Design and Implementation of a Write-based version of Exanet MPI

Michail Nikoloudakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *Polyvios Pratikakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Design and Implementation of a Write-based version of Exanet MPI

Thesis submitted by
Michail Nikoloudakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: [illegible]
Michail Nikoloudakis

Committee approvals: [illegible]
Polyvios Pratikakis
Associate Professor, Thesis Supervisor

[illegible]
Konstantinos Magoutis
Associate Professor, Committee Member

[illegible]
Vasileios Papaefstathiou
Assistant Professor, Committee Member

[illegible]
Emmanouil Ploumidis
PhD, Thesis Advisor

Departmental approval: [illegible]
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, April 2022

Design and Implementation of a Write-based version of Exanet MPI

Abstract

MPI is one of the leading communication protocols used in HPC (High Performance Computing) suites today due to its portability and scalability. Many HPC applications make use of MPI in order to enable communication between different processes. In the scope of the ExaNeST project, an HPC prototype was deployed in the CARV Laboratory of FORTH consisting of 512 ARMv8 cores coupled with FPGA logic. This prototype makes use of special network primitives designed to allow the low latency transmission of control messages as well as the efficient transfer of large data through the Exanet network. In order to exploit the aforementioned capabilities of the prototype, a highly optimized MPI implementation (Exanet MPI) was developed in the scope of the same project prior to our work. This implementation makes use of the prototype's communication primitives and manages to outperform the well known MPI implementation, MPICH (TCP/IP) by achieving up to 30x lower latency. Exanet MPI supports both an eager and a long communication protocol used for short and large MPI transfers respectively. The long protocol depends on emulated DMA reads and supports exclusively sender initiation. Sender initiation is defined as the ability of the sender of an MPI message to initiate the communication with the receiver by issuing an appropriate control message. Despite its simplicity, sender initiation does not let us exploit scenarios in which the receiver posts its request earlier than the sender. In addition, the use of emulated reads requires the receiver to notify the sender about the end of a DMA transfer through the use of an Ack control message which incurs extra latency. In this thesis, we design and implement from scratch a write-based version of the Exanet MPI that supports both sender and receiver initiation. With the use of DMA writes, we render the sender able to determine the end of a DMA transfer by itself without the need of acknowledgment from the receiver. Additionally, we take advantage of cases where a receive request gets posted earlier than a matching send request by letting the receiver initiate communication by asynchronously transferring its DMA related information to the sender. Consequently, a sender that posts its send request after the receiver, can immediately transfer data without the need of further synchronization with the receiver. However, simply adding receiver initiation support to the long protocol also infers some complications including (but not limited to) the significant increase of the eager protocol's latency. We propose our method for successfully facing the complications that arise from the support of receiver initiation and we also further optimize the long protocol by eliminating the need of some control messages. In order to break down the performance gain caused by our optimizations we develop in total 4 variants of the write based Exanet MPI. In each variant, we provide implementations for most point-to-point, collective as well as communicator manipulating functions. We describe the use cases of each developed variant and evaluate them against the already optimized read based original version of Exanet MPI on the HPC prototype. We offer insight into the ways our control path optimizations improve performance and the factors that let our implementation show more benefit. For the evaluation we use both microbenchmarks and real scientific applications. We show that our implementation can outperform the read based protocol by up to 50% in communication latency while also reduce the total execution time of specific applications by up to 10% (depending on the percentage of communication time they contain).

Σχεδιασμός και υλοποίηση μιας έκδοσης του Exanet MPI βασισμένης σε εγγραφή

Περίληψη

Το MPI είναι ένα από τα κυρίαρχα πρωτόκολλα επικοινωνίας που χρησιμοποιούνται στις πλατφόρμες HPC (Υπολογισμού Υψηλής απόδοσης) σήμερα λόγω της φορητότητάς και της κλιμακοσιμότητάς του. Πολλές HPC εφαρμογές κάνουν χρήση του MPI για να καταστήσουν εφικτή την επικοινωνία μεταξύ διφορετικών διεργασιών.Στα πλαίσια του έργου ExaNeST, ένα νέο HPC πρωτότυπο αναπτύχθηκε στο εργαστήριο CARV του I.T.E αποτελούμενο από συνολικά 512 ARMv8 πυρήνες συνδυασμένους με λογική FPGA. Το πρωτότυπο αυτό κάνει χρήση ειδικών μονάδων επικοινωνίας σχεδιασμένων να επιτρέπουν τη διάδοση μηνυμάτων συγχρονισμού με πολύ μικρή καθυστέρηση (latency) καθώς και την αποδοτική μεταφορά μεγάλων δεδομένων μέσω του δικτύου Exanet.

Προκειμένου να αξιοποιηθούν οι προαναφερθείσες ικανότητες του πρωτοτύπου, αναπτύχθηκε μια υψηλά βελτιστοποιημένη MPI υλοποίηση (Exanet MPI) στα πλαίσια του ίδιου έργου, πριν από την αρχή της δουλειάς μας. Η υλοποίηση αυτή κάνει χρήση των μονάδων επικοινωνίας του HPC πρωτοτύπου και καταφέρνει να επιτύχει καλύτερη απόδοση από την γνωστή MPI υλοποίηση MPICH επιτυγχάνοντας μέχρι και 30x μικρότερη καθυστέρηση διάδοσης δεδομένων (latency). Το Exanet MPI υποστηρίζει ένα eager και ένα long πρωτόκολλο επικοινωνίας για μικρές και μεγάλες μεταφορές δεδομένων αντίστοιχα. Το πρωτόκολλο long βασίζεται σε προσομοιωμένα DMA reads (read based) και υποστηρίζει εκκίνηση της επικοινωνίας αποκλειστικά από τον αποστολέα. Η "εκκίνηση από τον αποστολέα" ορίζεται ως η ικανότητα του αποστολέα ενός MPI μηνύματος να ξεκινήσει την επικοινωνία με τον παραλήπτη εκδίδοντας το κατάλληλο μήνυμα συγχρονισμού. Παρά την απλότητα της, η εκκίνηση αποκλειστικά από τον αποστολέα δεν μας επιτρέπει να εκμεταλλευτούμε περιπτώσεις όπου ο παραλήπτης καταχωρεί την αίτηση του νωρίτερα από τον αποστολέα. Επιπρόσθετα, η χρήση προσομοιωμένων DMA reads απαιτεί από τον παραλήπτη να ενημερώσει τον αποστολέα σχετικά με το τέλος μιας DMA μεταφοράς μέσω ενός Ack μηνύματος συγχρονισμού, κάτι που επιφέρει επιπλέον καθυστέρηση (latency).

Σε αυτή την εργασία, σχεδιάζουμε και υλοποιούμε εξ αρχής μια έκδοση του Exanet MPI βασισμένη σε εγγραφή (write-based) η οποία υποστηρίζει εκκίνηση επικονωνίας από τον αποστολέα αλλά και από τον παραλήπτη. Με τη χρήση DMA writes, καθιστούμε τον αποστολέα ικανό να αντιληφθεί το τέλος μιας DMA μεταφοράς δίχως να χρειάζεται επιβεβαίωση από τον παραλήπτη. Επιπρόσθετα, εκμεταλλευόμαστε περιπτώσεις όπου η αίτηση παραλαβής καταχωρείται νωρίτερα από την αίτηση αποστολής επιτρέποντας στον παραλήπτη να μεταφέρει ασύγχρονα στον αποστολέα όλες τις πληροφορίες που χρειάζονται για μια DMA μεταφορά. Συνεπώς, ο αποστολέας που θα καταχωρήσει την αίτηση αποστολής μετά τον παραλήπτη, μπορεί να μεταφέρει άμεσα τα δεδομένα του χωρίς να υπάρχει ανάγκη για περαιτέρω συγχρονισμό με τον παραλήπτη. Παρ'όλα αυτά, η απλή προσθήκη της δυνατότητας εκκίνησης από τον παραλήπτη στο long πρωτόκολλο επιφέρει κάποιες επιπλοκές συμπεριλαμβανομένης και της μεγάλης αύξησης της καθυστέρησης στο eager πρωτόκολλο. Προτείνουμε δικές μας μεθόδους για την επιτυχή αντιμετώπιση των επιπλοκών που προκύπτουν από την υποστήριξη εκκίνησης από τον παραλήπτη καθώς επίσης βελτιστοποιούμε περαιτέρω το long πρωτόκολλο εξουδετερώνοντας την ανάγκη χρήσης κάποιων μηνυμάτων συγχρονισμού. Προκειμένου να αναλύσουμε το όφελος απόδοσης των βελτιστοποιήσεων μας, αναπτύσσουμε συνολικά τέσσερις παραλαγές του

Exanet MPI βασισμένου σε εγγραφή. Σε κάθε παραλλαγή, παρέχουμε υλοποιήσεις για τις περισσότερες εκ των point-to-point, collective, και communicator manipulating συναρτήσεων. Περιγράφουμε τις περιπτώσεις χρήσης της κάθε παραλλαγής και αξιολογούμε τις παραλλαγέστης υλοποίησής μας έναντι της αρχικής, ήδη βελτιστοποιημένης, read based έκδοσης του Exanet MPI στο HPC πρωτότυπο. Εμβαθύνουμε στους τρόπους με τους οποίους οι βελτιστοποιήσεις μας στο μονοπάτι συγχρονισμού βελτιώνουν την απόδοση καθώς και στους παράγοντες που επιτρέπουν στην υλοποίησή μας να δείξει περισσότερο όφελος. Για την αξιολόγηση χρησιμοποιούμε τόσο microbenchmarks όσο και πραγματικές επιστημονικές εφαρμογές. Δείχνουμε ότι η υλοποίησή μας μπορεί να ξεπεράσει σε απόδοση το read based πρωτόκολλο προσφέροντας έως και 50% μικρότερη καθυστέρηση (latency) ενώ μπορεί να μειώσει το συνολικό χρόνο εκτέλεσης ορισμένων εφαρμογών έως και κατά 10%. (ανάλογα με το ποσοστό χρόνου επικοινωνίας που περιέχουν)

Acknowledgments

First of all, I am grateful to my supervisor Polyvios Pratikakis as well as to my advisor Manolis Ploumidis for their constant support and guidance throughout this thesis. I would also like to thank Associate Professor Konstantinos Magoutis and Assistant Professor Vassilis Papaefstathiou for agreeing to be members of my thesis' examination committee.

Special thanks to the CARV Laboratory members Pantelis Xirouchakis, Fabien Chaix, Marios Asiminakis and Astrinos Damianakis for their useful tips regarding the HPC Prototype, its hardware blocks' userspace API and benchmark applications used in the evaluation chapter of this thesis.

Last but not least, I would like to thank my family for their mental support and help which made the process of completing my studies a lot easier.

στους γονείς μου

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

High Performance Computing (HPC) is the practice of using parallel processing units in order to achieve much higher performance and perform complex calculations. Through years of technological advancement, the computation power required for HPC applications has started moving towards exascale. This fact constitutes a motive towards a general reconsideration of the modern HPC suites' design in order to keep their cost viable. To this end, a new HPC prototype has been deployed in the CARV Laboratory of FORTH, which makes use of novel, low cost communication primitives developed in the scope of the ExaNeST project. The proposed architecture used in that prototype offers near optimal message latency with minimum kernel involvement while at the same time keeping the cost low. The new communication primitives make use of an accelerated a custom packet-based hierarchical interconnect network called Exanet, developed in the scope of the same project. In order to enable support for the majority of HPC applications, an MPI runtime has also been deployed called Exanet MPI [16]. MPI is currently the dominant communication protocol used in HPC due to its portability, scalability and high performance. The MPI standard provides definitions of a wide range of primitive functions which focus mainly on message handling and are extremely useful for the development of parallel applications while it also determines how the features of the interface must behave in any different implementation. More specifically, MPI library consists of functions related to point point communication (between two processes) as well as functions related to collective communication regarding groups of processes. MPI implementations typically use two different protocols for transferring messages depending on their size. For small messages an **eager** protocol is used in which the sender sends the entire message to the receiver, where the receiver provides sufficient buffering space for the incoming messages. On the other hand, for large messages that cannot fit in a control message, a rendezvous **long** protocol is used in which the sender and the receiver negotiate before the data transfer takes place. The data transfer of a long rendezvous protocol usually happens through a Remote Direct Memory Access (RDMA) engine. A rendezvous long protocol can be sender initiated, receiver initiated or hybrid. Sender initiation means that the sending process is responsible for initiating the synchronization process with the receive before the DMA transfer takes place. Respectively, receiver initiation gives the capability of initiating the communication to the receiving process.

Exanet MPI supports both an eager and a long protocol. Exanet MPI's long protocol uses exclusively sender initiation and relies on **emulated DMA reads**. Specifically, a sending process initiates the communication by advertising the address of the send buffer to the receiving process. Subsequently, the receiver transfers the contents of the send buffer to its receive buffer by initiating a DMA write from the sender's side with the use of control messages. At the end of the emulated read, it notifies the sender about the end of the transfer with an acknowledgment control message. Exanet MPI manages to outperform MPICH by achieving up to 96% lower latency and thus offers HPC applications a high performing communication interface in the aforementioned HPC prototype. However, sender initiation, despite being the dominant form of communication initiation used in the long protocol of most implementations, also has some drawbacks. First, in cases where a receiving process posts a receive request before the posting of matching send request by the sender, the protocol does not let the receiver initiate the communication. In that way, the intermediate time between the receive's and the send's posting is not exploited. Secondly, relying on emulated reads requires the receiver to notify the sender about the end of DMA transfers which incurs extra latency. Consequently, it was a topic of our research whether the already optimized Exanet MPI can be urther improve.

In this thesis, we designed and implemented from a scratch a write-based version of Exanet MPI. This new MPI implementation relies exclusively on DMA write operations and supports both sender and receiver initiation. With the use of DMA writes, we render the sender able to determine the end of a DMA transfer by itself without the need of an acknowledgment control message from the receiver. Specifically, the receiver is able to initiate communication by asynchronously transferring its DMA related information to the sender. As a consequence, a sender that posts its send request after the receiver can immediately perform a DMA write without the need of further synchronization with the receiver. Our implementation combines the benefits of both types of initiations of the long protocol while supporting an eager protocol as well. Through implementing four variants of the write-based implementation we managed to find out and face the complexities the coexistence of receiver and sender initiations infers and even outperform the already existing version of the Exanet MPI. In addition, we broke down the performance gain of our implementation by comparing all four variants with each other, each one of them contributing different optimizations. Our MPI implementation relies on reimplementing most of the point-to-point, collective and communicator manipulation MPI routines and delegating them through the Exanet network by making use of network primitives of the HPC prototype for inter-process communication. Initially, we created an MPI variant that combines sender and receiver initiation in the long protocol but also contains some overheads that arise from the coexistence of the two initiation methods. Secondly, we further optimized the long protocol of receiver initiation by eliminating the need of one control message and, in a third axis, we came up with a method of eliminating the overhead receiver initiations infers to the eager protocol. Lastly, we implemented an optimistic (speculative) variant of our implementation that makes some assumptions that may partly violate the MPI standard but can be used to improve performance with some applications. Our evaluation showed that our implementation can compete and outperform the read-based version of the Exanet MPI in both the long and eager protocol. Overall, we make the following contributions:

- We design a write-based protocol that utilizes the network primitives of the new HPC prototypes while supporting

- We detect and face all the complexities that arise by supporting both sender and receiver initiation

- We manage to optimize the control path of the long protocol in both sender and receiver initiation scenarios. Specifically, with our optimizations, the long protocol requires one less message in comparison to the read-based version's control path. In scenarios where a receive request gets posted first, our implementation uses 2 less synchronization messages compared to the read-based variant.

- We propose an optimistic variant that can suggest new changes to the MPI standard

- We underline the impact of the topology of MPI processes and how it can affect performance.

- We attempt to find a sweet spot for the cost and benefit each of our optimization provides and suggest the most suitable variant for different scenarios.

The rest of this thesis is organized as follows. We first give the necessary background information on MPI, the HPC prototype, the network primitives it uses and the Read-based MPI implementation. In Chapter 3 we provide a detailed analysis of our implementation as well as our thinking process towards aeach one of the variants implemented. Chapter 4 contains our thorough evaluation's methodology and results while in Chapter 5 we present related work from the academic literature. Finally, in Chapter 6, we sum up our conclusions and discuss future work.

Chapter 2

Background

2.1 MPI

As mentioned in the introduction, our work includes the reimplementation of several MPI routines. MPI is a specification for message passing libraries designed to function on parallel computer architectures and it is maintained by the MPI Forum[19]. Its purpose is to constitute a standard for writing message passing programs while offering portability, efficiency and flexibility. Originally, MPI was designed for distributed memory architecture but through years of development and technological advancement. MPI supports both distributed and shared memory as well as hybrid architectures. Up to this day, MPI has managed to replace all previous libraries regarding message passing and is considered a standard supported on virtually all HPC platforms. There are many different implementations [20, 21] of the MPI Standard which have emerged through the years by different vendors. However, due to the standard's portability, source code that uses MPI can be used with different MPI implementations with little or no modification at all. The MPI standard undergoes constant changes and improvements and, up to the time of writing, its latest version is MPI-3 which consists of more than 430 routines. MPI's communication routines can be split into two major categories: Point-to-point and Collective routines. Point-to-Point routines are functions that regard communication between two processes while collective routines include all the process of a communicator in the data exchange. A communicator can be defined as a subgroup of processes. Each MPI process of a communicator is assigned an **MPI Rank,** an integer that uniquely identifies it in that communicator. At the beginning of execution, a default communicator gets created, called MPI_COMM_WORLD, which includes all the running MPI processes. A brief look on some examples of point-to-point and collective functions can be helpful for understanding their implementation described in the rest of this thesis.
The most commonly used point-to-point functions are **MPI_Send** and **MPI_Recv** [22] which have the respective signatures:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

Both functions accept as arguments a pointer to a send/receive buffer, the datatype of the transferred elements (eg., MPI_INT, MPI_CHAR etc), the number of elements to be transferred (count), the MPI rank of the source or destination process, a communication tag and the respective communicator the transfer regards. These two function are both **blocking** functions which means that they will block until the user is able to use the buffers without worrying whether the communication has ended. However, MPI also offers non blocking functions like **MPI_Isend** and **MPI_Irecv** which return immediately and the user has to use other complementary functions like **MPI_Wait** in order to determine whether they can reuse their buffers.

While reimplementing communication routines, one must take into account that the guarantees of the MPI standard must not cease to apply. For instance, it is guaranteed by the MPI Standard that a send function will always match with a receive of the destination rank of the same communicator, which denotes the sending process' rank as source and uses the same communication tag. Thus, the source/destination **ranks**, the **communication tag** and the **communicator** are the **matching attributes** of point-to-point functions. It is also guaranteed that all messages of a specific rank, tag and communicator combination are going to get received by the receiver in the exact same order they were sent. Thus, the FIFO property is preserved in point-to-point communications. It is worth noting that receive functions do not need to denote the exact size of the data they want to receive but rather the maximum size they can receive. As a result, it is correct for a receive request to get matched by a send request of equal or smaller size. Receive requests can use wildcards like **MPI_ANY_SOURCE** and **MPI_ANY_TAG** in order to designate all other ranks and tags, respectively, as matching attributes. As one can observe, the MPI_Recv function has one more argument of type **MPI_Status**. MPI_Status is an internal data structure whose contents may vary among different MPI implementations. This object consists of at least three integers which represent a)the source rank, b)the communication tag and c) the actual size of the transfer. During a communication, these integers take the appropriate values in order to enable the user to find out the aforementioned information after the communication ends.

An example of a collective MPI function is MPI_Broadcast, which broadcasts a message sent by a root process to the rest of the processes of the communicator. All the processes must call the function with the exact same arguments (except for the buffer address) for the communication to be executed correctly.

