12-2019

Improving MPI Threading Support for Current Hardware Architectures

Thananon Patinyasakdikul
University of Tennessee, tpatinya@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Thananon Patinyasakdikul entitled "Improving MPI Threading Support for Current Hardware Architectures." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Michael Berry, Michela Taufer, Yingkui Li

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Improving MPI Threading Support for Current Hardware Architectures

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Thananon Patinyasakdikul

December 2019

To my parents Thanawij and Issaree Patinyasakdikul, my little brother Thanarat Patinyasakdikul for their love, trust and support.

Acknowledgments

I would like to express my deepest gratitude to Dr. Jack Dongarra for giving me the opportunity to become a graduate research assistant at Innovative Computing Laboratory (ICL) and start my journey into the world of high-performance computing (HPC). It is a privilege to work with him and have him as my advisor. His wisdom and experiences will always be the guidance for my future work and life.

Dr. George Bosilca, my project leader, is an outstanding teacher. I would like to thank him for taking his time to explain and teach me everything I know about MPI. I will always be impressed by his thorough understanding of many projects, and the ability to switch the topic, jumping into a different project on the deepest level on the fly. Outside of work, Dr. Bosilca is very friendly, funny and approachable. I always enjoy working, hanging out, and arguing with him.

I am grateful to my committee, Dr. Michael Berry, Dr. Michela Taufer, and Dr. Yingkui Li for agreeing to serve on my dissertation committee. I greatly appreciate their time, dedication, and invaluable guidance on this dissertation.

I would like to thank my parents for supporting my decision to leave the homeland to pursue my Ph.D. in the United States and providing me with emotional support, love, and understanding. I thank my little brother for taking his time to video call me on the weekends to give me the update on his exciting teenager life.

Last, but definitely not least, I would like to extend my great appreciation toward my colleagues at ICL, from staff scientists to fellow students. Dr. Aurelien Bouteiller, Dr. Thomas Herault, Dr. Damien Genet, Dr. Anthony Danalis, Reazul Hoque, Xi Lou, David Eberius and Yu Pei, and more, for having me in their company, and providing me with great

(and not so great) ideas in both work and recreational environment. I wish them all the best.

Abstract

The Message Passing Interface (MPI) has been the most popular programming paradigm in the high-performance computing (HPC) landscape. The MPI standard provides an efficient communication API with the capability to handle different types of data movements across a variety of network hardware and platforms.

Since the inception of the MPI standard, the trend in hardware has evolved; a higher number of CPU cores per node introduces more opportunity for thread-parallelism. Dealing with changes in the hardware landscape, threading support has been added to the MPI standard in a later version, with the goal of allowing the user to exploit thread parallelism in MPI applications. Without the need of explicit communication between threads within the same process, multi-threaded MPI is the approach that can relieve stress on the intra-node communication, allowing MPI to focus on only inter-node communication. Nonetheless, this approach comes with its own set of challenges and limitations, which are addressed in this work.

Threading support for MPI has been defined in the MPI standard since 2008. While many standard-compliance MPI implementations fully support multithreading, they still cannot provide the same level of performance as their non-threading counterpart. This leads to a low adoption rate from applications, and eventually, lesser interest in optimizing threading support for MPI.

In this work, I propose, implement, and analyze threading optimization of MPI by exploring different tools and approaches to leverage the power of thread parallelism. First, I showed that my multi-threaded MPI benchmark enables MPI developers to stress test their implementation and optimization designs. Second, this work addresses the interoperability between MPI implementations and threading frameworks by introducing a design that gives

the MPI implementation more control over user-level thread, creating more opportunity for thread utilization in MPI. This design shows up to 7× performance gain in comparison to the original implementation. In the final phase of this study, I propose, implement, and analyze several strategies to address the discovered bottlenecks in the MPI implementation. This novel threading optimization can achieve up to 22× the performance compared to the legacy MPI design in two-sided communication and over 200× in one-sided communication.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Message Passing Interface (MPI) is nearly ubiquitous in high-performance computing (HPC)—according to Bernholdt et al. more than 90% of Exascale Computing Project (ECP) and Advanced Technology Development and Mitigation (ATDM) application proposals use it either directly or indirectly. Therefore, the availability of high-quality, high-performance, and highly scalable MPI implementations which address the needs of applications and the challenges of novel hardware architectures is fundamental for the performance and scalability of parallel applications.

The MPI standard provides an efficient and portable communication-centric API that defines a variety of capabilities to handle different types of data movement across processes, such as point-to-point messaging, collective communication, one-sided remote memory access (RMA), and file support (MPI-IO) Forum (2015). This ensemble of communication capabilities gives applications a toolbox for satisfying complex and irregular communication needs in a setup that maintains portability and performance across different hardware architectures and operating systems. Owing to these characteristics, many scientific applications have adopted MPI as their communication infrastructure and, therefore, rely on the efficiency of the MPI implementation to deliver the best communication performance for their applications across different networking hardware on various platforms.

Recent hardware developments, with higher numbers of cores per chip, even with higher frequency, have shifted the balance of computation vs. communication in favor of computations, which have become faster and more energy efficient. Over the last

decade alone, theoretical node-level compute power has increased 19×, while bandwidth available to applications has seen an increase by a factor of 3× only, resulting in a net decrease in bytes per floating-point operation (FLOP) by 6× Rumley et al. (2017). An increased rate of computation needs to be sustained by a matching increase in memory bandwidth, but physical constraints—such as the conductivity and the thermal capacity of the network cables' materials—set hard limits on the latency and bandwidth of data transfers. The current solution to overcome these limitations has increased the number of memory hierarchies, with orders of magnitude variation in cost and performance between them. Essentially, current architectures represent execution environments where data movement is the most performance and energy critical component. This shift has greatly impacted the traditional programming approach where each computational core corresponds to a unique process and every data movement passes through a message-passing layer. As the intra-node and inter-process communication costs started to rise, efforts began to move applications toward a more dynamic and/or flexible programming paradigm.

Using the combination of processes and threads becomes one of the promising solutions, as the approach is capable of relieving the pressure on the memory infrastructure. One of the advantages of this approach comes from the benefit that no explicit communication between threads in the same process is necessary. Although the use of multiple threads to alleviate the pressure on intra-node data movement seems like an intuitive approach, it generates an entire set of new challenges in both programmability and communication levels.

Firstly, the challenge is to guarantee for the thread safety of the applications. Multiple threads are likely to create 'race conditions' if they try to access or update the same memory, affecting the correctness or even corrupting the state of the application if not handled appropriately. In order to provide thread safety and prevent potential race conditions in applications, several thread synchronization mechanisms are available. However, thread synchronization often involves the interlocked memory operations or a kernel-level transaction, translated into an extra overhead for the multi-threaded application. Moreover, other than in the application level, the MPI implementations, as the message passing layer, is also susceptible to the race-conditions from threads and has to be designed in a way that provides thread safety with minimal overhead in multi-threaded environments.

Secondly, the communication level challenge stems from the nature of the non-deterministic behavior of the threads. Generally, threads behave better when they are more independent or loosely coupled, but more flexibility translates into reduced ordering between actions in different threads—and unfortunately this also includes the communication. The out-of-order communication from threads can become a significant problem for the MPI implementations. It becomes a chronic symptom of the lack of send determinism in applications Guermouche et al. (2011). That being said, in a communication paradigm other than the MPI, this could have been a minor issue (as an example in an Active Message Eicken et al. (1992) context). The MPI standard mandates a first-in, first-out (FIFO) message ordering guarantee for simplicity of programming, relieving the user of the burden of maintaining their own messages order. Unfortunately, the MPI standard does not take thread non-deterministic behavior into consideration, as support for multi-threaded environment has been added in the later version. This poses more challenges to optimize the multi-threaded support of MPI. Furthermore, the lack of the interoperability between the threading frameworks and the MPI contributes to more optimization limitations for the MPI developers, as the MPI implementations do not have necessary thread information to evaluate and make optimization decisions accordingly.

Current state-of-the-art MPI implementations are struggling to support a large number of concurrent communications and are under-utilizing thread parallelism in multi-threaded environments, resulting in suboptimal performance in the communication. With that in mind, this study proposes several strategies to enhance MPI communication performance in multi-threaded environments through an increased concurrency on different levels of the MPI implementation for both one-sided and two-sided MPI communications.

1.1 Dissertation Statement

While most current state-of-the-art MPI implementations fully support threading environments in MPI, the performance of the existing threading environment is still far behind its non-threading counterparts. This disparity, in turn, creates the 'chicken-and-egg' problem—a low adoption rate for threading environments by the MPI users leads to a low interest in

threading optimization from the community of MPI developers. During the length of this study, I also found that there were limited choices of performance evaluation tools for MPI. While the existing tools adequately measure basic communication performance, they fail to capture several aspects of the MPI communication in the multi-threaded environment and are therefore unable to expose its shortcomings. This made the task of optimizing MPI threading performance even more challenging. Furthermore, the lack of interoperability between the threading frameworks and the MPI implementation adds to the existing challenges in performance optimization for threading in MPI.

Therefore, in this study, I work to optimize the multi-threaded environment from an MPI implementation standpoint. This is done by starting with a thorough investigation of prior studies on MPI, finding areas where existing MPI implementation cannot perform up to a satisfactory mark in a multi-threaded environment and thus require improvement, and, finally, design and implementation of solutions that enhance the performance of MPI in a multi-threaded environment.

1.2 Contributions

In this study, I contribute to the optimization of multi-threaded MPI in two broad ways: (1) By introducing a new tool for evaluating performance of an MPI implementation, a tool which addresses some of the shortcomings of the existing benchmarks and (2) by proposing, implementing, and analyzing a set of threading performance optimizations in a particular MPI implementation, Open MPI. The latter covers a number of novel and non-trivial strategies that highlight portable ways to fully utilize thread parallelism in an MPI implementation.

Enhancement of Performance Evaluation Tools for MPI

I address the lack of flexibility in performance evaluation tools for multi-threaded MPI and increasing the number of powerful toolkits with various capabilities of performance assessment. This is done by, first, investigating the currently available performance measurement tools for multi-threaded MPI, to assess their strengths and weaknesses. A

major challenge of using the existing benchmarks is that they offer very limited capabilities—for example, the lack of ability to adjust the workload or communication pattern. In order to better evaluate the multi-threaded performance, these gaps need to be overcome. This research proposes a solution–the Multirate benchmark Patinyasakdikul et al. (2019), which allows users to evaluate benchmarks on various aspects. This highly flexible benchmark exposes the shortcomings of a multi-threaded implementation of the MPI through multiple communication patterns and flexible workload, allowing MPI developers to quickly compare performance in a threading environment to a non-threading environment. The potential of the proposed benchmark is demonstrated by evaluating the current state-of-the-art MPI implementations such as Intel MPI, Open MPI and MPICH. I strongly believe that this will enable the current and future MPI developers to more efficiently optimize their MPI implementation.