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
```

Other MPI collective functions include MPI_Reduce, MPI_Scatter, MPI_Gather [22]etc. each providing functionality in accordance to the user's need.
Any MPI program should invoke **MPI_Init** or **MPI_Init_thread** before calling any other MPI routine.
MPI also supports multi-threaded applications and different threading modes. A multi threaded MPI program may request one of the following thread support levels:

- **MPI_THREAD_SINGLE** Only one thread will execute.
- **MPI_THREAD_FUNNELED** The process may be multi-threaded, but only the main thread will make MPI calls
- **MPI_THREAD_SERIALIZED** The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads
- **MPI_THREAD_MULTIPLE** Multiple threads may call MPI at the same time, with no restrictions.

Some thread support levels infer some changes in the development of our implementation, which we also discuss in the next chapters.

2.2 Packetizers and Mailboxes

As it will get more clear in the rest of the thesis, in order to achieve the correct matching between MPI requests, an MPI implementation should utilize a mechanism for the delivery of intermediate control messages required for the processes' synchronization. In order to achieve the delivery of such messages, our prototype uses special hardware blocks called *Virtualized Packetizers* and *Virtualized Mailboxes.* In this section, a very brief description of these blocks is provided.

Packetizers and Mailboxes have been designed for latency-critical operations and support the sending and receiving of messages respectively. More specifically, the prototype makes use of an environment in which all memory locations belong to a Global Virtual Address Space and can be addressed by network packets. A network packet is created by a Virtualized Packetizer and should denote a virtual address as its destination as well as specify a ***protection ID*** (PDID). The protection ID is process group-specific and is used by the hardware to safely check the initiator's access rights on particular locations of the virtual address space. Additionally, a network packet must contain a ***node ID ,***which constitutes the physical location of the node in which the packet's destination virtual address is contained. The virtualized packetizer offers a limited number of virtual interfaces (pages) that can be allocated to different threads and processes. In order to make use of that hardware block, a process may acquire a virtual interface of a packetizer from a kernel driver. The driver writes into a special hardware register the PDID of the requesting process and returns a virtual address which is mapped to the physical address of the packetizer page. Subsequently, that process is able to use the packetizer to target any location of the virtual address space or a virtual mailbox. At the time of writing, a packetizer can transmit messages of up to 64 bytes.

A virtualized mailbox is a hardware block responsible for receiving messages sent from a packetizer. Processes can again acquire mailboxes from a kernel driver which associates each virtual mailbox interface with the PDID of the corresponding process group. When a packetizer sends a message to a virtualized mailbox, the receiving hardware checks the packet's PDID and tries to match it against that of the virtual mailbox. A user can poll for new arrivals in their mailboxes by reading from a virtual address that has been memory-mapped to the physical address of their virtualized mailbox. It's worth noting that the described network primitives guarantee the FIFO delivery of control messages. This means that all messages sent from a process A will get received by a process B in the same order they were sent. This fact plays a crucial role in preserving MPI's FIFO property described in section **2.1**

2.3 RDMA Engine

While packetizers and mailboxes are sufficient for the transmission and receiving of short and low latency control messages, for large data transfers the prototype utilizes a simple virtualized RDMA engine, with coordinated units running at the sending and receiving endpoints. The RDMA engine provides an effective multi-path transport based completely on hardware allowing to bypass the kernel stack on I/O operations. An RDMA operation transfers a message between two locations of the aforementioned Virtual Address Space: the source, which in our prototype is always local to the sending point's engine that will realize the transfer, and the destination, which is local to the receiving point's engine. In order to signify the end of the transfer to the receiving point, the DMA engine delivers an additional notification message to an arbitrary virtual address local to the receiver, called **notification address** in the rest of the thesis. By polling on the notification data (i.e., the contents of the notification address), a receiving process is able to determine the end of an incoming transfer. The RDMA engine is used in this work for the transfer of large MPI messages it offers high throughput and requires no copies at all. For converting local buffer addresses to virtual Exanet addresses within the Global Virtual Address Space, special routines are provided that make use of a DMA offset, different for each node. More information about the DMA related routines is available in the next section.

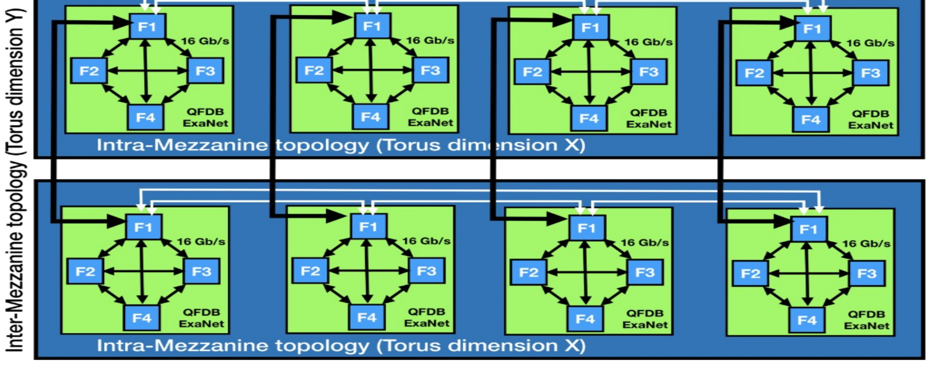
2.4 User Level Communication Libraries

For the utilization of the hardware blocks described in **2.2** and **2.3,** a user-space API that allows user-level access to them has been deployed. As a result, this communication library is widely used in this work. By making use of that API, a process is able to attach a virtual interface of the mailbox and packetizer that reside on the local compute node as well as acquire all the relevant information regarding the Ids and DMA offsets of the node. More precisely, the ***MBOX_ATTACH()*** and ***dma_alloc_chan();*** `routines are used in order to attach a mailbox/packetizer interface and allocate a DMA protection domain respectively. Both functions return handles which can be used by the user in order to initiate DMA transfers and poll for new messages in the mailbox. Additionally, functions like ***get_exanet_offset(dma_channel), MBOX_GET_PROTECTION_ID() and getBoardID()*** return the aforementioned node specific DMA offsets and IDs. Each node should be aware of this information regarding any other node it wishes to transfer data to. When a process has obtained the Protection ID and node ID of another node, it can derive the remote node's Mailbox Address and, subsequently, use the packetizer to send messages to its Mailbox by calling

__MBOX__ENQUEUE__64B(address, message) function of the User Level API. Respectively, a process is able to read its mailbox for incoming messages using dequeuing functions like ***__MBOX__TRY__DEQUEUE__64B*** and providing the mailbox handle returned from MBOX_ATTACH. It's worth mentioning that attaching functions like MBOX_ATTACH and dma_alloc_chan are the only functions of the API that involve the kernel. As far as the RDMA engine is concerned, a user can initiate DMA transfers without requiring kernel intervention by using ***dma__init__write(dma__handle__t *handle, void *src, void *dst, long size, void *notif_addr, uint64__t *notif_data)*** and passing to it as arguments the DMA handle returned by ***dma__alloc__chan()*** as well as the source and destination buffers, the size of transfer in bytes, the notification address of the receiver and the notification data to be written to the remote notification address. It should be noted that the destination buffer address and notification address should be valid Exanet addresses returned from the function **dma__get__exanet__addr(void *addr, void *offset);`** which transforms a local address (addr) to an Exanet address using the receiving node's DMA offset. The process that initiates the DMA transfer, can determine its completion state by calling ***dma__test__transfer(dma__transfer__t *transfer);`***and passing to it, as argument, the handle returned by ***dma__init__write.*** The receiver can also determine the end of the transfer by polling the contents of the notification address.

2.5 The HPC Prototype

In the scope of the ExaNeST project, a new HPC cluster has been deployed in the CARV Laboratoty of FORTH [23], which utilizes the network primitives described in the previous sections and gets used for the testing and evaluation of our implementation in this thesis. At the time of writing, the prototype consists of 8 mezzanines, each one of them carrying 4 Qaud FPGA Daughter Boards (QFDBs) for a total of 128 FPGAs.e FPGAs are Xilinx Zynq Ultrascale+ devices (ZCU9EG), featuring four (4) ARM-A53 [31], 16-GByte DDR4 each. As a result, our prototype contains in total 512 ARM v8 processors. The platform supports two different networks. The first one being a custom packet-based hierarchical interconnect realized over high speed serial links, developed by FORTH and INFN [24] called Exanet, which is used by the primitives described in Sections 2.2. and 2.3. The second network our prototype supports is a common 10G Ethernet interface. Within a QFDB, there is an all-to-all connectivity among all four of its FPGAs (called F1, F2, F3 and F4) through the Exanet interconnect offering a link capacity of 16 GB/s. In addition, all QFDBs are connected with each other in a 3D Torus topology using 10 GB/s serial links. Depending on the position of two FPGAs in the 3D Torus, a different number of network hops may exist between them. The number of hops between two FPGAs can play a crucial role in latency as it is shown in the next chapters. Figure 2.1 depicts an example of the prototype's 3D Torus topology. QFDBs are illustrated as green boxes which contain 4 FPGAs interconnected through 16 GB/s links while the links connecting different QFDBs of the same mezzanine as well as different mezzanines are depicted as white and black arrows respectively. Both types of inter QFDB links have a capacity of 10 GB/s as stated.


Inter-Mezzanine topology (Torus dimension Y)
F1
F2
F3
F4
16 Gb/s
QFDB
ExaNet
Intra-Mezzanine topology (Torus dimension X)
Figure 2.1: HPC prototype's Example topology

2.6 Read-based Exanet MPI Implementation

In order to exploit the described architecture and network primitives for the execution of HPC applications, an MPI implementation, prior to our work, was deployed. That MPI implementation is based on **emulated DMA reads** in order to transfer large messages while it uses the packetizer to transfer messages of size of less than 40 bytes between two MPI processes. This implies that the implementation supports two different kinds of MPI Communication protocols; The l**ong rendezvous protocol** and the **eager protocol**. The former protocol is used for the transfer of large messages and the eager protocol is restricted to small messages that can “fit” inside a packetizer message.

The long rendezvous protocol relies exclusively on sender initiation. More precisely, a sending MPI process is able to initiate the communication by transmitting an **RTS** (Request to Send) control message to the receiving process, notifying it about the posting of a send MPI primitive (eg. MPI_Send) on its side. The RTS control message should contain the address of the send buffer, the size of transfer as well as the matching attributes of the send request (tag, MPI_Rank of the sender) and the ID of the communicator). At this point, it should be mentioned that each MPI process preserves internal data structures that are used to store Posted and Received MPI requests. The **Posted MPI Requests** data structure regards communications posted by the process itself (by MPI_Send) while the **Received MPI Requests** data structure regards incoming control messages emerged from communications posted by other processes. For instance, a sending process should insert a new object on its Posted MPI Requests structure as soon as it calls MPI_Send. The receiving process that receives the RTS control message from the sender should search its Posted MPI Requests data structure for an already posted matching receive request. If no matching receive request exists yet, the receiving process stores the RTS message in the Received MPI Requests for future matching. In either case, when a receive requests gets matched by a matching RTS, it transmits a **CTS** (Clear to Receive) control message targeting an R5 AMR microprocessor [32] in the sender’s side and instructing it to perform a DMA write using the send buffer contained in the RTS message as source buffer and designating the receiver’s receive buffer as destination. Through the CTS control message, the notification address of the receiver is also conveyed. As soon as the R5 microprocessor receives a CTS message it initiates a DMA write and informs the receiver about its completion by writing into its notification address. Since the DMA write is actually initiated by the receiver, we call that operation an **emulated read** operation since actual DMA reads are not supported by our RDMA engine. In addition, since the sender process did not initiate the write itself, it cannot determine whether it is completed without receiving an acknowledgment control message from the receiver. As a consequence, the receiving process transmits an **Ack** control message informing the sender about the end of the emulated read.
In Figure 2.2, a scenario where the long rendezvous protocol is used is illustrated. We assume that the left vertical line depicts the execution of an MPI process with rank **S** while the right line depicts the execution of the process with rank **R**. At timestamp *t1,* the sending process invokes a blocking sending primitive (e.g., MPI_Send) denoting rank **R** as destination rank, **T1** as the communication tag and **bs** as the send buffer. As a result, it inserts a new request into the Posted MPI Requests data structure, issues an RTS control message that contains the matching attributes of the request and the address of the send buffer and block its progress waiting for an Ack message. At *t2,* the receiving process receives the RTS message and due to the fact that there is no receive request posted to match it yet, it stores it into the Received MPI Requests data structure (entry = {S, T1}). At *t3,* the receiver calls a receive MPI primitive (e.g., MPI_Recv) denoting rank S as source rank, T1 as communication tag and **br** as receive buffer (for simplicity, communicator ID is omitted. The process searches its Received MPI Requests and it manages to perform a match with the RTS stored at *t2.* Consequently, the receiving process transmits a CTS control message to the sender’s R5 microprocessor instructing it to perform a DMA write using bs as source and br as destination buffer. Afterwards, it blocks waiting for a change in the contents of the notification address that will indicate the completion of the transfer. At *t4,* the microprocessor initiates the transfer and at *t5,* the notification arrives. This fact makes the receiver unblock its progress and transmit an Ack message to the receiver notifying about the end of the emulated read. The Ack message also contains the request’s matching attributes in order to successfully match the send request at the sender’s side. At *t6*, the sender receives and matches the Ack message to the send request and

unblocks its progress. Ultimately, both processes clear the respective requests and the send and receive MPI primitives return.

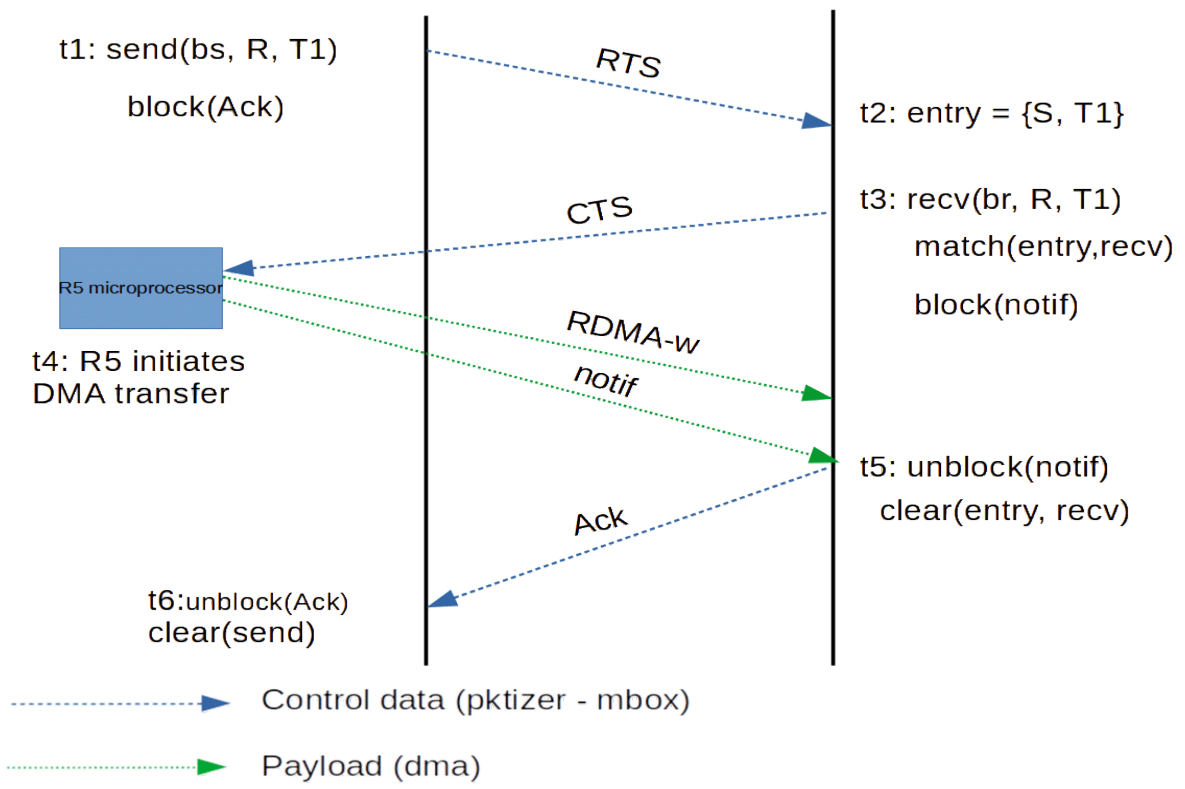


Figure 2.2 Long Protocol of the Read Based Exanet MPI

The eager protocol also relies on sender initiation and it is much simpler as it does not involve DMA transfers at all. A sending process that performs a send of size less than 40 bytes can immediately transmit the message to the receiver without requiring any rendezvous or synchronization beforehand. Instead, it piggybacks the data to get transferred in a packetizer message which also includes the request's matching attributes in order to get successfully matched with a receive request of a receiver. This new control message is called **Env+D** (Envelope and Data, where ***Envelope*** is another name used in scientific literature for the matching attributes of a message) Additionally, the sending process doesn't need to insert any request to the Posted MPI Requests since eager sends do not need to get matched by Ack messages.

Figure 2.3 illustrates an example of eager communication in the read-based protocol. The receiving process posts a blocking receive request at *t1* and denotes S as the source rank and T1 as the communication tag. After inserting the request into the Posted MPI Requests data structure, the process blocks its progress waiting for an RTS or Env+D control message. At *t2,* the sender posts an eager send and immediately transmits the Env+D message that includes as payload the contents of its send buffer and returns without inserting any object in the Posted MPI requests. At *t4,* the receiving process receives the Env+D and matches it against the receive posted at *t1*. Subsequently, it unblocks its progress, copies the payload of the Env+D message to its receive buffer, it clears its state and returns.

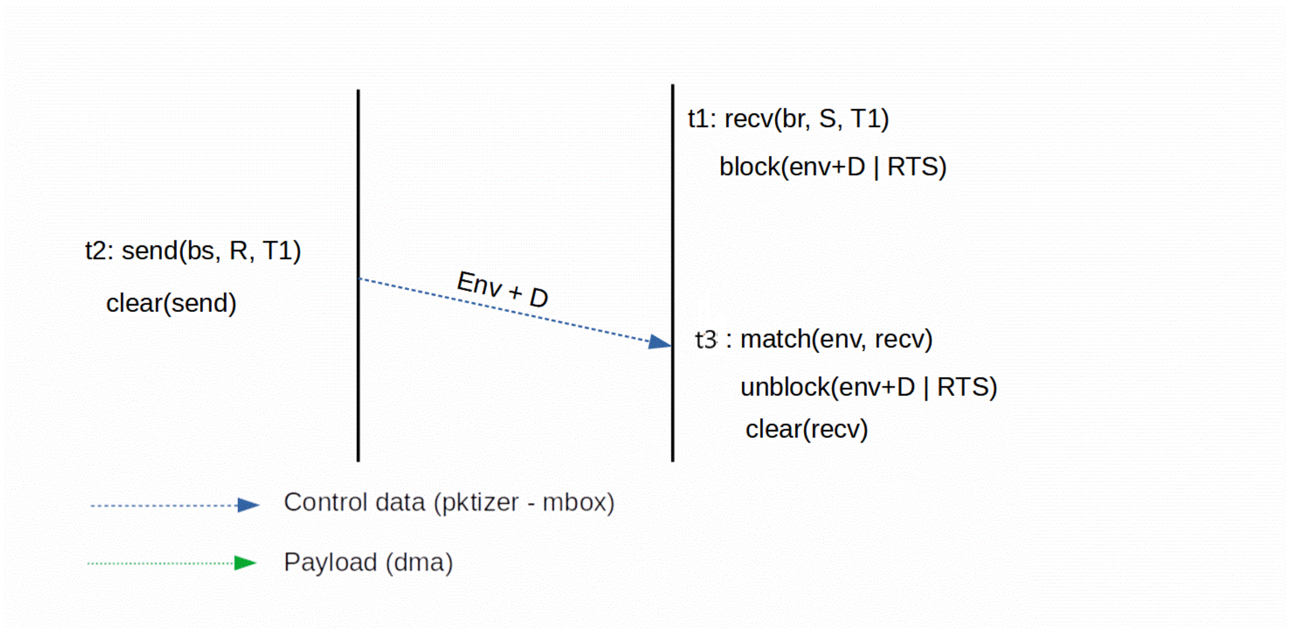


Figure 2.3 Eager protocol of the Read Based Exanet MPI

As one can notice, the eager protocol offers optimal performance since it constitutes the simplest form of communication possible between two MPI processes. Moreover, since it regards messages of small size (less than the size of a packetizer message, which is 64 bytes), the memory copy (from the Env+D message's payload to the source buffer of the receiver) it requires has not noticeable latency.

However, the long rendezvous protocol has some serious disadvantages. First of all, because of the fact that it is based on an emulated read, the receiver must inform the sender about the end of the transfer by issuing a control message (i.e., Ack) back to the sender. In addition, the inability of the receiver to initiate MPI communication itself does not let us exploit cases where the receive request gets posted earlier than a matching send. Instead, the receiving process has to wait for the sender to initiate the communication with the use of an RTS control message, thus not taking advantage of the intermediate waiting time. In the next chapter, we present our attempt to develop a write-based version of the Exanet MPI, addressing the drawbacks of the read-based variant.

Chapter 3

Design and Implementation of the Write-based Exanet MPI Protocol

Designing and implementing the write-based variant of Exanet MPI was a highly experimental process which resulted to the development of different variants of the write-based protocol itself. In this chapter, the basic variants of the protocol that emerged from this work are getting described and partly evaluated in an attempt to underline our thinking process towards the final version of the write-based protocol. Additionally, several details regarding the implementation of certain aspects of the protocol, common to all of the variants are getting presented.

3.1 Designing a preliminary sender and receiver initiated write-based protocol

3.1.1 Designing a preliminary sender and receiver initiated write-based long protocol

As already mentioned in the previous chapters, an important weakness of the read-based variant of Exanet MPI is the inability of a sending process to become aware of the end of a DMA write since the write gets initiated from the receiver's side in an attempt to emulate a DMA read operation. Thus, an acknowledgment (ACK) control message sent from the receiver is the sole way of the sender determining the end of the DMA transfer which is expected to cause measurable overhead. For that reason, one of our initial goals was to eliminate the need of that control message by rendering the sending process the initiator of the DMA write. In addition, we aimed to exploit the new DMA feature of delivering the end of a transfer to an arbitrary remote address of the receiving process' address space. More specifically, as described in Section 2.2, he DMA engine of the developing HPC prototype, supports notifying the receiver that a DMA transfer is over by performing an additional write to a remote address, called **notification address**. Subsequently, the receiving process can determine the end of the DMA data transfer by polling on the notification address and waiting for a change in its contents. Another one of our main motives for working towards a write-based protocol of the Exanet MPI is the fact that the preexisting read-based protocol relies exclusively on sender-initiated communication. This renders the read-based protocol unable to take advantage of scenarios in which the receiving process posts a receive request (e.g., MPI_Recv) before a matching send request (e.g., MPI_Send) gets posted by a sending process. This kind of early receive posting will be denoted in the rest of this thesis as ***fast receive***. In such a case, in the read-based protocol, the receiving process cannot initiate the communication by advertising its receive buffer and local notification addresses to the sender but instead it has to wait for an RTS control message from the sending process. As a consequence, the matching happens only in the receiving process' side. In order to address that drawback of the read-based protocol and allow the initiation of the communication from the receiving process's side, a new type of control message was introduced called Request-to-Receive (RTR). With the use of an RTR control message the receiver is able to advertise its receive buffer and local notification address to the sending process (just like with a CTS message) as soon as it posts its receive request., asynchronously, without waiting for the posting of a matching send as it happens in the read-based implementation. In addition to the DMA related information, an RTR control message also contains all the necessary matching attributes thus enabling the request matching to take place on the senders' side as well. After receiving an RTR message, a sender is aware of all the necessary information required to perform a DMA write operation without the need for further synchronization. In such a case, however, due to the absence of the RTS conrol message in the communication, the receiving process may be unable to fill its MPI_Status struct because it misses essential information like the tag of communication (in case MPI_ANY_TAG was used) or the size of the transfer. Additionally, without a control message from the sender, the receiver is not able to mark its receive request as matched before the communication ends which poses the risk of it becoming mistakenly matched by a future RTS message regarding another receive request with the same matching attributes. The aforementioned reasons render the use of another control message, called Envelope (Env), necessary. This type of message contains the information required to fill the receiver's MPI_Status struct and gets transmitted from the sender before the DMA transfer takes place. The use of an Envelope control message may seem to infer the same cost as the sender initiated scenario but in reality, fast receives still outperform the sender initiation used in the read-based protocol since they allow asynchronously transferring the matching iand DMA related information to the sender before it posts its send request. As a result, a typical long write-based protocol should contain the following control messages:

- **RTS** (Request to Send): The RTS control message constitutes the message containing all the attributes used to match the send request with a matching receive request at the side of the receiver as well as the envelope information needed in order to fill the MPI_Status struct of the matched receive (ie. Rank of the sender, communication tag, size of transfer). It gets transmitted from the sending process to the receiver and it is used to initiate the communication between them.
- **CTS** (Clear to Send): The CTS control message conveys, besides the necessary matching information, the address of the receive buffer which will be the destination of the sender's DMA write as well as the local notification address for that specific write . This message always gets transmitted in response to an already received RTS message and it indicates that receiver is ready to receive DMA data.
- **RTR** (Request to Send): The RTR control message contains the same DMA information as the CTS message. Unlike CTS, a receiving process issues an RTR message when no matching RTS has arrived at the time the receive request gets posted. It is, thus, a message used to initiate communication from the receiver's side.
- **Env (**Envelope): The Env control message contains the essential envelope information required for the completion of the MPI_Status struct. It gets issued by the sender just before the DMA transfer unless a RTS control message has already been issued for the same receive request before.

In Figure 3.1 the case where the sender arrives first at a matching send-receiver pair is presented. At *t1*, the sender posts a send request designating *bs* as a source buffer and denoting *R* as the destination rank. The corresponding communication tag is denoted as *T1.* Since no matching receive request exists in its Received MPI Requests, the sender posts a new request object in the Posted MPI requests and issues a request-to-send (RTS) message to the receiver specifying the communicator, tag and desired size of the transfer. It is assumed that the send request emerged from a blocking MPI primitive like MPI_Send. For that reason, the sending process blocks its execution waiting for either a matching CTS or RTR message. After receiving the RTS message, the receiver temporarily stores that received send request to an internal data structure used to store incoming requests at t2 (entry = {S, T1}). At t3>t1, the receiver posts a matching receive request and after matching it against the formerly received send request, it sends back a clear-to-send (CTS) message specifying the destination buffer (*br*) where it wishes to receive the data along with the virtual address where it will expect a notification of the DMA write's completion. We assume that the receive request emerged from a blocking MPI primitive like MPI_Recv, thus the receiving process blocks its execution while waiting for the DMA notification. Upon receiving the CTS control message, the sender uses the information that message contains in order to match it with the send it posted at t1. When the CTS message gets successfully matched, the sending process writes the data to *br* along the corresponding notification to the notification address. Note that since the DMA is initiated from the sender's side, the sender knows when the transfer is complete and does not require an acknowledgment back from the receiver, unlike the read-based protocol. At the end completion of the communication, both processes clear the send and receive requests from their respective data structures. More information regarding the mechanisms used in order to receive control messages as well as the data structures necessary for storing MPI communication requests are available in the next chapters.

In the scenario illustrated in Figure 3.2 the receiver posts its receive request at *t1* before the sending process posts its send request. Consequently, the receiving process, having received no matching RTS yet, initiates the communication by transmitting an RTR control message to the sender advertising the destination buffer (*br*) and the notification address, specifying the communicator, the communication tag and inserting a new receive request in its Posted MPI Requests denoting S as the source rank. After the issuing of RTR, the receiver blocks its progress waiting for either a matching Env or RTS message and subsequently for a change in the contents of the notification address. The sending process, after receiving the RTR message at *t2*, stores a specific receive request in its internal data structure regarding incoming requests since there is no posted send to match it yet. At *t3,* a matching send gets posted by the sender. By matching its send request with the previously received RTR successfully, the sender is aware of the destination buffer and notification address of the receiver and performs the DMA write operation after issuing the necessary Env message. As a result, the receiving process unblocks its progress after successfully receiving the Env message and observing the write the sender performed on the notification address' data. Both processes, ultimately, clear their respective requests.

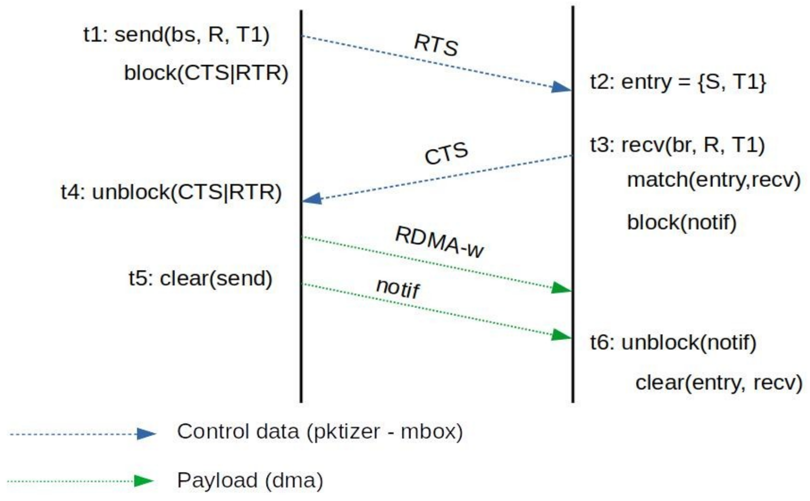


Figure 3.1: Sender initiated communication in the write-based Exanet MPI

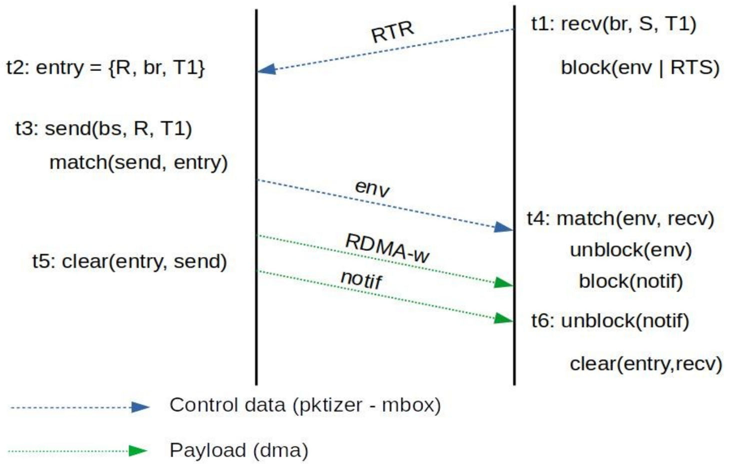


Figure 3.2: Receiver initiated communication in the write-based Exanet MPI

As it becomes apparent, the presented write-based long protocol offers more flexibility and timing exploitation capabilities since it allows the initiation of the communication from both the sender's and receiver's side. This fact renders the case where a receive request gets posted first nearly optimal, since the synchronization needed since the posting of a matching send request gets minimized. However, the possibility of both sender and receiver initiation for the same transfer can give rise to some complications which we examine in the next paragraphs.

At this point, we should remind that the MPI standard allows the use of a wildcard called MPI_ANY_SOURCE at the place of the source rank of a receive request, designating any rank of the denoted communicator as a possible sender of the expected message. This fact obliges to take into account several scenarios both in the design as well as the implementation of the write-based protocol. As it is easily conceivable, in the case where an MPI receive request gets posted earlier than the matching send request, it is not feasible to issue RTR messages towards all the other ranks of the communicator. Even in small communicators, sending the same RTR control message to all other ranks will most likely lead to an erroneous result since more than one sending rank may attempt to perform the DMA transfer to the same buffer. In addition, there is no way for each of the receivers of that RTR message to know which of them has received it first. As a consequence, the receiver initiation gets unavoidably suspended while receive requests that use MPI_ANY_SOURCE remain active in an MPI program.

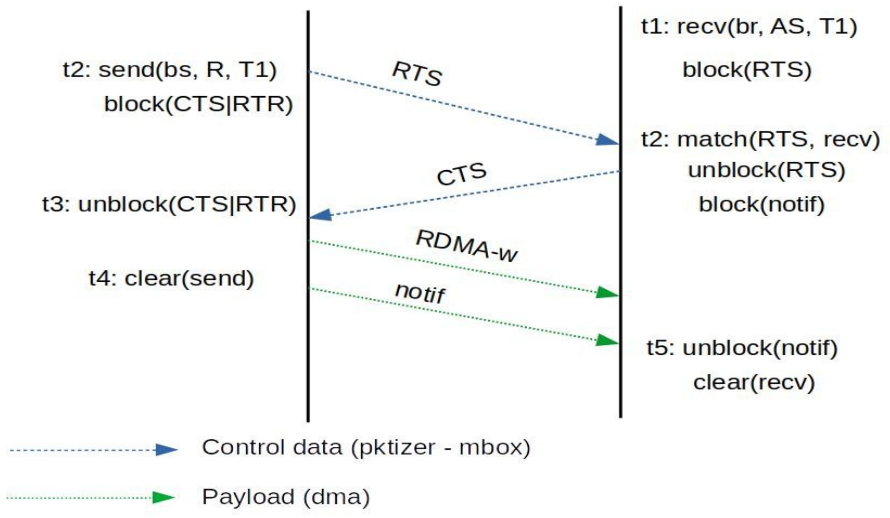


Figure 3.3: MPI_ANY_SOURCE receiver initiation suspension

In Figure 3.3 the case of a receiver posting its receive request designating MPI_ANY_SOURCE as source is depicted. Despite the fact that the receiver posted its receive request before the posting of the matching send request by the sender, it cannot initiate the communication with the issuing of an RTR control message due to the aforementioned complications. Instead, it inserts the receive request in its internal data structure regarding posted requests and waits for a matching RTS. On the sender's side, the progress is identical to the one depicted in Figure 3.1 where the sending process, being unaware of any matching posted receive request, attempts to initiate the communication by transmitting an RTS control message to the receiver. It is worth noting that the RTS control message also conveys the MPI Rank of the process initiating the communication, thus rendering the receiver able to log that information in the MPI_Status struct that regards the specific receive request. After successfully matching the received RTS message, the receiver issues a response CTS message thus advertising the necessary DMA information to the sender which in turn performs the DMA write. The communication ends in the exact same way it was illustrated in Figure 3.2 with both processes unblocking and clearing their state. At this point, it must also get clarified that the receiver initiation suspension does not regard only the receive request that uses MPI_ANY_SOURCE but rather **all** receives that get issued while MPI_ANY_SOURCE is still in use.

Figure 3.4 presents an example of a receiving process posting two non blocking receive requests at *t1* and *t2* respectively. The first receive request makes use of MPI_ANY_SOURCE while the second denotes rank S as the source rank. As we observe, both requests do not issue RTR messages, thus, RTR suspension is not limited to only the first request. If the second receive request did issue an RTR message, there would be the risk of the sender matching that RTR upon posting and perform the DMA write to the buffer of the second receive request and in that way violating the FIFO guarantee of the MPI Standard.

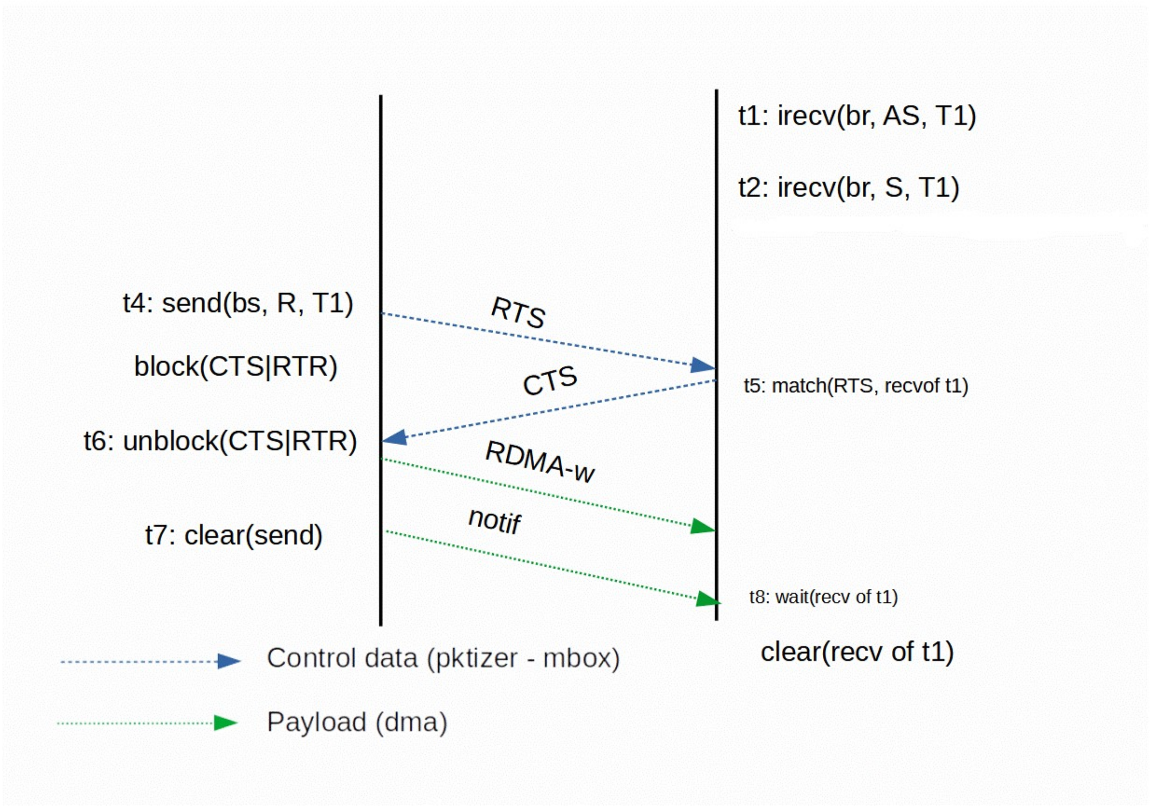


Figure 3.4: MPI_ANY_SOURCE receiver initiation suspension, non blocking receives

Another complication that may arise in a both receiver and sender initiated write-based protocol is the possible concurrent posting of both the send and the receive requests by the respective processes. In such a scenario, both of the processes issue their initiatory control messages (i.e., RTS, RTR). However, both type of messages get treated like an Env and CTS message respectively. More precisely, a receiver which has already sent an RTR message and receives a matching RTS will first check whether that RTS matches an already posted receive request before treating it as an attempt to initiate a new communication. If there is already a matching receive posted, then the receiver uses the information included in the RTS message in order to fill the MPI_Status struct and discards that RTS. Similarly, a sender which has already initiated the communication will treat an RTR control message, which matches that request, as a CTS message and will not store a new receive request in its internal data structure. The described case is illustrated in Figure 3.5. This scenario is pretty similar to the receiver initiated communication depicted in Figure 3.2 with the difference that the RTS message substitutes the Envelope message.

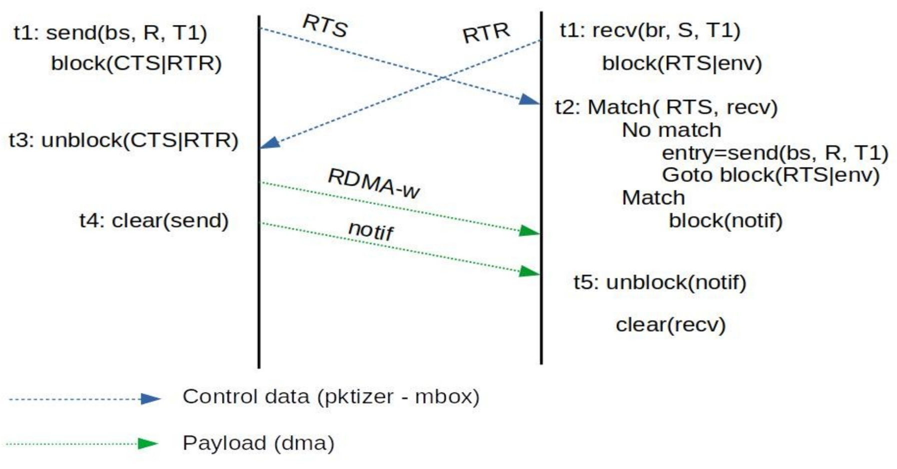


Figure 3.5 Concurrent Receiver and Sender initiation

3.1.2 Designing a preliminary sender and receiver initiated write-based eager protocol

As mentioned in 2.5, the read-based protocol uses an eager protocol in order to transfer messages of up to 40 bytes (a packetizer message has a maximum size of 64 bytes out of which, 24 bytes constitute the messages Envelope). The write-based and the read-based protocol coincide in this regard. For such small messages we choose to exploit the low latency mechanisms of the mailbox and packetizer hardware blocks described in 2.2 and, as a consequence, avoid the startup latency of the DMA engine. For that reason, an eager message gets packed as *payload* together with envelope data into a new control message call Envelope and Data (**Env+D**). This message gets issued by the sending process and has the receiving process as destination. Respectively, a receiving process that expects to receive a message smaller than 40 bytes should have no need to issue an RTR message since it useless for it to advertise DMA specific information as the eager communication takes place exclusively through the use of control messages. However, in the write-based eager protocol another complication arises from the fact that receiver initiation is also supported. As it is already mentioned in this chapter, the MPI standard allows the matching of receive and send requests of different sizes. More specifically, a receive request that regards a specific size may correctly match with a send request that regards equal or smaller size in bytes which means that a long receive (i.e., a receive request that expects a transfer of size bigger than 40 bytes) can match with an eager send. This fact renders the sending process issuing an eager send unable to know whether an incoming matching RTR message matches that eager send request or should match a future send request. Such a scenario is depicted in Figure 3.6 where the sender performs an eager sendat t1 while the receiver posting an eager receive at *t2* doesn't issue a control message. This implication made us to initially allow eager receives to also issue RTR messages, even if they don't need to advertise DMA information, in order to prevent mismatches.

Consequently, unlike the read-based protocol, the described version of the write-based protocol should not clear eager send requests before they get matched by a control message from the receiver while at the same time eager receives should send RTR messages if they get posted earlier than the matching send. In addition, eager receives should send acknowledgment (**Ack**) messages when they get posted after the eager message of the sender has already been received. This is essential in order to ensure a match on the sender's side and avoid future mismatches with other RTR messages. Figures 3.7, 3.8, 3.9 show some simple cases of eager communication.

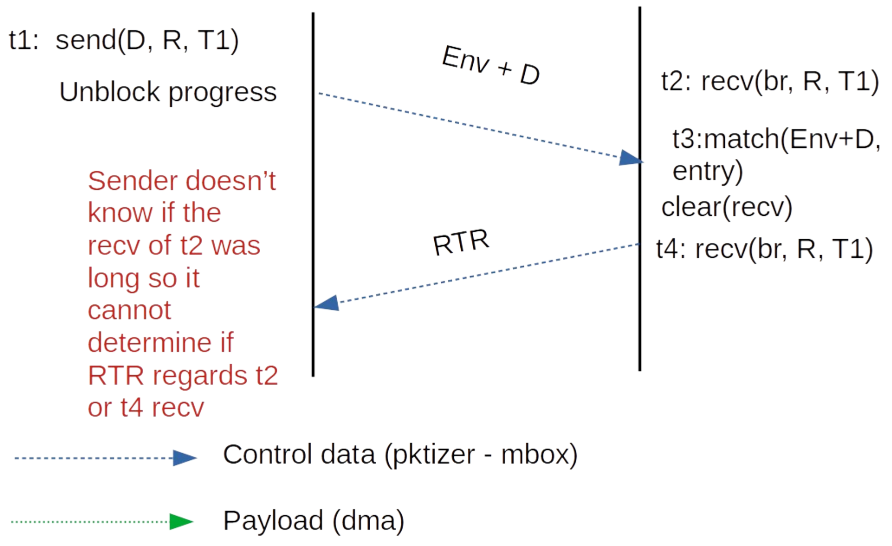


Figure 3.6 Problematic scenario in the write-based MPI's eager protocol

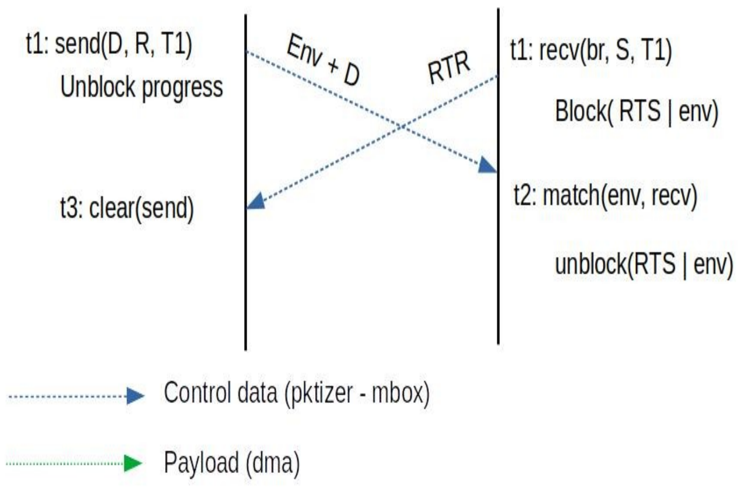


Figure 3.7: Eager protocol, Matching Eager Send and Receive requests posted concurrently

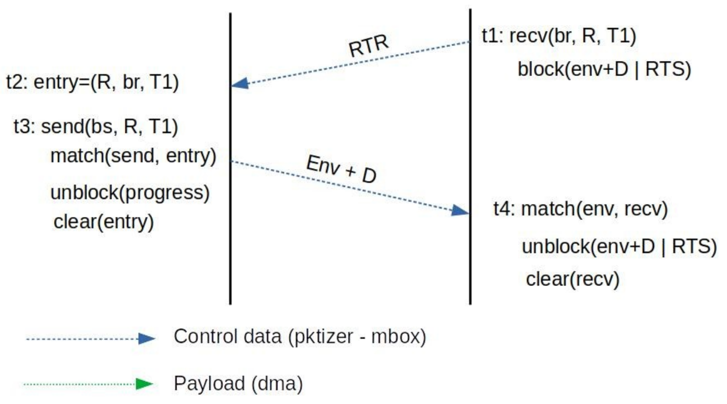


Figure 3.8 Eager Protocol, Eager receive request posted before matching eager send request

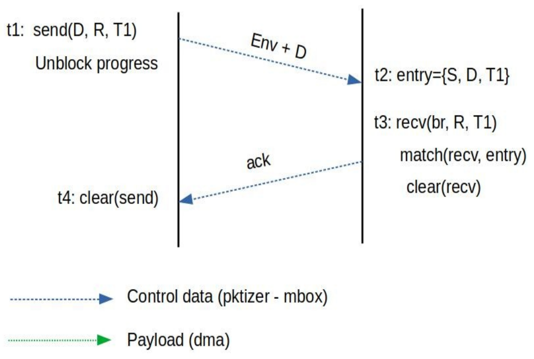


Figure 3.9: Eager Protocol, Eager send posted before matching receive request

Consequently, the eager protocol of the first version of the write-based MPI variant of the Exanet MPI contains the following new control messages:

- **Env+D** (Envelope and Data): The Env+D control message constitutes the message containing all the attributes used to match the send request, packed together with the data an eager send transfers (<40 bytes). In addition, it also contains the size of the data being transferred in bytes which is necessary information for the receiving process. It is used to initiate the communication between the sender and the receiver in case of an eager send.

- **Ack** (acknowledgment) The Ack control message is issued by the receiver of an eager send in the case the respective receive gets posted after the Env+D messaged is already received or the receiver initiation is suspended due to the existence of MPI_ANY_SOURCE. It contains the envelope information necessary in order to get matched with the correct eager send request on the sender's side. It gets issued only if a RTR message regarding the same receive request is **not** already issued.

As it is evident, the aforementioned facts render this version of the write-based eager protocol sub-optimal and under-performing in comparison to the read-based eager protocol. This is not only due to the transmission of extra control messages (i.e., RTR and Ack) but also due to the actions the sending and receiving of those messages infers inside the implementation of the protocol (e.g., locking, searching etc). More information regarding that aspect is available in the next chapters.

3.2 Basic description of the preliminary write-based MPI protocol's implementation

In this chapter, we will delve into some basic details regarding the implementation of the first variant of the write-based protocol. Basic components like the organization of the implementation in 2 threads and their interaction, the use of the mailbox ,packetizer and DMA API inside the MPI primitives will get covered. More technical information regarding implementation of the protocol is provided in next chapters.
Our MPI Library implements most of the point-to-point and collective MPI primitives in C language while primitives which regard one-sided and MPI-IO communications are delegated to a slightly modified MPICH library (going over the Ethernet network). Thus, the Exanet-MPI is a partial MPI implementation on top of MPICH.
It's worth noting that some of the implementation details mentioned in this chapter is identical to the other write-based variants presented in the next chapters.

3.2.1 Initiation of the MPI library

The MPI Standard requires any application that may make use of MPI to call one of the MPI_Init() and MPI_Init_thread() routines in advance. This is essential in order to allow the respective MPI implementation to initiate its internal state. Similarly, in our implementation certain actions must take place and specific information must be obtained with he use of one of those calls before the rest of the implemented MPI primitives can function. More precisely, due to the fact that all the control messages used in the MPI processes' synchronization get issued through the packetizer and are getting received using virtual mailboxes as described in **2.2** while long transfers utilize the RDMA engine, Exanet MPI makes use of the user level communication API of those hardware blocks. This renders the initialization of each hardware block necessary which in turn requires the invocation of the functions described in sections **2.4**. Specifically, dma_alloc_chan() should get invoked in order to allocate an RDMA channel for the MPI process which permit data transfers to virtual addresses without involving the kernel. It is reminded that this function returns a DMA handle which is required in all other routines of the DMA routines. Subsequently, the MPI process obtains the value of its DMA offset by calling dma_get_exanet_offset(dma_channel) and stores that information. Moreover, with calls to the routines MBOX_ATTACH(), MBOX_GET_PROTECTION_ID() and getBoardID(), the process attaches a virtual interface of mailbox and packetizer as well as extracts protection and node IDs respectively as described in **2.4**. Obtaining the information mentioned is crucial in order to render all the processes participating in the run able to reach each other. Since the necessary IDs of each process is initially unknown to the rest of them, the only way to advertise them is through the Ethernet network. Thus, MPICH is used and more precisely, the primitive PMPI_Allgather in order to achieve communication at this stage. After the following three calls, the MPI process possesses all the necessary IDs and DMA offsets needed to communicate with all other ranks of the MPI_COMM_WORLD communicator.

```
PMPI_Allgather(&my_node_id,  1, MPI_UINT64_T, node_ids, 1, MPI_UINT64_T, MPI_COMM_WORLD);

PMPI_Allgather( & protection_id,   1, MPI_UINT64_T, prot_ids, 1, MPI_UINT64_T, MPI_COMM_WORLD);

PMPI_Allgather( & offset,  1,  MPI_UINT64_T,  offsets,  1,  MPI_UINT64_T,  MPI_COMM_WORLD);
```

During the initialization of the MPI Library, the internal data structures needed are initialized. These data structures regard the following objects:

Posted MPI Requests: Objects that regard transfers emerging from the primitives the process has posted. For instance, in the case of an MPI_Send() that is getting posted before a matching RTR has yet arrived, the implementation will create a new object which will represent that send request and insert it into the data structure regarding its posted requests. In that structure, the send request will get matched by an incoming RTR or CTS message from a matching receiver.

Received MPI Requests: Objects that regard transfers emerging from the primitive of another process. These objects get created when a control message arrives from another process and regards a matching request our process hasn't already posted. For example, when a process receives an RTR and a matching send request cannot get found in its Posted MPI Requests, a new object gets created with the RTR's envelope and DMA information which will later should get matched by a future posted send.

Communicators Registry: A data structure that logs information regarding all created communicators besides MPI_COMM_WORLD. Each stored communicator is characterized by its Communicator ID, a 16bit value generated by MPICH. In that registry, information is gathered regarding the protection ID, board ID, mailbox addresses and DMA offsets for each rank of each communicator. We remind that this information is essential to enable packetizer-mailbox communication as well as DMA transfers between MPI processes as described in Chapter 2.

Finally, the initialization of our MPI implementation ends with the creation of the Progress Engine Thread which is necessary for the polling of the process' virtual mailbox and receiving control messages from other processes. More information regarding the utilities of the Progress engine is provided in the chapters.

3.2.2 Structure of Request Object in our Implementation

In order to represent an MPI transfer our implementation makes use of an object named Request which contains all the necessary information regarding that request. A Request object is the type of data stored in the Posted MPI Requests and Received MPI Requests data structures. In general, the most important fields of a request object contain (but are not limited to) the following:

- **int local_notif_address**[4] This arrays' position in memory is the address where the DMA engine writes the notification when the DMA transfer regarding that request object is complete. Notification data have a size of 16 bytes which is the reason their address' type is an array of four 32bit integers. Apparently this field is necessary in receive requests. The initial value of each integer is -1.
- **int type** Indicates the kind of request (eg. send, isend, recv etc.)
- **void * buffer** stores the buffer of the request. Useful for both send and receive requests when the progress engine needs to access the buffer either as source buffer for performing DMA writes or for copying eager messages' data to it.
- **int matched** Indicates the matching state of a request. In general, the value of 0 means that the request and currently unmatched by a control message and is available for matching. Any other value indicates that the request is already matched.
- **int tag** The communication tag of the request. Used for matching. It can take any positive value and MPI_ANY_TAG in receive requests.
- **inttarget_rank** The rank of the process our request has as target (either source for receive requests, or destination for send requests). Used for matching. It can take any positive value or MPI_ANY_SOURCE in receive requests.
- **uint16_t comm_id** 16bit unsigned integer representing the unique ID of a communicator. Used for

matching.

- **MPI_Datatype datatype** The datatype of the transfer
- **int size** The size of the transfer regarding the request in bytes. Useful when the progress engine thread performs DMA write (e.g., in MPI_Isend) or when a receiving process copies data of an eager message to its buffer or completes its MPI_Status structure.
- **char eager_data**[**40**] The data contained in the payload of a received eager message. When a receiving process receives an eager message before a matching receive is already posted, the Progress Engine stores a request object in the Received MPI Requests data structure and copies the eager data to that field. Later, when a matching receive gets posted, it will copy that data to its receive buffer.
- **int safe_to_get_cleared**; Special flag indicating whether a posted non blocking eager send request can get cleared or not. In the first variant of the write-based protocol, a non blocking eager send gets completed immediately without waiting for a matching receive to get posted. However, we must wait for either a matching RTR or Ack message to match it before the progress engine clears it. In addition, MPI_Wait (or similar primitive) will try to clear it too but it shouldn't do it before it has been matched by a control message. Similarly, the progress engine should not clear the request before the user has called MPI_Wait on it. This flag helps the progress engine and the MPI_Wait determine whether both of the conditions have been met for the request to get cleared (i.e., match with RTR or Ack and MPI_Wait call on it). Its initial value is 0 and each time one of the two necessary clearing conditions get met its value increments. When its value is 1, then either an MPI_Wait or the progress engine thread can clear it depending on which was the last which encountered the request.
- **MPI_Req * req_address** This field is used in non blocking requests and stores the address of the MPI_Request the user passed as argument when the non blocking function that created the request (e.g., MPI_Isend) got invoked. It is necessary for functions like MPI_Wait in order to match a provided MPI_Request * argument to an actual request in the Posted Request data structure.

3.2.3 General Control logic of Basic Point-to-Point primitives

As it is stated, our implementation requires the existence of at least two threads even for the execution of single threaded applications: The progress engine and the MPI User thread, which is actually the thread of the running application which calls our MPI primitives. Since both the progress engine and the user thread access the same data structures and the user thread needs to get notified on occasion about changes the progress engine makes, some synchronization mechanisms should be utilized. Additionally, in several cases the user thread needs to get notified about DMA writes another process performs which is achieved with the help of the notification address as already mentioned. In this chapter, we describe the control logic of the basic send and receive primitives in order to underline the presence of synchronization points inside our implementation.

Send Requests: As soon as the user posts a sending request (e.g., MPI_Send, MPI_Isend etc) the process should acquire a lock protecting both the posted requests and received requests data structure. This is compulsory in order to probe the received requests data structure for any matching requests emerged from incoming RTR messages. The mutual locking of both data structures is required in order to ensure the non existence of deadlocks since the progress engine also checks the same structures after receiving a control message from a receiving process and a race windows is possible when both the progress engine and user thread check the Posted MPI Requests and Received MPI Requests at the same time.
a)If no matching receive request exists, the process creates a new request object and inserts it in the Posted MPI Requests data structure. After the insertion of the new request in to the Posted MPI Requests, the process may release the control logic lock and subsequently issue an RTR or Env+D message to the transfer's destination rank according to the size of the message. It is worth noting that this applies to single threaded MPI applications or multi-threaded application which request the **MPI_THREAD_FUNNELED** or **MPI_THREAD_SERIALIZED** thread modes. If **MPI_THREAD_MULTIPLE** is selected instead, the release of the lock should happen only after the RTR or Env+D is issued. If the message was an eager one

then the send function returns (except for the case of MPI_Ssend) without cleaning the request since it still must get matched later by an Ack or RTR. In the case of a long message, whether the function returns or not depends on the kind of the MPI primitive. Specifically, if the send function is non blocking (eg. MPI_Isend), the process returns immediately. Otherwise, the process blocks waiting for either a matching CTS or RTR control message. Those messages do not get received by the function of the user thread itself but by the progress engine thread which subsequently notifies the blocking function. After the receiving of the expected control messages, the blocking send function unblocks its progress and performs the DMA write. Afterwards, the function clears the request object and returns. In the case of non blocking communication and specifically the MPI_Isend function, the DMA write gets performed by the progress engine thread.
b)If a matching receive request is found, the sending process releases the lock, issues the Env message to the receiver and performs the DMA transfer. Note that if the thread modei is **MPI _THREAD_MULTIPLE,** the lock should not be released before the transmission of the envelope message. Finally, the function clears the request object matched and returns.

It is worth noting that the sender may write any value other than -1 to the content's of the DMA notification's remote address in order to inform the receiving process that the transfer is completed.

Receive requests: When a receive request gets posted, the process acquires the control logic lock in order to check whether a matching send requests exists inside the Received Requests data structure.
a) If a matching send request doesn't exist then the user thread checks if the rank denoted as source of the receive equals MPI_ANY_SOURCE or if any active request that uses MPI_ANY_SOURCE in the specific communicator is currently active. If none of the above applies, it inserts a new request object in the Posted Request data structure and initiates the communication by issuing an RTR control message. Regarding the release of the control logic lock, the process unlocks immediately after inserting the new request object unless the thread mode is set to **MPI_THREAD_MULTIPLE** in which case the unlock happens after the RTR gets issued in order to ensure the preservation of the MPI Standard's guarantee that requests should match in the same order they are posted. After initiating the communication, if the receive function is blocking, the process blocks until the receiving of the Env or Env+D message and afterwards blocks for a change in the contents of the DMA notification address. It's worth mentioning that Env are Env+D messages are again received by the progress engine threads and not the user thread itself. In the case of an eager message (i.e.. the receiving of an Env+D control message) where the DMA is not involved, the progress engine threads copies the payload included in the control message to the receive request's receive buffer and also makes a minimal change to the local notification address of the receive request in order to simulate a DMA write and make it stop waiting for an actual DMA write to that address. In the case of the existence of MPI_ANY_SOURCE in the communicator the request regards, the process inserts a new request object in the data structure but does not initiate the communication using an RTR message. Instead it blocks for an incoming RTS/Env+D control message (if it is a blocking request) or returns (if it is a non blocking request). The rest of the process after receiving the expected control message from the sender is the same as if a received request already existed (described in the next bullet point). However, it should be noted that in the case of non blocking requests all the next steps are performed by the progress engine and not by the user thread.
b) If a matching send requests does exist, if that request is an eager send, then the user thread copies the payload of the Env+D message to the request's receive buffer and returns after clearing the matched request object. In the case of a long matching send request, the user thread issues a CTS message to the sender advertising its DMA information. Once again, the same assumption applies regarding the thread mode. The receiving function may only unlock after sending the CTS message regardless of threading mode. This applies due to the fact that the progress engine thread may send CTS messages in special occasions described in 3.2.4 and we need to be sure that the FIFO order is kept in all messages (i.e., a request that gets posted or gets matched first, issues a control message before a request that got posted or matched second).
It should get mentioned that a blocking receive function always fills its MPI_Status object (provided

MPI_STATUS_IGNORE is not used) and clears its state before returning. For non blocking receiving functions, the filling of the MPI_Status structure and the removal of the request from the data structure at the end of the communication are tasks of MPI_Wait and similar primitives.

Wait and Test Point-to-Point primitives: Primitives like MPI_Wait, MPI_Test, MPI_Request_get_status and all the relative functions (e.g., MPI_Waitany, MPI_Testsome etc) have similar behavior. All of these functions take as argument a pointer to an MPI_Request object and a pointer to an MPI_Status object. It should be mentioned that the MPI Standard's type **MPI_Request** should not be confused with the **request** structure of our implementation stated earlier. MPI_Request is defined as a 32bit integer in the MPICH implementation. It is reminded that all non blocking functions require the caller to provide a valid pointer to an MPI_Request object as argument. For instance, the non blocking receive function's signature is the following:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request * request)
```

Each time a process calls a non blocking send or receive primitive, the new request that gets added to the Posted MPI Requests data structure contains a field of type MPI_Request * called **req_address** which stores the address of the MPI_Request the user passed as argument to the non blocking call as described in 3.2.2. When a process invokes a wait or test primitive, our implementation uses the position in memory of the MPI_Request given as argument and searches the Posted MPI Request data structure in order to find an object whose **req_address** field has the same address as the provided MPI_Request. In that way, we manage to match the MPI_Request * argument of MPI_Wait to an actual request object in our data structures. It should be noted that this process requires locking and unlocking of the control logic lock which can be the cause of overhead. In General, send requests are considered completed if the value of the **MPI_Request** related to them is equal to 1 and uncompleted otherwise. Receive requests are considered partially completed when the value of its **MPI_Request** is 1 (which is set by the progress Engine when a control message like RTS, Env or Env+D matches the request) and fully completed when the first integer of its local notification data is other than -1 (as described in **3.2.2**). MPI_Wait blocks the progress until the request is fully completed and clears it afterwards. A case of high interest is that of a non blocking eager send. Eager sends are completed immediately after sending the Env+D control message without requiring the existence of a posted matching receive request on the receiver's side. This means that an MPI_Wait call shouldn't ever block on a non blocking eager send and instead return immediately. However, since in this version of the write-based variant eager sends need to get matched by an RTR or Ack messages for correctness reasons, it renders the MPI_Wait unable to clear the request in some occasions. As described in **3.2.2** the **safe_to_get_cleared** integer field indicates whether an RTR or Ack has matched the request. If the request is not matched at the time of MPI_Wait then the MPI_Wait function increments the integer and returns without clearing. Otherwise, MPI_Wait clears the request. An example of a non blocking eager send is illustrated in Figure 3.10.

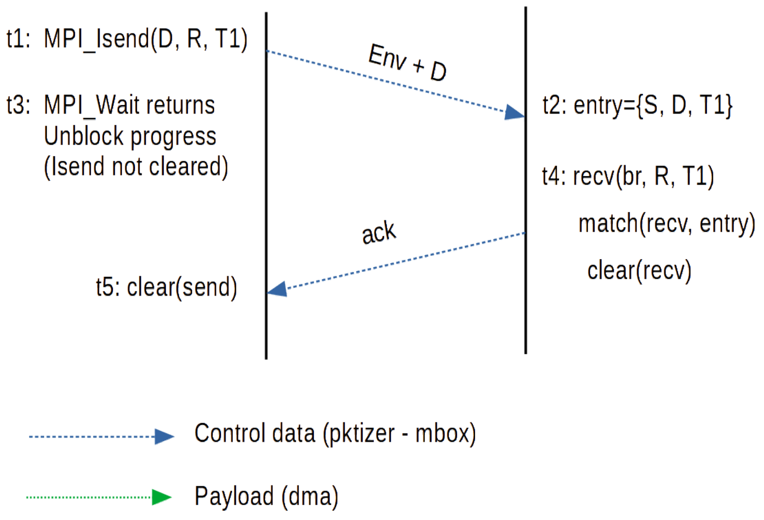


Figure 3.10: Eager protocol: Non blocking send and MPI_Wait

3.2.4 General Control logic of the Progress Engine

The main objective of the Progress Engine thread is the constant polling of the process' virtualized mailbox and the appropriate handling of each received control message. The progress engine thread is responsible for matching incoming control message from other processes with requests that exist in Posted MPI Requests of the process. More precisely, the progress engine executes an infinite while loop invoking *_MBOX_TRY_DEQUEUE_64B* atomic primitive in each iteration. Each message sent through the packetizer uses 4 of its 64 bytes to denote its type (e.g., RTS, RTR, Env etc). The progress engine, after dequeuing a message to a specific buffer, determines its type and proceeds to the necessary action depending on it.
More specifically:

In case of an **RTR message**: The progress engine locks the control logic lock and searches through the appropriate Posted MPI Request data structure for a matching send request. The tag, rank and communicator id attached in the message determine whether the message matches with an unmatched request or not.
If such send request exists, the progress engine unlocks the control logic lock sets its *matched* flag to 1 in order to prevent it getting matched by future messages. If the send request was a blocking eager one , the progress engine just clears it since eager sends complete immediately. If the request regards a blocking long send, then it updates the request object with information that regard the remote buffer and the remote notification address included in the RTR message, disconnects the request object from the Posted Requests data structure and notifies the user thread blocking for a CTS or RTR that such a message has arrived. Consequently, the user thread is ready to perform the DMA transfer. The reason the progress engine disconnects the request from the data structure is because such an action lets the user thread clear the request (i.e., free()) without having to acquire the control lock since the request getting cleared does not belong to any data structure any more. On the other hand, if the request is a non blocking eager send, the progress engine checks the value of the **safe_to_get_cleared** flag to determine whether to clear the request or let MPI_Wait (or similar primitive) handle the clearing. If the request is a non blocking long send, the progress engine performs the DMA transfer since the function from which the request emerged (e.g., MPI_Isend) has already returned. After the DMA transfer, the progress engine marks the request as completed but no. It is worth noting that if the destination and the source of the transfer is the same process, a memcpy is preferred over a DMA transfer. Ather the memcpy operation the progress engine writes to the address of the DMA notification as the DMA engine would

do to singal the receiver that the transfer is over. If no matching request exists then the progress engine inserts a new request to the appropriate Received MPI Requests data structure. The request carries the same matching attributes (tag, rank, comm_id) included in the RTR message as well as the DMA related information. Subsequently, the progress engine thread releases the control logic lock.
In case of a **CTS message** the exact same logic applies with the following exception. It is guaranteed that a CTS message will always match a long send request in the Posted MPI Requests. Anything else would be erroneous since the CTS control message is always a response to an RTS message there will always exist matching long send request in the Posted MPI Request.

Respectively, an **Ack** message will always match an eager send request and the progress engine will perform the same actions it would perform in the case of an RTR message.
In case of an **RTS message**: After locking the control logic lock, the progress engine searches the Posted Requests MPI data structure for a matching receive request.
a)If such a request exists, the request is marked as matched so that it cannot get matched by other messages and gets updated with information necessary for the completion of the MPI_Status object. If the request has emerged form a blocking receive, the user thread gets notified that the expected RTS arrived. In case of a non blocking receive request, the MPI_Request object related to the request is set to 1 to signify partial completion. An exceptional case is that of a non blocking receive request when MPI_ANY_SOURCE is active in at least one other request of the communicator (not necessarily the one matched). In such a scenario, it should be reminded that issuing of RTR is suspended for all receive requests of the process regarding that communicator. As a result, an MPI_Irecv would have returned without issuing any RTR messages. This fact renders the progress engine thread responsible for issuing a CTS message for that receive request after matching the RTS message against it. It should be noted that in that specific case, the control logic lock gets released only after the sending of the CTS message even in single threaded applications as mentioned in subsection 3.2.3.
b)If no matching requests gets found, the progress engine threads posts a new send request to the Received MPI Requests data structure and unlocks the control logic lock
In case of an **Env message:** The progress engine will always match an Env control message to a long receive request after acquiring the control logic lock. After the match, the lock is released and ilf the request is blocking, it notifies the user thread waiting for it about its arrival. In the case of a non blocking request, it sets the MPI_Request object related to that request to 1, indicating partial completion. In both cases, the request learns all the information required for the completion of its MPI_Status.
Similarly, an **Env+d message** will cause the progress engine to lock the control logic lock and search for a matching receiving request in the Posted MPI Requests data structure.

- In case such a request is found, the control logic lock gets released, and the payload data of the Env+D message gets copied to the matched request's receive buffer. In addition, the request gets updated with the necessary information in order to get rendered able to complete its MPI_Status object.
- If no such request exists, a new send request gets inserted into the Received MPI Requests data structure to be later matched by a future receive primitive of the user thread. This new request consists of all the matching attributes contained in the Env+D message as well as the payload data stored in the **eager_data** field of the request.

3.2.5 Implementation of Synchronization and Locking Mechanisms

In the previous paragraphs it became evident that our implementation is highly dependent on locking mechanisms as well as mechanisms allowing the progress engine thread notify the user thread for certain changes that take place. During our research we came across various ways of achieving synchronization each one of which resulted to significant performance changes. Specifically, our first attempt to achieve synchronization between the progress engine thread and the user thread(s) included the use of the broadly used

POSIX library and more precisely, the **pthread__mutex** and **semaphore** objects. The control logic lock was initially implemented as a **pthread__mutex** while each time a user thread blocked its progress waiting for a specific change, it was using **sem__wait()** function on a binary semaphore which constituted a field of our **request** structure. For instance, when a MPI_Send() primitive needed to wait for a CTS message to match the send request it posted, it would call **sem__wait(&new__request→ received__CTS);** where **new__request** is the request inserted into the Posted MPI Requests and **received__CTS** is the name of the binary semaphore included in the request structure. Similarly, the progress engine would increment that semaphore after matching a CTS message against that specific request in order to notify the user thread to unblock its progress. We preferred the use of binary semaphores over that of another mutex due to certain limitations of the pthread_mutex's API like the fact that only the thread that locked a mutex may also unlock it. In contrast, semaphores can get incremented by any thread regardless of which thread created them. While the use of the POSIX library for that reason proved reliable in terms of correctness and functionality, in our eyes it seemed essential to further experiment with more ways of synchronizing since both mutexes and semaphores frequently enter kernel mode which could prove very costing in terms of performance. Specifically, pthread_mutexes of POSIX are implemented based on the Futex POSIX library which makes extensive use of system calls and requires the intervention of the kernel which can put threads to sleep or wake them. Using a similar method, semaphores also involve the kernel space in their attempt to suspend a thread's execution.
Our first attempt to minimize the expensive thread suspensions was to replace all POSIX semaphores with simple integers. The blocking user thread would busy wait on a volatile integer's value until the progress engine changes its value. By trying this alternative, there was a significant improvement on our implementation's latency as seen Figure 3.11. In eager sizes (0-40 bytes) we observe a 300% performance gain while in bigger sizes there is 100% improvement.

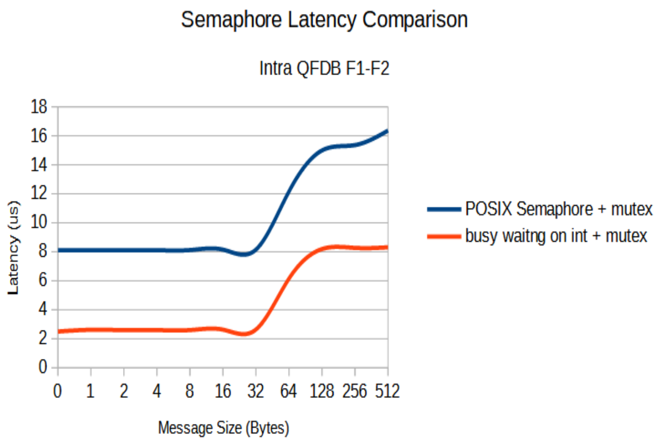


Figure 3.11: Comparison of POSIX Semaphore and busy waiting latency

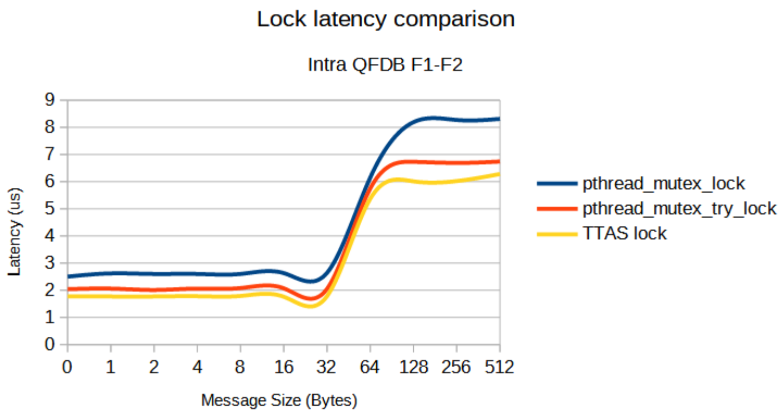


Figure 3.12: Comparison of different locking mechanisms' latency

Our next step was the further reduction of our implementation's latency by trying different mechanisms for locking. In order to avoid the sleeping of the user thread as much as possible we tried using successive calls to **pthread_mutex_trylock** instead of **pthread_mutex_lock** in order to acquire the control logic lock. In both cases, releasing the lock was achieved using **pthread_mutex_unlock.** Even this change offered an observable performance gain. However, our experimenting continued with the benchmarking of our implementation using our own TTAS spinlock that depends on GNU Compilers Collection (GCC) builtin Atomic functions like **___sync_lock_test_and_set** and **___sync_fetch_and_sub.** The TTAS lock proved to be the most efficient mechanism as seen in Figure 3.12.

The results depicted in the Figures regard OSU Latency Microbenchmarks run in one QFDB with the 2 processes residing on F1 and F2 FPGAs. Message sizes from 0 to 512 bytes are included in the results since in bigger sizes the difference emerging from changes in synchronization and locking gradually becomes less apparent as the message size grows.

3.2.6 Data Structures Design

We use in total 4 data structures for storing MPI requests:

- Posted MPI send requests
- Posted MPI receive requests
- Received MPI send requests
- Received MPI receive requests

However, for reasons of simplicity, in this thesis we usually mention two more generic data structures; the Posted MPI Requests and Received MPI requests.
Since the MPI Standard guarantees the FIFO property described in Section 2.1, one sees that the best choice for the implementation of a data structure that regards MPI requests is a FIFO queue. Initially, we used four FIFO queues implemented as doubly linked list (for supporting disconnecting elements inside them when they get matched) for representing Posted MPI send Requests, Posted MPI receive requests, Received MPI send Requests and Received MPI receive requests. However, this choice offers poor performance since it can lead to congestion of the data structure when many concurrent requests exist. As a consequence, we switched to using a hashtable of FIFO queues for each one of our data structures. Each hashtable entry uses as hash the combination of the rank, tag and communicator its requests regard. Unfortunately, due to the existence of MPI_ANY_SOURCE and MPI_ANY_TAG we cannot use a hashtable for Posted MPI receive requests and Received MPI send requests since these data structures get accessed by MPI_Recv/MPI_Irecv primitives, which must be able to find all requests in a FIFO order. This renders the use of a hashtable impossible for these structures since receive primitives wouldn't be able which bucket of the hashtable when MPI_ANY_SOURCE/MPI_ANY_TAG gets used. As a result for these two data structures we use a simple FIFO queue.

3.3 Designing an improved sender and receiver initiated write-based long protocol: Elimination of the Env Control message

As observed in the previous chapters, the frequent locking and the constant need of synchronization between the user thread and the progress engine can constitute one of the biggest causes of overhead in our implementation. Consequently, eliminating some control messages was one of our main goals as far as optimization of the write-based long protocol was concerned. In order to achieve this, we investigated whether any of the control messages that are not existent in the read-based prototype can also be absent in the write-based prototype after improving or exploiting already existing components of the protocol. In the scope of this work, we investigated whether the Env Control message can be omitted. As described previously, the objective of the Envelope message is to convey information required to complete the MPI_Status structure of a receiver in case of a *fast receive* where an RTS control message does not get issued. Specifically, in any MPI implementation the MPI_Status structure of a receive request should contain:

- *The size of the completed transfer in bytes*
- *The communication tag*
- *The source rank that performed the transfer*

When a fast receive takes place (and thus an Env message is needed) only the first two of the aforementioned elements can be unknown to the receiver. This applies due to the fact that when a request using MPI_ANY_SOURCE is active, no receive request can initiate the communication. This implies that all receive requests that designate MPI_ANY_SOURCE as the transfer's source will always be initiated by the sender and,thus, will learn the sender's rank from the received RTS message. Consequently, a mean of transferring only two integers (size of transfer and communication tag) from the sender is needed.

The DMA notification used to notify the receiver of the completion of a DMA write, as described in **2.3**, has a size of 16 bytes which is more than enough to hold that information. Thus, in the second variant of write-based MPI, there is no need for an Env message during a fast receive since all the information such a message would carry is instead conveyed by the sender through the DMA notification of the RDMA engine. It should get mentioned that every new receive request that gets created, has the first integer of its notification data (first 4 bytes) which represent the size of the transfer set to -1. As a result, a receiving process waiting for a DMA notification, polls the first integer of the notification until its value changes and learns the information an Envelope message would carry by reading its notification data and the transmission of such a control message by the sender is omitted. However, such a reformation of the write-based protocol also infers some consequences that is explained in the next paragraphs.

In Figure 3.13 a scenario of a fast receive is illustrated. The receiving process arrives first to a matching send-receive pair at *t1* and, since there is no MPI_ANY_SOURCE request active, initiates the communication by issuing an RTR control message after inserting the receive request into the appropriate Posted Request data structure. Afterwards, it blocks waiting for the DMA notification since the Env control message has been omitted. The RTR control message arrives at the sending process at *t2* and gets dequeued by the sender's progress engine thread. Since no matching send request exists in the Posted Request of the sender, a new request object gets created and added into the Received Request data structure containing the matching attributes and DMA information included in the RTR message. At *t3,* the sending process invokes a sending primitive and searches its Received Request for a matching received receive request. A match is successful with the request received at *t2* so the sender performs the DMA write and instead of writing arbitrary data to the remote notification data address, it instead writes the size of the transfer and the communication tag. Subsequently, it returns cleaning its state. The receiving process unblocks after receiving the DMA notification and has therefore access to all information conveyed through it in order to correctly fill the MPI_Status structure. Finally, it also returns cleaning its state.

Regarding the case of a sender-initiated communication, the second version of the write-based variant is not differentiated from the first version in any aspect. As shown in Figure 3.14, the sender initiates the communication since the send request posted earlier than the receive request at *t1.* While the sender is blocking its progress waiting for a CTS or RTR control message, the receiver receives the RTS message at *t2* and stores the respective request in its Received Requests FIFO queue. At *t3,* the receiving process posts its receive request which matched with the previously received RTS. After, writing the appropriate in the MPI_Status object, it issues a CTS message advertising its receive buffer and DMA notification address. The sender normally executes the DMA transfer after receiving the matching CTS message.

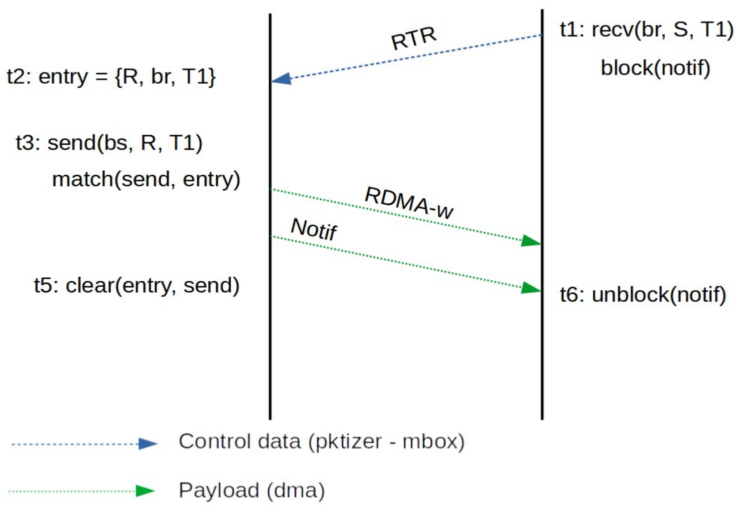


Figure 3.13: Receiver initiation with omitted Env control messsage

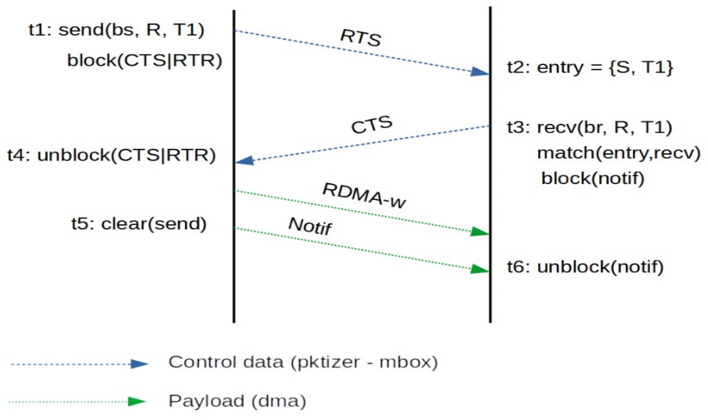


Figure 3.14: Sender initiation with omitted Env control message

The realization of a common fast receive as well as a common sender initiated communication in the second variant of the write-based protocol is simple and reliable as described in the previous paragraphs. Unfortunately, in more complex cases, there are some issues to be resolved. An example of high interest is the case of simultaneous sender and receiver initiations from both the sender and receiver processes. In the previous example of Figure 3.13 where the communication is initiated by the receiver, we observe that the receiving process never blocks expecting an Envelope or RTS message. However, there is a serious possibility that the sending process has also issued an RTS control message which should match with the correct receive request, avoiding mismatches. One could argue that blocking progress waiting only for the DMA notification is sufficient for the receiver since an RTS control message always gets issued before the DMA transfer takes place which ensures that when a DMA transfer is completed, the RTS control message would have already matched the receive request. While that argument sounds valid, it is not safe to ignore waiting for an RTS message due to the fact that the receiver process uses one thread for polling its mailbox (i.e., Progress Engine Thread) and a different thread for polling the DMA notification address (i.e., the main User Thread). This can lead to unpredictable outcomes regarding the order in which the process determines the arrival time of each of the two synchronization elements. An explanatory example is illustrated in Figure 3.15.

In Figure 3.15 the receiver progress gets split visually to two threads in order to render each thread's actions more apparent. Both the sender and the receiver process arrive concurrently at *t1* and attempt to initiate the communication by issuing their respective initiatory control messages (i.e, RTS and RTR). Afterwards, the sender blocks waiting either for a CTS or an RTR message while the receiver blocks waiting for the DMA notification, omitting the waiting for an RTS or Env message. The polling of the mailbox takes place always in the Progress Engine thread of every process while the user thread waits for a change in the notification address' contents made by the DMA engine. At *t2*, the sending process matches the RTR message, unblocks the user thread's progress and performs the DMA transfer. At the receiving process' side, due to the CPU the progress engine runs on being slow or just because of a context switch, the user thread gets aware of the DMA notification **before** the progress engine dequeues and matches the RTS message. As a consequence, the user thread clears the request at *t5* assuming that any RTS that may have got issued has already matched it. At *t7,* the progress engine tries to match the dequeued RTS control message but due to the receive request being already cleared, it will either treat the RTS as unmatched (and insert a send request into the Received Requests data structure) or match it against a wrong matching receive request. In both cases the execution of the program becomes erroneous. On the other hand, making the user thread always wait for a control message from the sender before it waits for the DMA notification is also wrong since there is no guarantee that the sender will issue an initiatory message for that specific request.

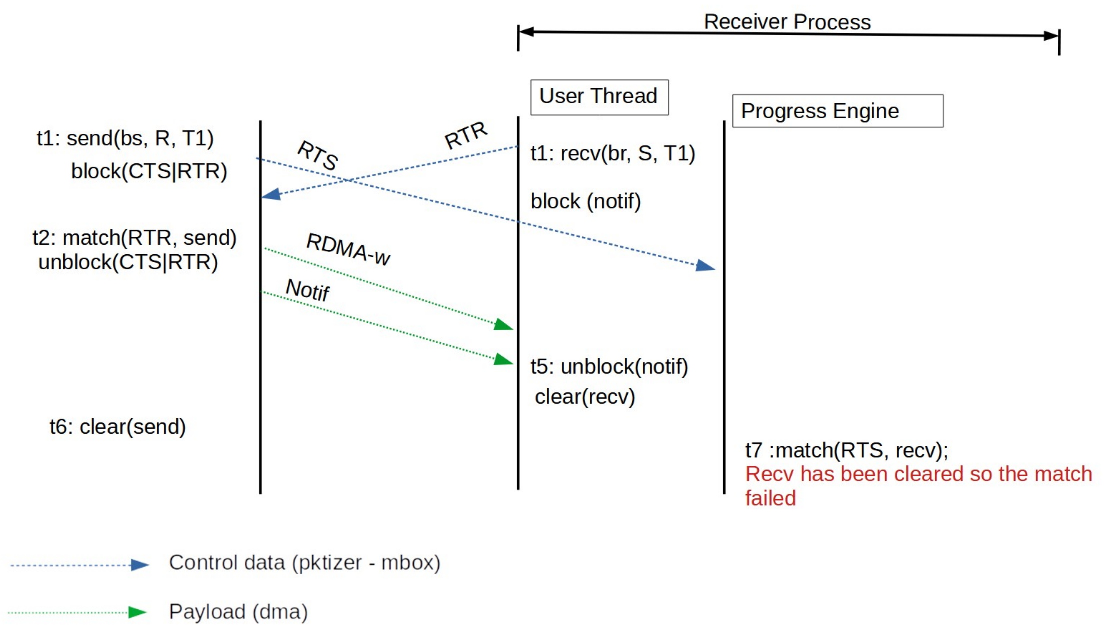


Figure 3.15: Problematic scenario of concurrent sender and receiver initiation with omitted Env control message

In order to resolve the issue outlined in the previous paragraph, we adopted an elegant solution which makes use of some of the rest of the bytes in the DMA notification data. Specifically, we organize the 16 byte space of the notification data as an array of four 32bit integers. As already mentioned, the first two integers take the values of the size of the data transfer and the communication tag respectively. We use the third

integer in order to inform the receiving process which expects a DMA notification whether it should also expect an RTS message or not. More precisely, the value of 1 in that integer denotes that an RTS has also been issued for the receive request and it should wait for it if not already received. Otherwise, the value of the integer is 0. It's worth noting that the problematic scenario depicted in Figure 3.15 happens extremely rarely. Most of the times, the progress engine dequeues and matches the RTS message before the user thread gets the DMA notification and clears the request. This implies that the RTS will most likely be already matched when the user thread reads the notification data and there will be no observable performance loss attributed to waiting for an RTS. The fourth integer inside notification data still remains unused at this point. Figures 3.16, 3.17, 3.18 illustrate the correct versions of sender and receiver initiated communications in the second variant of our implementation respectively. It should be observed that when the sending issues an RTS for the communication it also sets the third integer of the notification data (*rts_sent* in the figures) to 1.
Another complication that arises in the second variant of the write-based protocol is the fact that since a fast receive request can complete without receiving a single control message from the sender, it is technically viewed as unmatched. Consequently, a future matching RTS normally directed to another receive request may accidentally match it. In order to address such a case, we make all the new created receive requests have the third integer of their notification data array set initially to -1. This value represents that no sender has performed any DMA write for that receive request. Additionally, the matching function called in order to match an RTS message has the following change. It no longer considers a request that only has its *matched* field set to 0 as unmatched but also checks for the third integer of the request's notification data. That integer's value must either be -1 or 1 for the request to be considered available for matching. Recall that the value of 1 in the third integer of notification data means that the DMA transfer has been completed but the user thread should also wait for an RTS message issued by the sender while 0 conveys the meaning that no RTS is headed for that specific receive request. In order for this technique to be functional it should be guaranteed that no issuing of RTS message can take place simultaneously with a DMA transfer. In our implementation this assumption holds true for all Threading Modes except for MPI_THREAD_MULTIPLE. For that threading mode, we don't use the second variant of the write-based long protocol.

The problem that is caused by the existence of multiple threads that can call MPI functions concurrently is described more thoroughly in Figure 3.19. The receiving process issues an RTR message at *t1* and blocks its progress waiting for the DMA notification. At *t2,* a send function is getting called on the sender's side thread 1 which matches with the receive request received, thus, the DMA transfer takes place. After the successful match, the thread releases the control logic lock which lets a second sending thread initiate another send request issuing an RTS message. That RTS message happens to also match the receive request posted at *t1.* The first sending user thread has not yet completed its DMA transfer which implies that the third integer of the notification data of the receive request is still set to -1. Consequently, the RTS message of the second send request mistakenly matches the receive request. One could argue that this error would not occur should the first thread did not release the control logic lock before finishing the DMA transfer. However, this would inflict extra overhead since neither any other user thread nor the progress engine thread can access the internal data structures while the lock is acquired which would render the performance poorer especially in big DMA transfers.

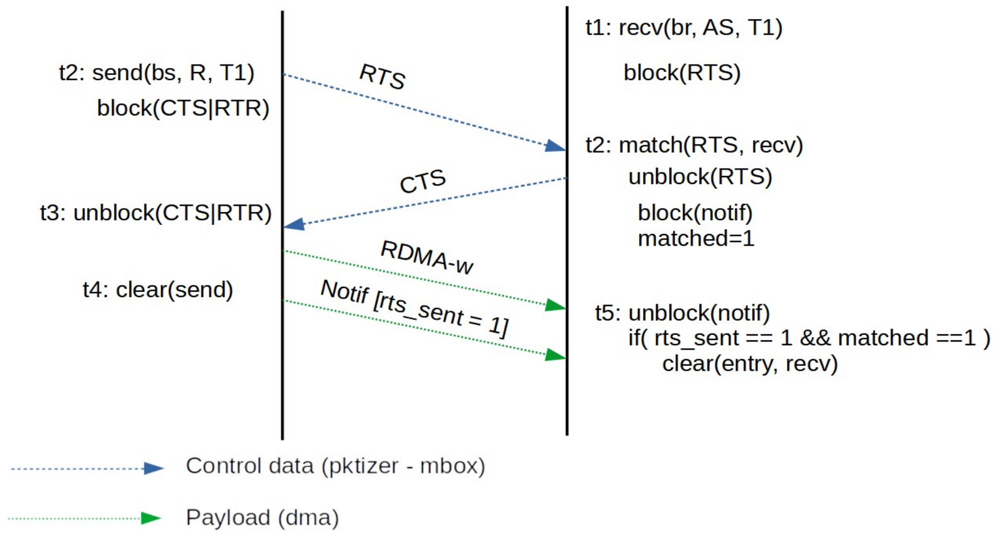


Figure 3.16: Sender initiation with omitted Env control message, correct execution

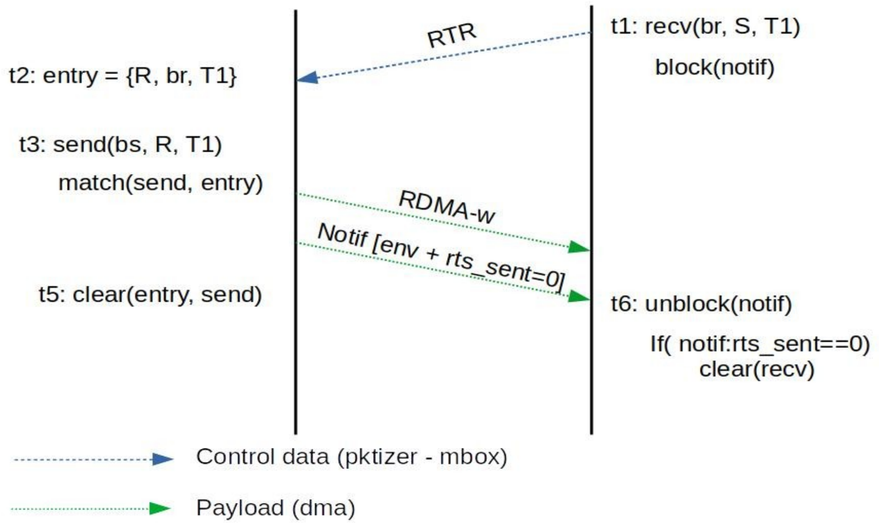


Figure 3.17: Receiver initiation with omitted Env control message, correct execution

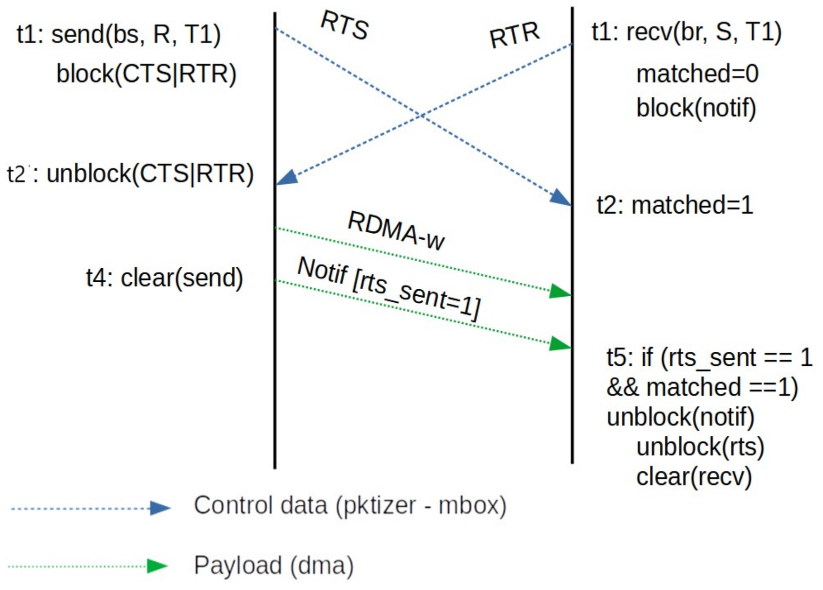


Figure 3.18: Concurrent sender and receiver initiation with omitted Env control message, correct execution

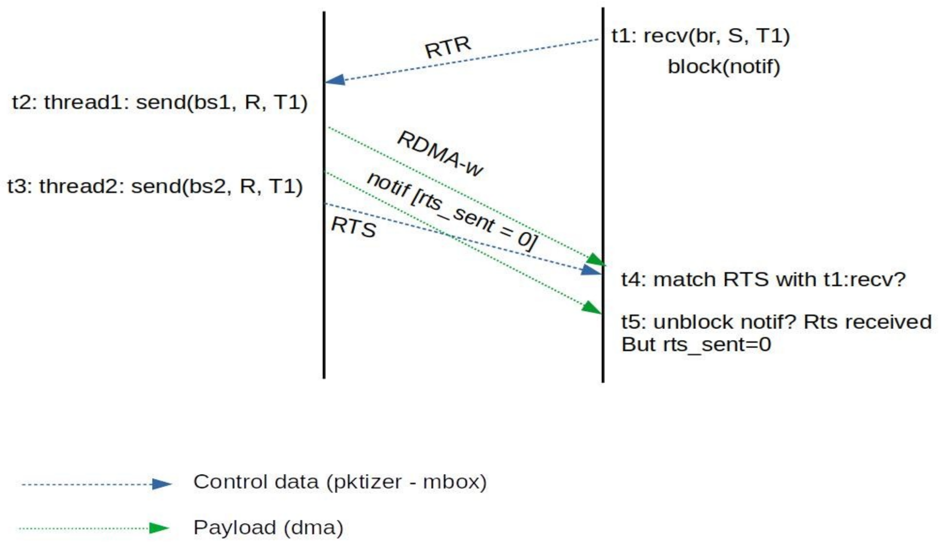

RTR
t1: recv(br, S, T1)
block(notif)
t2: thread1: send(bs1, R, T1)
RDMA-w
t3: thread2: send(bs2, R, T1)
notif [rts_sent = 0]
RTS
t4: match RTS with t1:recv?
t5: unblock notif? Rts received
But rts_sent=0
Control data (pktizer - mbox)
Payload (dma)

Figure 3.19: Problematic scenario with two sending threads and omitted Env control message

One may argue that the problem illustrated in Figure 3.19 can also be present in scenarios where non blocking sends are used in single threaded applications. However, this is not true since non blocking send primitives of our implementation issue an RTS message before returning if no matching receive request already exists in the Received MPI Requests. This ensures that a matching receive request in the receiving will get matched by the correct send request. When there is already a matching receive request in the Received MPI Requests data structure at the time our non blocking send primitive gets posted then no RTS message gets issued bu the the function does not return until the DMA transfer is over. When the DMA transfer completes, the third integer of the request's notification data in the receiver's side has the value 0 which renders it unmatchable by RTS messages. Since the non blocking function will return only after completing the transfer, it is guaranteed that no future send request can issue a matching RTS message that will accidentally match the receive request on the receiver's side. An example is depicted in Figure 3.20. The MPI standard demands that a non blocking sen request like MPI_Isend must not block until a matching receive gets posted but if such a receive already exists, an implementation is allowed to complete the data transfer before returning. In addition, some bugs in the current version of the DMA API, make it safer to wait a transfer to complete before returning.