Strategies to Optimize Threading Performance in MPI

I propose several strategies to optimize the threading performance by addressing the lack of interoperability between the MPI implementation and the threading frameworks. I also propose a novel design with different approaches to harness the power of thread parallelism for the MPI implementation.

- **Better Thread Synchronization:** Current thread synchronization approaches in modern MPI implementations are highly inefficient, resulting in threads being inadequately organized. This creates unnecessary contention in the critical MPI components, which results in time wasted causing degradation in overall performance. To equip the MPI implementation with more control over user-level threads and give the MPI implementations more opportunity to optimize for threading support, I introduce the concept of the thread synchronization object. The benefits of my design is demonstrated by employing the synchronization object to commandeer the access to the MPI progress engine, thereby reducing the unnecessary lock contention from the original approach. I also demonstrate other potential use cases of the thread synchronization object in real-world applications.

- **Better Resource Management:** I propose and implement a design of Communication Resource Instances (CRIs) Patinyasakdikul et al. (2019) for an efficient allocation of resources. I studied the impact of resource contention in multi-threaded MPI in the current design which led to the discovery of a number of shortcomings. This acted as an inspiration to come up with solutions that can fill the voids that the legacy resource allocation strategies have left. My implemented design provides MPI with a simpler design to allocate more resources for threads and help alleviating the resource contention in the MPI implementation. I also propose multiple strategies to incorporate CRI into MPI core functionality to extract more performance from threads. I discuss its impact on both one-sided and two-sided communication in multi-threaded MPI.

- **Optimization Suggestions:** I summarize my study and propose a compiled list of suggestions to both the MPI developers and the MPI users to fully harness the power of threading in MPI. The optimization presented in this dissertation has been incorporated into the Open MPI development branch.[1] It is to be noted that from a software perspective, all my optimization work has been accepted by the Open MPI community and released publicly with Open MPI 4.0.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows:

- Chapter 2 provides the high-level details from sections of the MPI standard which are related to the focus of this study. I discuss the trends from the prior studies around the topic of the use and the optimization of the threading environment in the HPC community, and ultimately, in the MPI communication.

- Chapter 3 discusses the the motivation for my proposed multi-threaded MPI benchmark along with its design, and evaluate its capabilities by performing measurements on the current state-of-the-art MPI implementations, assess their strengths and weaknesses, in both threading and non-threading environments.

[1] https://github.com/open-mpi/ompi

- Chapter 4 presents my design and implementation of the thread synchronization object, and illustrates how the design can provide the MPI implementation with more control and better utilization over the user-level threads, and the ability to redirect them. I demonstrate the potential of the thread synchronization object design, evaluate its performance and discuss ongoing research collaborations originated from this design.

- Chapter 5 presents my solution for better resource allocations in threading environments through the design of the CRIs, along with the assignment strategies for better thread concurrency in different levels of the MPI implementations. I evaluate the CRI implementation in both one-sided and two-sided communications. I discuss my findings and provide additional suggestions for optimizing the MPI in multithreaded environments.

- Chapter 6 concludes this dissertation with the summary of my findings, with suggestions for the multi-threaded support in the MPI, and I discuss my future directions.

Chapter 2

Background and Literature Review of Related Works

2.1 Overview

This chapter presents the high-level background knowledge related to the topics of this study. It focuses mainly on the basic point-to-point communication, both one-sided and two-sided. I discuss the MPI standard, especially the threading support of MPI, along with the high-level design of Open MPI, an open-source MPI implementation used as the base MPI implementation for this study, and finally, present the prior studies of several aspects, challenges and proposed solutions for optimizing multithreaded performance in MPI.

2.2 The MPI Standard

The first MPI standard Forum (2015) was published by the MPI forum in 1994 as a revolutionary programming paradigm for high-performance computing. The MPI forum is continuously maintaining and releasing the new specifications and adds more functionality to the MPI API to the present day. The current version of MPI standard as of this study is MPI standard 3.1, published in June of 2015. The threading support in the MPI standard was originally not well defined. The official threading support from the MPI standard begins from the standard version 2.1 in September, 2008. In this section, I discuss the MPI standard

API of interest for the scope of this study, from point-to-point communication to the newly added one-sided communication, and the multithread environment supports from the MPI.

2.2.1 Point-to-Point Communication

The point-to-point communications are the communication between two MPI processes. The operation involves sender and a receiver, always in matching pairs. It is the most basic form of communication defined with the original MPI standard from 1994. Other than the user-level API, point-to-point communication also serves as the bedrock to a more sophisticated communication provided with the MPI standard, such as collective operations. It is important to optimize the performance of point-to-point communication as it may translate into better performance overall for the MPI implementation.

The MPI standard provides multiple flavors of the API for point-to-point communications, allowing the user to be more specific on the behavior of the communication and optimize the application to their needs. However, in this study, I only discuss the most basic point-to-point communication API.

Send and Receive API

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

int MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request)

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

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

The MPI standard provides message matching by tag and also guarantees that every message will be received in a FIFO order. In short, the first send will always get matched

with the first receive of the same tag, relieving the user of the burden of tracking message sequences.

The API comes in two modes: synchronous and asynchronous (asynchronous API has a prefix of 'I'). Some might refer to synchronous as blocking, and asynchronous as non-blocking, due to its behavior. As the name suggests, the synchronous call waits until the operation is completed, at least locally, before returning from the call, while asynchronous only issues the intent for communication and return to the user immediately. The asynchronous API gives the user a "request," an opaque handle associated with the operation for the user to check for its completion later with MPI_Wait or MPI_Test variants. The asynchronous API allows for more flexibility of the application as the user can ask for the message completion only when it is needed, and avoid the implicit synchronization that usually comes with the synchronous (blocking) communication while also provide the possibility of the overlap between computation and communication.

Wait and Test API

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])

int MPI_Waitany(int count, MPI_Request array_of_requests[],
                 int *index, MPI_Status *status)

int MPI_Waitall(int count, MPI_Request array_of_requests[],
                 MPI_Status *array_of_statuses)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

int MPI_Testsome(int incount, MPI_Request array_of_requests[],
```

```
                int *outcount, int array_of_indices[],
                MPI_Status array_of_statuses[])

int MPI_Testany(int count, MPI_Request array_of_requests[],
                int *index, int *flag, MPI_Status *status)

int MPI_Testall(int count, MPI_Request array_of_requests[],
                int *flag, MPI_Status array_of_statuses[])
```

For asynchronous operation, the MPI standard provides two major ways of checking for completion through *wait* and *test* API. Similar to the point-to-point API, the *wait* API is a synchronous routine and will only return when the condition is met (number of completed requests) while *test* is an asynchronous routine, which will return immediately but provide the means for the user to get the information of the requests. The MPI standard offers 4 flavors of the wait/test operation, a single request and multiple requests (some, any, all). As the name suggests, for *'some'*, the user can test for a subset of requests by providing the desired number with the API. The user can check for the completion of one or more requests through *'any'* API and completion of every request through *'all'* API.

2.2.2 One-Sided Communication

In addition to two-sided communication, the MPI-2.1 standard provides support for one-sided RMA communication. This support allows an MPI implementation to directly expose hardware Remote Direct Memory Access (RDMA), a feature which is present on some high-performance networks, e.g., Infiniband, and Cray Aries. This allows the MPI implementation to offload communication directly to the hardware. In addition, the one-sided model separates communication (data movement) from the synchronization (completion). The standard defined API for one-sided communication are the following.

Window Initialization/Finalization

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
```

```
                MPI_Info info, MPI_Comm comm, MPI_Win *win)
int MPI_Win_free(MPI_Win *win)
```

Data Movement

```
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win win)

int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win win)

int MPI_Accumulate(const void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

Operation Completion (Examples)