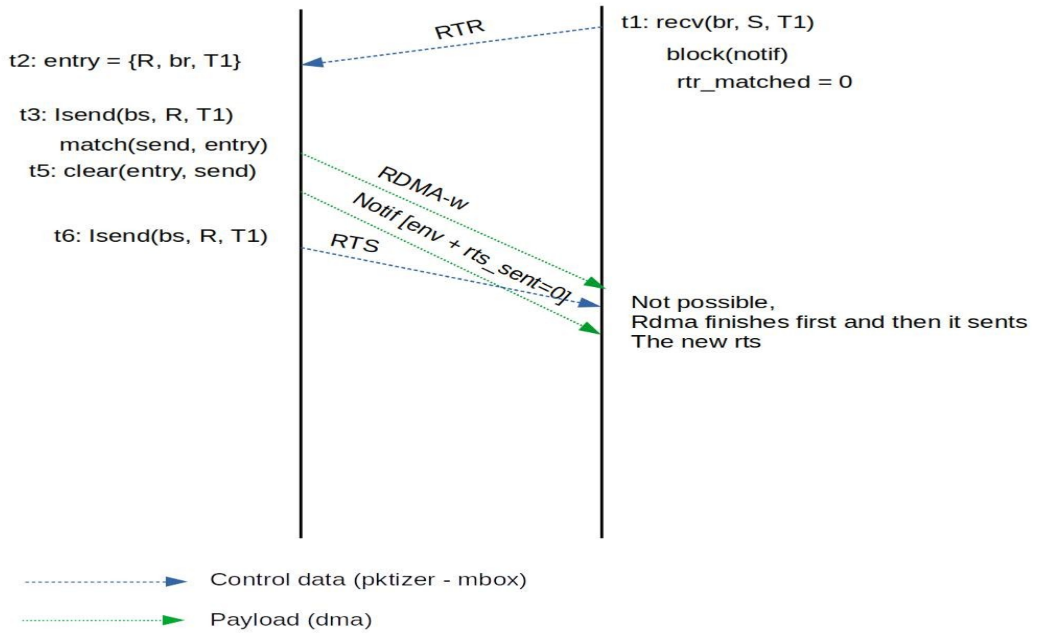


Figure 3.20: Non blocking sends with omitted Env control message

3.4 Basic description of the second write-based MPI variant's implementation

In General the implementation details of all variants of the write-based MPI are the same with those described in 3.2. Only substantial differences will get presented in this section.

3.4.1 Structure of Request Object

The same details stated in 3.2.2 also apply. An important difference is that any new request object created has specific initial values in the array which symbolizes the contents of its local notification address (**int local_notif_address**[**4**]). In specific, the first and the third integer, which now represent the size of transfer and the flag indicating whether an RTS has been issued for that transfer respectively, should be always initially set to -1. A negative initial value in the size of transfer is necessary since a change in that values is what notifies the user thread that the DMA transfer has been completed. Should we chose 0 or any positive value, it would pose the risk of it coincidentally matching the size of transfer which implies that the sender would write the exact same initial value to it after the transfer completed. This would render the user thread polling on the value of that integer unable to determine the end of communication. Regarding the third integer, as it was explained in the previous chapters, its initial value to -1 plays a crucial role in avoiding mismatches with RTS messages headed for other receive requests in cases where the correctly matched send request does not issue an RTS message.

3.4.2 General Control logic of Basic Point-to-Point primitives

Similarly, the same basic control logic of point-to-point primitives described in 3.2.3 apply in this variant. However, certain details regarding some changes emerged from the elimination of the Env message need to get highlighted.

Send Requests: As it is apparent, in this variant send requests no longer issue Env messages in a receiver initiated communication except for the case of the thread mode **MPI_THREAD_MULTIPLE** where this change does not apply as already explained (Figure 3.19). In addition, each process should write the correct value to the third integer of the receiver's notification data. Specifically, when a long send request gets posted before a matching receive exists in the Received MPI Requests it issues an RTS message and, consequently, has to write 1 to the third integer of the notification data ,which indicates whether an RTS has been issued or not, when the DMA transfer takes place. In other cases when no RTS message (and no Env) is needed, that value gets set to 0.

Receive Requests: A receive request should not be required to get matched by a sender's control message when the communication is receiver-initiated. The process reads the third integer (rts_sent) of the notification data in order to determine whether it should block for an RTS or not. If the integer's value is 0, the receive function must disconnect the request from the Posted MPI Requests before cleaning as since no control message will ever match it and the progress engine won't have a chance to disconnect it. Again, the thread mode **MPI_THREAD_MULTIPLE** poses an exception to what was described above since Env messages are not omitted.

Wait and Test Point-to-Point primitives: In the second variant of Exanet MPI, the MPI_Wait and relative functions have an additional task to perform in a certain occasion. In a fast receive (receiver initiated communication), if the sending process does not issue an RTS message, the receiver's progress engine does not match any control message with that specific receive request and thus has no chance to disconnect it from the Posted MPI Requests data structure. In addition, since the function from which the request emerged was non blocking (MPI_Irecv) it has already returned. As a consequence, the only function which can remove the complete receive request from the data structure is the MPI_Wait() or a relative function. This renders necessary for the Wait, Test etc functions to also check the third integer of the notification data (written by the DMA engine) when they handle a completed receive request. Should that integer be 0 (ie. no RTS sent in that communication), the Wait, Test function must remove it from the respective data structure before cleaning it.

3.4.3 General Control logic of the Progress Engine Thread

The same logic described in 3.2.4 also applies in this variant. However, unless the thread mode **MPI_THREAD_MULTIPLE** is used, the progress engine no longer needs to handle the receiving of Env Control messages as they are actually replaced by notification data. In addition, when an Env+D eager message is received, if it matches an eager receive request already posted, the progress engine manually changes the third integer of its notification Data to 1 since no DMA transfer takes place in eager communication. The value of 1 normally conveys the meaning that an RTS has been sent but in this case it regards the Env+D message and not an RTS. By reading the value of 1, the receiving user thread knows that it has no need to disconnect the request from any structure since it is done by the progress thread engine after the match of a control message against it.

3.5 Designing an improved sender and receiver initiated eager based protocol: Elimination of the Ack control message and receiver initiation in eager communication

In the previous chapters, we presented a new write-based protocol that lacks the main drawbacks of the preexisting read-based protocol. More precisely:

1. We allow receiver initiation in scenarios where a receiving process posts its request earlier than the sender (fast receive). As a consequence, receives do not need to wait for an RTS message to initiate the communication. Instead they can advertise their DMA related information asynchronously by issuing RTR messages.

2. The new protocol is write-based which renders the sending process able to determine the end of a DMA transfer without the need for an Acknowledge message.

3. We exploit the DMA notification component which helps the receiver determine the end of a DMA transfer.

However, we see that the enabling of receiver initiation also constitutes the source of new weaknesses in the protocol. Specifically:

1. The use of an extra Env message was initially required when sender initiation did not take place

2. Eager receives (i.e., receives that regard transfers of sizes of up to 40 bytes) still need to transmit RTR messages while they have no reason to advertise DMA information.

3. Eager sends need to create requests that must get matched either by RTR or Ack messages in order to preserve the integrity of the protocol and not cause mismatches while in the read-based protocol they do not need to insert any new requests into any data structure.

Recall that the reasons behind this design are thoroughly described in subsection **3.1.2**.
The second write-based variant has managed to eliminate the need of an Env message, however, the drawbacks regarding eager communication make the up-to-now proposed write-based protocol worse in performance in comparison to the read-based one regarding messages of up to 40 bytes. The performance difference is illustrated in Figure 3.21. As one can observe in the figure, the latency of the write-based protocol,measured using the OSU Latency Microbenchmark, is significantly higher than the respective latency of the read-based protocol since all 7 eager message sizes take about 1,8 microseconds while the read-based protocol achieves a latency of about 1,3 microseconds. As stated, the cause of that difference is the fact that in the write-based protocol, eager receives issue RTR messages when posted earlier than the respective send functions. At the same time, eager receives that are posted after an eager message has already received, have to issue Ack control messages back to the sender. Besides the transmission of the extra messages themselves, the locking and data structure searching their receiving infers is also a considerable source of latency. Note that Figure 3.21 illustrates the latency of both implementations in a single QFDB (in the FPGAs F1 and F2). In scenarios where the two processes reside in different QFDBs or different Mezzanines, the latency of a control message's transmission is worse. This results to an even bigger difference between the two protocols (in favor of the read-based one) depending on the distance of the two nodes in the network. In addition, it is worth noting that the write-based eager protocol's overhead is partially attributed to the fact that eager sends insert new request objects into the Posted MPI Request structure in order for them to later get matched by the aforementioned control messages.

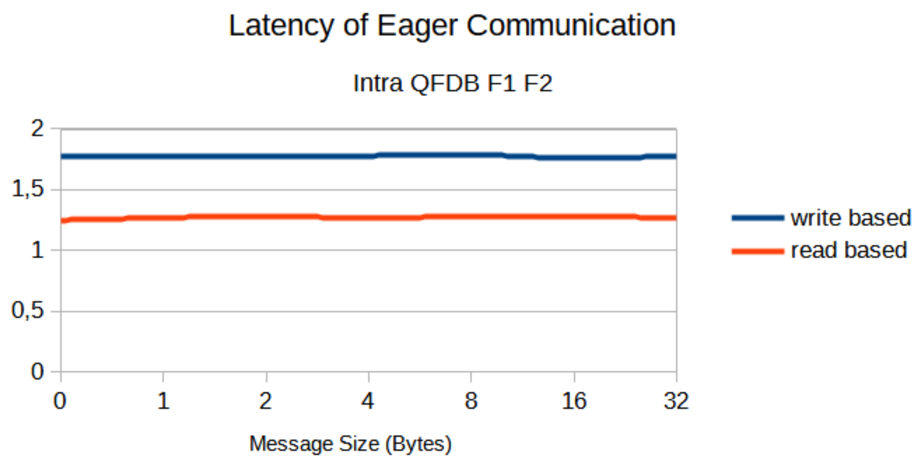


Figure 3.21: Comparing read-based and write-based MPI in eager communication latency

Due to those reasons, our main objective was improving the eager write-based protocol in order to eliminate the overhead of the RTR and Ack messages. Note that the reasons eager receives issue RTR messages in our protocol is a) because a long receive may match with an eager send as it is allowed by the MPI Standard and b) in order to avoid a mismatch between a long receive's RTR and an eager send that normally should match with an eager receive as described in 3.1.2. Considering the aforementioned fact, one understands that if the RTR messages of eager receives are removed from the protocol then the RTR messages of long receives should carry extra information which will prevent them from getting mismatched with eager sends they are not headed for, on the side of the sending process. In the process of determining this kind of information, we initially made the following changes in the protocol:

- Each receiving process preserves communicator, rank and tag specific counters indicating the number of eager receives the process has issued denoting the specific rank of that communicator as source using that specific communication tag.
- Each time an eager receive gets posted with those matching attributes, no RTR message gets transmitted (since there is no need to advertise DMA related information in eager communication) but the respective counter gets incremented. Ack messages are also not issued for eager receives that get posted after the eager message has arrived.
- The sending process inserts new send request objects into Posted MPI Requests data structure for eager sends as in the previous variants of the protocol.
- When a receiving process issues a long receive before a matching send has arrived, it initiates the communication issuing an RTR message (in cases where MPI_ANY_SOURCE is not in use) but inside the RTR message, it attaches the value of the aforementioned counter piggybacked. Since the counter is also tag specific, in this new variant of the protocol RTR suspension happens also when MPI_ANY_TAG is in use (as with MPI_ANY_SOURCE).
- When a receiving process issues a long receive after a matching send is already received:
  - ◆ If the send was a long one, the receiver issues a CTS message.
  - ◆ If the send was an eager one, the receiver just copies its payload data to its receive buffer and increments the corresponding counter. Generally, that counters indicates the number of all (long and eager) receives that didn't issue control messages.
- Each time a receiver piggybacks a counter to an RTR or a CTS message for this communicator, rank, tag triple, the counter's value is reset to zero.
- A sending process that receives a piggybacked RTR or CTS, it extracts the number of the piggybacked counter and clears that number of send requests the message can match with, before actually

performing a match. It is certain that all cleared messages will be unmatched eager sends pending to get cleared.

Using this technique, we manage to prevent RTRs of long messages from matching the wrong eager sends. This eliminates the need of issuing RTR and Ack messages for eager receives.

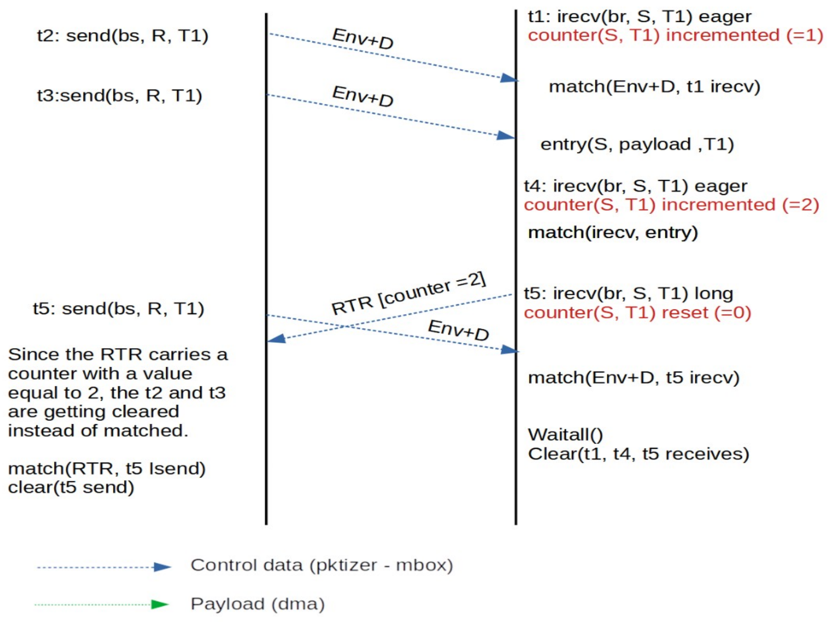

t2: send(bs, R, T1)
t3:send(bs, R, T1)
t5: send(bs, R, T1)
Since the RTR carries a counter with a value equal to 2, the t2 and t3 are getting cleared instead of matched.
match(RTR, t5 lsend)
clear(t5 send)
Env+D
Env+D
RTR [counter =2]
Env+D
t1: irecv(br, S, T1) eager
counter(S, T1) incremented (=1)
match(Env+D, t1 irecv)
entry(S, payload ,T1)
t4: irecv(br, S, T1) eager
counter(S, T1) incremented (=2)
match(irecv, entry)
t5: irecv(br, S, T1) long
counter(S, T1) reset (=0)
match(Env+D, t5 irecv)
Waitall()
Clear(t1, t4, t5 receives)
Control data (pktizer - mbox)
Payload (dma)

Figure 3.22: Eager protocol: Piggybacked counter in RTR optimization, match of long receive with eager send

The process we described becomes more clear in the scenario illustrated in Figure 3.22. Initially, the receiving process posts a non blocking receive request at *t1*. As observed, the process **omits transmitting an RTR message** for the eager receives of *t1* and *t4* and instead increments the counter regarding Rank S and tag T1. The counter's value is now set to 1. It is worth reminding that the counter is also communicator specific but the communicator is omitted in the Figures for simplicity. At *t2,* the sending process (with rank S) issues an eager send with tag T1 and transmits the Env+D message to the receiver with rank R. The Env+D message matches the receive request posted at *t1* and its payload data get copied to the receive buffer used in the former receive request. At *t3,* the sender posts another eager send and transmits the respective Env+D message to the receiver. The receiver receives the Env+D and stores a send request representing it, into its Received MPI Requests data structure. At *t4*, the receiving process posts an eager receive request which matches the received send request. The eager send's payload data get copied to the new receive request's receive buffer. **Instead of sending an Ack control message** back to the sender, the receiving process increments the counter regarding rank S and tag T1. The counter's value is now 2 while the eager sends of *t2* and *t3* remain unmatched by control messages up to this point. At *t5*, the receiver posts a long receive (i.e., an irecv that can receive data bigger than 40 bytes) that needs to advertise its DMA related information to the sender through an RTR control message. Since that long receive also regards rank S and tag T1, it piggybacks the value of the respective counter in the RTR. At the same time, the sender posts another eager send which happens to match with the last long receive. After it sends the Env+D message, the sending process receives the RTR message from the receiver. The sends of *t2* and *t3,* are still not marked as matched at the sender's side. *t2* send could match with the newly received RTR message since it has the same matching attributes and lies first in the FIFO queue of Posted MPI Requests of the receiver. However, the progress engine reads the value of 2 piggybacked in the RTR message and clears the first two matching send requests found before performing the match. Consequently, the RTR message clears the two first eager sends and achieves a match with the third one. In this way, a mismatch between *t2* or *t3* eager sends and the RTR headed for the send posted at *t5* gets prevented.

In Figure 3.23 a similar scenario gets illustrated. The sole difference being that at t5 the send posted is not eager but long. Subsequently, it issues an RTS message to the receiver. At *t6*, the receiving process posts a long non blocking request which matches with the send posted at t5, thus a CTS message is issued. The CTS message does not contain any counter piggybacked. However, its issuing also resets the respective counter. When the sender receives the CTS message, since in a correct implementation no CTS can match eager sends, it cleans all intermediate matching eager sends it encounters during the matching process. Ultimately, the DMA transfer gets performed.

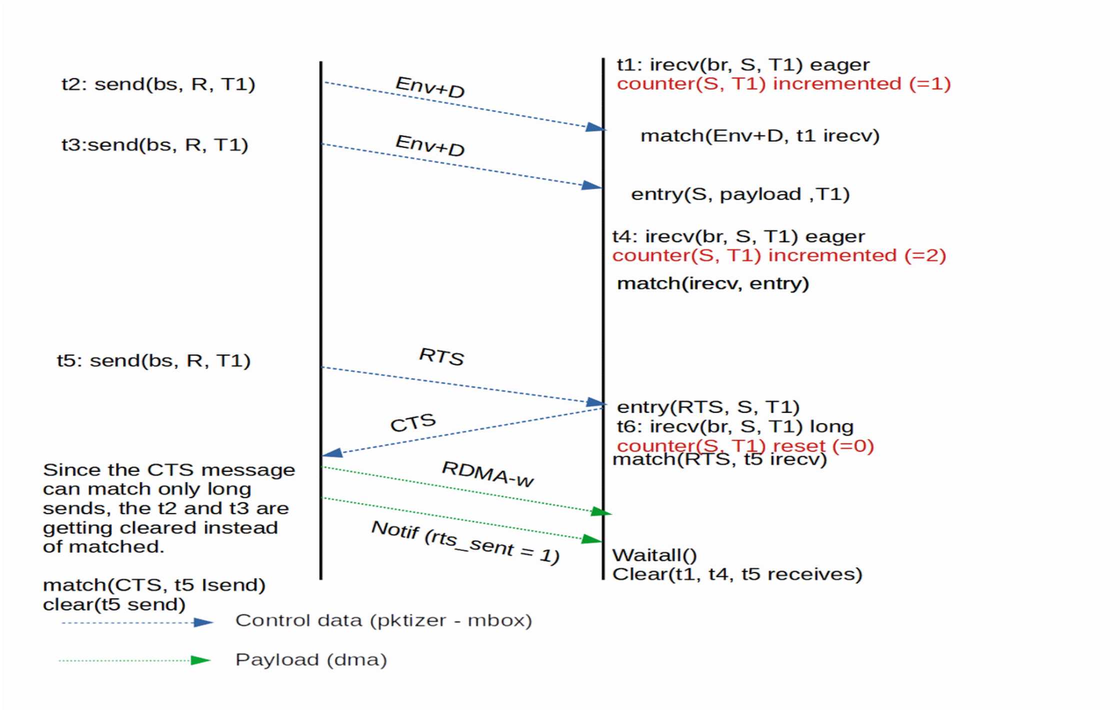

t2: send(bs, R, T1)
t3:send(bs, R, T1)
t5: send(bs, R, T1)
Env+D
Env+D
RTS
CTS
RDMA-w
Notif (rts_sent = 1)
t1: irecv(br, S, T1) eager
counter(S, T1) incremented (=1)
match(Env+D, t1 irecv)
entry(S, payload ,T1)
t4: irecv(br, S, T1) eager
counter(S, T1) incremented (=2)
match(irecv, entry)
entry(RTS, S, T1)
t6: irecv(br, S, T1) long
counter(S, T1) reset (=0)
match(RTS, t5 irecv)
Since the CTS message can match only long sends, the t2 and t3 are getting cleared instead of matched.
match(CTS, t5 lsend)
clear(t5 send)
Waitall()
Clear(t1, t4, t5 receives)
Control data (pktizer - mbox)
Payload (dma)

Figure 3.23: Eager protocol: Piggybacked counter in RTR optimization, CTS message clears pending eager sends

A case of high interest is presented in Figure 3.24 where the receiving process posts two non blocking eager receive requests followed by a long receive request while the sender has not posted any requests yet. In this occasion, the RTR control message, issued by the long receive, conveys the value of 2 of the counter that regards rank S and tag T1 piggybacked, indicating that 2 eager sends should get cleared on the sender's side before a match can happen. However, since the sender has not posted any matching eager send yet, the progress engine will insert two “crafted” receive requests (functioning as placeholders) in the Received MPI Requests FIFO queue before it stores the actual receive request derived from the RTR control message. These two “crafted” requests are are added in order to cause an immediate match (and clearing) when the sender posts the two eager sends the value of the counter piggybacked in the RTR was referring to. As seen, in the figure, the sends posted at *t4* and *t5* can get cleared right after they are posted since a match happened at the time of their issuing. Finally, the send posted at *t6* correctly matches the receive request containing the RTR message's DMA information and performs the DMA transfer.

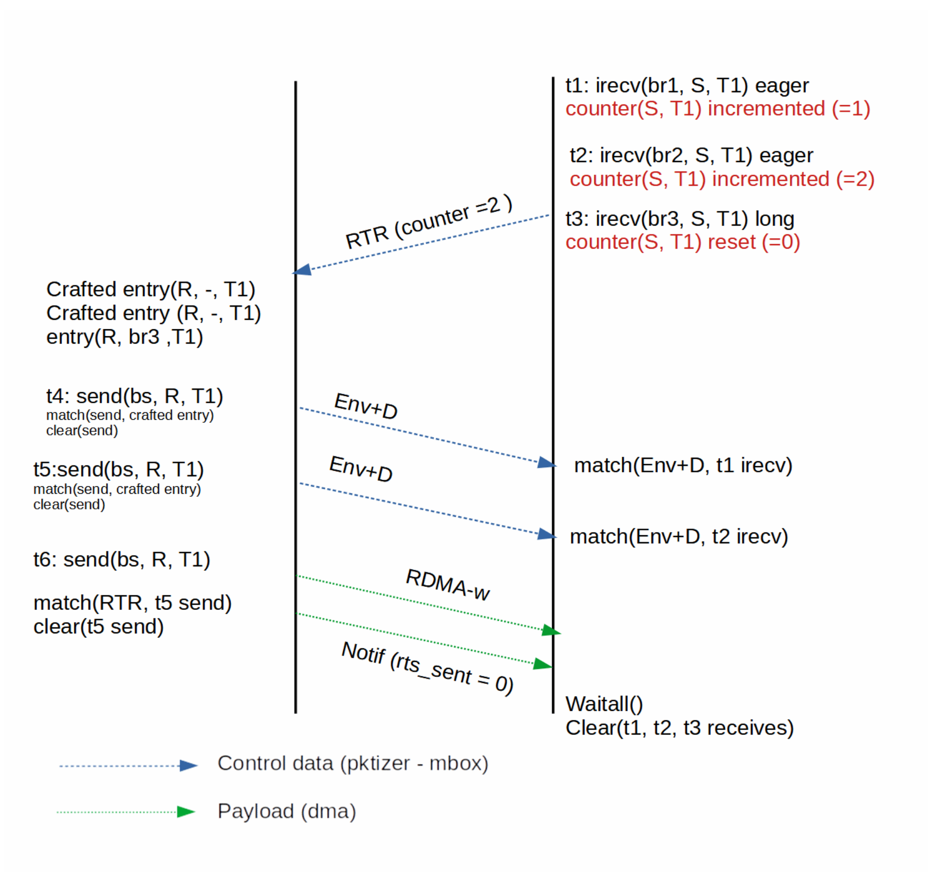


Figure 3.24: Eager protocol: Piggybacked RTR issued before any matching eager send request is posted. Use of crafted requests

As it is evident, this solution lets us fully omit the Ack and RTR messages an eager receive should issue. This fact render the latency of the eager communication of the write-based protocol significantly improved as seen in Figure 3.25 where we can observe a 20% improvement in latency which is attributed to the elimination of both the transmission of the omitted messages and the synchronization actions their receiving inferred (e.g., locking of the control logic lock by the progress engine thread). The figure shows the results of the OSU Latency microbenchmark for eager sizes. In addition, in scenarios when many eager receives precede a long one, the RTR message emerging from it will cause significant overhead at the sender's side due to the big amount of either the eager sends that will get cleared or the number of crafted receive requests that will be inserted. Although the idea of preserving a counter for each communicator, rank and tag a program uses indeed re, there was still room for improvement since eager sends still need to acquire the control logic lock, search the Received MPI Requests structure upon creation for matching RTR messages and also insert new send requests into the Posted MPI Requests data structure representing the transfer, actions which notably increase the latency of an eager send.

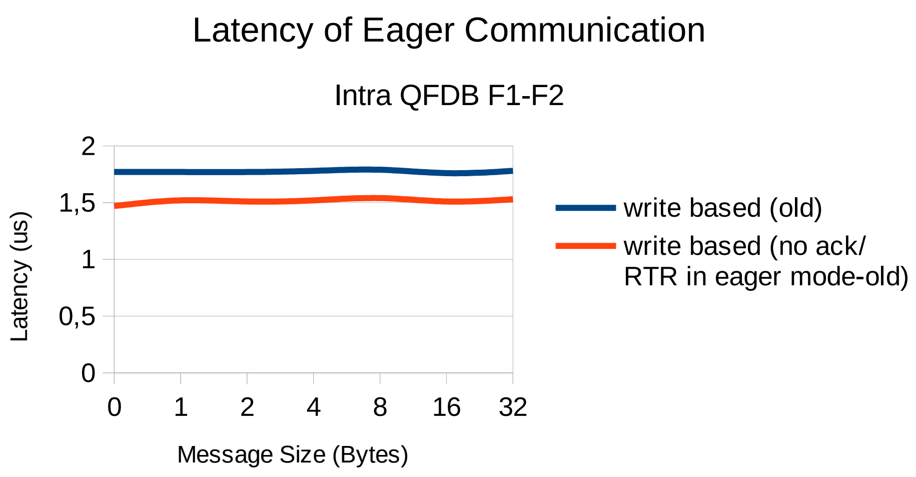


Figure 3.25: Comparison of the initial write-based MPI eager protocol with the optimized eager protocol in terms of latency

To solve this, we came up with a different and more elegant solution. In this new solution, we enrich the idea of communicator, rank and tag specific counters. More precisely, for each combination of communicator, rank and tag we maintain two counters. They serve the purpose of counting the number of posted receive and send requests with these matching attributes respectively. Each time a new send or receive MPI primitive gets called, the respective counter increments and a ticket ID equal to the current value of the counter gets assigned to the request the primitive regards. Each send and receive request gets assigned a ticket ID indicating its order among requests of the same type and same matching attributes (e.g., the 5th send of a process with tag 1 and destination 3 will get assigned the value 5, the 3rd send of a process with tag 0 and destination 1 will get assigned the value 3 etc). In a correct MPI program, it is guaranteed that a send request, assigned a specific ticket id, is going to match with a receive request that bears the same ticket id. Due to that fact:

- An RTR control message no longer piggybacks the number of receives that haven't issued a control message. Instead it carries the ticket ID of the request it regards.

- When an RTR is getting matched, the sending process no longer uses the counter value in order to clean pending eager send requests or create crafted receive requests. Instead it determines whether to perform a match or not with the first matching send request encountered, based on the equality of their ticket IDs

- If no matching send is found for an RTR message, the sending process compares the ticket ID in the RTR message with the value of the respective send counter the process maintains locally. If the ticket ID is bigger than the counter's value then a new receive Request gets inserted into the Received MPI Requests data structure. Otherwise, the RTR gets discarded since it is implied that it was headed for an eager send.

- Eager send requests no longer need to insert request objects into the Posted MPI Requests data structure
  in most sending modes (e.g., Send, Isend etc. While Ssend may be an exception for reasons explained in Section **3.11**)

- Eager sends also avoid acquiring the control logic lock and searching the Received MPI Requests structure since they do not need to match any RTR message as they dot make use of the DMA.

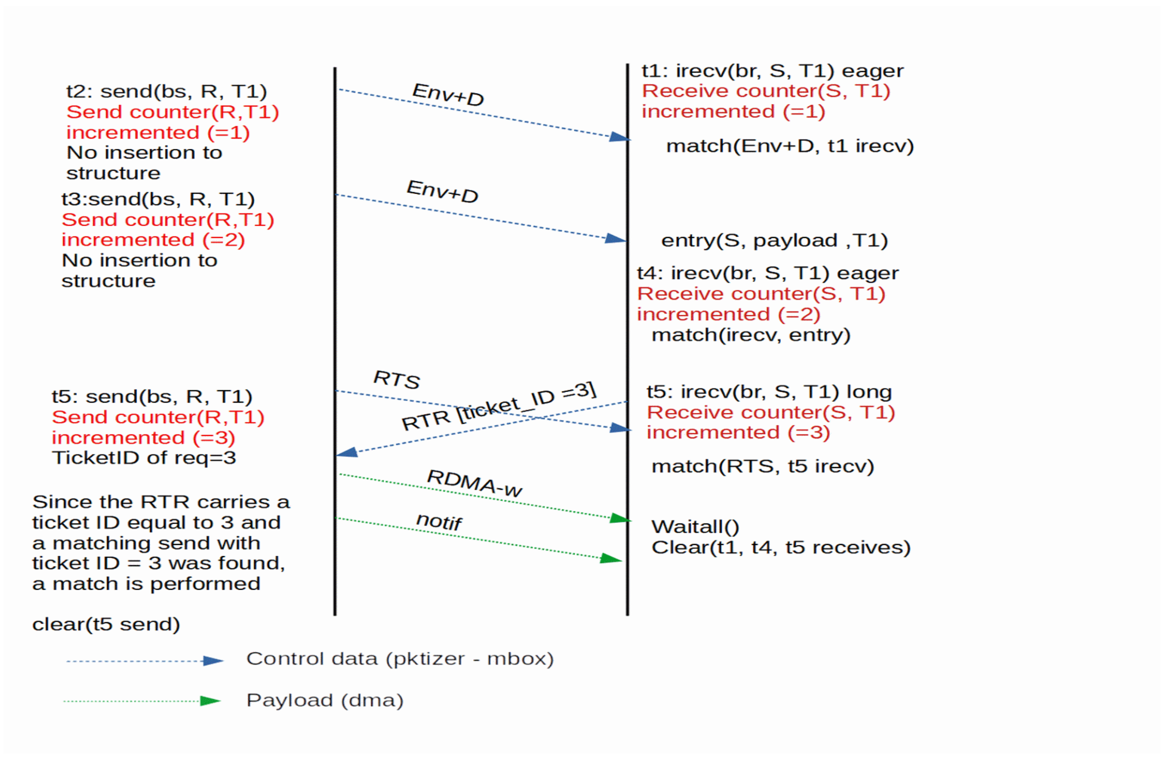

t2: send(bs, R, T1)
Send counter(R,T1) incremented (=1)
No insertion to structure
t3:send(bs, R, T1)
Send counter(R,T1) incremented (=2)
No insertion to structure
t5: send(bs, R, T1)
Send counter(R,T1) incremented (=3)
TicketID of req=3
Since the RTR carries a ticket ID equal to 3 and a matching send with ticket ID = 3 was found, a match is performed
clear(t5 send)
Env+D
Env+D
RTS
RTR [ticket_ID =3]
RDMA-w
notif
t1: irecv(br, S, T1) eager
Receive counter(S, T1) incremented (=1)
match(Env+D, t1 irecv)
entry(S, payload ,T1)
t4: irecv(br, S, T1) eager
Receive counter(S, T1) incremented (=2)
match(irecv, entry)
t5: irecv(br, S, T1) long
Receive counter(S, T1) incremented (=3)
match(RTS, t5 irecv)
Waitall()
Clear(t1, t4, t5 receives)
Control data (pktizer - mbox)
Payload (dma)

Figure 3.26: Piggybacked ticket_ID in RTR optimization, long receive matching long send

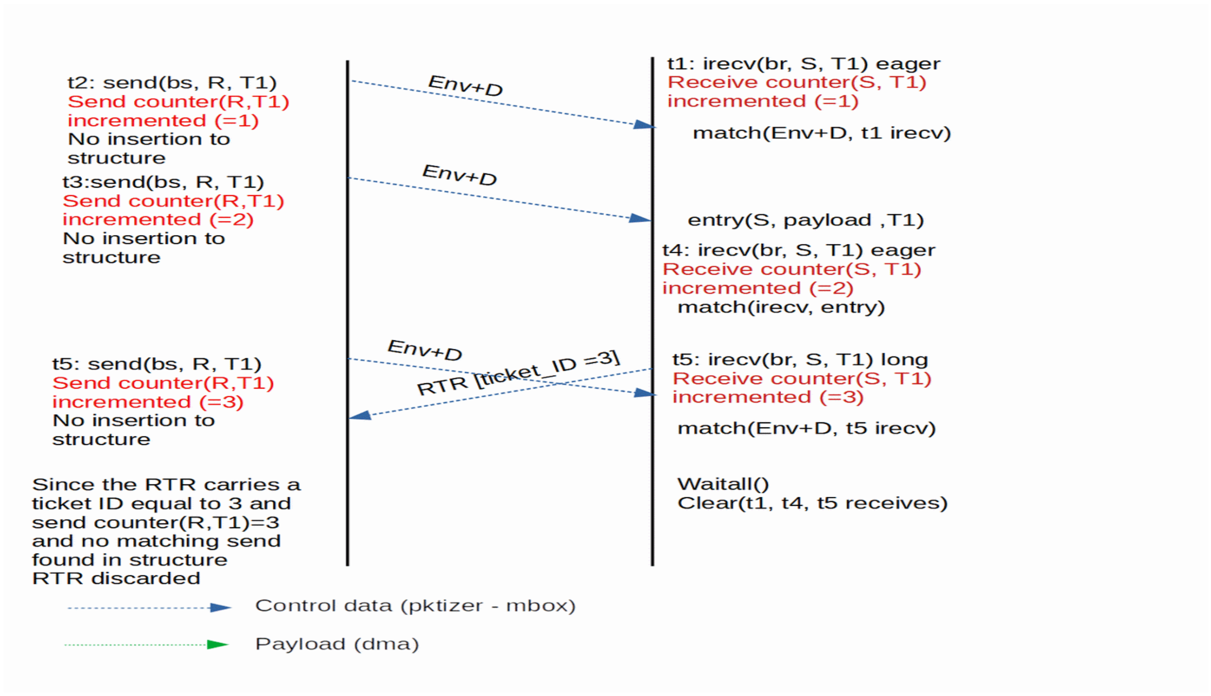
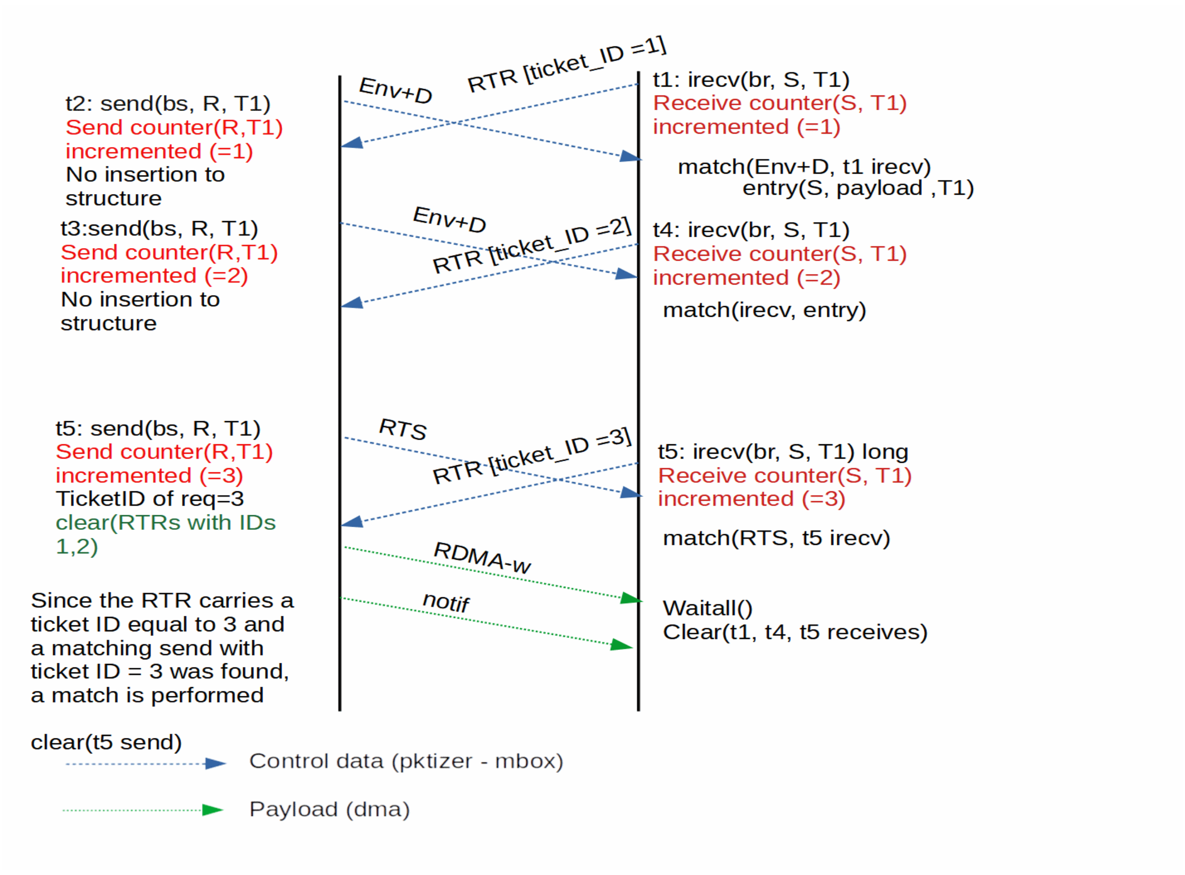

t2: send(bs, R, T1)
Send counter(R,T1) incremented (=1)
No insertion to structure
t3:send(bs, R, T1)
Send counter(R,T1) incremented (=2)
No insertion to structure
t5: send(bs, R, T1)
Send counter(R,T1) incremented (=3)
No insertion to structure
Since the RTR carries a ticket ID equal to 3 and send counter(R,T1)=3 and no matching send found in structure
RTR discarded
Env+D
Env+D
Env+D
RTR [ticket_ID =3]
t1: irecv(br, S, T1) eager
Receive counter(S, T1) incremented (=1)
match(Env+D, t1 irecv)
entry(S, payload ,T1)
t4: irecv(br, S, T1) eager
Receive counter(S, T1) incremented (=2)
match(irecv, entry)
t5: irecv(br, S, T1) long
Receive counter(S, T1) incremented (=3)
match(Env+D, t5 irecv)
Waitall()
Clear(t1, t4, t5 receives)
Control data (pktizer - mbox)
Payload (dma)

Figure 3.27: Piggybacked Ticket ID optimization, eager receive matching eager send

In Figure 3.26 at *t1* the receiving process issues a non blocking eager receive request denoting S as source rank and T1 as communication tag. That receive requests increments the respective receive counter of the receiving process and gets assigned a ticketID with value 1 and does not transmit an RTR message. At the sender's side, at t2 and t3, the sending process posts 2 eager send requests. None of those sends insert any object in the Posted MPI Requests but instead increment the respective send counter. As a result, the send counter regarding rank R and tag T1 has the value of 2. The Env+D messages issued get matched on the receiver's side with the receive requests posted at t1 and t4. At t5, the sender posts a long send request, inserting a new request object in the Posted MPI Requests data structure that gets ticket ID equal to 3 while at the same time the receiver also issues a long receive request with ticket ID also equaling 3. Since the send request and the RTR message both have the same ticket ID and matching attributes, a match takes place and the sender performs the DMA transfer.