```
int MPI_Win_flush (int rank, MPI_Win win)
int MPI_Win_flush_all (MPI_Win win)
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
int MPI_Win_lock_all(int assert, MPI_Win win)
int MPI_Win_fence(int assert, MPI_Win win)
```

The API offers three ways of data movement. MPI_Put and MPI_Get offer remote write and read operation while MPI_Accumulate allows the user to perform atomic mathematical operations such as addition or multiplication on the target buffer. The operations are carried out by the source without involving the target. However, since anyone can read or write to the same target buffer at any time, the user is responsible for keeping track of their data accessing order through the synchronization.

The MPI API provides the means of synchronizing through an abstraction of 'window'. Every MPI process exchanges the necessary information beforehand at the window creation through MPI_Win_create. The example of the window operations are listed above. The entire window operation API is listed in the appendix C. Unlike the two-sided communication, the information of peers and target buffer is already exchanged in this window creation process. Hence, the message matching operation is no longer required after the actual communication. There are multiple flavors of synchronization on the window, but in short, performing any synchronization operation on the window will complete outstanding operations associated with that window. For example, fence completes every outstanding operation on the window for every peer, while lock completes the operation for between the calling process and the target process only.

2.2.3 Threading Support

The MPI-3.1 standard Forum (2015) provides four levels of threading support. During MPI initialization, more precisely during MPI_INIT_THREAD, users can marshal with the MPI implementation the desired thread level for the application.

- MPI_THREAD_SINGLE: The most restrictive setting where a single thread must exist in the application, independent if they make MPI calls or not;
- MPI_THREAD_FUNNELED: Multiple threads can coexist in the application—but only one, the master thread (i.e., the one that initialized the MPI library), is allowed to performing MPI operations;
- MPI_THREAD_SERIALIZED: Multiple threads can coexist in the application, and the application is responsible for serializing their MPI calls, in order to guarantee that only a single thread will perform MPI operations at any time;
- MPI_THREAD_MULTIPLE: Multiple threads exist in the application and every thread can perform MPI operations at any time, without restriction on the ordering or serialization.

From a purely pragmatic point of view, most of these thread support levels have little reason to exists nowadays, but it is acknowledged that it might be needed on some esoteric hardware, with extremely restrictive thread support from the operating system. Anyhow, with the current hardware architectures, there is no possibility of race conditions for the SINGLE, FUNNELED, or SERIALIZED mode. Thus, current MPI implementations are not providing any protection for these modes as there is no need to spend the unnecessary cost. Thread safety is therefore only provided if the user initializes MPI with MULTIPLE. This study is focused only on the MPI_THREAD_MULTIPLE mode, as it is the only mode that allows for thread concurrency.

The benefit of multi-thread environments has been explored since its inception with the MPI-2.1 standard. One of the proposed benefits is to decrease the memory footprint of the MPI application. By taking advantage of thread memory space, every thread in the same process can access the same space of memory, reducing the need for multiple copies of the same data. Moreover, utilizing the multi-thread environment reduces the need for intra-node explicit communication as the threads can simply access other threads' memory. Another benefit of threading is to increase the throughput for messages of smaller size. As MPI implementations are highly optimized for sending large messages through sophisticated pipeline algorithms, sending a high volume of smaller messages is still a challenging aspect to optimize for. This study contributes to the efforts to improve small message throughput by utilizing threads to send multiple messages in parallel.

2.3 The Open MPI library

While this study is generic and can be applied to any MPI implementation, all designs and engineering aspects were implemented in Open MPI. In this section, I present multiple aspects of interests of Open MPI for this study. The Open MPI Gabriel et al. (2004) is one of the MPI implementations that is fully compliant with the MPI standard 3.1. It is an open source MPI implementation that, like most open-source projects, is developed and maintained by an active community of volunteers, the Open MPI community. This community consists of a large number of volunteers, together with participants from a variety

of organizations from both academia and industry. With its open-source nature, Open MPI has been used as the base of multiple vendor's MPI implementations such as Fujitsu, Bull, and IBM Spectrum MPI, driving some of the most powerful supercomputers in the world at the time of this study, *Summit* Vazhkudai et al. (2018).

2.3.1 Modular Component Design

Open MPI employs a modular design where multiple components can work independently through a well designed, standardized API with frameworks (Figure 2.1). The components can be plugged easily into a framework as long as they provide the necessary API for the framework to operate. The framework-component design allows for multiple implementations of the same functionality. For example, the *coll* framework provides the functionality of collective operations. Any developer can create a component under *coll* to plug their implementation of collective operations such as the broadcast or allreduce operation, and have Open MPI invoke their component for the MPI_Broadcast or MPI_Allreduce call.

Operating in a multi-threaded environment presents a different level of challenges to the Open MPI components, as the individual components serve in a different capacity, and in some cases, are designed to interact with the different set of hardware with different capabilities and limitations. This study mainly focuses on the multi-threaded operation of 2 major Open MPI frameworks which provide the basic communication support; the Point Messaging Layer (PML) and the Byte Transporting Layer (BTL).

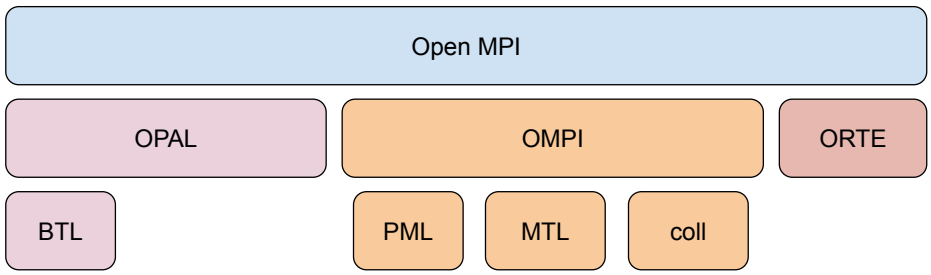


Figure 2.1: Example of Open MPI framework layers.

Point Messaging Layer (PML)

PML is a framework for point-to-point data movement. The framework provides functions such as send, receive, and message matching for the Open MPI. Currently, there are multiple components in the PML framework such as UCX and OB1 (Obiwan). This study focuses mainly on *pml/ob1*, as the OB1 component utilize the BTL framework for the actual data movement while the OB1 itself provides high-level operation such as message matching, message pipelining, and sequencing. By utilizing the BTL framework, OB1 becomes the component that provides the high-level algorithm design and allowing other developers to easily plug their network implementation with only basic data movement functionality.

Byte Transporting Layer (BTL)

The interaction between the BTL and pml/ob1 is illustrated in Figure 2.2. The Byte Transporting Layer is a framework with only basic data movement capabilities such as memory allocation, send, and read for message completion. It is designed to work with a higher level framework such as pml/ob1. The BTL itself does not have the context of any message that it is sending or receiving, and it will let the higher level handle the completion of the message. The simplicity of the BTL framework allows multiple network hardware vendors to create their basic component and easily integrate it into Open MPI without implementing the entire process of MPI communication such as message matching and ordering guarantees. The example of the components in the BTL frameworks are: *btl/tcp* (socket communication), *btl/ugni* (Cray's GNI), *btl/openib* (ibverbs), *btl/vader* (shared-memory communication).

2.3.2 Progress Engine

The MPI standard does not provide the explicit API for progressing operations, but most underlying network protocol requires an explicit progress routine. The MPI implementation, as a middleware between the network protocols and the user, has to provide a solution to comply with the standard. Most MPI implementations address the progression by creating a centralized routine—the progress engine for progressing the communication of network protocols and also internal MPI events. The standard mentioned that the user can expect the

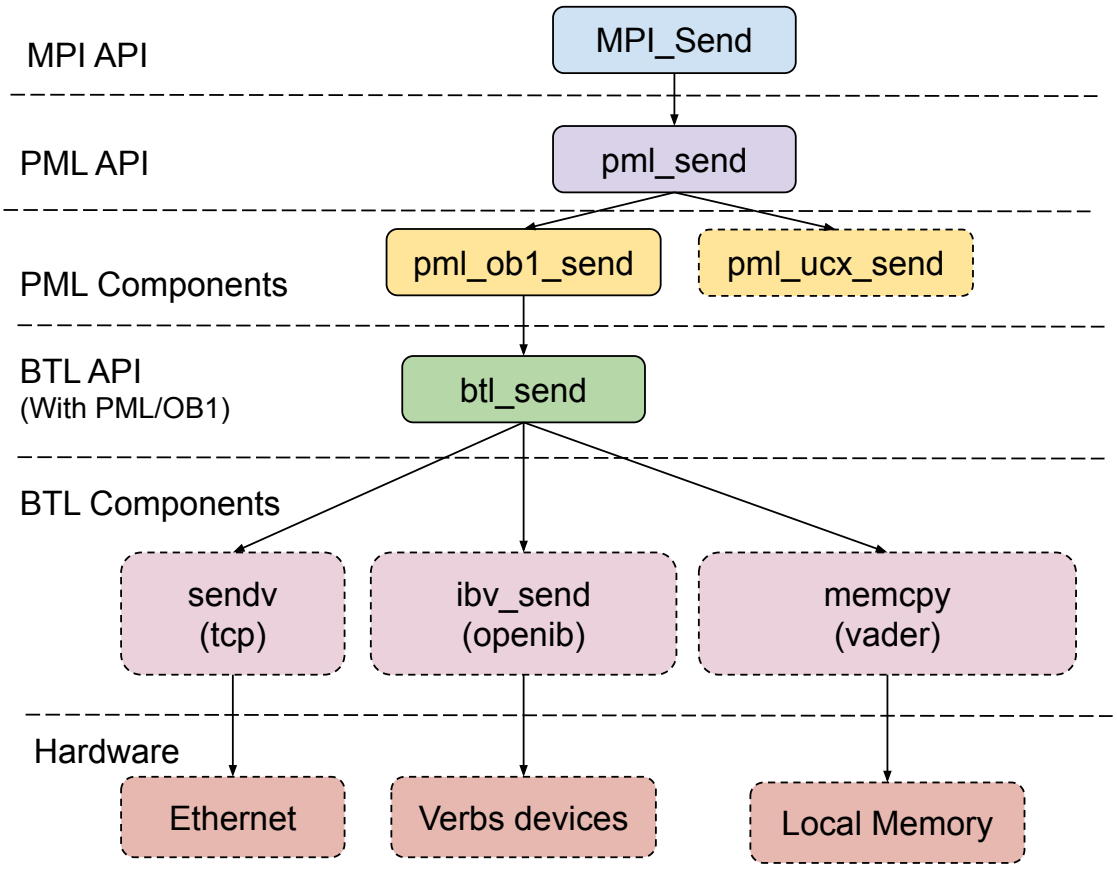


Figure 2.2: Interaction between BTL framework and pml/ob1 in Open MPI.

progression of their asynchronous operation with every call into the MPI library. However, the decision to invoke the call to the progress engine is entirely up to the MPI implementation. Usually, in the blocking operation such as MPI_Recv or MPI_Wait the MPI implementation will keep invoking the progress engine until the associated operation is completed.

For Open MPI, every component registers their progress routine to the centralized progress engine (OPAL progress). When invoked, the progress engine will execute every registered routine. Figure 2.3 illustrates the example of components who register their progress routine to the Open MPI progress engine. For example, the BTL has its network polling or reading for message completion registered to the progress engine. Since the progress engine is involving handling the message completion event, it plays a crucial role in the message receiving path of MPI, as the matching process is mostly performed in this part. The details of the message matching process is discussed in the next section.

In multi-threaded environments, Open MPI ensures thread safety by serializing the call into the progress engine to eliminate the possibility of any race condition that might occur. The serialization is implemented by a process-wide lock, *progress lock* with the pthread condition variable to synchronize between threads.