In Figure 3.27 a similar scenario is depicted. In contrast to the previous scenario, the send posted at *t5* is also -an eager one. As a consequence it doesn't cause any insertion of any request object into any Posted MPI Requests FIFO queue. The RTR message emerging from the receive request posted at t5, finds no matching send request so its Ticket ID gets compared with respective send counter's value and since they are equal the RTR message gets discarded since it was headed for an eager send.
If the ticket ID of the RTR message was larger than the value of the respective counter in a different scenario, a new receive request would have been inserted into the Received MPI Requests structure of the sender. That RTR should get matched by a future send.

It's worth noting that in order to render eager sends as fast as possible, an eager send (except for MPI_Ssend) issues an Env+D message and returns immediately without acquiring any lock and searching the Received MPI Requests since it does not need to learn any DMA related information. This strategy can lead to some received RTRs of fast long receives that match with eager sends being left marked as unmatched. For this reason, long sends also clear received receive requests that have a smaller ticketID than the ticketID they got assigned. This normally does not cause any considerable overhead since eager sends matching long receives without intermediate matches between long sends and long receives are especially rare in existing applications. An example is illustrated in Figure 3.28



t2: send(bs, R, T1)
Send counter(R,T1) incremented (=1)
No insertion to structure
t3:send(bs, R, T1)
Send counter(R,T1) incremented (=2)
No insertion to structure
t5: send(bs, R, T1)
Send counter(R,T1) incremented (=3)
TicketID of req=3
clear(RTRs with IDs 1,2)
Since the RTR carries a ticket ID equal to 3 and a matching send with ticket ID = 3 was found, a match is performed
clear(t5 send)
Env+D
RTR [ticket_ID =1]
Env+D
RTR [ticket_ID =2]
RTS
RTR [ticket_ID =3]
RDMA-w
notif
t1: irecv(br, S, T1)
Receive counter(S, T1) incremented (=1)
match(Env+D, t1 irecv)
entry(S, payload ,T1)
t4: irecv(br, S, T1)
Receive counter(S, T1) incremented (=2)
match(irecv, entry)
t5: irecv(br, S, T1) long
Receive counter(S, T1) incremented (=3)
match(RTS, t5 irecv)
Waitall()
Clear(t1, t4, t5 receives)
Control data (pktizer - mbox)
Payload (dma)

Figure 3.28 Posted Send request clears useless received requests (RTRs)

This protocol design improves upon the first two variants regarding eager communication of the write-based protocol. As we can see in Figure 3.29, in comparison to the first write-based variant (**write-based (old)** in the figure) our solution offers up to 33% performance gain as far as latency is concerned. In addition, the final version has also another 14% lower latency than the first eager optimization presented in this chapter (**write-based (no Ack/RTR in eager mode-old)** in the figure). This performance gain is attributed to the elimination of locking, searching Received MPI Requests and inserting new request objects to the Posted MPI Requests during an eager send emerged from functions like MPI_Send. We see that with our solution, the write-based eager communication latency became almost even with the read-based protocol's latency.

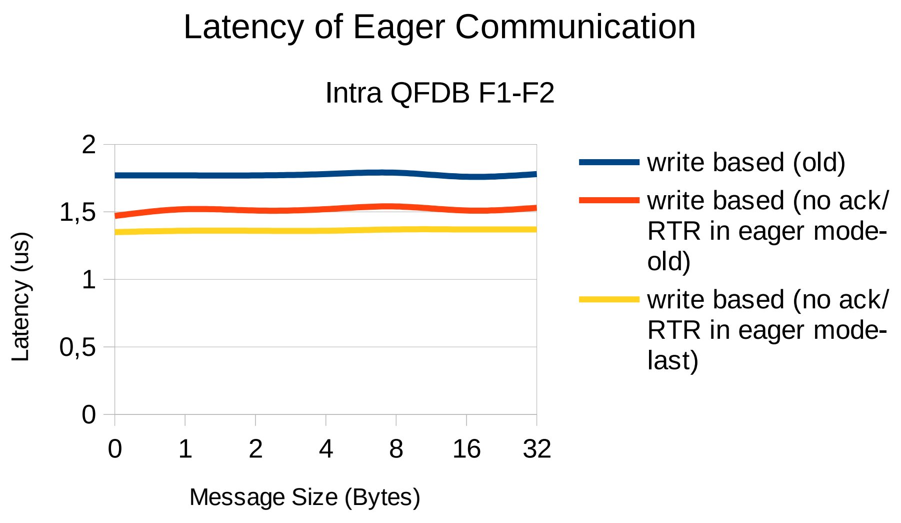


Figure 3.29: Comparison of the original eager protocol of the write-based MPI with the two different optimizations

3.6 Basic description of the third write-based MPI variant's implementation

In General the implementation details of all variants of the write-based MPI are the same with those described in 3.2. Only substantial differences will get presented in this section.

3.6.1 Structure of Request Object

The same details stated in 3.2.2 also apply. The main change in this variant is the existence of the **ticketID** in all request objects that regards point-to-point communications. As explained in the previous chapter, requests use this value in order to determine whether they should match with a received RTR message or not.

3.6.2 General Control logic of Basic Point-to-Point primitives

Send Requests: Generally, all send requests increment the respective send counter that regards their destination rank, tag and communicator and get assigned a ticketID equal to its value. In long send requests, the increment of the counter and the assignment of a ticketID takes place after the acquiring of the control logic lock. Hence, no additional synchronization is required to protect the counters. Long sends also check the ticketID of a received RTR before they perform a successful match. This value must be equal to the send request's ticketID else no match takes place. Should a long send request encounter matching received receive requests with ticketID lower than their own, they clear these requests, since they were headed for past eager sends. In the case of eager sends, no lock gets acquired except for the case of the thread mode **MPI_THREAD_MULTIPLE or the primitive MPI_Ssend/MPI_Issend.** The reasoning behind this decision is the following: Since eager sends emerged from most MPI send primitives (besides Ssend) do not require to get aware of any information contained in an RTR control message, they do not check the Received MPI Requests structure. Also, they do not insert any request object in the Posted MPI Requests structure to get matched by incoming RTR messages. These facts render the acquiring of control logic lock useless and thus we let those eager sends increment their respective counter without locking which is safe in single threaded applications or with applications using any thread mode other than MPI_THREAD_MULTIPLE. One argument against that decision may be the existence of the progress engine thread, besides the user thread, which also reads the value of the counters. However, in the next chapter ,regarding the control logic of the progress engine, we show that this design choice remains safe assuming no out-of-thin-air produced during races. In case the MPI_THREAD_MULTIPLE is used, acquiring of control logic lock is necessary even for eager sends in order to avoid race conditions when two simultaneous eager sends may try to increment the same counter.

Receive Requests: The control logic of receive requests is similar to that of the second variant. The only addition is that each receive request increments the respective receive counter associated with its source rank, tag and communicator, gets assigned a ticketID and attaches its ticket ID to the RTR message, if it issues one. CTS messages do not need to carry a ticketID since it is guaranteed that no CTS may ever match with an eager send. Additionally, receives with maximum receive size less than 40 bytes do not issue RTR messages since they do not need to advertise DMA related info. Instead, they immediately block for an incoming Env+D if not already received (if they are blocking). It is also worth reminding that, as with MPI_ANY_SOURCE, receiver imitation suspension also applies when MPI_ANY_TAG is in use in this variant. In that case, the receive requests that use MPI_ANY_SOURCE/MPI_ANY_TAG cannot determine which counter to increment. Additionally, the rest of the requests cannot increment their respective counter either because they may violate the FIFO order the MPI Standard guarantees. In such scenarios ,receive requests increment their counters only upon the matching with an RTS or Env+D message when they learn the source rank and tag. When all requests that used MPI_ANY_TAG/MPI_ANY_SOURCE get cleared, RTR suspension is lifted.

Wait and Test Point-to-Point primitives: Wait, Test and Relative functions return immediately on eager send requests since they no longer insert any object to any data structure. Regarding all other kinds of requests, these primitives maintain the functionality described in the previous variant.

3.6.3 General Control logic of the Progress Engine Thread

The more significant change regarding the progress engine in this variant is the handling of incoming RTR control messages. When an RTR message arrives, the progress engine locks the control logic lock as before and tries to match it with a send request in the Posted MPI Request structure. However, instead of relying solely on the matching attributes and the ***matched*** flag of the requests, the progress engine also checks for their ticket ID. Only if the ticket ID of the RTR message is equal to the ticketID of a matching send request, a match takes place.

- If no matching request is found, the progress engine compares the RTR's ticketID with the value of the respective send counter of the MPI process. If the ticketID of the RTR is less than or equal to the value of the counter, then this RTR is an obsolete message which regards an eager send that did not insert any object into the Posted MPI Requests structure and gets discarded. If the ticketID of the RTR is larger than the value of the respective send counter, then this RTR was headed for a future send request not posted yet and a new receive request object gets inserted into the Received MPI Requests. This new object contains the matching attributes and ticketID conveyed by the RTR message. Since eager sends increment the send counter without locking (in applications with no use of MPI_TRHEAD_MULTIPLE), there is a chance of a counter getting incremented after the progress engine reads its value. Even if something like that happens, the worst outcome possible is that the progress engine will insert a useless receive request into the Received MPI Requests structure. That redundant request will not cause any mismatches (due to its ticketID value) and will most likely get cleared by a future long send.
- If a matching request is found, the progress engine makes the same steps described in previous variants.

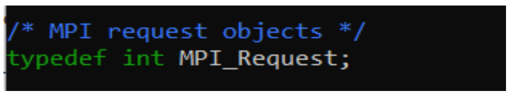
One other interesting case is when MPI_ANY_SOURCE/MPI_ANY_TAG is in use and receiver initiation is suspended. In that case, posted receive requests cannot increment their respective receive counters and get assigned ticketIDs. Due to this fact, the progress engine increments the respective receive counter and assigns ticket IDs to receive requests when they get matched by RTS or Env+D control messages. It is worth noting that receiver initiation suspension applies only for communicators that have requests with MPI_ANY_SOURCE active while requests regarding other communicators can still use receiver initiation. Similarly, MPI_ANY_TAG suspends receiver initiation only for receive requests that denote source ranks also denoted by other requests that use MPI_ANY_TAG. For instance, if no MPI_ANY_SOURCE is used and a process posts the following request:

MPI_Irecv(buf, 1000 , MPI_INT, 5, MPI_ANY_TAG, MPI_COMM_WORLD, &request)

allowing any tag to match it from rank 5, all other requests that denote source ranks other than 5 **can still utilize receiver initiation.** The suspension applies only for requests that denote rank 5 as source rank.
This last variant of the write-based MPI is the final design of our implementation and will be used in experiments described in the Evaluation Chapter unless otherwise stated.

3.7 Changing type of MPI_Request

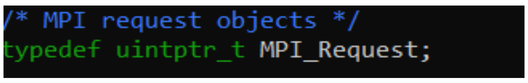
As already stated, the MPI_Request type used in MPI is actually a 32bit integer in MPICH. This is clearly defined in the mpi.h header file which exists in all MPI Implementations.



```
/* MPI request objects */
typedef int MPI_Request;
```

Figure 3.30: Original MPI_Request type in mpi.h

Pointers to MPI_Request objects are passed as arguments to non blocking send and receive primitives as wells as non blocking collectives. Additionally, wait and test functions use such arguments in order to wait for or test the completion of a request respectively. Since these primitives may also complete the MPI_Status object of a non blocking receive request as well as read its notification data in order to determine completion, one can easily understand that an implementation needs to somehow map the MPI_Request to a request object stored in one of the implementation's data structure. As explained in previous chapters, in our implementation each request object contains a MPI_Request * field which stores the location in memory of the MPI_Request used in the non blocking primitive which created it. In that way, by searching our Posted MPI Requests data structure for the request object that has an MPI_Request * that shows to the address of the MPI_Request the user supplied as argument we map MPI_Requests to request objects successfully. However, this method requires acquiring the control logic lock and searching our respective data structure. In order to avoid this process, we decided to edit the mpi.h header file of MPICH and change the type of MPI_Request to **uintptr_t,** an unsigned integer of size equal to the size of a pointer. As a result, we can set the value of a MPI_Request object to the location in memory the correct request object resides in our implementation. For instance, a non blocking receive will insert a receive request into the Posted MPI Requests data structure and subsequently set the value of MPI_Request to the location of the request it just inserted. When MPI_Wait takes a pointer to that MPI_Request as argument, it can, just by dereferencing it, learn the location of the receive request object without acquiring any locks and searching any data structure.



```
/* MPI request objects */
typedef uintptr_t MPI_Request;
```

Figure 3.31: Modified MPI_Request type in mpi.h

This change renders the implementation of wait and test primitives more trivial and also offers a very slight improvement in performance. The osu_bw microbenchmark of OSU Microbecnhmarks contains exclusively non blocking send and receive MPI calls. Sizes from 64 to 256 bytes are presented in Table 3.1 since in these sizes the performance difference is more apparent.
This optimization is used in both the second and third variant of the write-based implementation. (described in Sections 3.3 and 3.6 respectively)

Table 3.1: Comparison of bandwidth with original and modified MPI_Request

osu_bw, OSU MicroBenchmarks	Vanilla MPICH MPI_Request	Modified MPICH MPI_Request
64 bytes	15,61 MB/s	15,71 MB/s
128 bytes	26,83 MB/s	26,92 MB/s
256 bytes	52,65 MB/s	52,73 MB/s

3.8 Handling Communicator manipulation

MPI offers users the ability to distribute processes into subgroups called Communicators. A communicator can be described as a world of MPI Processes that can communicate with each other. Collective functions always include all processes of a communicator. The default communicator which exists at the start of an MPI program and includes all the running MPI processes is called MPI_COMM_WORLD. During the execution of the program, a user can create more communicator consisting of some of the processes of MPI_COMM_WORLD using communicator manipulation primitives like: ***MPI_Comm_split, MPI_Comm_dup, MPI_Comm_create*** etc.

Our implementation maintains a data structure called **Communicators Registry** as mentioned in Section **3.2.1.** This data structure is actually a hashtable of 1000 buckets. Each bucket contains a list of objects each one of them describing a communicator. Each communicator object carries as a key the communicator's ID and it's put into the appropriate bucket in accordance to that key. The information contained in a communicator object residing in the Communicator's Registry is the following:

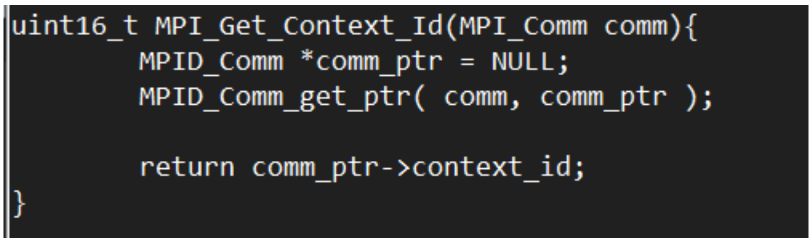
- **uint16_t comm_id** *The communicator's 16bit ID*
- ***uint64_t * node_ids*** *Array of size equal to the size of the communicator. Contains the node ID of each process participating in the communicator in the respective cell of the array. ie. The node ID of rank i is the ith element of the array.*
- **uint64_t * protection_ids** *Array of size equal to the size of the communicator. Contains the protection ID of each process participating in the communicator in the respective cell of the array.*
- **uint64_t * mailbox_addresses** *Array of size equal to the size of the communicator. Contains the mailbox address of each process participating in the communicator in the respective cell of the array.*
- **uint64_t * offsets** *Array of size equal to the size of the communicator. Contains the DMA offset of each process participating in the communicator in the respective cell of the array.*
- **int my_rank** *The MPI rank of the process in that Communicator (An MPI process doesn't necessarily have the same MPI rank in all communicators it is a member of)*
- **struct counter_queue * mysend_counts** *Array of size equal to the size of the communicator. Each element of the array represents a destination rank and contains a queue of counters. The size of the queue in a specific point in time is equal to the number of different tags that have been used with this destination rank up to that time and each counter represents the number of sends the process has posted in that communicator with the respective destination rank and communication tag. This structure is necessary for the implementation of the third variant of the write-based MPI described in section **3.6.***
- **struct counter_queue * myrecv_counts** *Array of size equal to the size of the communicator. Each element of the array represents a source rank and contains a queue of counters. The size of the queue is equal to the number of different tags that have been used with this source rank and each counter represents the number of receives the process has posted in that communicator with the respective source rank and communication tag. This structure is necessary for the implementation of the third variant of the write-based MPI described in the 3.6 chapter of this thesis.*

In our implementation, MPI_Comm_split, MPI_Comm_dup, MPI_Comm_create, MPI_Comm_create_group and MPI_Cart_Create are supported for creating new communicators. Each function, after the creation of the new communicator, makes use of the **PMPI_Allgather** primitive in order to update the newly created communicator object that gets inserted into the Communicator's Registry with information regarding DMA offsets, node and protection IDs etc. as done in MPI_Init(). (Described in Section 3.2.1)
It's worth mentioning that in MPICH, the 16 bit communicator ID of a communicator is not normally accessible to users and remains a hidden component of the internal MPICH implementation. As a result, we

had to edit the source code of MPICH in order to get able to access it. More precisely, MPICH contains an internal C structure called **MPID_Comm** which represents a communicator. This C structure contains a field call **context_id** which is the 16bit ID of the communicator and is **not** available to users. MPICH also includes a function, that maps a MPI_Comm object (i.e., the usual datatype used for communicators and accessible to users) to the respective hidden MPID_Comm object by assigning a pointer to the MPID_Comm object's localtion. The signature of the function is the following

*MPID_Comm_get_ptr(MPI_Comm comm, MPID_Comm * comm_ptr);*

By creating a new function that takes an MPI_Comm as argument, calls the *MPID_Comm_get_ptr* and returns the context_id field of the MPID_Comm object we manage to render the ID of a communicator accessible in our implementation.



```
uint16_t MPI_Get_Context_Id(MPI_Comm comm){
        MPID_Comm *comm_ptr = NULL;
        MPID_Comm_get_ptr( comm, comm_ptr );

        return comm_ptr->context_id;
}
```

Figure 3.32: Added function to support communicator ID exporting

We include this function in the **comm** folder of the MPICH's source code folder and also include its signature in the mpi.h header file. It is reminded that the communicator's ID is also used in matching between control messages and requests as it is a basic matching attribute.
Our implementation does not support intercommunicator communication.

3.9 Memory Allocation Optimizations

Each time a send or receive request gets posted in our implementation, there is a high chance that it will need to insert a new request object in a Posted MPI Request queue. In addition, when the progress engine threads dequeues control messages from the mailbox that do not match any posted request, it must allocate and insert new request objects into one of the Received MPI Request queues. These facts make our implementation in constant need of memory allocation for objects of known size (i.e., the size of a request object). In order to improve the performance of our implementation, we decided to make use of a memory pool which gets initialized during the MPI initialization. Precisely, we allocate a big memory block in the heap memory equal to 200.000 request objects (can be configured) at the program's startup and each time a new request object is needed we use fragments of that space instead of calling memory allocation functions like malloc again and again since they can involve the kernel in occasion and worsen performance significantly. When a request object that was allocated from the memory pool gets cleared, its memory location gets added to a list of pointers for future reuse. In addition, we use stack allocation (instead of heap) for posted request objects emerging from blocking primitives like MPI_Send or MPI_Recv since their lifetime is equal to the lifetime of the respective function's stack (i.e., blocking primitives clear their request objects before returning).

3.10 Supported MPI Send modes

MPI supports many different send modes each one suited for specific user purposes. Our Implementation supports the following MPI send primitives:

- **MPI_Send/MPI_Isend** The normal sending mode most commonly used. It utilizes the eager protocol for messages smaller than 40 bytes and the long protocol for larger messages as described in the previous sections.

- **MPI_Rsend/MPI_Irsend** (Ready Send). This primitive requires the user's knowledge that a matching receive is already posted. It is the user's responsibility to ensure this condition before using the primitive. It is implied that this send mode always uses eager mode in some implementations regardless of the message's size. However, in our implementation its behavior is identical to that of MPI_Send/MPI_Isend since the prototype's packetizer cannot be utilized for messages larger than 40 bytes.

- **MPI_Bsend** (Buffered Send). This primitive returns immediately and the user can use their send buffer after the return. However, the message may not yet be delivered. This is achieved with the use of a preallocated buffer to which the contents of the send buffer get copied before the primitive returns. The user must preallocate such a buffer with the use of **MPI_Buffer_attach.** In our implementation, this primitive is identical to MPI_Send for eager messages or for long messages that get posted when a matching RTR is already received. However, when a matching RTR does not exist in the Received MPI Requests data structure, this primitive issues an RTS message to the destination rank, copies the contents of the send buffer to the preallocated buffer and returns so the user is free to use their send buffer again immediately. The DMA transfer takes place by the progress engine thread as soon as a matching RTR or CTS control message gets received.

- **MPI_Ssend/MPI_Issend** (Synchronous send). This type of send primitives may not return unless a matching receive has been posted by the receiver. In our implementation, the behavior of MPI_Ssend is identical to that of MPI_Send for long messages since the receiving of a matching RTR or CTS control is a sufficient indication that a matching receive is indeed posted by the destination rank. For eager messages, this primitive will acquire the control logic lock and search the Received MPI Requests for a matching RTR considering the fact that an eager send is allowed to match a long receive. If a matching RTR exists, the eager message is sent and the function returns immediately. In case no matching RTR exists, the primitive cannot return immediately after the delivery of the eager message since there is no way to know whether a matching receive is posted. In such a case, the Ssend function inserts a new send request into the Posted MPI Requests and issues a slightly different control message called **Env+D_S**. This message contains the same information an Env+D message contains (Envelope with the payload data concatenated) but its type informs the receiver that this message emerged from a synchronous MPI primitive (i.e., MPI_Ssend/MPI_Issend). Consequently, the receiver should issue an Ack control message back to the sender in order to render them aware that a matching receive is posted. This applies only to eager receives or long receives which have not issued RTR messages yet (e.g., because of MPI_ANY_SOURCE/MPI_ANY_TAG or because they got posted after the send). Should a receive request has already issued an RTR message for that communication, no Ack is needed. When the Ack message gets received by the sender and the progress engine thread matches it against the correct send request, the send primitive returns. As one can see, this type of send primitive makes it compulsory for eager sends to also acquire the control logic lock, search the Received MPI Requests and even insert a new object in the Posted MPI Requests even in single threaded applications in contrast to MPI_Send as pointed out in **3.6** and **3.7.2**

3.11 Support for Persistent Point-to-Point MPI Requests

The MPI Standard offers support for persistent MPI_Request objects. These persistent requests are extremely useful in situations where a user may need to call multiple send or receive functions using always the same arguments (i.e., buffer. Destination rank, count, datatype etc). In such cases, MPI gives the user the opportunity to create a persistent MPI_Request that will survive when the communication it represents is over

and can thus be reused as it is.
For example, supposing a user needs to call the same MPI_Send function with the exact same arguments multiple times within a loop as seen in the following example:

```
int i=0;
while(i++<100){
		MPI_Send(buf, count, datatype, dest, tag, comm);


}
```

With the use of a persistent MPI_Request, the user can activate the same request repeatedly without innvoking MPI_Isend each time. In the following example the use of persistent request is demonstrated in a scenario identical as the one of the previous example

```
MPI_Request request;
MPI_Send_init(buf, count, datatype, dest, tag, comm,, &request);

while(i++<100){
		MPI_Start(&request);
		MPI_Wait(&request, MPI_STATUS_IGNORE);

}
```

In the example above, a persistent MPI Request is created with a call to **MPI_Send_Init** and gets bound with the arguments the user supplies to the function. Later, the user can initiate the exact same request with MPI_Start without having to call MPI_Isend and supplying the same arguments again and again. This capability of persistent requests exists in MPI standard in order to let implementators offer better performance when the same arguments are being used. In our implementation we support **MPI_Send_Init** and **MPI_Recv_Init** primitives while persistent requests for the rest of send modes is trivial to implement in the future. The main performance gain of using persistent requests in our implementation is the fact that a **request** object gets allocated only once with **MPI_Send_init/MPI_Recv_init** and initially is not put in any data structure. MPI_Start inserts the allocated request object into the Posted MPI Requests queue while MPI_Wait disconnects it from it. Subsequently, the user can repeat the same transfer with the same arguments without causing our implementation to allocate a new request object for each new iteration. In contrast, should the user used multiple calls to MPI_Send, a new request object would get allocated internally in each loop iteration.

3.12 Support for Probing Primitives

Another feature the MPI library provides to its users is the ability to probe for incoming messages without posting receive requests. Specifically, one can use the functions **MPI_Probe** or **MPI_Iprobe** in order to check whether a send, with the matching attributes denoted in the arguments of those functions, has been posted before actually receiving it. The difference between the two probing functions is that the former blocks until such a send request gets posted while the other returns immediately. The signature of MPI_Probe is

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_Iprobe's signature is almost identical with the only difference being that it also requires one more argument, a pointer to a flag integer the value of which will indicate whether a posted send was found. As one notices, the first three arguments are the preferred matching attributes of the send request and the last argument is a pointer to an MPI_Status object. This object is used in order to enable the user find out the size

of the probed send as well as the source rank and tag in case MPI_ANY_SOURCE/MPI_ANY_TAG were used as matching attributes.
In our implementation, MPI_Iprobe simply locks the control logic lock and searches the Received MPI Requests for a matching send request. If a matching request is found, it does **not** get marked as matched since that would prevent a future receive from matching it. The value of the flag integer is set to 1 and the MPI_Status object gets updated accordingly. In case no matching send received request exists, the flag's value is set to 0. In both cases the function unlocks returns immediately.
MPI_Probe also searches the Received MPI Request once, and if a matching send request exists, it also returns immediately. On the other hand, if no matching request exists, the function avoids constantly locking, searching and unlocking. Instead, it inserts a pseudo-receive request in the Posted MPI Requests and waits until it gets "matched" by an RTS or Env+D control message. Notice that if the progress engine thread finds a matching pseudo-receive request while trying to match an RTS or Env+D message, it doesn't consider it as a real match. When this happens, the blocking probing function returns having completed the MPI_Status object correctly.

3.13 Support for Collective Primitives

Besides point-to-point functions, MPI also offers communication functions that involve all the MPI processes in a communicator. These functions are called collective function and they are usually implemented on top of point-to-point primitives. Some of the common collectives are MPI_Bcast, MPI_Reduce and MPI_Barrier[22]. In Exanet MPI, most blocking collective functions had been implemented (on top of simple point-to-point functions) before the beginning of this work and have been compiled as a shared library. As a result, one can support collective functions in our MPI implementation by making use of that shared library in combination with the library of the write-based MPI. This process, which is described in the Evaluation chapter, causes each blocking MPI collective encountered in a program to get delegated into our collectives library and subsequently into the point-to-point primitives of our implementation.

MPI also supports non blocking versions of collective functions like MPI_Ibcast. These functions return immediately and one can wait for their completion using a wait or test primitive like MPI_Wait. Their implementation is not trivial since it would require either the use of more simple algorithms than the ones used in the blocking versions (which would destructive for performance) or either the implementation of part of their logic into the progress engine. Instead, we chose to implement non blocking collectives as follows; We make use of an extra thread called a worker thread which contains a list with tasks. When a non blocking collective function is called by the user thread a new task is getting added into that list and the function returns. The worker thread dequeues a task and executes the blocking version of the collective the user has invoked. The user thread returns immediately so the process is transparent to the user who can then wait for completion using MPI_Wait. When the blocking version of the collective is over, the worker masks the MPI_Request it regards as completed. In order to avoid the control messages of the worker thread to get mixed with those of the user threads, the non blocking collective gets executed in a duplicate of the denoted communicator (created by Mpi_Comm_dup) and not the same communicator provided by the user. Non blocking collectives are not implemented in the read-based variant.

3.14 Support for an Optimistic version of the write-based variant

During our research and while experimenting with the different variants of our implementation, we thought of some speculative scenarios that may not fully comply with the MPI Standard but could improve performance and efficiency. For instance, as we have already noted, the existence of MPI_ANY_SOURCE and MPI_ANY_TAG seriously limits our ability to use optimized data structures like hashtables for preserving the Received Send requests. This happens because we should keep all Received send requests in the same queue in order to guarantee the MPI's FIFO property for a posted receive that will denote MPI_ANY_SOURCE or MPI_ANY_TAG as source rank or communication tag respectively. Thus, we see that if these wildcards are not supported by an implementation, one could hold the implementation's state in more efficient data structures. In addition, one could argue that sender initiation is not needed when MPI_ANY_SOURCE or MPI_ANY_TAG do not exist, since then all MPI communications (except for eager messages) would get initiated by the receiver. The receiving process would always initiate communication with the use of RTR messages (for long receives) and would not have to take into account the possibility of sender initiation. Sender initiation would only have meaning if our DMA engine did support actual (and not emulated) DMA reads. In that case, the receiver would use the RTS message in order to learn the address of the send buffer and directly read from it. Since this is not possible, RTS messages constitute unnecessary overhead in an implementation that does not support wildcards and MPI_Probe primitives and does not suspend receiver initiation. Additionally, we speculated that the implementation of a receiver initiated protocol would indeed be easier if the MPI Standard did not let eager sends match long receives. In such a scenario, no counters and ticket IDs would be required. As a consequence, MPI_ANY_TAG would not suspend receiver initiation.

Considering the speculations mentioned, we developed an optimistic variant of the writed based Exanet MPI which has the following attributes:

1. It does not support MPI_ANY_SOURCE
2. It assumes that eager send always match with eager receives only
3. It does not support MPI_Probe, MPI_Iprobe
4. It does not use sender initiation for long sends
5. It does not use counters and ticket IDs since there is no risk of eager sends matching long receives
6. Long receives do not insert anything into the Posted MPI Requests since no RTS get issued to match them and they also cannot get matched by Env+D messages *because of point 2). They only need to allocate a notification address in order to determine DMA write completion. Thus, they never acquire the control logic lock.

We evaluate this variant in the Evaluation section with applications that respect the aforementioned attributes.

Chapter 4

Evaluation

The evaluation of our implementation took place on the custom prototype described in Section 2.5. In order to evaluate the write-based implementation of the Exanet MPI, both micro benchmarks and benchmark applications were used. For each experiment, we implemented our own profiler in order to obtain the following information for each problem:

- The percentage of fast receives emerging in an average run of the microbenchmark/application (since our implementation is expected to present most of its benefit in cases when a receive gets posted before the matching send)
- The ratio of computation and communication in the execution time of an application
- The MPI primitives the application uses
- The average size of a message transferred during the execution of the application.

Our profiler's source code is very similar to that of the third write-based variant described in Section 3.6 enchanted with the appropriate functions and variables used to measure time and keep statistics during each MPI call. Specifically, **MPI_Wtime()** is mainly used in order to measure time spent in communication routines while some internal counters keep the number of fast receives during an execution. A fast receive gets counted on the sending side each time a sending primitive finds an already received matching RTR message in its Received MPI Requests as soon as the send gets posted.
After profiling each application's behavior, we proceeded to the actual evaluation of our implementation against the read-based variant. In order to build our microbenchmarks and applications we used MPICH 3.2.1 modified in the way described in **section 3.9** to allow the communicators' ID extraction. Additionally, for the third write-based variant, we performed the modification described in **section 3.8** regarding the type of MPI_Request. For building MPICH, **-O3** optimization was used in all cases. Our implemented variants are built, using **Ofast** optimization, as shared libraries (.so files in Linux) available for dynamic linking. Regarding the read-based implementation, we had an already built so file using the same optimization level. During the execution of each experiment, we delegate all the point-to-point and supported collective functions into the Exanet MPI implementation (either write or read-based) by making use of **LD_PRELOAD** which dynamically links a provided library and intercepts the aforementioned functions. As a result, each time a point-to-point or collective function gets encountered in a program, the Exanet MPI implementation of the function gets executed instead of the vanilla-MPICH one. MPI functions that are not implemented in Exanet MPI, are normally executed through MPICH.

In general our execution command looks like this:

```
LD_PRELOAD=/path/to/exanet_mpi.so:/path/to/collectives_wrapper.so mpirun -np <number of processes> -f
<hostfile containing all available machiines (nodes)> ./executable
```

Unfortunately, certain restrictions and bugs in the DMA implementation and API not corrected until the time of writing, prevented us of from taking advantage of all 512 cores of the prototype. Trying an intra-FPGA DMA write (i.e., write between two processes both residing in the same FPGA) causes undefined behavior in the write-based version so we had to limit our experiments' execution to one MPI process per FPGA, thus utilizing only 128 nodes. Each experiment was carried out at least 15 times. After discarding the worst and best result, we compute the standard deviation present the average results.

For the remainder of this chapter and unless otherwise noted, the following holds:

- **write_based 1.0** refers to the implementation of the first write-based variant described in **Section 3.1** without any memory optimizations and without the change of MPI_Request type described in **Section 3.8.**
- **write_based 2.0** regards the second write-based variant described in 3.3 with the MPI_Request optimization.
- **write_based 3.0** regards the third variant described in 3.6 which also makes use of the memory optimizations and the change of MPI_Request. May be referred simply as **write_base**d.
- **write_based optimistic** refers to the optimistic implementation described in 3.15 which assumes that sends match only with receives that use the same protocol (i.e., eager or long) and that MPI_ANY_SOURCE, MPI_ANY_TAG and MPI_Probe primitives are not used. Memory and MPI_Request optimizations are included

Microbenchmarks

In order to compare the performance of the variants described in Sections **3.1, 3.3, 3.6, 3.13** and the preexisting read-based variant in the point-to-point primitive level we used the OSU Microbenchmarks, version 5.6.2. More specifically, our experiments included the following microbenchmarks:

- **osu_latency - Latency Test**
  This benchmark resembles a simple ping pong test. The sender rank sends a message to the receiver rank and waits for a reply from the receiver. After receiving the initial message, the receiver receives sends back a reply with the same size. Multiple iterations of this process take place and average one-way latency numbers are obtained. Small message sizes perform 1000 timed iterations while 100 iterations are performed for large sizes. MPI functions used: MPI_Send and MPI_Recv

- **osu_bw - Bandwidth Test**
  During bandwidth tests the sender sends out a fixed number of messages to the receiving rank and then waits for a reply from the receiver. The receiver sends the reply only after receiving all these messages. This process is repeated for 1000 iterations for large messages and 100 iterations and the bandwidth is calculated based on the elapsed time and the number of bytes sent by the sender. MPI functions used: MPI_Isend, MPI_Irecv and MPI_Waitall

Next, in Figure 4.1 we show the results of OSU Latency for eager sizes carried out in two FPGAs of the same QFDB. The x axis represents the message size evaluated by the microbenchmark while the y axis represents the reported latency in microseconds. We can notice that in all sizes, write_based 1.0 and write_based 2.0 are equal and have significantly worse latency than the rest of the tested variants. This is attributed to the fact that those two variants demand an RTR or Ack message issuing even for eager receives as well as the locking of the control logic lock and data structure manipulation and search needed for eager sends as described in 3.1. Due to the elimination of the aforementioned sources of latency in combination with memory optimization, write_based 3.0 manages to slightly outperform even the read-based protocol which has the simplest design for eager messages. Write_based optimistic has even lower latency since it omits all counter increments and ticketID assignments for eager sends and receives. It's worth noting that size 0 shows slightly decreased latency since it avoids any copy of the send buffer to the Env+D message in the side of the sender. The same applies for the receiving process.
In Figures 4.2, 4.3 and 4.4 we see the same test for the rest of the OSU message sizes. In Figure 4.2, we can notice that in most sizes, except for 128 and 256 bytes, write_based 3.0 and write_based_optimistic

outperform the rest of the variants. The read-based variant seems to achieve a slightly better (3%) latency than write_based 3.0 with sizes of 128 and 256 bytes. This is most likely attributed to the fact that in the read-based variant, send requests do not need to search Received MPI Requests upon getting posted since receiver initiation is not possible. On the other hand, the write-based implementation requires sends to lock and search the Received MPI Requests in order to check for received RTR messages before beginning the synchronization with the receiver. At this point, we should note that according to our profiling, the percentage of fast receives in the OSU Latency microbenchmark is extremely low (~0,1 %) in both ranks which critically limits the potential benefit of our implementation in this experiment. However, in most the sizes we can clearly see that the write-based implementation still outperforms the read-based one. For instance, for sizes from 512 to 4096 bytes, the write-based 3.0 protocol has up to 8% improved latency in comparison to the read-based protocol as well as the write-based 1.0 and write-based 2.0 protocol. The main reason for this difference in performance is the deployment of memory optimizations described in **section 3.10** which apply in write-based 3.0. We confirmed this fact by trying disabling these optimizations. As a result, the latency of these variants became almost even. In addition, even in the absence of fast receives, the read-based protocol uses one more synchronization message (Ack) which is redundant in the write-based protocol. In intra-QFDB communication (like the one depicted in the figures we are describing), the transmission overhead of that control message is not very apparent. Note that in Figure 4.4, where the biggest sizes are depicted, the read-based implementation shows a significant decrease in performance. We suspect that this performance difference is attributed to the internal mechanism used to detect the completion of a DMA transfer in the receiver's side. Most likely, the read-based implementation uses some sleep routine while waiting for long DMA transfers which renders the receiving process slow in determining the transfer's end and, subsequently, notifying the sender by issuing the Ack control message. It's also worth noting that write_based 2.0 is slightly better than write_based 1.0 in some sizes (e.g., 512, 1024, 2048 bytes of Figure 4.2 etc) due to the fact the former doesn't use Env messages in fast receives. As the message size grows, the difference between write_based 1.0 and write_based 2.0 become less apparent.