2.3.3 Matching Process

For two-sided communication, MPI offers tag matching in the standard as a means to pair a sent message with an expected reception, instead of working with a simple stream of data. The tag matching provides the user with better control over their communication, as they can use the tag to distinguish between different messages with the same size or use them as a label for each message. The matching process can occur at two points; When the user posts a receive (through MPI_Recv, MPI_Irecv and other variations), and when the MPI process extracts a message from the network. Therefore, the matching process involves two queues, expected and unexpected.

Figure 2.4 shows the matching process implementation. In the case of the user posting receive, the MPI will search for the message in the unexpected queue, a buffer queue when the message arrives before the receiver posts the receive. If there is a match, the receive operation is complete immediately. If not, the request will be added to the expected queue

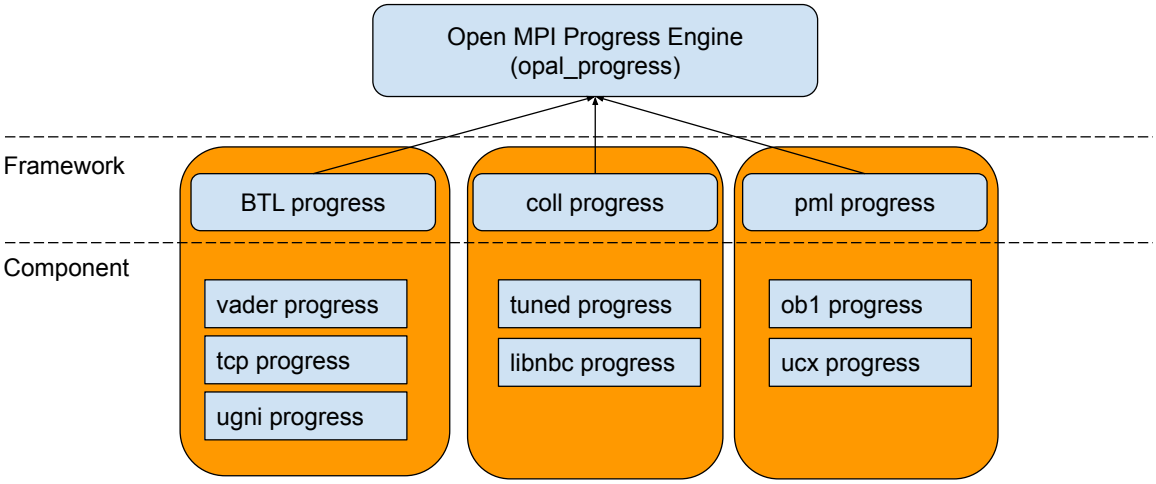


Figure 2.3: Every component registers their progress routine to the progress engine.

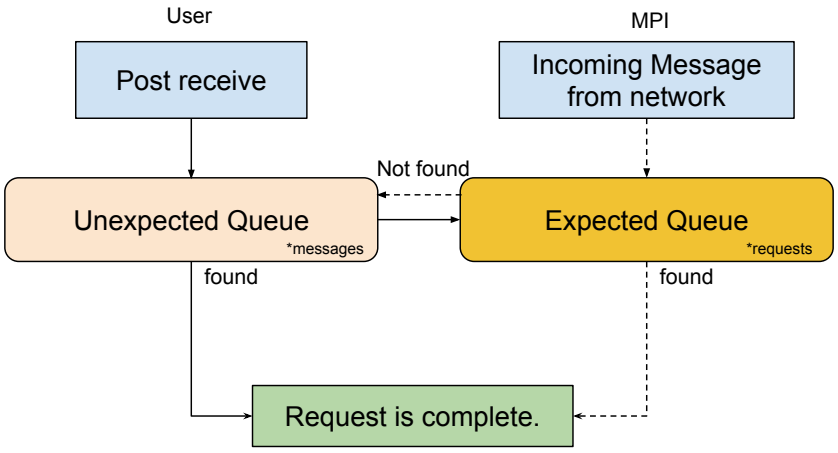


Figure 2.4: Matching Process implementation

to match with incoming message later. On the other hand, in case of receiving an incoming message from the network, the MPI implementation will try to match the message's tag with the request in the expected queue. If there is a match, then the matched request is marked as completed. If not, the message will be added to the unexpected queue for matching with the request later.

The MPI standard also guarantees that the message will be matched in the FIFO ordering. In short, the MPI will match the message by the order of the posting of the reception, as well as the order of the sending. For example, if the message has the same tag, the first receive will always get matched with the first send, the second receive with the second send, and so on. This guarantee provides the simplicity for the user to program. However, there is no such guarantee from the network level as to whether implementing the ordering from the hardware level will hinder the overall hardware performance capacity. Thus, the MPI implementations have to provide the software solution for the ordering guarantee to the user. The message ordering implementation is different for each MPI implementation, but the general idea is to issue a sequence number for each message. The sender will include the sequence number with the message header, and the receiver can check if the message's sequence number is expected. By imposing the monotonically increasing sequence number on both sides, the MPI implementation can ensure that the message is matched in the described FIFO order.

Figure 2.5 illustrates the matching process implementation with the FIFO ordering guarantee by utilizing the message sequence number. When the receiver receives an incoming message with non-FIFO ordering (incorrect sequence number) from the network, the MPI buffers the message and adds it to the out-of-sequence queue without performing any matching operation on it. Every time the MPI successfully matches a message to a request, the anticipated sequence number changes. Before returning to the user, the MPI implementation will try to search for the message with the anticipated sequence number in the out-of-sequence queue. If the message with the anticipated sequence number is found, the usual matching process is performed on the message. There can be two outcomes: if the user posted receive for that message, the request associated with the receive is marked as completed; if not, the message gets moved into the unexpected queue.

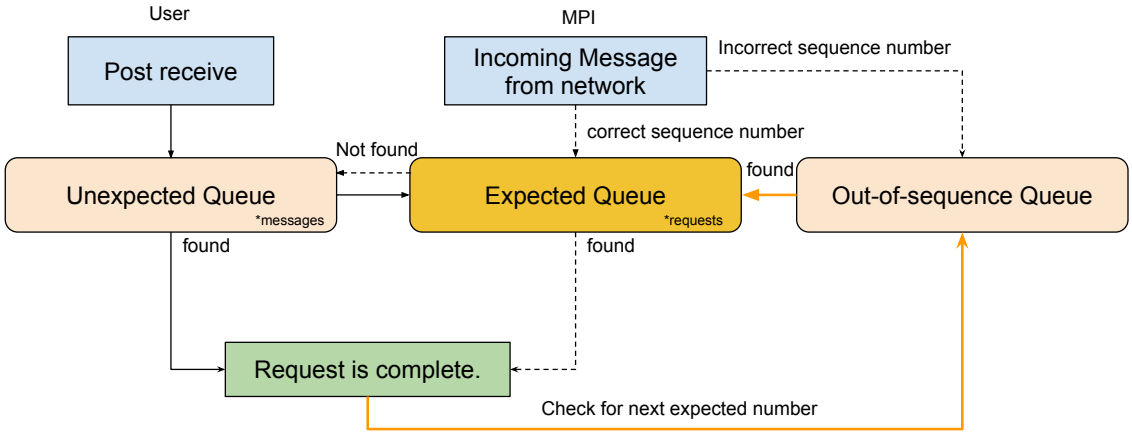


Figure 2.5: Matching process implementation with ordering guarantees

Usually, buffering the message in the unexpected queue or out-of-sequence queue is a costly operation, as the MPI implementation has to allocate proper memory to store the message in the middle of the time-critical message extraction process, but the occurrence is expected to be very minimal as the network devices rarely rearrange the send order under normal circumstances. Another interesting metric for the matching process is the unexpected queue length, as the matching operation is essentially a serial queue search, where the cost of searching the queue grows with the length of the queue itself. Therefore, applications with a high volume of unexpected messages (not pre-posting receives) will be susceptible to longer matching time.

There are more complications in the multi-threaded scenario as the matching process has to be serialized, at least per communicator, to ensure correctness. The critical parts, such as adding or removing an object from a queue, cannot be performed concurrently. In the context of MPI implementation, two threads can post a receive at the same time, or one thread is inside the MPI progress engine receiving the incoming message while another thread is posting the receive. The MPI implementation usually protects the matching process with a process-wide lock or *matching lock* to serialize the access to all queues.

2.4 Literature Reviews

Multiple studies have been conducted investigating ways to improve the efficiency of multi-threaded MPI. There are several moving parts contributing to the abysmal performance of multi-threaded MPI implementations. This section discusses several research topics related to multi-threaded MPI communication and the efforts to improve its performance from several perspectives.

Threading Frameworks and HPC

There are research interests in threading performance optimization in HPC other than the use of the classic Portable Operating System Interface (POSIX) threading framework such as pthread Lewis and Berg (1998). OpenMP Dagum and Menon (1998) is the threading framework that provides the high-level abstraction that is user friendly but still based on the fork-join programming model. However, the OpenMP specifications in recent years added more support for current trends in HPC such as tasks and job-stealing design. From the large-scale parallelism perspective, Wheeler et al. (2008) proposed Qthread, a lightweight, portable threading framework, which is more suitable for the HPC environment with smaller memory footprints per thread, with lightweight thread control and synchronization while not relying on any specific hardware or the platform capability, allowing for more scalability and portability of the multi-threaded HPC applications. In the same year, Intel released its Thread Building Block (Intel TBB) Pheatt (2008) which presented the similar idea of threading and its portability by providing its own user-level threading runtime that evaluates the system it is currently running on, and performs the load balancing for threads accordingly.

In recent years, MassiveThreads Nakashima and Taura (2014) presented the threading framework with the same API as pthread but offer more delicate thread management design for better load balancing and workload prediction to avoid the high cost of unnecessary context switching, resulting in lower overhead compared to the existing solution such as Qthread. Argobots Seo et al. (2017) are refining the approach of the lightweight threading framework, which directly focuses on the HPC usage while providing the high-level threading

capability on its own, or it can be used as the low-level threading infrastructure under other threading frameworks.

The works presented in this section show high interest in incorporating the multi-thread environment into the HPC landscape with an increasing number of CPU cores per chip. This naturally calls for the combination of MPI and threads (MPI+threads) to handle larger scale applications where MPI is used for inter-node communication, while using threads to perform the computation task locally on the node, and spurs interests in optimizing thread support in the MPI.

Lack of Interoperability Between the MPI and the Threading Frameworks

The MPI standard does not provide an API for the MPI implementation to exchange the information with the threading frameworks of the application. The MPI implementation cannot differentiate between threads that make calls into the MPI library, thus limiting the optimization strategies that it can employ. Si et al. (2014) proposed interoperability between the de facto OpenMP threading runtime and the MPI implementation to share the information of the threads between the application and the MPI layer in many-core architectures. When equipped with threading information from the threading framework, MPI can incorporate the idling application threads for internal communication without user intervention, allowing for better computing resources utilization.

On the other hand, there is a movement to extend the MPI standard to address this interoperability problem. The study of Sridharan et al. (2014) and Dinan et al. (2014) shows interest in creating multiple endpoints per MPI process for multiple threads to communicate. This is a step towards better interoperability by providing the user with a standardized way to provide hints to the MPI implementation for better optimization. The MPI implementation can be utilizing this information for better resource allocation internally. Unfortunately, at the time of this study, the MPI forum has not yet accepted the proposal, but the MPI implementation can still offer the non-standardized way for the user to provide hints such as using environment variables, among other means. Nonetheless, I believe that interoperability between the MPI implementation and the threading framework is desperately needed for better performance optimization of multi-threaded MPI.

Resources Contention in Multi-threaded MPI

First, the resource contention in multi-threaded environment: it has been identified as a major roadblock that prevents the MPI implementation from harnessing true thread parallelism. The MPI implementation has to create a critical section—a section that can be executed by a single thread at a time, to prevent the race condition that might occur when multiple threads are making updates to the same memory location. The race condition from threads can affect the correctness of the communication, or even escalate to corrupt the state of the MPI software and crash the whole application. Thus, the critical section also becomes a funnel that mitigates the potential performance gain from thread parallelism. Balaji et al. (2008) study multiple granularities of the critical section in MPI by simulations with MPI_PROC_NULL (no actual data movements), and suggested that coarse-grain critical sections such as global lock—even for a short duration per-thread, per-access—can create bottlenecks that significantly affect the overall communication performance. The study also suggested MPI implementations should shift towards per-object lock or even lockless data structure to avoid the massive lock contention.

Amer et al. (2015) suggested another approach to minimize the lock contention by creating a systematic mechanism to assign the resources to a thread. Their study imposes priority to threads, and managing the load balancing between threads by several strategies. The study suggested that multithreading can benefit from a well-designed load balancing algorithm. However, it has been noticed that the MPI standard does not provide any interoperability between the MPI and the threading framework, which might render the load balancing more challenging for the MPI implementation. In another study, Goodell et al. (2010) illustrate the impact of an often overlooked aspect: the reference counting of the object. Reference counting is a common practice to track the usage of a shared object by a simple counter addition and subtraction. Once the reference count of an object becomes zero, the object is marked for garbage collecting purposes. However, the cost of reference counting increases drastically in multi-threaded environments, as simple addition becomes an atomic operation. Goodell's study proposes a design to reduce the cost of the reference counting and demonstrates its impact on overall MPI communication. Dózsa et al.

(2010) proposed a design of multi-channel communication for MPI; the MPI implementation creates multiple channels for communication with different peers, allowing multiple threads to perform multiple communications in parallel. Their study paves the way for achieving true thread concurrency in MPI implementation, and suggested that the MPI implementation has to implement a way to extract the incoming message in parallel. The study uses a parallel receive queue for multiple threads to drain at the same time; the performance evaluation in their study claims good performance scaling with an increasing number of threads.

There has been a trend to completely avoid resource contention by delegating every communication to a dedicated communication thread. By using a single thread to communicate, the application does not have to initialize the MPI library in multi-threaded mode, thus removing all the cost of thread safety that comes with it. There are several ways to achieve this model. The user can manually program their application accordingly, or there can be a middle layer between the user and MPI such as Vaidyanathan et al. (2015)'s study. Vaidyanathan proposed software offloading, intercepting every MPI call from user threads, and funneling them into a dedicated communication thread via lockless command queue. While the study cannot fully utilize the thread parallelism for communication, it shows significant improvement over the current coarse-grain critical section design in some MPI implementations. Grant et al. (2015) proposed another approach to avoid the resource contention by accumulating smaller messages from multiple threads into a large buffer, and use a single thread to perform the communication, utilizing the highly sophisticated message pipelining mechanism on the larger message to achieve better performance while avoiding the unnecessary contention. The work shows an impressive performance boost; however, it requires user-level involvement in the initialization stage.

Progress Threads

Another trend in utilizing threads in MPI is the *progress thread*. The main attraction of the progress thread design is computation and communication overlap. There is a common misconception related to asynchronous communication in MPI, as the users expect the MPI to progress the communication in the background. In reality, the MPI has to explicitly progress the network through different protocols to get completion and MPI itself cannot

do that in the background. Currently, for MPI to progress the communication, the user has to make a call into the MPI library to give the chance for the MPI implementation to execute the progress engine. The progress thread approach is a design where there is a thread executing the progress engine in the background for the MPI implementation to progress the outstanding communication while the user thread is executing application code, providing the overlap between the two.

That being said, the progress thread approach often comes with the design question of where the MPI implementation should bind the progress thread. If the progress thread shares the physical CPU core with the application, it takes crucial computation power away from the application; but if the progress thread is bound to a dedicated physical core, there will be no interference with the application, but every MPI process has to take additional CPU cores and ultimately cut the computation power of the application in half. Hoefler and Lumsdaine (2008) experimented with multiple strategies of progress thread design and proposed a shared core design with low performance impact on the application while providing a good percentage of overlap. Wittmann et al. (2013) uses the standardized PMPI interface to implement the progress thread from outside of the MPI implementation, allowing a portable progress thread implementation for communication and IO overlap. Lu et al. (2015) proposed a design to utilize user-level threads as a temporary progress thread, circumventing the need for MPI implementation to spawn the progress thread by itself and avoid the resource management problem entirely. The work of Potluri et al. (2010) demonstrates the application-level benefit from communication overlap in real-world seismic modeling applications.

Matching Process

The matching process is another critical piece of the two-sided communications infrastructure (send and receive), which is the bedrock of the MPI communication. While this study does not contribute to the topic of matching process optimization, multiple challenges in this study stem from the current design of the matching process. This section presents the efforts from the HPC community in optimizing the matching process by improving its efficiency and releasing synchronization constraints.

Brightwell et al. (2005) and Ferreira et al. (2018) present the characteristic of real-world applications regarding the usage of the MPI matching queues (expected and unexpected queues) and its effect on the overall performance. Brightwell suggested that one of the interesting metrics is the queue length, as the cost for the MPI implementation to search through the queue increases linearly with it. Since the matching process is mandatory for two-sided communication, speeding the process up will be beneficial to both single- and multi-threaded modes of the MPI implementations.

There are multiple studies of matching process optimization in the past. Flajslik et al. (2016) suggested a binned matching algorithm to alleviate the contention of the matching process. The study presents a significant speedup over the traditional matching process and suggested that the approach can be easily adapted for multi-threaded environments as they are more fine-grained and suitable for per-object lock. Schonbein et al. (2018) proposed the use of Intel vector instruction to implement a fuzzy matching algorithm for the global matching queue of MPI. This, in turn, allows multiple messages matching at the same time and shows vast performance improvements. Bayatpour et al. (2016) proposed an adaptive algorithm for the tag matching to use different tag matching designs for different workloads.

Another interest in optimizing the matching process is the possibility to offload it to the hardware, relieving the MPI of the matching duty entirely, and supposedly speeding up the entire communication process. The work of Underwood et al. (2005) pioneered the concept of specialized hardware accelerated queue in MPI. Moreover, Hemmert et al. (2007) and Gupta and Abels (2006) demonstrate the benefit of moving the matching process entirely into the network hardware itself. Currently, we can see several high-performance network hardware vendors such as Mellanox, Cray, and Intel incorporate the tag matching capability into their hardware design. While offloading the tag matching process to the hardware can be beneficial for the MPI implementations, it still poses some set of limitations such as the lack of the possibility to cancel the messages correctly, or the ability to split the matching process for different MPI communicators.

One-Sided Communication

This dissertation focuses mainly on two-sided communication, but also touches the topic of multi-threaded one-sided communication in Chapter 5. The past studies on one-sided communication emerged as soon as the MPI-2.1 standard was published. Barrett et al. (2007) studied several approaches to implementing the one-sided communication support in Open MPI. Barrett's study suggested that if the network hardware is equipped with remote memory access capability, one-sided communication can achieve higher bandwidth and lower latency than the two-sided communication in some cases. In the same year, Gropp and Thakur (2007) studied one-sided communication performance on various platforms with various MPI implementations, and their study also gives the conclusion in the same direction as Barrett's. However, the MPI-2.1 standard poses some limitations for one-sided communication as it is written vaguely. In 2009, Tipparaju et al. (2009)'s study proposed the improvements in one-sided communication to the MPI standard which resulted in MPI-3.0 standard. Kumar and Blocksome (2014) implemented the MPI-3.0 one-sided communication on Blue Gene/Q computer and evaluated the performance. Their study concluded that MPI-3.0 one-sided communication has lower latency than the older MPI-2.2 standard while performing at the same level as two-sided communication. In their remarks, they also show interest in extending the support for multi-threaded one-sided communication and utilizing the internal progress thread to increase communication overlap. In 2016, where several MPI implementations are fully standard compliance, Dinan et al. (2016) presents the implementation and performance evaluation of the MPI-3 one-sided communication and suggested that the new MPI-3.1 standard allow MPI implementation to be fully equipped, and fully utilize the hardware capabilities in remote memory access.