We also tried changing the distance of the 2 MPI processes that participate in the run of OSU Latency and rearrange them in the prototype's 3D Torus Topology.

Figure 4.5 shows the latency of small messages as reported by OSU Latency when the two processes reside in different QFDBs between which there is a distance of three network hops. One can notice that the latency of all message sizes is inflated. However, the write-based implementation shows more benefit than in the intra-QFDB scenario. More precisely, in Figure 4.5 we can see that eager messages' latency is almost doubled in comparison to the latency of the intra-QFDB case. The write-based implementation has a slight advantage offering less than 5% performance gain in the eager sizes, mainly due to the use of the memory pool. The next sizes show a bigger interest since we can clearly see that the write-based implementation achieves around 11% lower latency than the read-based implementation. We remind that in the intra-QFDB case there was hardly any difference between the two implementations while the read-based appeared to perform better in 128 bytes. In this case, though, the bigger distance between the two processes signifies the value of omitting the Ack control message of the receiver which we achieve in our implementation regarding **non** fast receives. Note that in this inter QFDB experiment, the percentage of fast receives also remains below 0,5% yet we manage to see considerable performance gain even without them. Figure 4.6 evaluates the larger messages where we see no considerable difference between the intra-QFDB scenario regarding the performance gain of our implementation. This is expected since, in this work we make optimizations regarding the control path and not the data path that is the main source of latency in larger messages. We note that write_based 1.0, write_based 2.0 and write_based optimistic show the same increase in performance gain regarding small messages. Eager messages in write_based 1.0 and write_based 2.0 are an exception because of the additional control messages required in their eager protocol. As a result, their performance drops considerably in eager sizes. Figures 4.7 and 4.8 depict a scenario where the two processes lie on different QFDBs in a distance of 5 hops. Here, the results favor the write-based variant even more as in the range of 64-512 bytes it appears having up to 30% lower latency. In the rest of the sizes, this performance gain gradually gets less observable and the differences resemble those seen in the intra-QFDB case.

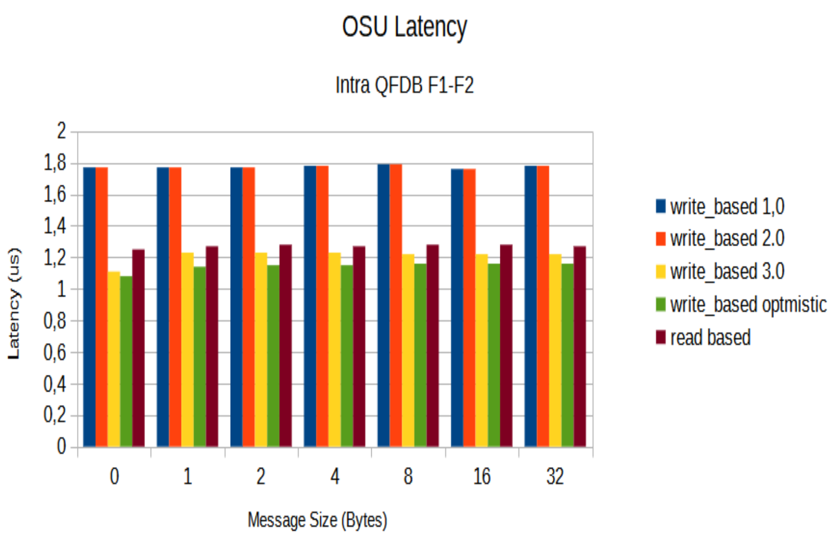


Figure 4.1 OSU Latency: Comparison of all Exanet MPI variants, Eager Messages

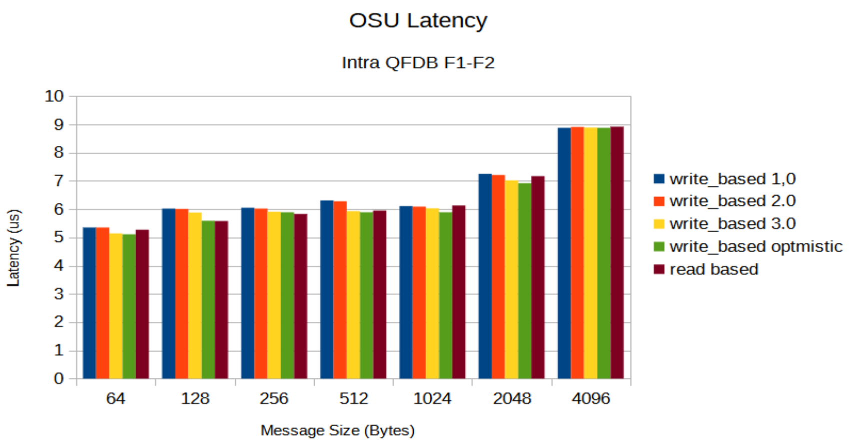


Figure 4.2 OSU Latency: Comparison of all Exanet MPI variants, Short Messages

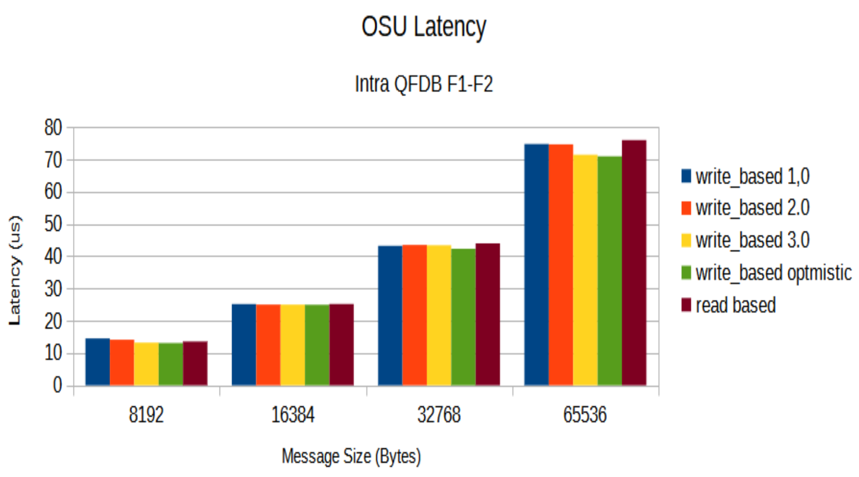


Figure 4.3 OSU Latency: Comparison of all Exanet MPI variants, Medium Messages

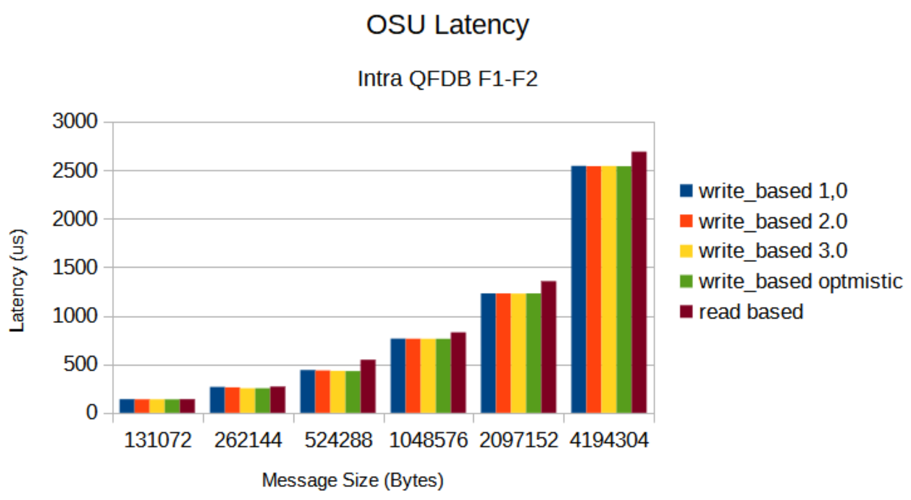


Figure 4.4 OSU Latency: Comparison of all Exanet MPI variants, Large Messages

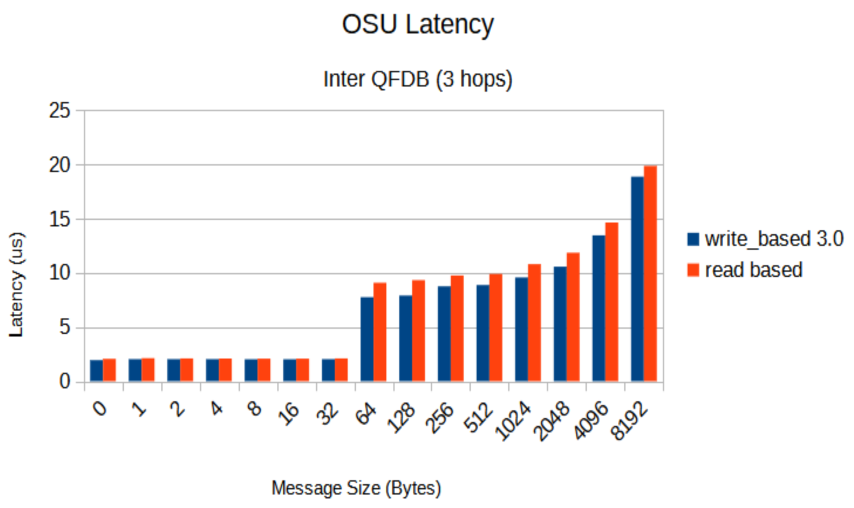


Figure 4.5: OSU Latency: Comparison of small messages' latency between 2 QFDBs in a 3 hops distance

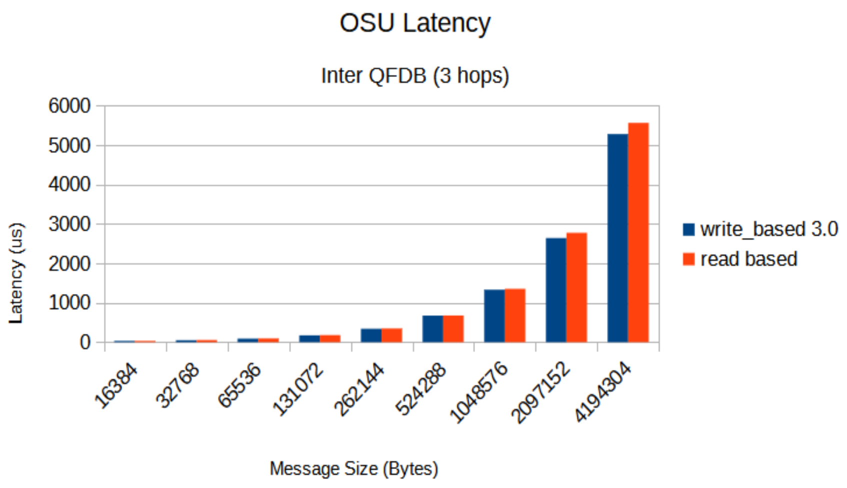


Figure 4.6: OSU Latency: Comparison of big messages' latency between 2 QFDBs in a 3 hops distance

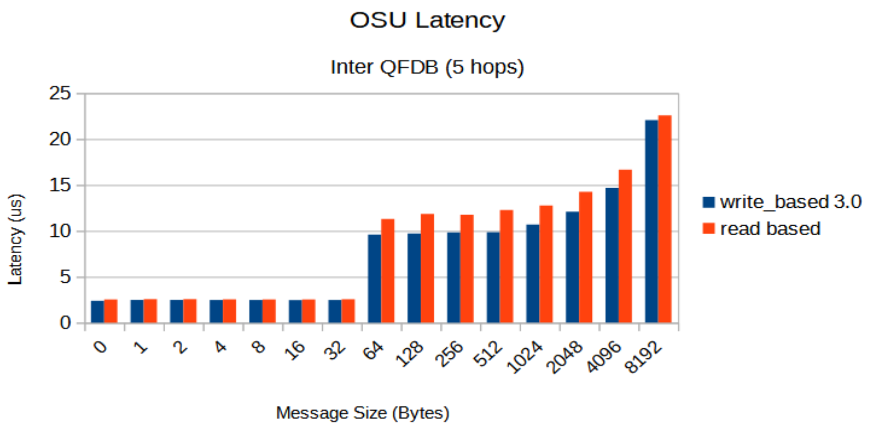


Figure 4.7: OSU Latency: Comparison of small messages' latency between 2 QFDBs in a 5 hops distance

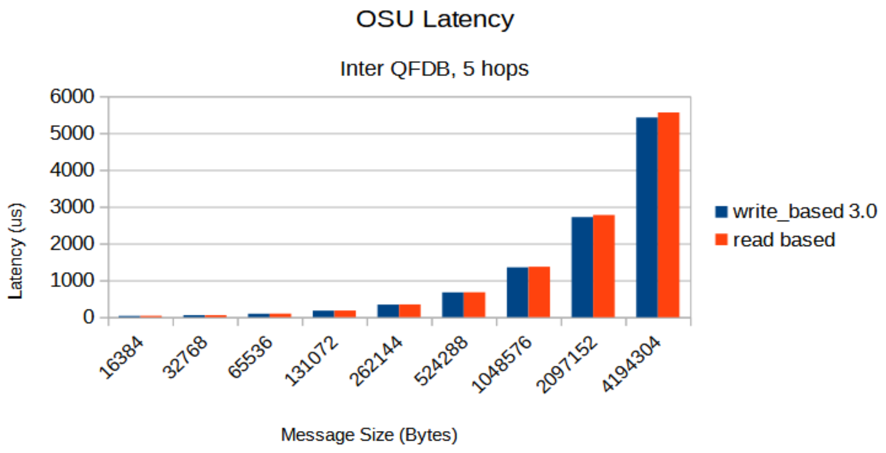


Figure 4.8: OSU Latency: Comparison of big messages' latency between 2 QFDBs in a 5 hops distance

Figures 4.9 and 4.10 illustrate the performance of all the variants of Exanet MPI as reported by the OSU Bandwidth Microbenchmark running in one QFDB. By default, the microbcnchmark uses a window size of 64 pipelined transfers. By observing Figure 4.9 we can see that in eager sizes (0-32 bytes) write_based 1.0 and write_based 2.0 significantly under-perform in comparison to the rest of the variants (up to 50% lower bandwidth). This is expected taking into account the worse eager protocol write_based 1.0, write-based 2.0 use (described in **Section 3.1**). It's also notable that write_based 3.0 and write_based optimistic outperform the read-based protocol in eager messages even if the read-based has a simpler eager protocol. For example, in the size of 32 bytes, the write_based 3.0 variant has 46% higher bandwidth than the read-based variant. Write_based 3.0 may have an improved eager protocol competent with the read-based variant's protocol but it still needs to increment counters and assign ticketIDs which was expected to render it worse than the read_based protocol in that regard. However, since write_based 3.0 makes use of memory optimizations described in **Section 3.10**, it manages to avoid malloc() and free() invocations which is the reason for outperforming the read-based variant in eager messages. Thus, we see that memory optimization plays a very crucial role in the implementation's bandwidth. The same applies for write_based optimistic(i.e., due to memory optimizations, it manages to outperform the read-based protocol as well). One can notice a sudden drop of bandwidth for all variants in the size of 64 bytes. This happens because the eager protocol cannot be applied for messages bigger than 40 bytes so the implementations switch to the long rendezvous protocol which utilizes the DMA engine. In general, for all subsequent sizes in Figures 4.9 and 4.10 we can observe that the write_based 3.0 and write_based optimistic outperform the rest of the variants with the exception of 256 bytes in which the read_based variant performs better. The same pattern has been noticed in the OSU latency microbenchmark between 2 processes in the same QFDB. Write_based 1.0 and write_based 2.0 have slightly worse performance than write_based 3.0 due to the lack of memory optimizations, since their long protocols are almost identical the sole difference being that write-based 1.0's includes the Env message. Like the OSU Latency microbenchmark, in big sizes (524288-4194304 bytes) the read-based variant has the worst performance between all variants. This intensifies our assumption that the read-based implementation is problematic in detecting the end of a big transfer by the receiver and notifying the sender. We also note that OSU Bandwidth also has about the same percentage of fast receives as OSU Latency (~0,1%). Changing the distance of the two MPI process in the prototype's topology did not resulted to any significant difference, unlike the OSU Latency test. The bandwidth of all variants got worse but the performance difference between the variants remained in the levels of the intra-QFDB scenario. This fact shows us that the distance between the nodes gives our implementation only a latency advantage. That latency gets masked out in the presence of pipelined sends.

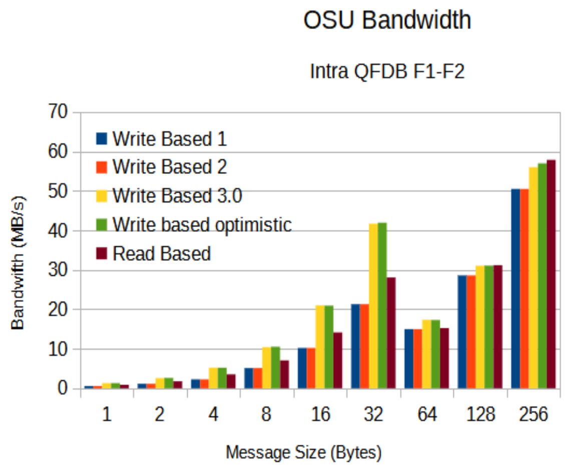


Figure 4.9: Comparison of all Exanet MPI variants' bandwidth, small messages

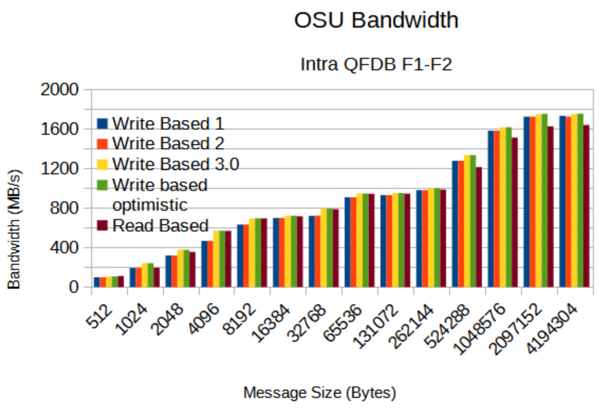


Figure 4.10: Comparison of all Exanet MPI variants' bandwidth, big messages

As we noticed in the 2 previous microbenchmarks, the occurrence of fast receives was extremely rare between 2 MPI processes exchanging messages. In order to be able to evaluate the optimizing potential of the fast receive case in our implementation, we designed our own microbenchmark which forces a fast receive and subsequently evaluates the latency of the MPI_Send primitive. Specifically, our microbenchmark requires two MPI processes, a sender and a receiver. In each iteration, the receiver posts a non blocking receive (MPI_Irecv) and immediately after, it invokes MPI_Barrier. The sender starts its execution by calling MPI_Barrier and subsequently, it invokes an MPI_Send that matches the MPI_Irecv posted by the receiver. Calls to MPI_Wtime are placed before and after the MPI_Send in order to measure its latency. Since the sender waited on MPI_Barrier first, it is guaranteed that the receiver has issued an RTR message which has been received by the sender. This applies only in the write-based implementation since the read-based one does not support receiver initiation. In this way, we force the occurrence of a fast receive. The described process gets iterated 1000 times per run for any size. Figures 4.11, 4.12 and 4.13 present the results of the benchmark in different topologies. Write_based 3.0 and the Read-based variant are evaluated in message sizes in the range of 64-8192 bytes. Shorter messages utilize the eager protocol while longer messages show no significant difference in fast receives between the implementations due to the big cost of the DMA transfer which masks any potential benefit.

Figure 4.11 evaluates the write-based and read-based implementations in forced fast receive scenarios inside the same QFDB. The x axis represents the Message Size while the y axis shows the latency of MPI_Send for the respective message size. We see that the write-based variant outperforms the read-based variant by 25% in the 64-1024 bytes range. Gradually, the difference becomes smaller as the message size grows since the DMA write cost also gets bigger. At 8192 bytes, the write-based implementation still outperforms the read-based one by 10% while in larger sizes there is no apparent difference between a fast and a non fast receive. As already mentioned through this thesis, the read-based implementation cannot exploit scenarios where the receive gets posted earlier since receiver initiation is not supported. A similar performance pattern appears in Figures 4.12 and 4.13 which regard inter QFDB communication with 3 and 5 intermediate hops respectively. In both figures, read_based shows initially a 50% higher latency than write_based. This difference gets less observable faster than in the previous figure since in inter QFDB cases the data path of the DMA also gets significantly worse for both implementations. We remind that in a fast receive, the write-based long protocol has 2 control messages less than the read-based protocol. In the write-based variant, the sender immediately performs a DMA write after receiving the RTR control message from the receiver. On the other hand, the read-based implementation performs a full synchronization (issuing of RTS, CTS and finally Ack) even in the case of an early posted receive. This experiment shows the potential of receiver initiation as well as the importance and impact of the percentage of fast receives in a program's performance.

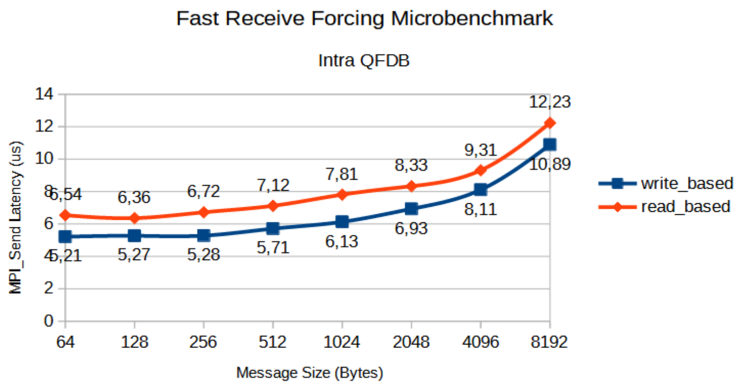


Figure 4.11: Comparison of Read and Write Based MPI in fast receives, intra QFDB

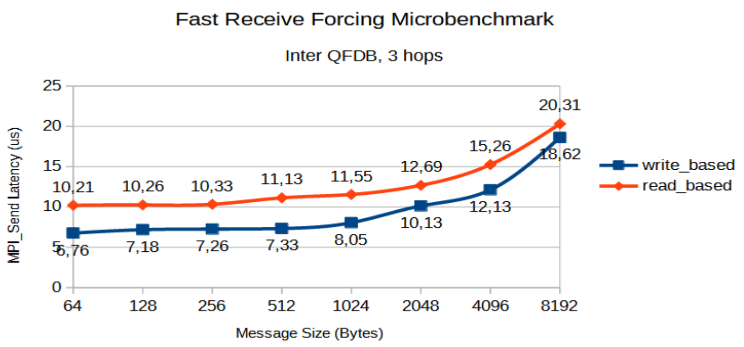


Figure 4.12: Comparison of Read and Write Based MPI in fast receives, 3 hops distance

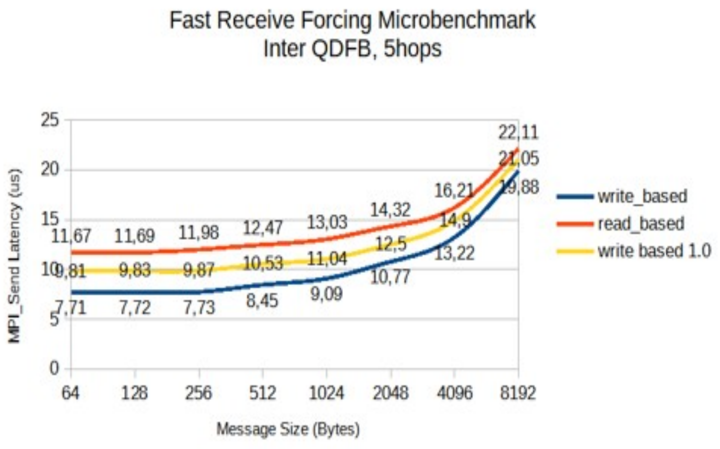


Figure 4.13: Comparison of Read and Write Based MPI in fast receives, 5 hops distance

After comparing all developed write-based variants with the read-based MPI variant, we continue evaluating only the write_based 3.0 and write_based optimistic against the read-based implementation. Our next experiments include the following OSU Collective Benchmarks:

- osu_allreduce - MPI_Allreduce Latency Test
- osu_barrier - MPI_Barrier Latency Test
- osu_bcast - MPI_Bcast Latency Test
- osu_reduce - MPI_Reduce Latency Test
- osu_scatter - MPI_Scatter Latency Test

Each one of these benchmarks makes multiple iterations of the respective collective function for a specific message size and reports the average latency for each size. For sizes up to 8192 bytes 1000 timed iterations are performed while for bigger sizes the number of timed iterations is 100. Recall that all of the implemented collective functions are finally delegated to point-to-point MPI calls. Thus, by evaluating the collective functions, in reality we evaluate the same primitives evaluated in the previous experiments. However, different collective algorithms offer different combinations of message sizes as well as varying fast receives percentage which can result to different performance patterns. For this reason, we chose the aforementioned collective functions as indicative among all collective functions that are available.
In Figures 4.14-4.23 the results of OSU_Broadcast are illustrated for a scale of 16, 64 and 128 ranks. For all the message sizes we use the broadcast binomial tree algorithm, which has a complexity of *$O(log_2N)$*, where N is the number of processes. As one can notice, in all of the 3 cluster configurations, the write-based implementation has a significant advantage over the read-based one especially in messages of small size (64-4096 bytes). For instance, in Figure 4.14 the write-based implementation has a performance gain of 25-30% in each size in the 64-1024 bytes size range. The same pattern can be noticed in Figures 4.18 and 4.21 for the same sizes. This result is attributed to the relatively big percentage of fast receives that appear during the execution which offers a serious benefit for our implementation since the synchronization needed by MPI_Send is minimal in that case. Specifically, for all 3 numbers of ranks (16, 64, 128) the percentage of fast receives was about 40% while there were a few ranks that performed no fast receives at all. The performance gain due to fast receives gets gradually less observable as the message size grows (Figures 4.15, 4.17, 4.19, 4.20, 4.22, 4.23). Note that even eager messages have better latency in the write-based implementation due to the memory optimizations applied in the write-based variants. In Figure 4.13 we can see an improvement of 10% in the latency of eager sizes (0-32 bytes). We should also note the fact that the optimistic version of the write-based

implementation outperforms write-based 3.0 since it omits counter incrementing and ticket ID assignment. In addition, it allows an optimization in its data structures since it does not havc to support MPI_ANY_SOURCE. Moreover, since no sender initiation exists in the long protocol, the receiver doesn't need to insert long receive requests to the Posted MPI Requests which eliminates some overhead. These facts render the optimistic variant of the write-based implementation about 5% more efficient than the standard write-based MPI in terms of latency. This difference can more easily be observed in the 64-512 bytes size range.

Figure 4.16 evaluates the small message sizes using a **shuffled hostfile**. Normally, our hostfile includes all the available nodes in an optimal order. For instance, the four first nodes are the first 4 FPGAs of the first QFDB, the next 4 nodes are the 4 FPGAs of the second QFDB in the 3D Torus and so on. MPI assigns MPI Ranks in the order it reads the nodes in the hostfile. This facts makes processes with neighboring MPI Ranks also be neighbors in the topology of our prototype. However, when the hostfile gets randomly shuffled, such a guarantee does not exist. We see that with a shuffled hostfile we trigger more inter QFDB transfers and thus the write-based implementation outperforms the read-based variant by up to 50% in the 64-1024 bytes range. The difference between the write and read-based implementation also appears inflated up to the size of 8192 bytes. Note that the shuffling of the hostfile did not affect the percentage of fast receives at all in this experiment. We observe the exact same pattern when we shuffle the hostfile in runs that use 64 and 128 ranks as well.

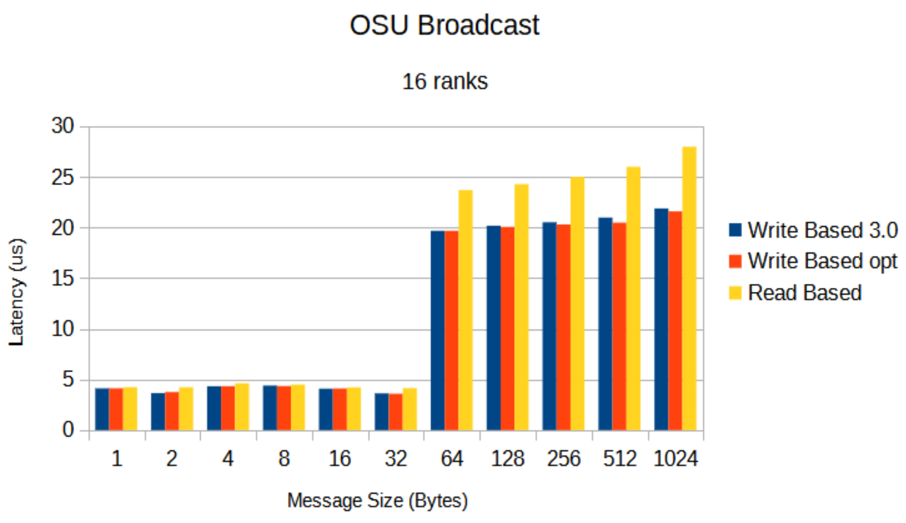


Figure 4.14: OSU_Broadcast latency comparison, small messages with 16 ranks

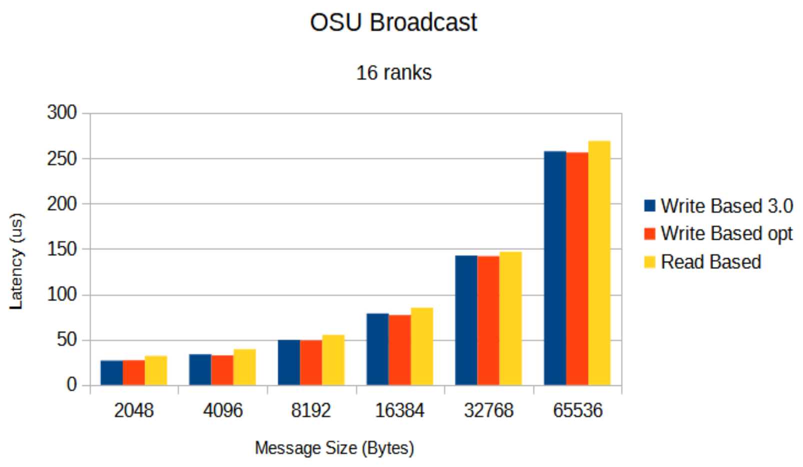


Figure 4.15: OSU_Broadcast latency comparison, medium messages with 16 ranks

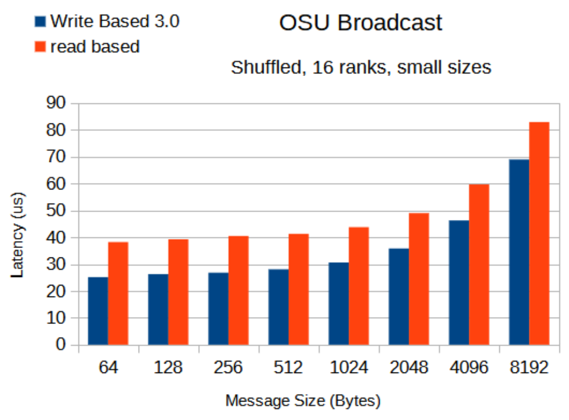


Figure 4.16: OSU_Broadcast latency comparison, small messages with 16 ranks with shuffled hostfile

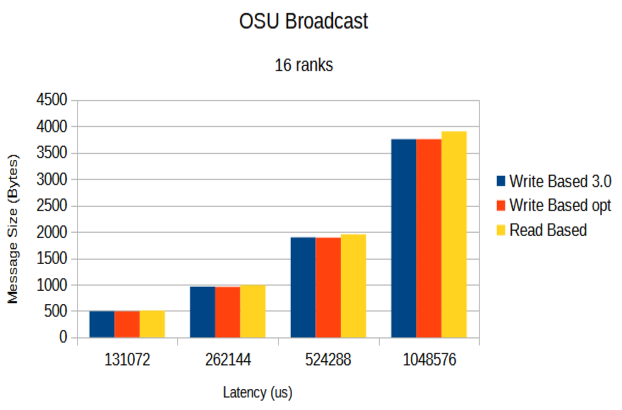


Figure 4.17: OSU_Broadcast latency comparison, big messages with 16 ranks

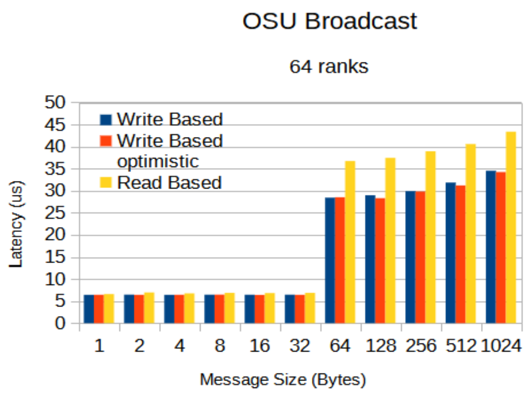


Figure 4.18: OSU_Broadcast latency comparison, small messages with 64 ranks

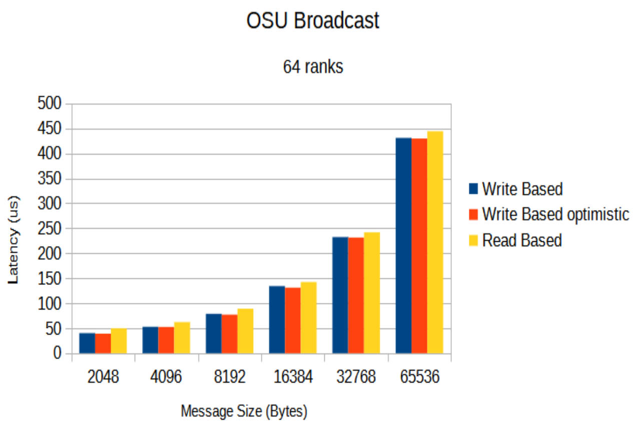


Figure 4.19: OSU_Broadcast latency comparison, medium messages with 64 ranks

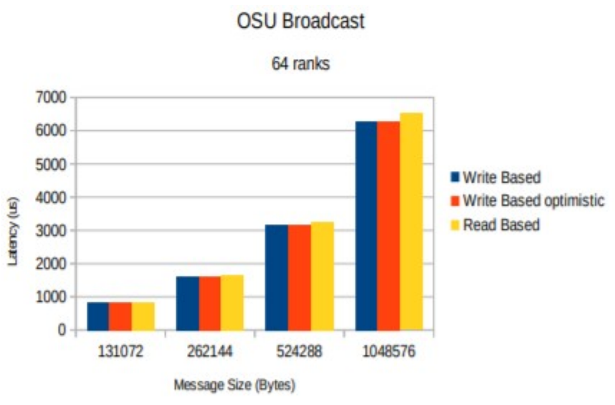


Figure 4.20: OSU_Broadcast latency comparison, big messages with 64 ranks

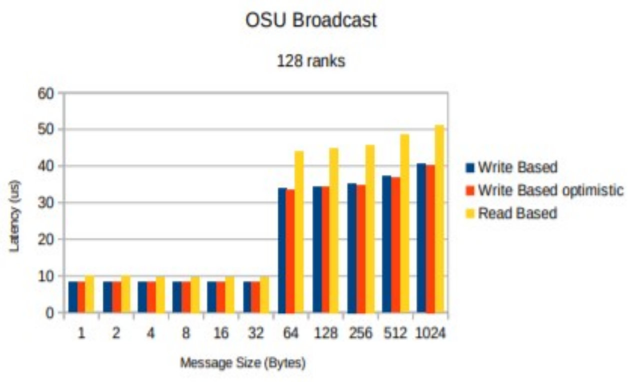


Figure 4.21: OSU_Broadcast latency comparison , small messages, 128 ranks

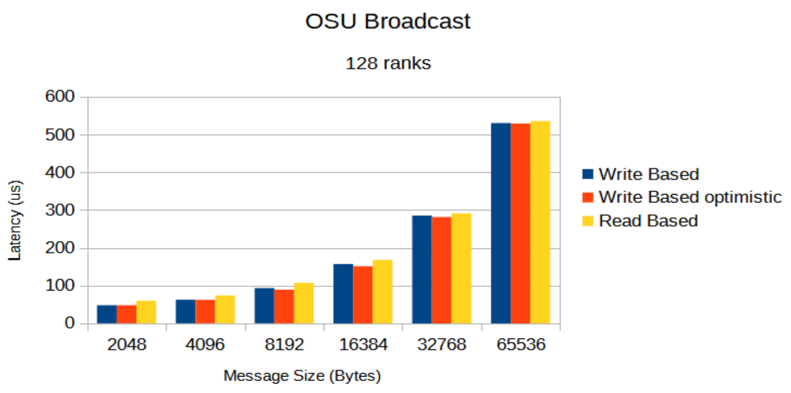


Figure 4.22: OSU_Broadcast latency comparison, small messages with 128 ranks

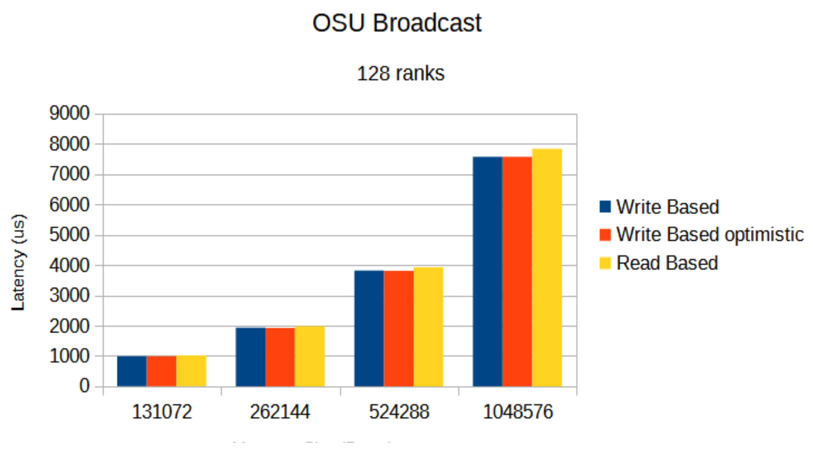


Figure 4.23: OSU_Broadcast latency comparison, big messages with 128 ranks

Figures 4.24, 4.25, 4.26 compare the performance of the write-based and read-based variants of Exanet MPI using the OSU Reduce Microbenchmark with 16 processes. We notice that in eager sizes the write-based implementations have a 10% improved latency compared to the read-based variant. We attribute this performance gain to the memory optimizations described in **Section 3.10** since the difference ceases to exist if we do not make use of them. One can also observe that an improvement of 7-10% appears in medium sizes (64-8192 bytes) mainly due to the presence of a considerable amount of fast receives which favors our implementation. However, the latency reduction is less than that observed using the OSU Broadcast microbenchmark. This happens because here the percentage of fast receives is lower (20-25% depending on the algorithm used) but also due to the fact that MPI_Reduce functions also include the computation emerging from the specified reduce operation (MPI_SUM in the OSU microbenchmarks) as well as the allocation and deallocation of temporary buffers inside the implementation of the primitive. For sizes less than 2048 bytes, a classic binomial tree reduce algorithm is used while for larger sizes we deploy the Rabenseifner's Reduce algorithm also used in MPICH as it is optimal for large messages. Rabenseifner's algorithm makes significant use of temporary buffers allocated and deallocated in each iteration as well as memory copies. Figures 4.25 and 4.26 evaluate the MPI implementations using larger messages and thus forcing the use of the Rabenseifner's algorithm. We see that the write-based implementation has a slight advantage offering around 6% less latency in most sizes when Rabenseifner's algorithm is used. The three last sizes depicted in Figure 4.26 show the read-based implementation performing again significantly worse in big sizes and having 10% more latency than the write-based one. Note that the optimistic version of the write-based variant does not show considerable improvement in comparison to the standard write-based version except for the medium message sizes (64-512 bytes). Similar performance patterns are preserved when we use 64 and 128 ranks for the same microbenchmark as we see in Figures 4.27, 4.28, 4.29, 4.30, 4.31 and 4.32. However, we observed that the percentage of fast receives gets reduced slightly with the increase of the number of ranks. Precisely, in the 64 and 128 ranks runs, the percentage of fast receives usually drops to 19-20% of total receives. However, this does not infer noticeable reduction in the difference between the implementations' results. We can see in Figures 4.27 and 4.31 that the write-based variant has again below 10% lower latency for sizes between 64 and 8192 bytes. This benefit gradually fades as the size grows. In Figures 4.29and 4.32, we see that at the last two large sizes a difference reappears in favor of our implementation. As already mentioned, this difference is most likely attributed to other implementation details in the read-based variant's code since it cannot be explained with the optimizations of the control path, which we contribute in this work.

Figure 4.33 shows the evaluation of the write and read-based implementations with OSU Reduce(128 nodes) using a randomly shuffled hostfile for the message sizes of 64-8192 bytes. We see that the difference between the variants' latency gets inflated reaching up to 20% (in the 512 bytes message size). We also report that the shuffling of the hostfile slightly reduced further the percentage of fast receives but at the same time forced more inter QFDB communications. Shuffling the hostfile for 16 and 64 ranks produces a very similar effect.

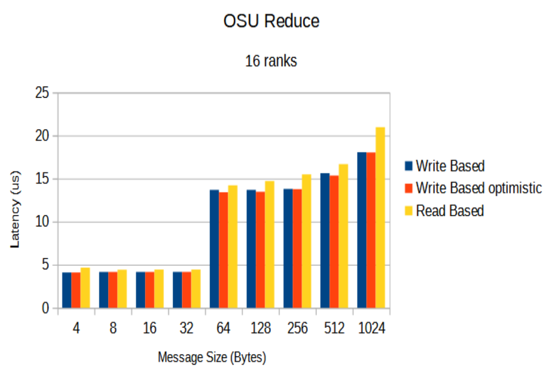


Figure 4.24: OSU Reduce: latency comparison, small messages with 16 ranks

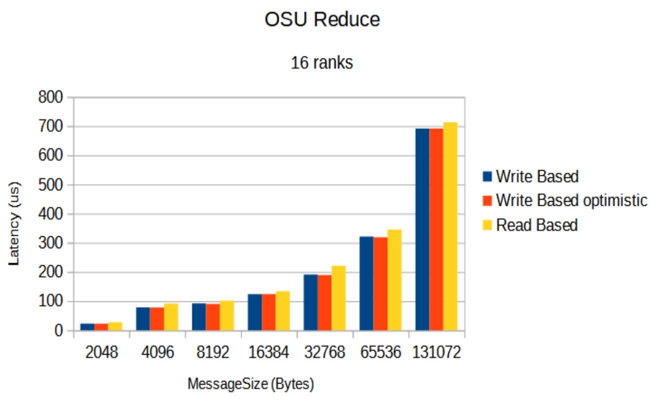


Figure 4.25: OSU Reduce: latency comparison, medium messages with 16 ranks

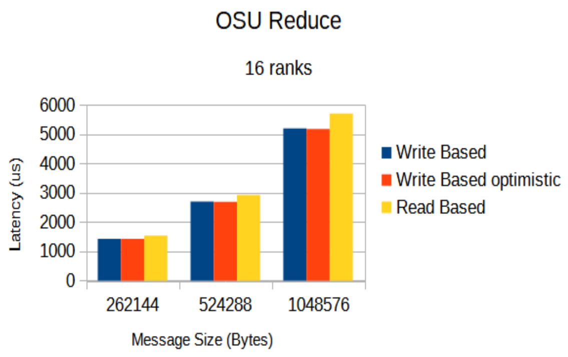


Figure 4.26: OSU Reduce: latency comparison, big messages with 16 ranks

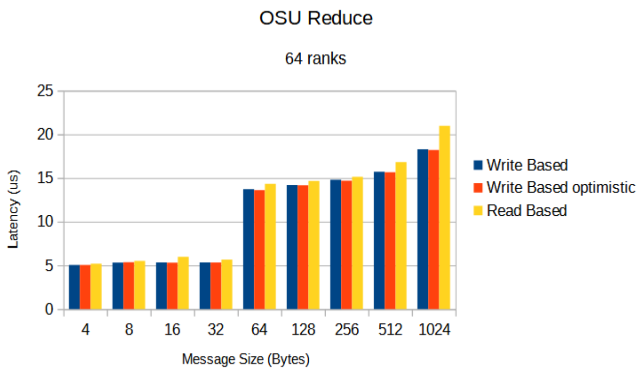


Figure 4.27: OSU Reduce: latency comparison, small messages with 64 ranks

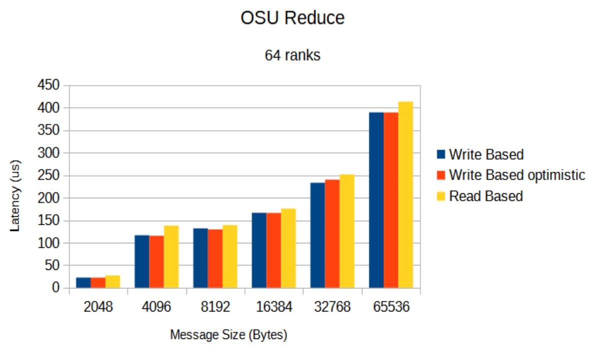


Figure 4.28: OSU Reduce: latency comparison, medium messages with 64 ranks

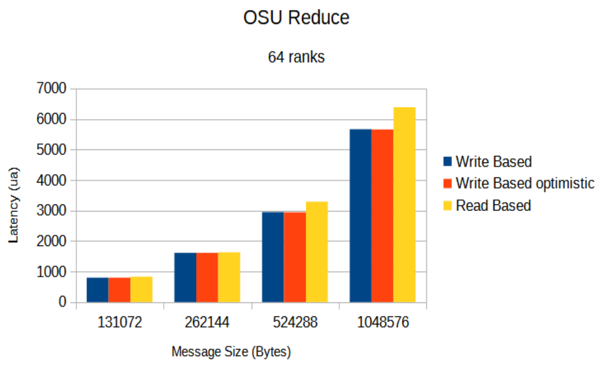


Figure 4.29: OSU Reduce: latency comparison, big messages with 64 ranks

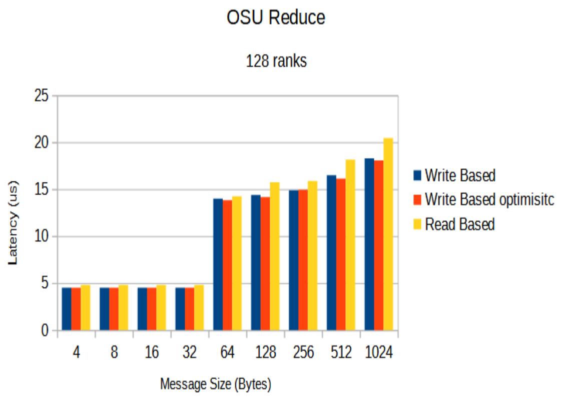


Figure 4.30: OSU Reduce: latency comparison, small messages with 128 ranks

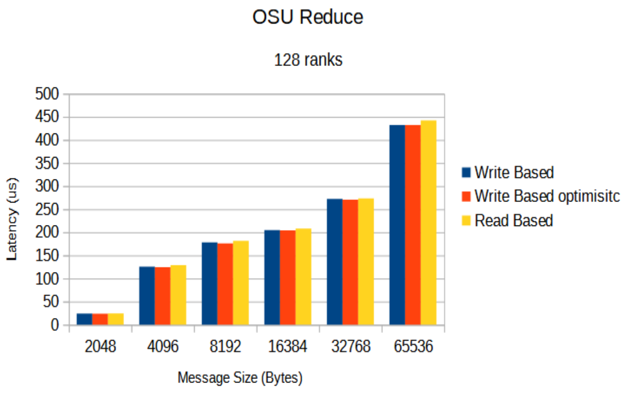


Figure 4.31: OSU Reduce: latency comparison, medium messages with 128 ranks

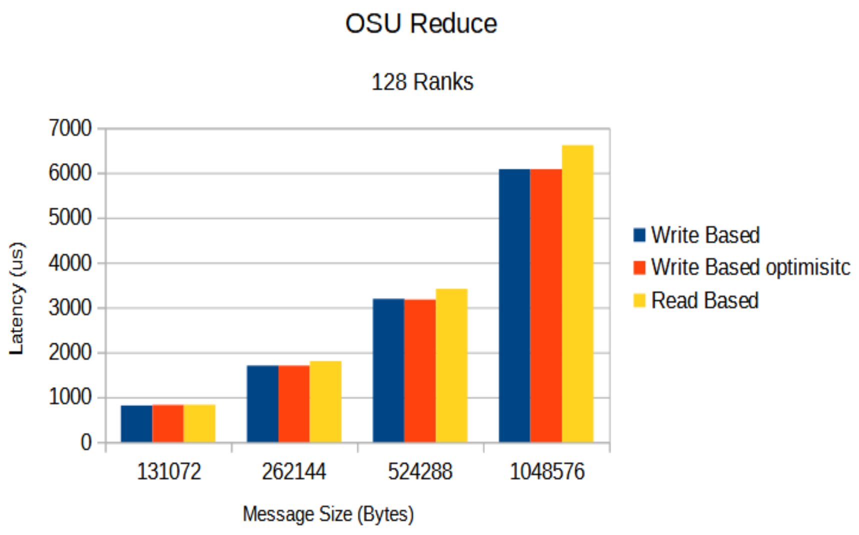


Figure 4.32: OSU Reduce: latency comparison, big messages with 128 ranks

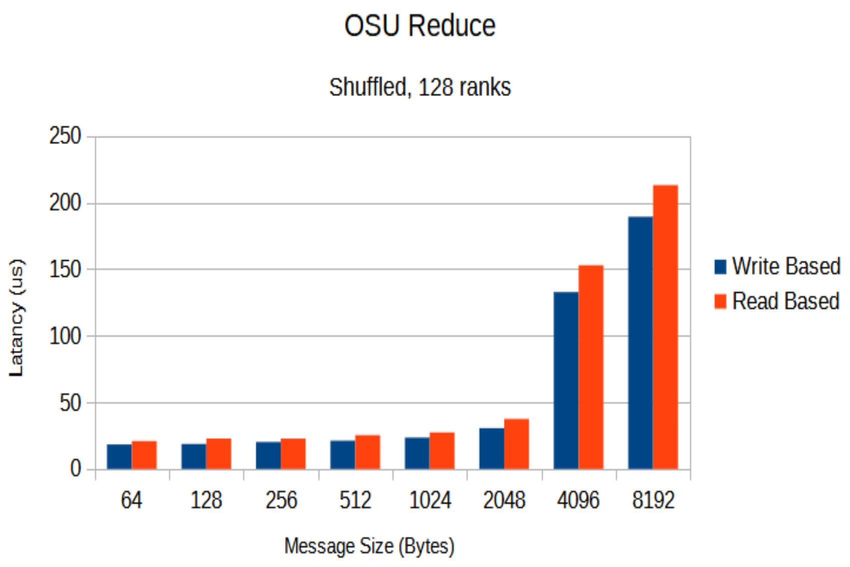


Figure 4.33: OSU Reduce: latency comparison, small messages with 128 ranks in a shuffled hostfile

Figure 4.34 shows the evaluation of the OSU_Barrier microbenchmark. For the implementation of MPI_Barrier we use a simple dissemination algorithm which utilizes only eager messages. However, we can see that a substantial difference exists between the write-based and read-based variants. This difference is attributed to the memory optimizations described in **Section 3.10**. We avoid the use of malloc and free functions for the allocation and deallocation of received send requests and we use stack allocation (instead of heap) for posting receive requests into the Posted MPI Requests.

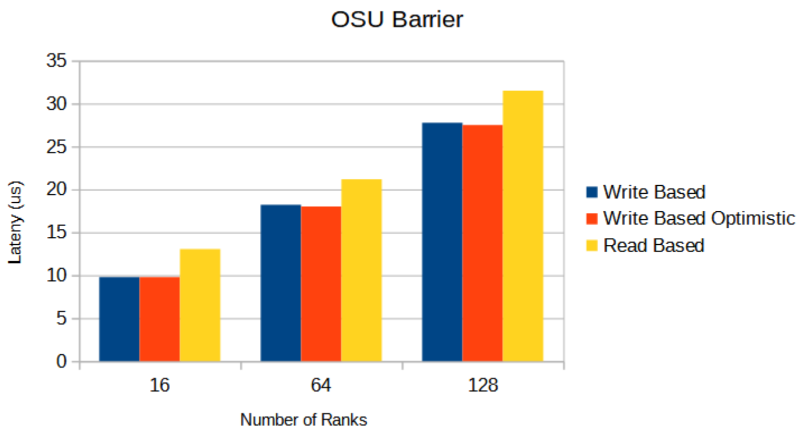


Figure 4.34: OSU Barrier: latency comparison

Figures 4.35-4.43 evaluate the Exanet MPI variants using the Osu_Scatter microbenchmark. MPI_Scatter [22] is implemented using a binomial tree algorithm, similar to Broadcast but with the appropriate data reordering the primitive requires. In Figure 4.35 we see that the read-based variant has a 20%-36% higher latency than the write-based variants in sizes 64-1024 bytes. This fact can be explained by the percentage of fast receives the microbenchmark achieves (~45% of total receives in some ranks). Eager messages present also a noticeable improvement in the write-based version like in the rest of the microbenchmarks. In Figures 4.38 and 4.41 which regard the same size range, we notice that the write-based variant still outperforms the read-based one but the difference between them is smaller. In our profiling we figured that in the runs using 64 and 128 ranks, some of the ranks had no fast receives at all while the rest of them maintained a decent percentage. In Figures 4.35, 4.38, 4.41 we observe that the three variants have similar latencies with the difference of the write-based variant from the read-based one never surpassing 7% in percentage. In Figures 4.36, 4.39, 4.42 we see that the read-based implementation performs significantly worse in big message sizes without that being explainable by our optimizations. We should also notice that the optimistic variant of the write-based implementation offers a slight improvement in the latency of MPI_Scatter hardly observable in the figures. At this point, we should mention that shuffling the hostfile, sometimes caused some undefined behavior in both the read and write-based implementation which resulted in the constant need for resetting the prototype nodes. An explanation for that might be some undocumented hardware bug which does not allow DMA transfers between specific FPGAs that happen to never occur without a sorted hostfile. We are confident that it is not an issue related to our code since it affects the read-based variant as well. This incident is not that frequent but it critically limits our ability to perform the necessary number of tests to present a reliable result. Consequently, we avoid evaluating with shuffled hostfiles in the following experiments unless otherwise noted.

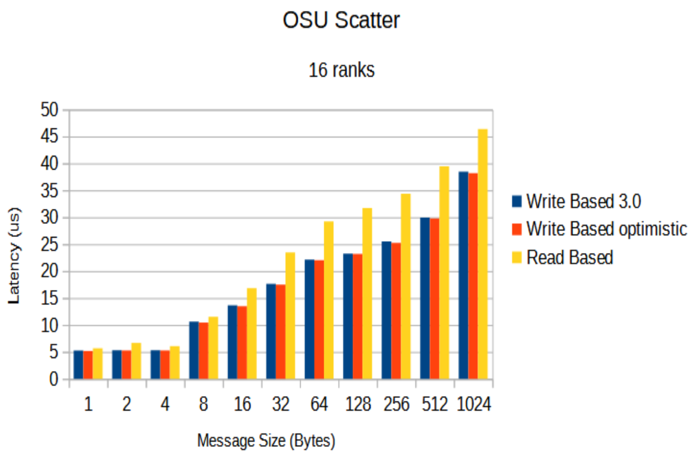


Figure 4.35: OSU_Scatter latency comparison, small messages with 16 ranks

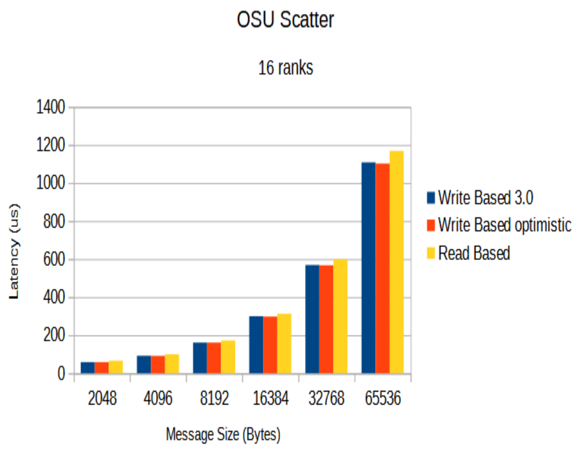


Figure 4.36: OSU_Scatter latency comparison, medium messages with 16 ranks

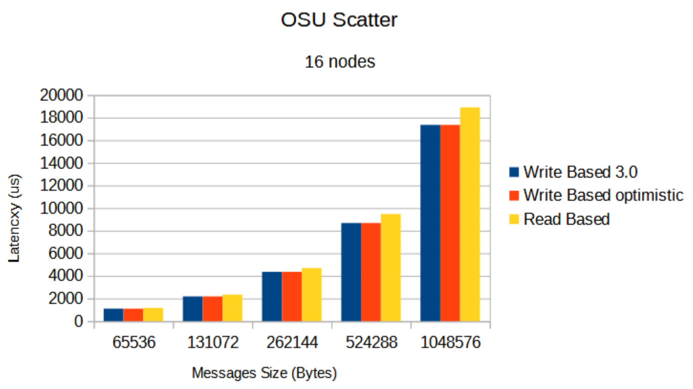


Figure 4.37: OSU_Scatter latency comparison, medium messages with 16 ranks

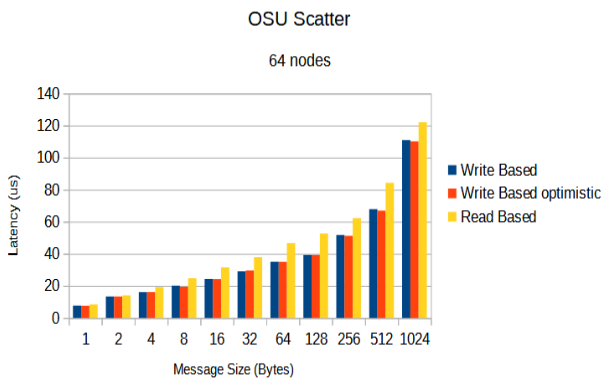


Figure 4.38: OSU_Scatter latency comparison, small messages with 64 ranks

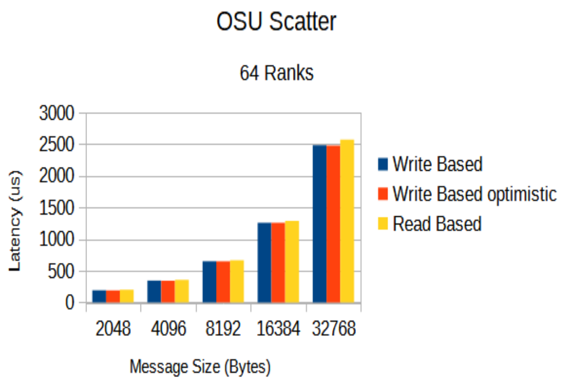


Figure 4.39: OSU_Scatter latency comparison, medium messages with 64 ranks

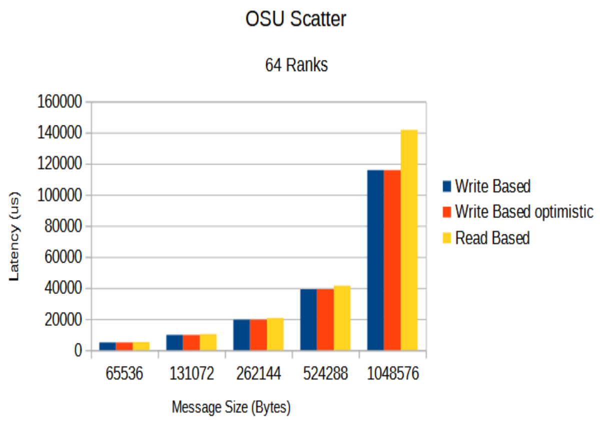


Figure 4.40: OSU_Scatter latency comparison, big messages with 64 ranks

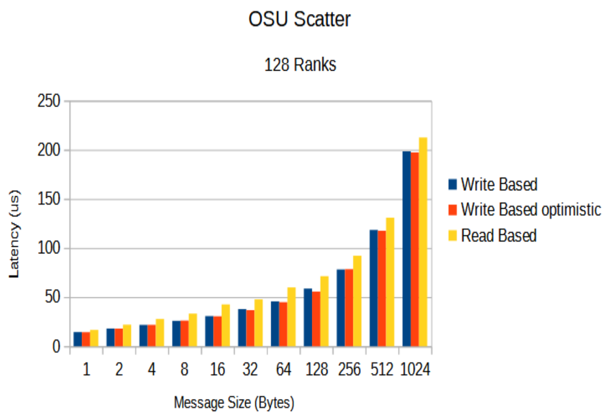


Figure 4.41: OSU_Scatter latency comparison, small messaes with 128 ranks

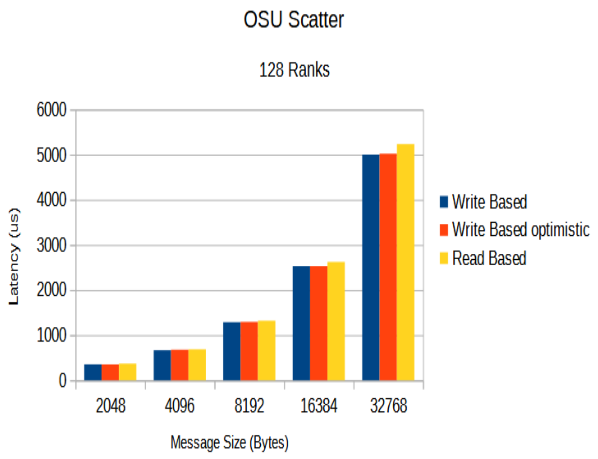


Figure 4.42: OSU_Scatter latency comparison, medium messages with 128 ranks

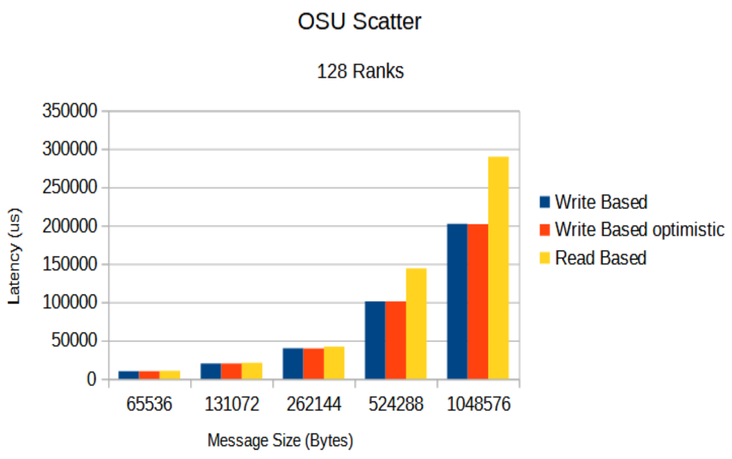


Figure 4.43: OSU_Scatter latency comparison, big messages with 128 ranks

Figures 4.44-4.52 present the evaluation of the MPI_Allreduce primitive using the osu_allreduce microbenchmark. Osu_Allreduce offers a fair percentage of fast receives (25-30% of total receives) in all 3 cluster configurations (16, 64, 128 ranks). As we can see in Figures 4.44, 4.47, 4.50, the read-based Exanet MPI generally achieves an up to 20% higher latency than the write-based variant in small messages (64-8192 bytes) in any cluster configuration. It is worth noting that even medium message sizes (16384-65536 bytes) show a significant performance improvement in favor of the write-based implementation. This difference gets signified especially in the 64 and 128 ranks executions in spite of the fast receive percentage not varying among cluster configurations. We explain this fact by noting that MPI_Allreduce forces the communication of any process with most of others at some point in its execution which triggers inter QFDB and inter-mezzanine transfers without shuffling the hostfile. In other words, MPI_Allreduce does not limit the processes to communicate only with processes that carry nearby MPI_Ranks but also includes communication between distant nodes. This type of communication benefits our implementation as shown in the previous paragraphs. For sizes up to 2048 bytes, we use a recursive doubling algorithm while for larger message sizes another version of the Rabenseifner's Algorithm is used. In Figures 4.46, 4.49, 4.52, we see that our implementation keeps outperforming the read-based variant in big messages. We should also mention that the optimistic variant of the write-based MPI keeps very slightly outperforming write_based 3.0 in most small messages.

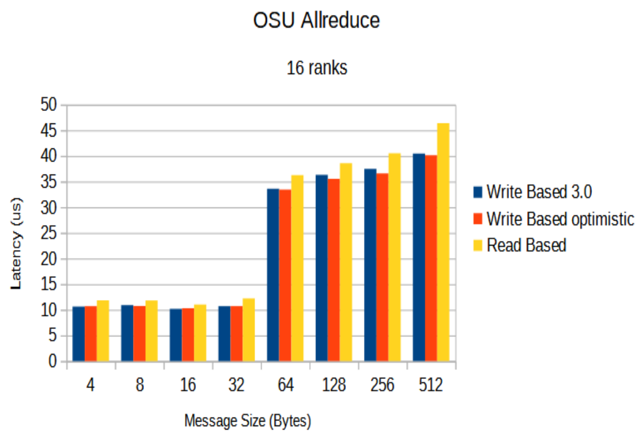


Figure 4.44: OSU_Allreduce latency comparison, small messages with 16 ranks

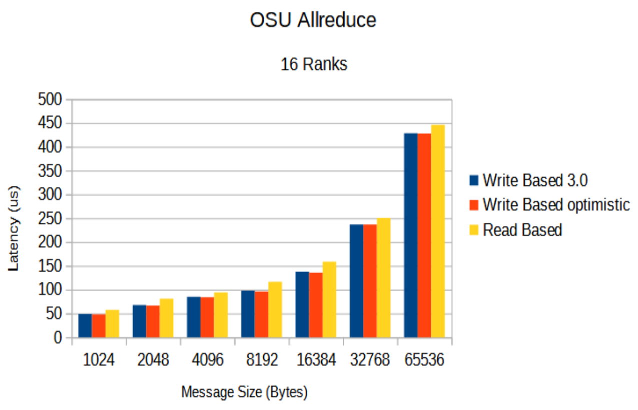


Figure 4.45: OSU_Allreduce latency comparison, medium messages with 16 ranks

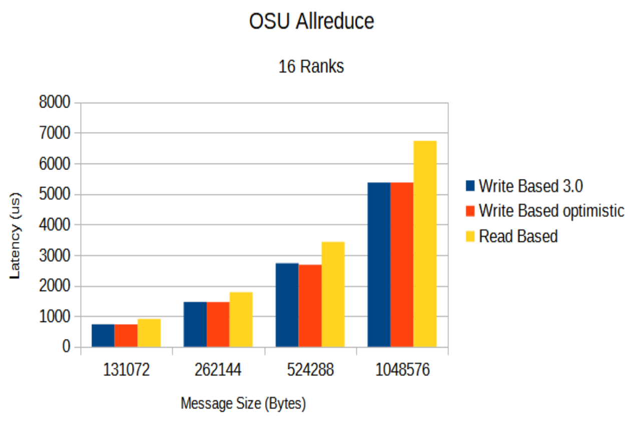


Figure 4.46: OSU_Allreduce latency comparison, big messages with 16 ranks

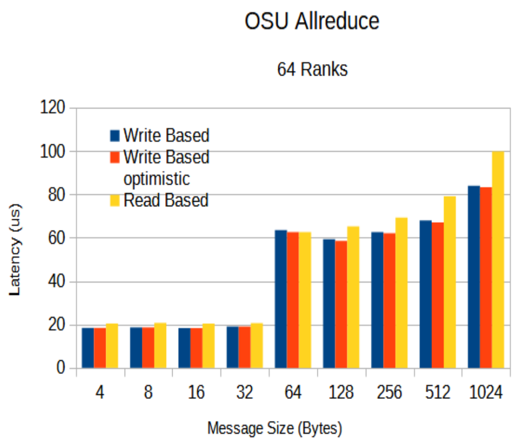


Figure 4.47: OSU_Allreduce latency comparison, small messages with 64 ranks

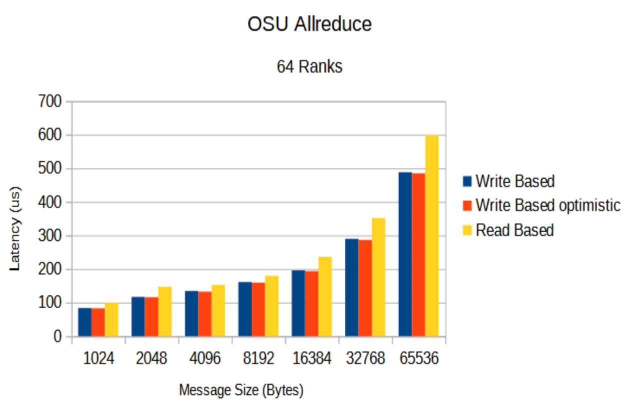


Figure 4.48: OSU_Allreduce latency comparison, medium messages with 64 ranks

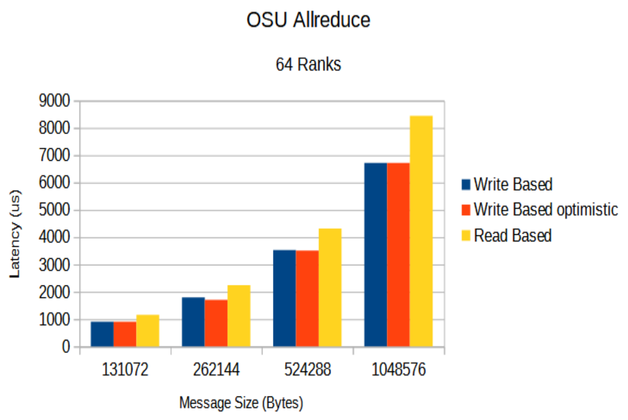


Figure 4.49: OSU_Allreduce latency comparison, big messages with 64 ranks

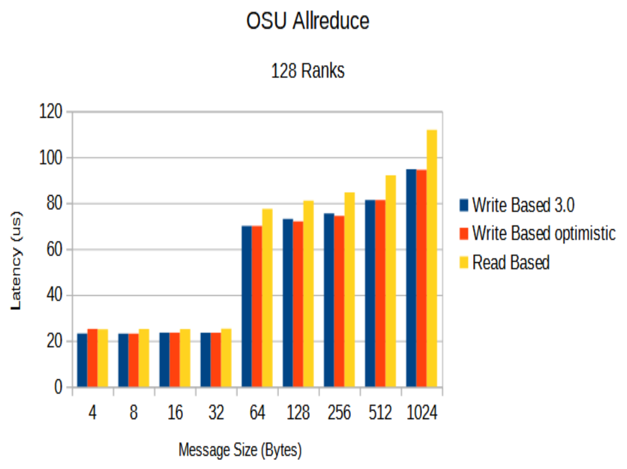


Figure 4.50: OSU_Allreduce latency comparison, small messages with 128 ranks

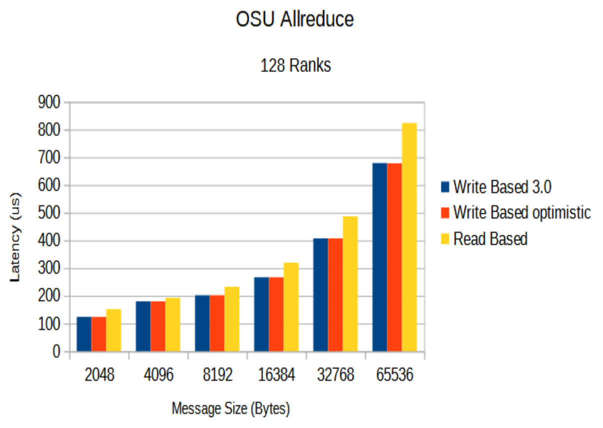


Figure 4.51: OSU_Allreduce latency comparison, medium messages with 128 ranks

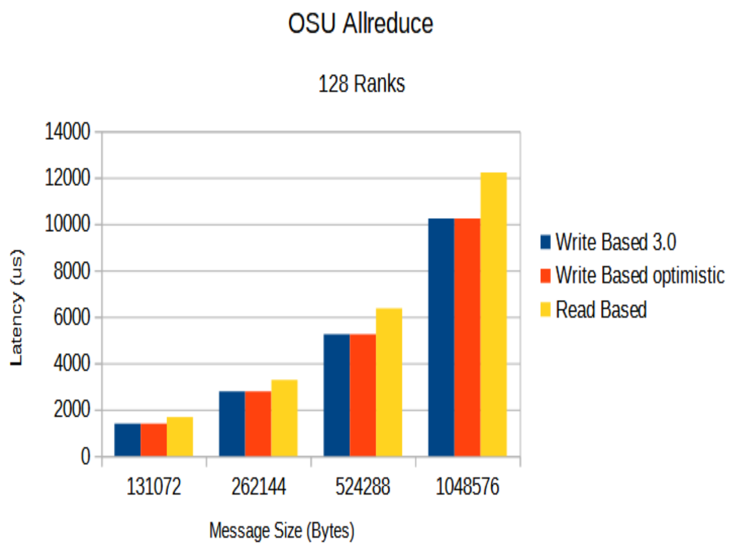


Figure 4.52: OSU_Allreduce latency comparison, big messages with 128 ranks

NAS Parallel Benchmarks

For the evaluation of our implementation using real scientific applications we used, among other benchmarks, some problems included in the NAS Parallel Benchmarks [26] (NPB) package. NAS Parallel Benchmarks are benchmarks derived from computational fluid dynamics (CFD) applications. They are developed by the NASA Advanced Supercomputing Division and are widely used to evaluate the performance of parallel supercomputers. NPB are not limited to MPI communication but they contain versions of the problems which use other programming models like OpenMP, serial etc. Out of the MPI problems constituting the NPB-MPI package, we chose the following benchmarks based on communication primitives used as well as their computation and communication ratio:

LU (Lower-Upper Gauss-Seidel solver, implemented in Fortran)

MG (Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive, implemented in Fortran)

IS (Integer Sort, random memory access, implemented in C)

SP (Scalar Penta-diagonal solver implemented in Fortran)

Each of the aforementioned problems exists in different sizes called *classes.* For a specific problem we choose a certain class which offers the best communication and computation ratio without being too small in duration. In general, increasing the size of a NAS benchmark makes it more computation intensive while communication time percentage remains in the same levels among different classes. The available classes for each benchmark are **S**-Small, **W**- 90's Workstation Size, **A**, **B**, **C** (medium sizes) and **D, E, F** (large sizes).

Figure 4.53 shows the performance of the LU problem, class A. LU solves a synthetic system of Nonlinear partial differential equations using the mathematical algorithm of Gauss-Seidel and specifically the successive over-relaxation (SSOR) variant. The benchmark organizes the processes in a pipeline manner. A processor makes its computation and subsequently, forwards the result to the next process. The next process starts its own computation after receiving the result. At the same time, the first process has advanced to its second computation. Such a process is followed by all the participating MPI processes, utilizing non blocking receives (MPI_Irecv and MPI_Wait) and blocking sends (MPI_Send). Among the available classes we chose class A since it has a fair percentage of communication time in comparison to the rest of the classes. Class A regards a 64 x 64 x 64 grid and performs 250 iterations. In the figure, we see the average executions of LU using the write and read-based implementation with 16, 32, 64 and 128 ranks. The x axis show the total duration in milliseconds while the y axis the number of ranks used in the respective run. Using 16 ranks, the write-based variant outperforms the read-based variant by 1,5%. This difference may seem small but it regards the total duration of the problem including both computation and communication. According to our profiling, the communication time constitutes 15% of the total experiment's duration with an average of 1500 bytes sent per message when run with 16 nodes. Additionally, the percentage of fast receives is about 40% for the first 3 numbers of ranks (16, 32, 64) and around 30% for 128 ranks. Using 32 ranks, we see a difference of 3,8% in total duration with communication time constituting the 25% of the execution time. When the benchmark uses 64 or 128 MPI processes, the difference between the two variants comes close to 7% while the communication time percentage becomes 30% and 33% respectively.
Figure 4.54 compares only the communication time of class A of LU between the two implementations. The x axis shows the number of nodes (and MPI Ranks) while y axis shows the average communication time per rank in seconds. We notice that the write-based variant has a 10% lower communication latency in the 16 node run while it achieves about 20% lower latency in the 32 and 64 nodes runs. We attribute this fact to the fact the 16

node run is limited to one mezzanine. On the other hand, the next runs include inter mezzanine communications which benefit more from our implementation as shown in the Microbenchmark experiments. In the 128 node run, the write-based variant having 15% less latency difference most likely due to the reduced percentage of fast receives. Note that due to the nature of the benchmark, the average communication time per rank gets reduced as the number of nodes increases over 16 nodes. By shuffling the hostfile, we managed to slightly increase the communication difference between the variants by 15% in favor of the write-based implementation. Figure 4.55 shows the average mathematical operations per second for each process. This metric primarily depends on the node's hardware's computing capabilities but communication can also have a small impact to it. We see that in the write-based variant, we achieve from 1% to 2.5% improvement compared to the read-based variant.