Recently, there is increasing interest in multi-threaded one-sided communication. First, the effort to create a multi-threaded one-sided MPI communication benchmark by Dosanjh et al. (2016) signals the interest of MPI developers in pushing for multi-threaded RDMA support. The follow-up work includes Hjelm et al. (2018) where the authors investigate several techniques to improve one-sided communication in multi-threaded scenarios. Hjelm is also one of the collaborators of this work.

Chapter 3

Measuring MPI Performance

3.1 Overview

This chapter presents my effort to expand the number of powerful tools for the MPI developers to evaluate their MPI implementation. I introduce my novel Multirate benchmark: a highly flexible benchmark, capable of stressing multiple performance points of an MPI implementation. The Multirate benchmark is designed to provide fast assessment and comparison between MPI in a normal process-to-process communication and thread-to-thread communication. This chapter discusses the motivation and design of the Multirate benchmark, and shows that my tool is capable of exposing the optimal and sub-optimal point in the MPI implementations. I demonstrate Multirate capability by evaluating three current state-of-the-art MPI implementations and discuss my findings.

3.2 Introduction

Several MPI implementations are available today, with varying capabilities and support for lesser-used MPI features. While it is tempting to make a distinction between vendor implementations and open-source implementations, the software evolution in HPC has led to only two major flavors, MPICH and Open MPI—both of which are open source—surrounded by a series of derivative implementations with small differences compared with their underlying open-source versions.

Current state-of-the-art MPI implementations are categorized into two groups, vendor-specific MPI with platform optimization and aftermarket support such as Intel MPI, Cray MPICH, or IBM Spectrum MPI, and open-source implementations such as MPICH, MVAPICH, and Open MPI where the community voluntarily maintains and contributes.

Each MPI implementation differs in design, as MPI developers are free to optimize their implementations. A particular MPI implementation might perform well in one aspect but not in others. Moreover, with different communication patterns required for each type of application, it is essential for the application developers to know which MPI implementation gives the best performance for their application.

Benchmarking is one of several approaches that application developers can leverage to quickly evaluate the performance of their MPI implementation and tailor it to suit their needs. It has the added benefit of enabling MPI developers to validate and assess the performance impact of changes made to their implementation and gives them the ability to compare performance between released versions.

There are several approaches to benchmarking the performance of MPI implementations. One is a simple communication test where the benchmark uses basic patterns like one-to-one. By doing so, the benchmark can show the general behavior of the MPI implementation, but the communication patterns in the benchmark may not be similar to the application, which can be misleading in some cases.

Another approach is to imitate a real-world workload by creating a small subroutine that includes real computation and an actual communication pattern from the application itself. This approach can provide a good performance assessment of the interaction between the application and the MPI implementation as the network engine. This approach, however, puts a burden on the application developer to create a benchmark that reflects their workload and communication patterns, which would change for each application and is a non-trivial task.

The MPI standard offers many communication functionalities, and benchmarking every aspect of MPI would be difficult. This work only focuses on point-to-point communication, as this is the most commonly used feature in MPI applications and also serves as a building block for more complicated schemes, like collective operations Luo et al. (2018). The goal is

to offer performance measurements at the point between a simple communication test and an application-specific benchmark. I build a tool that is flexible in communication patterns, workload, and mode of operations, which can be beneficial to both application and MPI developers in evaluating MPI implementations.

I propose Multirate, a flexible benchmark that offers multiple communication patterns that can be mapped to real-world application needs using an adjustable workload to enable developers to perform early assessments of their MPI implementations. Multirate offers three modes of operation—processes, threads, and hybrid—and enables a quick comparison of performance between the three modes. The results can be used to identify possible bottlenecks in MPI implementations, which can be highly beneficial to MPI developers looking to improve multi-threaded support for MPI.

3.3 Background

3.3.1 Metrics

For MPI performance metrics, we are usually interested in bandwidth, latency, and message rate, which most MPI benchmarks—including Multirate—will be measuring. While the instrumentation for every metric mentioned is very similar, each metric can be more or less critical depending on a given application.

Many scientific applications, such as fast Fourier transform (FFT) or a general distributed machine learning framework, usually perform matrix transpose operations with data transfer to their peers after each iteration. The communication workload in this type of application is usually substantial, making it reliant on the network bandwidth, as it reflects the capacity of the network stack. However, for large messages, the long transfer time often overshadows the overhead from MPI. Most MPI implementations should be able to provide comparable performance.

On the other hand, some applications rely heavily on the small to medium sized messages. However, it is very challenging for MPI to reach peak network bandwidth with small sized messages, as the actual transfer time is short and effectively makes MPI overhead the

bottleneck in the overall communication time. Since additional MPI overhead is added with each message, sending multiple small messages will incur more overhead than sending one large message. The ability to transfer multiple smaller messages simultaneously to hide the overhead and increase the communication throughput has been one of the main attractions of multi-threaded MPI (Figure 3.1). Message rate is the metric that involves the size of the message and might be more suitable for measuring performance in some cases. It can also be used to evaluate the MPI overhead directly.

Latency is another important metric for most applications, as the faster the message is received, the faster the application can move into the next stage of computation, thereby reducing overall run time. Latency differences between MPI implementations can often be used to compare the quality of the optimization or the algorithm used in each of case.

3.3.2 Workloads

Communication performance can vary drastically under different workloads. Usually, we can describe the communication workload in two ways: (1) the message size and (2) the number of messages. The message size is an extremely important parameter, used by most MPI implementations as a trigger for different low-level communication protocols (eager vs. rendezvous). To be able to accurately quantify the protocol switching points, most MPI benchmarks offer presets for message sizes or make them freely adjustable in some cases.

The number of concurrent messages is another factor that can affect MPI communication performance. Having a high number of messages posted at the same time might stress MPI's internal message handling (matching costs, the fairness of receives, the fairness of progress) and load balancing capabilities. Some MPI benchmarks offer an adjustable number of concurrent messages in the form of "window size" or the number of posted messages per iteration. Using both message size and window size to adjust the workload can offer a more comprehensive assessment of MPI performance.

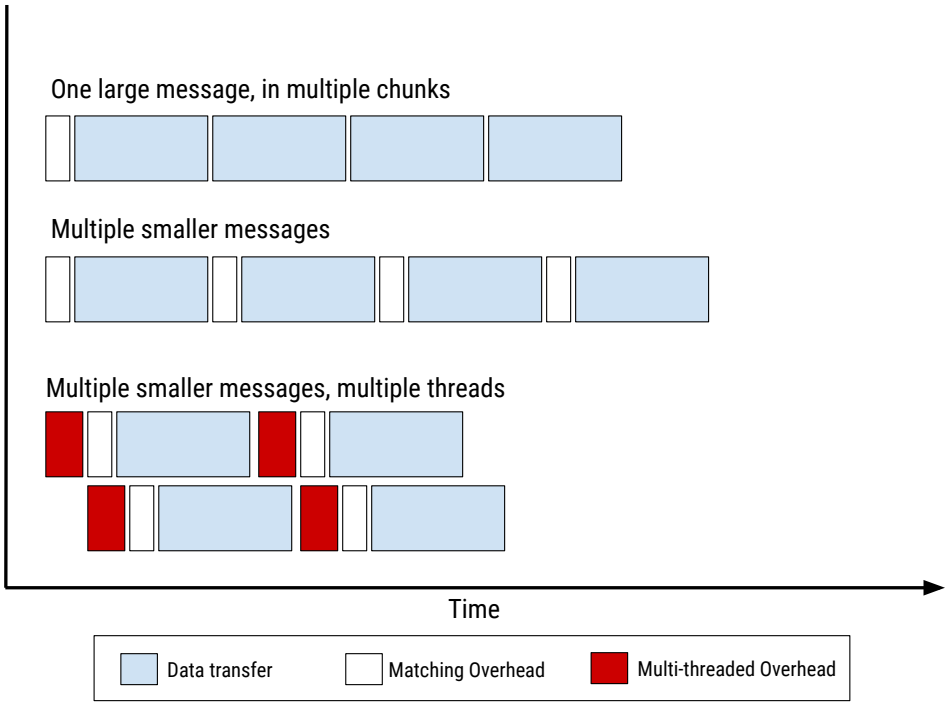


Figure 3.1: The use of multiple threads to increase communication throughput.

3.3.3 Communication Patterns

Most MPI point-to-point benchmarks usually perform communication between two MPI ranks. While this approach certainly gives some insightful performance data on trivial communication patterns, it rarely represents a realistic communication pattern that is used by complex parallel applications.

Communication between two MPI ranks with smaller messages might not be enough to achieve peak hardware bandwidth. However, MPI users can mitigate this problem by using multiple ranks or threads to send messages simultaneously, thereby hiding the MPI overhead by having multiple messages ready to be sent as soon as the network finishes the transfer of the previous message. However, existing MPI benchmarks often do not offer multiple ranks and one-to-one communication patterns in their testing.

Additionally, non-deterministic, thread-based task-based applications, such as MADNESS Thornton et al. or rootSIM Pellegrini et al. (2011), and task-based runtimes, such as StarPU Augonnet et al. (2011) or PaRSEC Bosilca et al. (2013), rely on coordination

messages between peers, which can exhibit dynamic communication patterns between stages of the application and between different executions. A single MPI rank might be the target of messages from multiple peers at the same time, or be one of the sources of messages to a particular target. Similarly, most collective operations also make use of similar point-to-point communication behavior. This behavior can be generalized into simple communication patterns like *one-to-many* or *many-to-one*.

One-to-many and *many-to-one* (Figure 3.2) are interesting communication patterns, as they can be used to test MPI implementations under the asymmetric workload often presented in the real-world application. For example, the stencil, or the halo neighbor exchange, is commonly used in scientific applications. These communications can be mapped into *one-to-many* and followed by *many-to-one*. An imbalance between the message injection and extraction can become a major bottleneck for the overall communication: the sender might inject the message with a higher rate than the receiver can effectively extract from the network layers, introducing significant delays in message delivery to the user level. Performing tests on these patterns can help in exposing the strengths and the weaknesses of the MPI implementations.

3.3.4 Threading in MPI

Ideally, running on the same hardware, MPI with and without threading support should have comparable performance. However, in practice, the cost of serialization from MPI

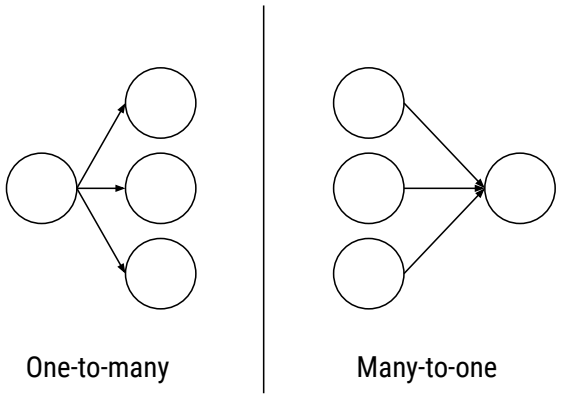


Figure 3.2: One-to-many and many-to-one communication pattern

often overshadows the gained benefits from the implicit intra-node communication. As a result, not many applications are adopting a full MPI threading model, despite significant interest.

The early adoption of the multi-threaded MPI in scientific applications comes after the MPI standard officially defined the multi-threaded environment support in 2008. Shan et al. studied the performance of the MPI threading environment for a large-scale molecular simulation, NWChem Valiev et al. (2010), and concluded that a hybrid combination of processes and threads provides the best performance in this particular application Shan et al. (2015). This evaluation process can be done on a much smaller scale by utilizing a flexible benchmark to represent similar communication patterns without computation.

Currently, there are several multi-threaded MPI benchmarks available in the market. Most of the benchmarks only offer a one-to-one communication pattern and perform the communication in the same way as they do between ranks. The significant difference between process and thread mode in the resources allocated for the operations also needs to be considered. In process mode, each process allocates the resource instance, while thread mode usually spawns multiple threads from a single process, and threads are likely to race against each other for access to limited resources, causing a slowdown.

3.4 Existing Benchmarks

There are several MPI benchmarks already available. NetPIPE Snell et al. (1996) is a set of tools used to measure communication performance. NetPIPE-MPI offers one of the most efficient MPI performance tests for point-to-point communication between two MPI ranks. It also offers bare-bones performance analysis for latency and bandwidth. However, the limitation of NetPIPE is that it only tests communication between two MPI ranks. Sandia microbenchmark Lawry et al. (2002) offers MPI message rate measurement with variable message size, window size, and number of peers, but only offers one communication pattern.

Aside from straightforward tests, several benchmarks (e.g., GRID Boyle et al. (2015) or Parallel Research Kernels der Wijngaart (2016)) offer more specific communication

patterns that are commonly used by scientific applications, including neighbor communication (stencil), parallel matrix transpose, and a distributed linear algebra subroutine. These specialized benchmarks could help application developers make early assessments of their communication patterns by showing them the expected performance from MPI implementations using a practical workload. Several applications from the US Department of Energy's Exascale Computing Project provide proxy applications Proxy, which act as miniature versions of a project to represent the workload, thereby enabling a more accurate performance assessment.

With increasing interest in improving MPI performance in a many-thread environment, several well-known threading benchmarks have emerged. Thread tests Thakur and Gropp is one of the popular MPI multi-threaded benchmarks and offers a fundamental, point-to-point send with the ability to increase the number of threads. OSU microbenchmark OSU offers multiple measurements of point-to-point communications, including bandwidth, bi-directional bandwidth, latency, and message rate with variable window size and multiple communication pairs. However, in threading mode, it only offers latency testing.

3.5 Multirate Benchmark

The goal is to offer performance measurement at the point between a simple communication test and an application-specific benchmark. Multirate is flexible and can adjust the mode of operation (process, thread, and hybrid) and the size of the workload with various communication patterns; this enables application developers to perform an early assessment of their communication needs and provides a quick comparison between different settings for the MPI developer to help him or her identify problems or bottlenecks in the implementation.

3.5.1 Communication Patterns

Multirate provides multiple communication patterns that can be used to map the application behavior or used directly to test capability and identify bottlenecks of MPI implementations. The patterns currently offered are *Pairwise*, *one-to-many*, *many-to-one*, and *many-to-many*.

Pairwise (Figure 3.3) is one-to-one mapping from sender to receiver. The goal of this pattern is to extract the best possible communication performance from MPI with the balanced workload between message injection and extraction. However, for the small-sized messages, using only a single one-to-one communication pair might not be able to achieve the peak network bandwidth. Deploying multiple one-to-one communication pairs concurrently to keep the network hardware occupied might be beneficial to the overall bandwidth utilization.

many-to-one and *one-to-many* (Figures 3.4b and 3.4c) can be useful for detecting bottlenecks in MPI implementations. For example, a single sender to multiple receivers will test the capability of message injection from the sender while mitigating the bottleneck from message extraction, as multiple receivers can split the receiving workload among them. On the other hand, *many-to-one* overwhelms a single receiver with incoming messages from different peers at the same time, solely testing the capability of message extraction. In *many-to-many* (Figure 3.4a), the user can choose any arbitrary number of sender and receiver entities. It is useful for identifying the optimal workload for particular MPI implementations.

3.5.2 Communication Entities

MPI process mapping can be different in multi-threaded mode, as the users are usually advised to map one MPI process onto the entire node or socket (Figure 3.3) and let

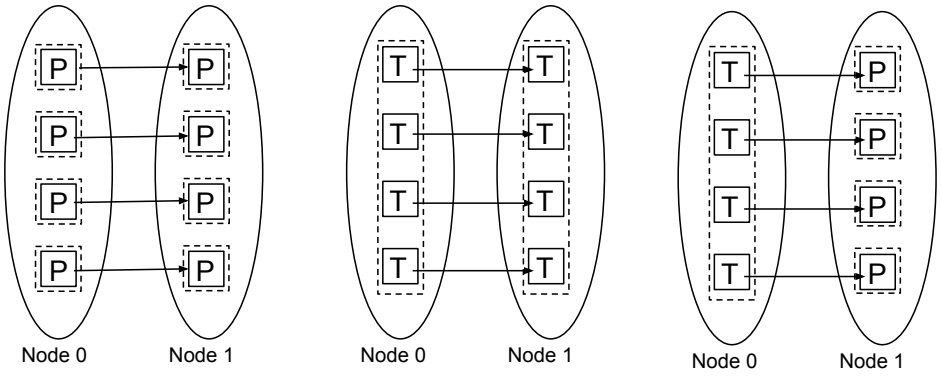


Figure 3.3: CPU core mapping to MPI ranks for pairwise pattern.

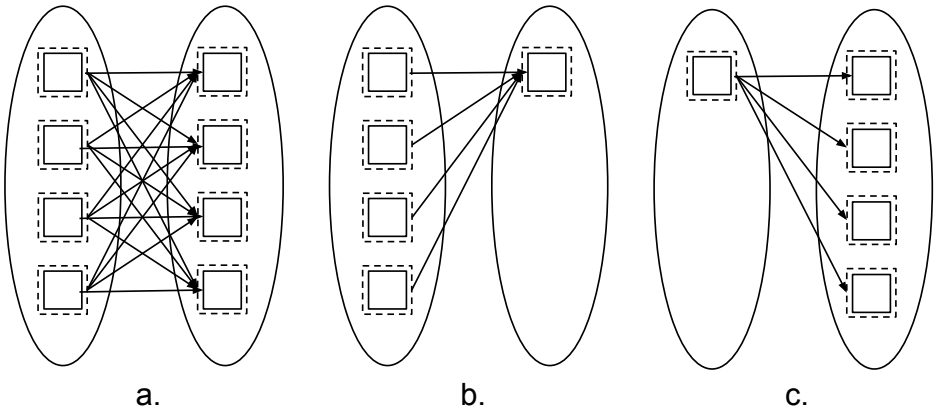


Figure 3.4: All-to-all communication can be used to make sub-patterns, such as *many-to-many* (a), *many-to-one* (b), and *one-to-many* (c).

the operating system schedule threads automatically. The communication entity is an abstraction level that the Multirate benchmark uses to describe a communication body. An entity can be mapped to a single MPI process or to a single thread.

Communication entity abstraction enables Multirate to perform the test in multiple modes of operation, such as thread to thread communication (thread mode), process to process communication (process mode), or combinations of thread to process communication (hybrid mode).

Performance results from process mode and thread mode can be compared to demonstrate the overhead from initializing MPI with full threading support, while hybrid mode can be used to further pinpoint the performance degradation from threading support by fixing one side to process entities and alternating between thread and process entities on the other side, for comparison.

3.5.3 Communicator's Effect

From the user's perspective, the communicator is an MPI-defined abstraction to refer to a group of MPI processes. In each communicator, the participating processes are assigned an individual 'rank' number related to that particular communicator. The communicator is used in every MPI communication API, including the collective operations, as the reference

to an MPI process group, with the rank number to identify the individual process in the communicator. The basic communicator MPI_COMM_WORLD is provided by MPI by default with every MPI process inside. The user can also freely create and manipulate the communicator to create any arbitrary MPI process group for better communication management, such as creating separate communicators for 'odd' and 'even' MPI process number to only communicate among themselves.

In MPI implementations, such as MPICH, and its descendants such as Intel MPI, the matching process for any communication is handled in a global matching queue, regardless of the communicator, while Open MPI handles the matching with a local per-communicator matching queue. The two approaches can inflict different performance impacts when it comes to multi-threaded environments in MPI, as the threads race for access to the communicator and the matching process. In the latter design, increasing the number of the communicator might alleviate the stress on each communicator and provide a more level playing field for a multi-threaded environment. Multirate provides the option for the user to switch between utilizing a single communicator, or using an exclusive communicator for each of the communication pairs, to provide a way to detect the communicator congestion problems of the MPI implementations.