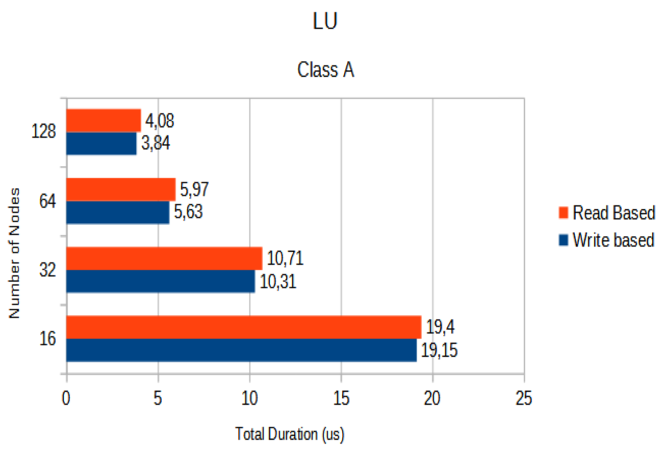


Figure 4.53: Total duration comparison, LU NAS Benchmark

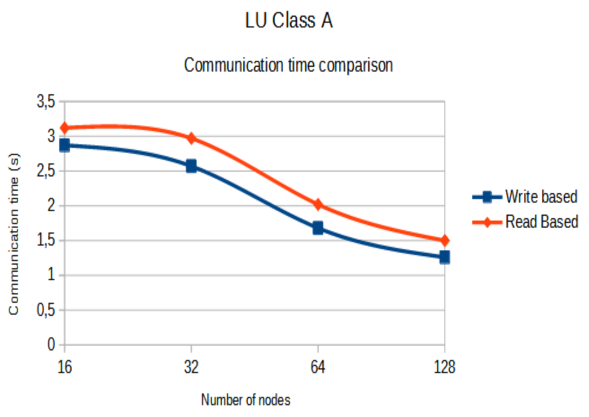


Figure 4.54: Communication time comparison, LU NAS Benchmark

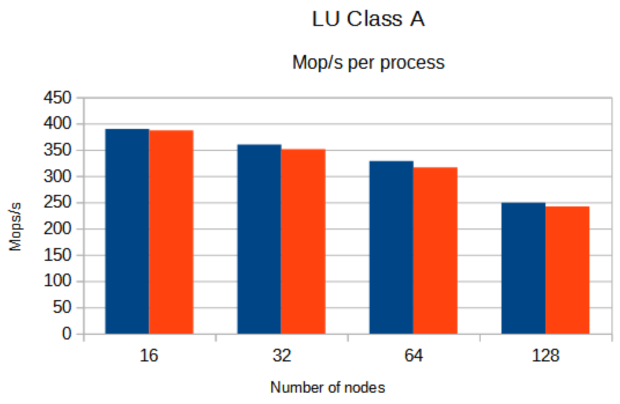


Figure 4.55: Mops/s comparison, LU NAS Benchmark

Figure 4.56 shows the results of the MG Benchmark's execution. MG uses the Multi-Grid method (V-Cycle) to approximate a solution to a 3-dimensional cubic domain decomposed into a regular grid. This benchmark requires point-to-point communication to update every processor's boundary values for each dimension that is distributed. As a result, the benchmark makes use of the primitives mpi_send, mpi_irecv while it also utilizes communication between distant nodes. We chose to evaluate class B since smaller classes have extremely small duration while bigger ones have worse communication to computation ratio. Class B regards a grid of size 256 x 256 x 256 and performs 20 iterations. In the figure, we see that the write_based implementation achieves a lower duration by 1% (16 nodes run), 4% (32 nodes run) and 3% (64 nodes run). The run that uses 128 ranks shows no difference between the two implementations, which indicates that the benchmark cannot be scaled more communication-wise. Our profiling indicates that the communication time constitutes 5% of the total execution time in the 16 nodes run while in 32, 64 and 128 nodes runs it equals 8%, 9% and 2% of execution time respectively. We see that the small differences in the total execution time are not representative of the performance difference between the MPI implementations since the difference of the two implementations in communication time are significant as seen in Figure 4.57. The write-based variant has 29%, 33% and 36% lower communication duration than the read-based variant in runs with 16, 32 and 64 nodes respectively. This fact can be explained by the average percentage of fast receives that is around 35%. In addition, the benchmark performs many inter QFDB transfers. As a consequence, shuffling the hostfile infers no actual difference to execution times of this benchmark with any number of MPI processes. As already seen, inter QFDB transfers have better performance in the write-based implementation. Also, as the number of ranks increases, the average message size that gets sent in the benchmark gets smaller, thus making our advantage in inter QFDB transfers more apparent. In Figure 4.58 we see that the difference in Mops/s doesn't exceed 2,3% (64 nodes run).

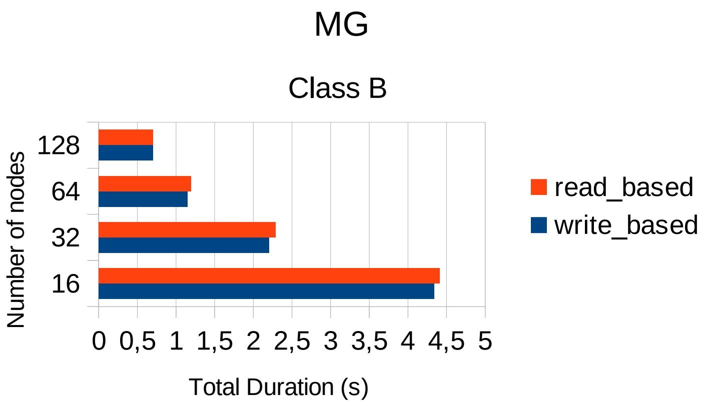


Figure 4.56: Total duration comparison, MG NAS Benchmark

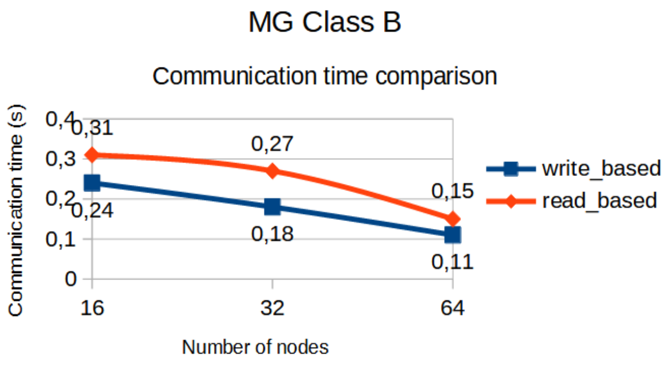


Figure 4.57: Communication time comparison, MG NAS Benchmark

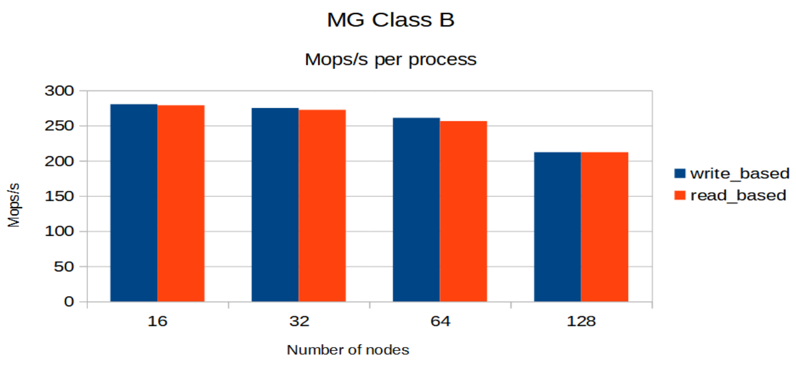


Figure 4.58: Mop/s comparison, MG NAS Benchmark

Next, Figure 4.59 shows the evaluation of the two variant's performance using the class C of IS Benchmark. IS performs bucket sorts in many keys using random memory access. During its execution, it utilizes all to all communication and MPI_Allreduce calls in order to get the bucket size totals to determine the redistribution of the keys. Class C sorts 2^{25} keys with a max value of 2^{21} and performs 20 iterations. This benchmark has a considerable amount of communication time. Specifically, in a run with 16 nodes the computation time constitutes the 24% of total execution time while in the runs with 32 nodes the percentage increases to 41%. In runs with 64 and 128 nodes the communication to computation ratio reaches 50%. However, the benchmark uses very large messages (from 60.000 to 800.000 bytes on average depending on the number of ranks). The percentage of fast receives does not surpass 25% in any of the runs. We see that when 16 ranks are used, the write-based variant has a 5% better execution time than the read-based variant. In the rest of the experiments the write-based implementation outperforms the read-based on by almost 10%. These results are expected by taking into account the results of the OSU_Allreduce microbenchmark presented earlier since they indicate a performance gain of about 20% for our implementation. Figure 4.60 shows the difference in communication time. We see that the write-based variant has a 15-20% lower communication latency in all runs. We note that we did not see significant improvement in the Mops/s between the two variants in any run. In addition, shuffling the hostfile did not infer any change in the difference between the two implementations since the benchmark uses all to all communication forcing all the processes to communicate with each and every other process at some point. This means that a process will communicate with all other ranks and not just the ones with a nearby MPI_Rank.

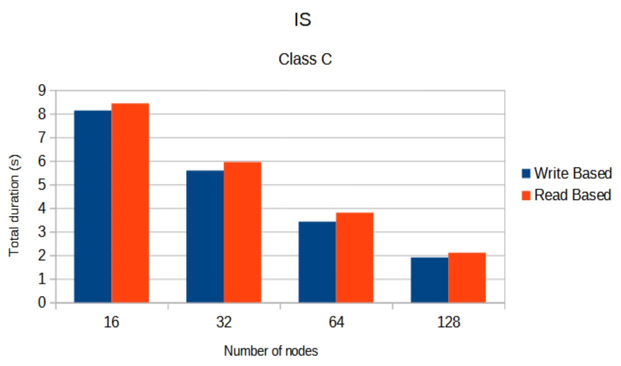


Figure 4.59: Total duration comparison, IS NAS Benchmark

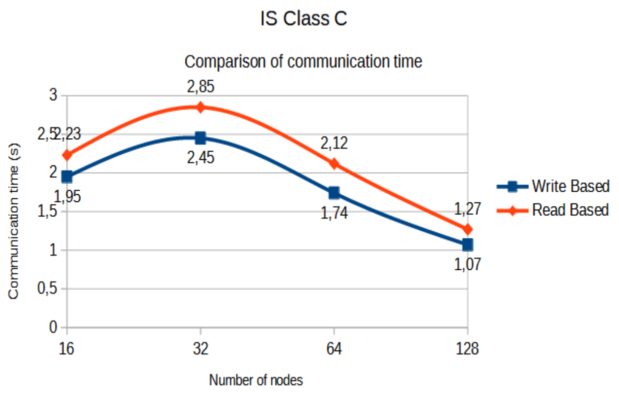


Figure 4.60: Communication time comparison, IS NAS Benchmark

Figure 4.61 depicts the evaluation of the SP Benchmark, class A. SP solves a synthetic system of non linear partial differential equations, like LU. However, SP uses a scalar pentadiagonal solver instead of successive over-relaxation (SSOR). The benchmark makes use of MPI_Send, MPI_Irecv and wait primitives in order to inform neighbors about the results of their computations during the computation of the problem. During our profiling we saw that in this benchmark has a high percentage of fast receives using 25 ranks. More precisely, fast receives constitute about 75% of total receives when run with 25 ranks. That percentage touches 22% and 18% for the 64 and 121 ranks runs respectively. Note that the benchmark requires the number of ranks to be a number with an integer square root. In the figure we see that the read-based variant achieves about 1,5% more total execution time than the write-based variant in all runs. We should note that these results regard total execution time and not communication time. SP is a computation intensive benchmark. Using 25 ranks, the communication time is the 6% of the total execution time while this percentage rises to 10% when we use 64 and 121 ranks. Figure 4.62 shows the performance difference regarding only communication time. As we can notice, the case with more fast receives offers more significant performance differences. We should also note that in the 25 ranks run, the average size of a sent message was about 7000 bytes. With smaller messages an even better result could have been achieved.

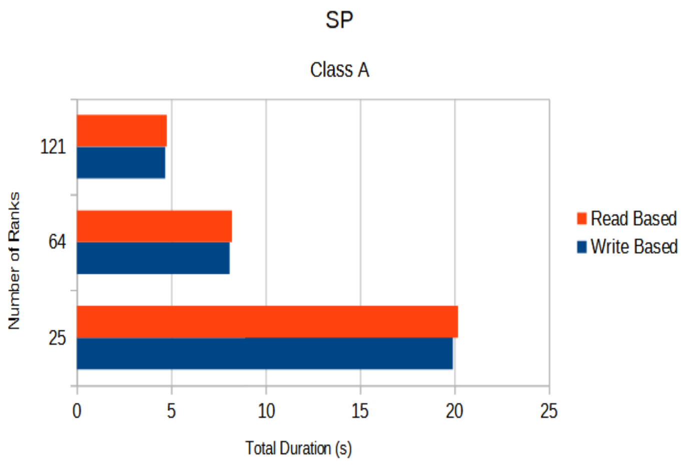


Figure 4.61: Total duration comparison, NAS SP Benchmark

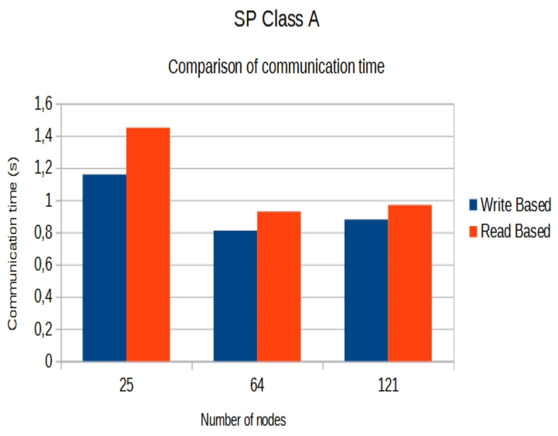


Figure 4.62: Communication time comparison, NAS SP Benchmark

HPCG

HPCG [27] (High Performance Conjugate Gradients) is another scientific application we used in order to evaluate our implementation. This benchmark uses a preconditioned conjugate gradient (PCG) algorithm to measure the performance of HPC platforms. Its implementation uses a regular 27-point stencil discretization in 3 dimensions of an elliptic partial differential equation (PDE) with zero Dirichlet boundary condition. The 3D domain is mapped onto a 3D virtual grid of MPI processes. The main communication pattern of the benchmark is the Halo Exchange that happens between the processes constituting the 3D grid. For that exchange, the primitives MPI_Irecv, MPI_Send and MPI_Wait are used. In addition, the benchmark uses MPI_Allreduce in order to obtain maximum global residuals, or the sums of the numbers of non zero values of each process through the execution. The communication time percentage of the benchmark does not surpass 3% in sizes bigger than proof of concept which results in a very little small between the two MPI implementations regarding execution time. However, we measure specifically the average Halo Exchange time inside the application. In addition, the benchmark reports the MPI_Allreduce latency in its output as well as GFLOP/s.

In Figure 4.63 we see the GFLOP/s results of the HPCG benchmark using a problem of medium size 64x64x64 running for approximately 200 seconds. In our profiling we see that with this problem size, the fast receives make up over 65% of total receives for any cluster configuration. As we can see in the figure, the difference is existent but never over 3% of GFLOP/s in favor of our implementation. We remind that GFLOP/s is a metric which primarily regards CPU capacity but can also up to some degree get affected by the communication between the processes. In Figure 4.64 we can see the evaluation of the two MPI variants by using the average latency of the Halo Exchange stage for each implementation. We see that the write-based variant has a clear advantage in all runs outperforming the read-based one by up to 46% of the Halo exchange. The combination of a high fast receive percentage in combination with the existence of inter QFDB transfers is sufficient explanation for this result. Lastly, Figure 4.65 shows the comparison of average Allreduce latency reported by the benchmark. We can see the write-based variant outperforms the read-based implementation by 12-20%, which is equal to the performance improvement the osu_allreduce microbenchmark. The same pattern can be observed in different problem sizes of HPCG with the main difference that the communication time becomes an even smaller part of the execution time as the problem size increases.

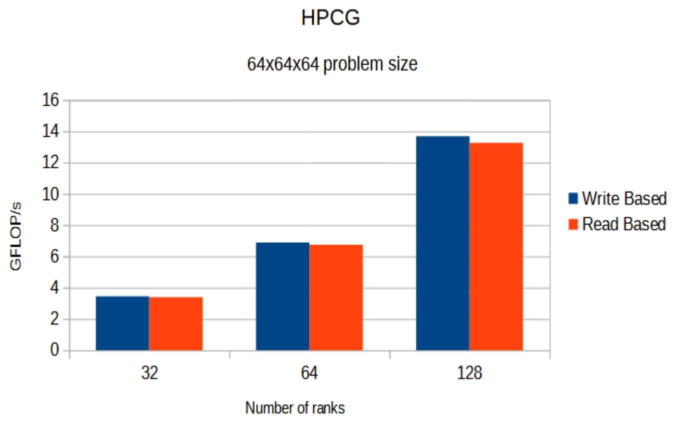


Figure 4.63: GFLOP/s comparison, HPCG Benchmark

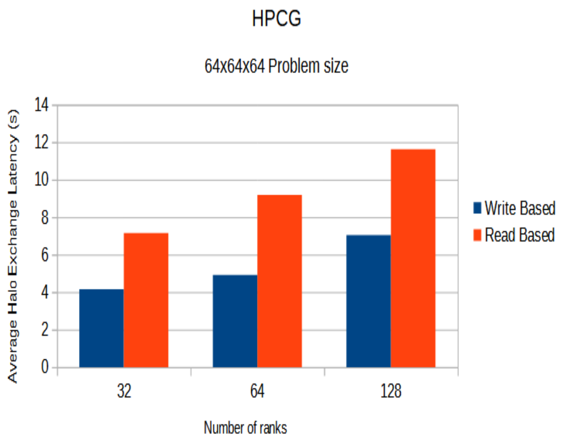


Figure 4.64: Halo Exchange average latency comparison, HPCG Benchmark

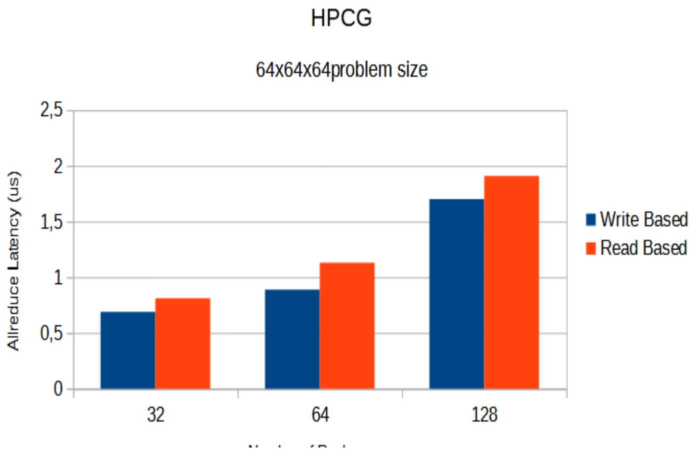


Figure 4.65: All Reduce average latency comparison, HPCG Benchmark

MPI Graph

MPI Graph [29] is an MPI benchmark designed to measure message bandwidth and inspect the scalability of HPC systems while exposing them to heavy load. The benchmark arranges all MPI processes in a logical ring and performs as many steps as the number of processes minus one. In each step, each MPI process transfers data to the process D units to the right and receives data from the process D units to the left. The value of D starts at 1 and increments at each step. By the end of the process, each MPI process has communicated with any other process except for itself. At the end of the run, the benchmark reports both the send and receive average bandwidths. In each step, the application uses MPI_Isend and MPI_Irecv primitives for the transfer of data while subsequently it waits for their completion using successive calls to 2 different MPI_Testall primitives, one for MPI_Irecv and another for MPI_Isend requests. The main peculiarity in MPI Graph's code is the fact that all of its MPI_Irecv calls make use of MPI_ANY_TAG. As mentioned in **Section 3.6**. write_based 3.0 suspends receiver initiation in the presence of MPI_ANY_TAG besides MPI_ANY_SOURCE. As a result, we choose to show the comparison of the read-based MPI with the optimistic variant of the write-based MPI. The performance of write-based 3.0 is practically identical to that of the write-based variant's. Since the application does not use MPI_ANY_SOURCE and all the receive requests have the exact same size with matching send requests, the use of write_based_optimistic is permitted. Additionally, our profiling shows that MPI Graph has an unusually high percentage of fast receives (~99%).
In Figures 4.66, 4.67, we see the results of MPI Graph for the send bandwidth. We can see that while the receive bandwidth is almost the same with both implementations, the send bandwidth shows an improvement

with the write-based MPI of the class of 10%. We attribute this difference to the fact that almost all the receives of the application are fast receives which improves the performance of the matching send. We also note that since MPI Graph does not use eager messages (with the default configuration) one could also use the second variant of Exanet MPI combined the memory optimization and achieve the same result as write_based_optimistic.

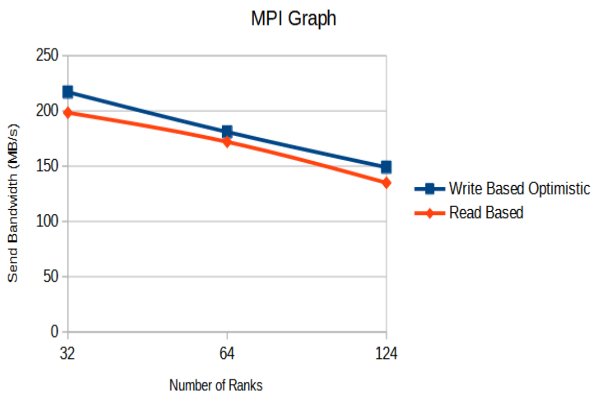


Figure 4.66: Comparison of MPI Graph send bandwidth

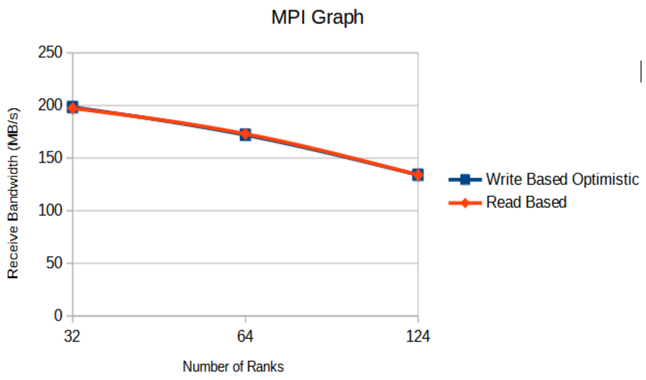


Figure 4.67: Comparison of MPI Graph receive bandwidth

LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [28] is a molecular dynamics program from Sandia National Laboratories, which makes use of MPI. Among the problems contained in the LAMMPS suite, we evaluate our implementation using eam (metallic solid, Cu EAM potential with 4.95 Angstrom cutoff) and Chute (granular chute flow, frictional history potential with 1.1 sigma cutoff). Both problems make use of neighbor lists to keep track of nearby particles as well as spatial decomposition to partition the simulation domain into small 3d sub-domains. As a consequence, they make use of both blocking and non-blocking point-to-point primitives and MPI_Bcast and MPI_Allreduce collective functions. For both problems we used the default input files provided which set the number of atoms to 32,000. LAMMPS report throughput in the form of Timesteps/s at the end of the execution. In Figures 4.69 and 4.70 we see the runs of the two problems using both implementations. The performance pattern of the two problems is almost identical. One can see that the write-based implementation achieves a slightly better throughput in all cluster configurations. The difference becomes more evident in the 128 nodes run in which the write-based implementation achieves 2-3% higher throughput than the read-based variant. At this point we should mention that throughput measured by LAMMPS is also highly dependent on hardware characteristics and not solely on MPI communication. Should we be able to use all the cores available in the future, we may manage to produce different executions that will underline the advantage of our implementation more.

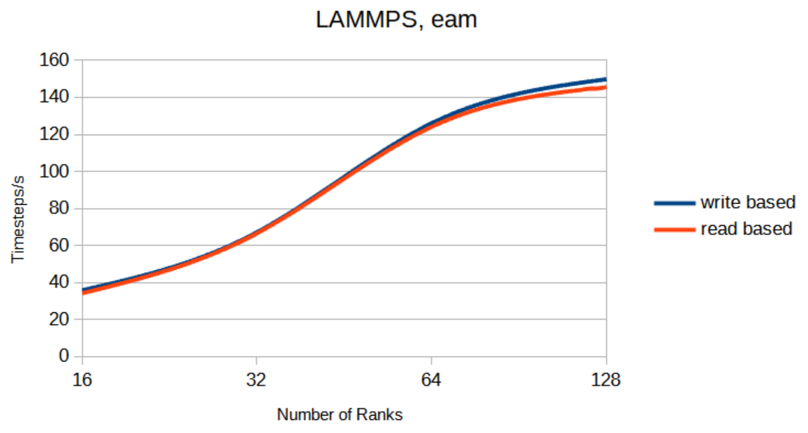


Figure 4.69: LAMMPS eam Problem,Throughput Evaluation

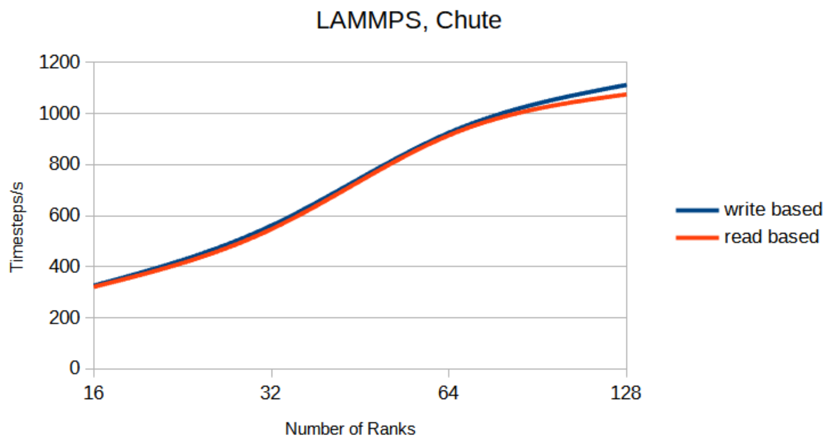


Figure 4.70: Figure 4.70: LAAMPS Chute Problem, Throuhput Evaluation

In conclusion, we can safely argue that the write-based version of Exanet MPI offers significant improvement in comparison with the read-based variant in most cased. However, one must be able to determine the best variant of the write-based implementation for each application. For this reason, a profiling of the application's source code is needed before the choice of the most suitable variant in order to achieve the biggest possible benefit. As we saw in this evaluation process, in general, write_based 3.0 is the most suitable variant in cases where an application does not make extensive use of MPI_ANY_TAG or contains a very high number of eager requests. On the other hand, when MPI_ANY_TAG is extensively used, one might prefer to make use of write_based 2.0 (enchanted with the memory optimizations of **Section 3.10**) which does not suspend receiver initiation in such cases. However, it should be taken into account that the existence of many eager sends in an application may overwhelm the benefit of avoiding the suspension of receiver initiation in the presence of MPI_ANY_TAG. Ultimately, when an MPI program ensures that matches between eager sends and long receives never happen as well as that no MPI_ANY_SOURCE is used then the write_based_optimistic is the optimal choice for a user.

Chapter 5

Related Work

Design of MPI communication protocols is a topic of research that dates back to the very early days of Message Passing. In this chapter, we attempt to present an indicative list of related work which mainly regards, but is not limited to, the support of receiver initiation in long rendezvous protocols in scientific literature. Since sender initiation was used in the first forms of long protocols and is generally present in any implementation, we will try to focus on works that are not limited to it.

Works [1] and [2] constitute two of the earliest works introducing receiver initiated rendezvous protocols as means of transferring data. The former was published in 1998 and had as its main goal to address the overhead incurred by the use of sender initiated protocols. At the time of that paper's writing, sender initiation was the only form of long protocols deployed, In addition, sender's control messages induced interrupts on the receiver's side which further worsened performance. The authors offer a simple receiver initiated long protocol as a solution, which lets the receiver notify the sender about its intention to receive a message while also allowing the send-receive matching to take place at the sender's side. It's worth noting that they also underline the inability of the receiver to initiate the communication in the case of MPI_ANY_SOURCE while they do not examine the possible coexistence of an eager and long protocol. The latter work propose a new whole communication architecture called FCI which also contains a subset of MPI routines in its API. The new architecture suggested focuses on the use of specialized hardware and software primitives that offer zero copy transfers, user space communications as well as some native implementations of various MPI functions (barriers, reduce, non blocking send/receive functions) integrated in FCI's internals. An interesting part of the work is their support for a communication protocol that does not allow unexpected send requests (ie. sends posted before the matching receive) but require senders to write data immediately to the receiver's memory address space without previous synchronization. Receivers build tables of requests in a global address space visible by the senders, which in turn use those tables to learn the destination memory address of a transfer.

Authors in [30] introduces the combination of an eager and a long protocol. However, the long protocol, unlike our implementation, supports only sender initiation in order to avoid the complication the receiver initiation induces to an eager protocol. The long protocol also depends on a DMA write followed by a notification signaling the end of the transfer. [9], [11], [12] make an attempt to improve the latency of the rendezvous protocol by using receiver initiation but they do not offer any optimizations for short messages (eg. an eager protocol). [11] comments on the thread safety of MPI routines as well as on ways to optimize the progress engine thread by polling on different locations concurrently. [6] additionally discusses the case of concurrent issuing of RTS and RTR messages by the sender and the receiver respectively. In order to face that case, the require the sending of Acknowledgment messages each time an RTR or RTS gets received by a process. [7] is a very interesting work that proposes an eager protocol very similar to ours that can be combined with a receiver initiated long protocol. They utilize counters in a similar way we do in our implementation but still require eager sends to insert objects in the Posted Requests and queue and acquire locks in all cases, unlike Exanet MPI. They also state some circumstances in which the FIN control message (equivalent of Env in our work) may be omitted. However, they render such an elimination possible only in cases where the message's size is equal to the size denoted by the receiver while in our implementation the Env gets omitted in all single threaded programs. Thread safety is not mentioned at all in that work.

Works like [3] and [13] are some relatively recent attempts to increase the communication and computation overlap when non blocking functions are used. They examine optimizations regarding the progress engine thread as well as lock contention prevention. In addition, they propose methods to reduce the number of context switches between threads. In our implementation, we did not focus as much in optimizing contention between the main user thread and the progress engine as we were not allowed to run more than one MPI process in one FPGA which means that there was no scenario in which the cores would be oversubscribed. Additionally, none of the benchmarks we used uses multiple MPI threads. However, we took into account the effect of multiple MPI threads in theory and redesigned our protocol as described in **Chapter 3.** [13] also proposes some new communication protocols that derive from the combination of already existing protocols. For instance, in some cases they suggest a sender initiated protocol in which the sender writes half of the data while concurrently the receiver reads the other half, thus combining both read-based and write-based protocols.

We should also mention [14] and [15] which do not contribute in the development of new protocols but they offer insight of other aspects of MPI development like the implementation of non blocking collective functions and the optimization of internal data structures respectively. [14] suggests the method we also used to implement non blocking versions of collective functions among others.

These works do not constitute an exhaustive list of related work but are rather indicative and selected by us as some of the most influential in their respective topics.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work we designed, implemented and further optimized a write-based version of the Exanet MPI. Specifically, we initially studied the drawbacks and weaknesses of the read-based protocol used in the preexisting Exanet MPI implementation and came up with a write-based protocol design which additionally supports receiver initiation. In order to find out if our protocol is competent with the read-based protocol, we implemented from scratch a new Exanet MPI implementation and evaluated it using the already optimized read-based implementation as a baseline implementation. Subsequently, we deployed improved versions of our protocol, each one offering a different optimization or countering complications raised from the support of receiver initiation. We managed to develop in total 4 write-based Exanet MPI variants, break down the performance gain achieved by each one of them and finally find the sweet spot regarding the trade off between cost and benefits of each optimization. We proposed our methods for improving the fast path of a fast receive and for making our eager protocol competent with the read-based protocol even when receiver initiation is supported. Moreover, we showed that MPI_ANY_SOURCE and MPI_ANY_TAG have indeed a negative impact on receiver initiated protocols. This fact gets signified by the slightly improved performance the optimistic variant of our implementation achieves. Our evaluation showed that in general, the write-based implementation can achieve up to 50% latency improvement compared with the read-based variant of Exanet MPI as well as take significant advantage of the early receive case. In addition, a write-based implementation is benefited from the long distance that can exist between nodes as it infers shorter control paths in all cases. As a consequence, the total execution time of certain scientific applications using the write-based MPI can be reduced by up to 10% of the duration achieved with the Read-based variant.

6.2 Future work

First and foremost, we plan to evaluate our implementation using all cores available to the HPC prototype (described in Section 2.5). Currently, as already mentioned, intra-FPGA DMA writes are not supported using the PL-DMA API. However, they are supported in the read-based implementation which makes use of the R5 microprocessor to perform the emulated reads. This significantly weakens our capability to increase the communication potential of applications by using more MPI processes per execution. Additionally, we intend to further investigate the factors that cause the percentage of fast receives in an application to change and manage to find ways to inflame that value when possible. Another important part of future work is the full support for MPI derived datatypes in receiver initiation scenarios. We plan to render a sending process able of allocating memory buffers remotely on the receiver's side which will help in the allocation of temporary receive buffers for the receiving of packed data when necessary. Moreover, we are also interested in developing our own MPI primitive which will support DMA transfers without the need for a rendezvous protocol. This is possible since in the third variant of our implementation each send and receive request gets assigned a ticket ID (provided MPI_ANY_SOURCE and MPI_ANY_TAG are not used). This can make a send request able to determine a pre-agreed receive buffer and notification address for a matching receive request without the need of receiving RTR and CTS messages. The buffers' address would emerge from the receive request's ticket ID, its MPI rank, communicator and tag combination. Lastly, the process of figuring which variant of our implementation is the most suitable choice for an application is quite manual. It would be helpful if there was some automated way of picking the best variant without human intervention in the future.

.

Bibliography

[1] Osamu Tatebe , Yuetsu Kodama , Satoshi Sekiguchi , Yoshinori Yamaguchi. Highly efficient implementation of MPI point-to-point communication using remote memory operations. In Proceedings of the 1998 International Conference on Supercomputing (ICS98)

[2] Stephan Brauss, Martin Frey,Martin Heimlicher,Andreas Huber, Martin Lienhard, Patrick Müller, Martin Näf,Josef Nemecek,Roland Paul,Anton Gunzinger. Highly efficient implementation of MPI point-to-point communication using remote memory operations. In SC '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing

[3] Amit Ruhela Hari, Subramoni Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha , Dhabaleswar K. Panda. Efficient Asynchronous Communication Progress for MPI without Dedicated Resources. In EuroMPI'18: Proceedings of the 25th European MPI Users' Group Meeting, Pages 1-11

[4] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett & A. Lumsdaine . TEG: A High-Performance, Scalable, Multi-network Point-to-Point Communications Methodology. In EuroPVM/MPI 2004: Recent Advances in Parallel Virtual Machine and Message Passing Interface pp 303–310

[5] Anthony Danalis, Aaron Brown, Lori L. Pollock, Martin Swany, and John Cavazos. Gravel: A communication library to fast path mpi. Pages 111–119, 09 2008.

[6] Mohammad J. Rashti; Ahmad Afsahi. Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects In 2008 22nd International Symposium on High Performance Computing Systems and Application

[7] Matthew Small, Xin Yuan. Maximizing MPI point-to-point communication performance on RDMA-enabled clusters with customized protocols. In ICS '09: Proceedings of the 23rd international conference on SupercomputingJune 2009 Pages 306–315

[8] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In International Journal of Parallel ProgrammingVolume 32Issue 3June 2004 pp 167–198

[9] Scott Pakin, Receiver-initiated Message Passing over RDMA Networks In 2008 IEEE International Symposium on Parallel and Distributed Processing

[10] Samuel K. Gutierrez, Nathan T. Hjelm; Manjunath Gorentla Venkata; Richard L. Graham, Performance Evaluation of Open MPI on Cray XE/XK Systems. In 2012 IEEE 20th Annual Symposium on High-Performance Interconnects

[11] Sayantan Sur, Hyun-Wook Jin, Lei Chai, Dhabaleswar K. Panda
.RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programmingMarch 2006 Pages 32–39

[12] Mohammad J. Rashti; Ahmad Afsahi. Assessing the Ability of Computation/Communication Overlap and Communication Progress in Modern Interconnects. In 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)

[13] S. Chakraborty; M. Bayatpour; J. Hashmi; H. Subramoni; D. K. Panda. Cooperative Rendezvous Protocols for Improved Performance and Overlap. In SC18: International Conference for High Performance Computing, Networking, Storage and Analysis

[14] Torsten Hoefler, Andrew Lumsdaine, Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007

[15] Judicael A. Zounmevo; Ahmad Afsahi An Efficient MPI Message Queue Mechanism for Large-scale Jobs. In 2012 IEEE 18th International Conference on Parallel and Distributed Systems

[16] Manolis Ploumidis, Nikolaos D. Kallimanis, Marios Asiminakis, Nikos Chrysos, Pantelis Xirouchakis, Michalis Gianoudis, Leandros Tzanakis, Nikolaos Dimou, Antonis Psistakis, Panagiotis Peristerakis, Giorgos Kalokairinos, Vassilis Papaefstathiou, and Manolis Katevenis. Software and Hardware Co-design for Low-Power HPC Platforms. In 5th International Workshop on Communication Architectures for HPC, Big Data, Deep Learning and Clouds at Extreme Scale

[17] M. Katevenis, N. Chrysos, M. Marazakis, I. Mavroidis, F. Chaix, N. Kallimanis, J. Navaridas, J. Goodacre, P. Vicini, A. Biagioni, P. S. Paolucci, A. Lonardo, E. Pastorelli, F. Lo Cicero, R. Ammendola, P. Hopton ,P. Coates, G. Taffoni, S. Cozzini, M. Kersten, Y. Zhang, J. Sahuquillo, S. Lechago, C. Pinto, B. Lietzow, D. Everett, G. Perna. The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems. In 2016 Euromicro Conference on Digital System Design (DSD)

[18] ExaNeST https://exanest.eu/

[19] MPI Forum https://www.mpi-forum.org

[20] MPICH High-Performance Portable MPI https://www.mpich.org/

[21] Open MPI: Open Source High Performance Computing https://www.open-mpi.org/

[22] Web Pages for all MPI Routines https://www.mpich.org/static/docs/v3.2/www3/index.htm

[23] CARV Laboratory, FORTH https://www.ics.forth.gr/carv/

[24] INFN https://home.infn.it/en/

[25] OSU Microbenchmarks https://mvapich.cse.ohio-state.edu/benchmarks/

[26] NAS Parallel Benchmarks - NASA Advanced Supercomputing Division https://www.nas.nasa.gov/software/npb.html

[27] HPCG https://www.hpcg-benchmark.org/

[28] LAMMPS https://www.lammps.org/\

[29] MPI Graph https://github.com/LLNL/mpiGraph

[30] Jiuxing Liu., Jiesheng Wu, Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand In ICS 2003

[31] ARM Cortex A53 https://developer.arm.com/Processors/Cortex-A53

[32] ARM Cortex R5 https://developer.arm.com/Processors/Cortex-R5