3.6 Experimental Evaluation

While Multirate offers bandwidth, message rate, and latency measurement, only the message rate measurement is presented in this section, as it is the most representative metric for the quality of multi-threaded communication. Nonetheless, the goal of the experiments is to demonstrate the potential of Multirate in evaluating MPI performance and pinpointing possible bottlenecks in current MPI implementations.

The experiment's results are from the University of Tennessee, Knoxville's Alembert cluster. Each Alembert node consists of two sockets of Intel Xeon E5-2650v3 (Haswell) 10-core CPUs, running at 2.3 GHz and configured with hyper-threading, with 64 GB of DDR4 2, and 133 MHz main memory. The interconnect is an InfiniBand (IB) EDR running at 100 Gbps. The MPI implementations in the experiments are: MPICH 3.3 Gropp (2002),

Intel MPI 2018.1 IMPI, and Open MPI 4.0 Gabriel et al. (2004). MPICH and Open MPI are configured with Unified Communication X (UCX) Shamis et al. (2015) library version 1.5.0 in multi-threaded support mode with all optimization turned on. The entire software stack, including the MPI libraries, was compiled with GNU Compiler Collection (GCC) version 7.3.0 with maximum level of optimization flags from the provided package configure script, while Intel MPI is only available in a pre-compiled binary from the vendor and assumed to be operating with high efficiency on the platform.

The MPI process binding policy is bind to core (one core per MPI rank) for process mode experiments, and every necessary precaution has been taken to ensure all experiments were using identical bindings and thread placements. The thread mode experiments spawn only one MPI process on each node with a "floating" (no binding/bind to all available cores) policy and manually bind the threads to their corresponding cores. The default communication module of every MPI implementation is used, unless stated otherwise. The message sizes used in these experiments were selected based on ongoing optimization efforts for MADNESS and PaRSEC, by taking the sizes of the most representative communications. The performance data points presented are the average of 30,000 runs, and, where meaningful, the standard deviation is reported in the graph as error bars. Experiments that did not complete, either due to a segfault in the MPI library or to a deadlock, identified by an expiring allocation limit, were not reported and can be seen by the lack of data in the graphs.

To give a better understanding of the order of magnitude of the results, some graphs report the theoretical upper limit of the message rate calculated by dividing the peak hardware bandwidth of 100 Gbps by the corresponding message size. The computation of the theoretical upper limit ignores all local overheads, and is, therefore, an unattainable upper bound, toward which the message rate should asymptotically converge. The theoretical upper bound on the message rate is 12.5 Mmsg/s for 1,024 bytes messages and 3.125 Mmsg/s for 4,096 bytes messages (M stands for Millions).

3.6.1 Communication Patterns

This section investigates how different state-of-the-art MPI implementations handle different communication patterns available in Multirate. Most figures in this section will illustrate

the performance of both process and thread mode for the same communication pattern in the same figure. However, the discussion is organized as follows: First, the performance of the process mode in each communication pattern, which is a general use-case for most applications. It also serves as the practical upper bound for the thread mode since the threading performance with additional overhead is unlikely to achieve the better performance than the process mode in the same settings. The discussion is then followed by the thread mode performance and comparisons at the end of this section.

Pairwise

Pairwise communication performance is illustrated in Figures 3.5 and 3.6. For the experiment, the message size and the window size (number of messages per iteration) is fixed, while the number of communication pairs varies. Since the communication pattern is pairwise, the number of sender and receiver entities is the same. Every communication pair has the same amount of workload.

The solid lines (Figures 3.5 and 3.6) show the performance in process mode for three MPI implementations. The message rate increases with the number of process pairs, as anticipated. Since each individual process pair communicates independently without any contention, the more pair added gives higher aggregated message rate, until the performance comes close to the theoretical limit of the hardware device and the message rate flattens out. This indicates that in this mode, the MPI implementations can operate very efficiently and push the hardware to its limit. Each MPI implementation reacts slightly differently, but all of them asymptotically reach the peak message rate at a number of communication pairs. In this experiment, for 1,024-byte messages, Open MPI reaches the peak with 7 communication pairs, while Intel MPI needs 11 pairs, and MPICH needs 16 at around 11M msg/s, with the calculated bandwidth of 90.12 Gbps or 90% of the theoretical network bandwidth. Although the MPI implementations reach the peak performance at different point, the result firmly suggested that the MPI requires more than one communication pair to satisfy the peak hardware bandwidth for small size messages.

At 4,096 bytes, the MPI implementations reach the peak message rate earlier (Figure 3.6), as by increasing the message size, the actual transfer time increases and the software

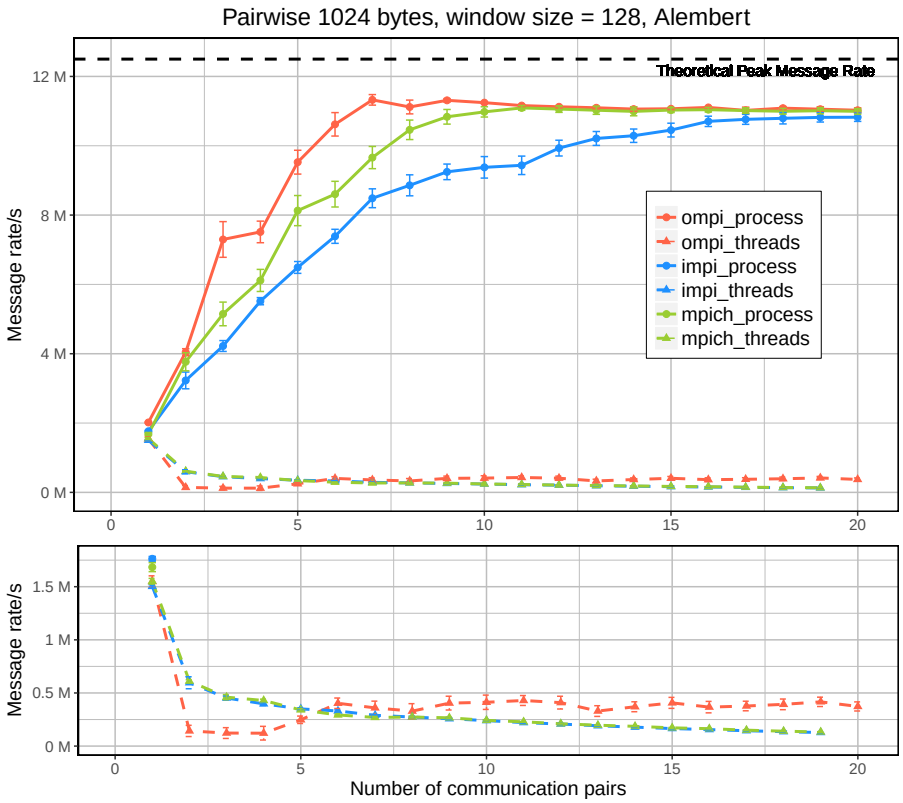


Figure 3.5: Pairwise message rate for a message size of 1,024 bytes, w = 128.

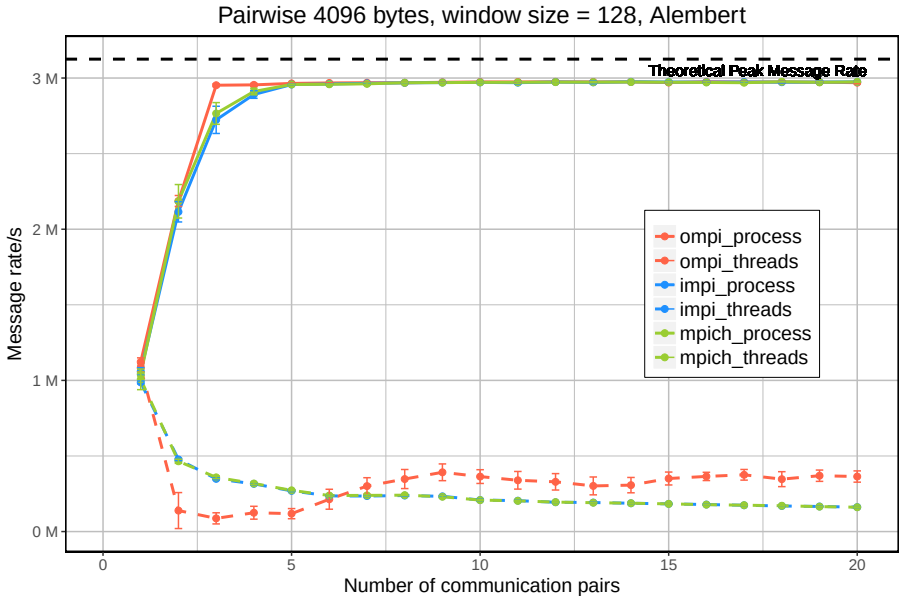


Figure 3.6: Pairwise message rate for a message size of 4,096 bytes, w = 128.

overheads have lesser impact on the overall performance. With larger message sizes, it requires smaller numbers of concurrent messages to reach the peak hardware bandwidth (Around 3M msg/s, the calculated bandwidth is 98.3 Gbps, 98.3% of the theoretical bandwidth), and the differences between MPI implementations are mostly negligible.

The performance results of the process mode in pairwise pattern shows that with truly concurrent communication, the MPI implementation can attain the peak performance of the network device. We can use this process mode performance as the reference point, as an attainable performance or 'practical peak' for multi-threaded communication on the same hardware. It should be noted that threading communications perform extremely suboptimally for every MPI implementation in this experiment, with a global message rate decreasing as the number of peers increased—opposite to what we observe from non-threading communication—despite running on the same set of hardware. The message rate for thread mode never goes above 1/12th of the practical peak and suggests massive optimization opportunities for every MPI implementation.

Many to One

In this experiment, the number of receiver entities is fixed to one, while varying the number of the sender entities to measure the limit of the message extraction capability of a single receiver entity. Generally, the message rate should keep increasing with the number of senders until it reaches the capacity of the receiver to handle the incoming messages, then the message rate should stay flat from that point.

The solid lines on Figure 3.7 show the performance in process mode for 1,024-byte messages. Intel MPI performs the best and shows a performance improvement with an increasing number of senders. This result suggests that one receiver process is capable of extracting more incoming messages than a single sender can inject. From the result, the Multirate benchmark can expose the optimal point of operation for Intel MPI at 2.8M msg/s with 8–12 senders. Open MPI shows steady performance at 1.5M msg/s despite the increasing number of senders, indicating the limit of message extraction from the receiver. The result indicates that there is more room for improvement in the message extraction path of Open MPI. On the other hand, MPICH demonstrates a performance decrease with

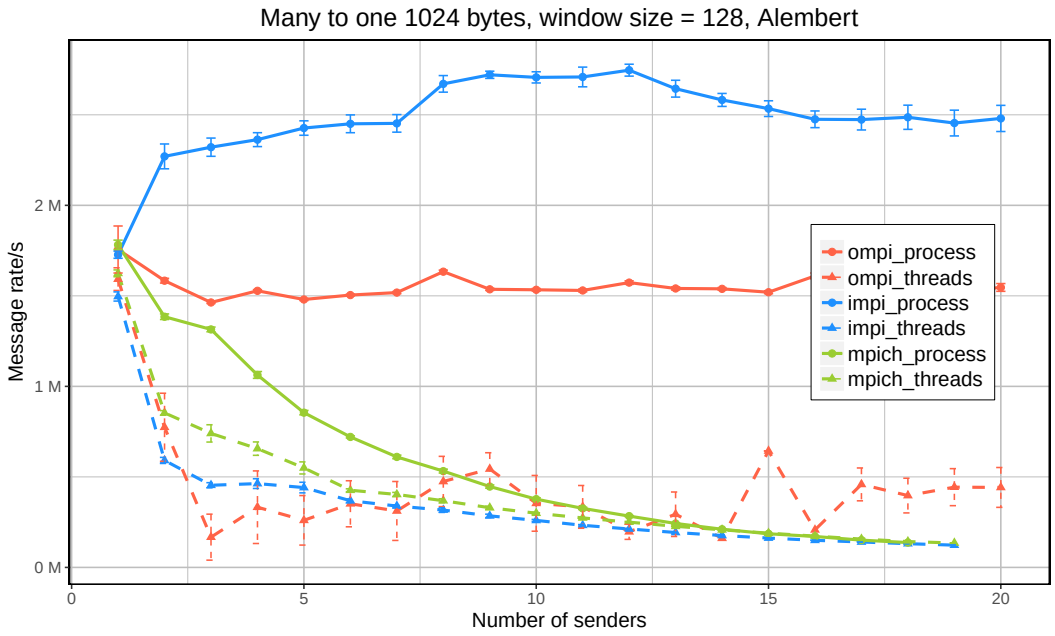


Figure 3.7: Many-to-one message rate with a message size of 1,028 bytes, w = 128.

higher number of senders, which indicates the internal bottleneck that introduces the delays proportionate to the volume of incoming messages, decreasing the rate of extraction, leading to sub-optimal performance—the behavior that is not anticipated.

For 4,096-byte messages (Figure 3.8, solid lines), Open MPI demonstrates better performance early on but drops off with an increasing number of senders—entirely different behavior from the smaller 1,024-byte messages. The behavior shows that the same MPI implementation reacts differently depending on the message sizes. The change of behavior can originate from the protocol change at the MPI implementation level or even from the underlying network library (in this case, Open UCX). On the other hand, for both message sizes, MPICH and Intel MPI show similar behaviors. Specifically, Intel MPI shows the same pattern of a sharp rise in message rate when increasing the number of senders from 7 to 8. This information can become valuable for the application developer in deciding their MPI process layouts to get the most performance out of their MPI implementation. From an MPI developer's perspective with inside knowledge of the code base, this information can be useful for further optimization.

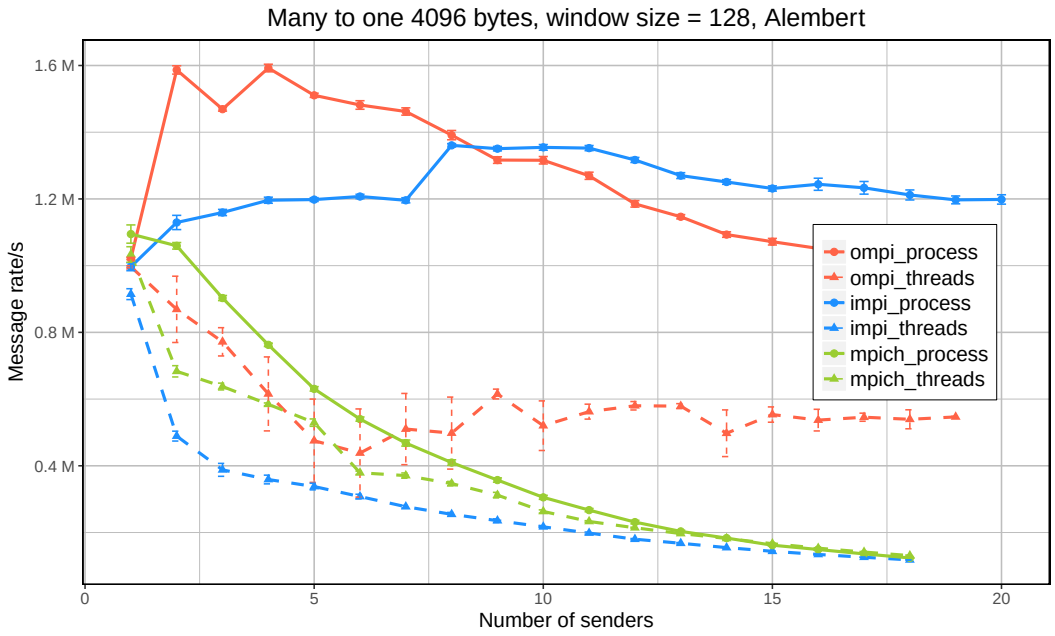


Figure 3.8: Many-to-one message rate with a message size of 4,096 bytes, w = 128.

The *many-to-one* experiments show that the Multirate benchmark can successfully identify the problem in the message extraction path, and expose the optimal performance point for a specific workload from the receiver's perspective.

One to Many

This experiment utilizes one sender to send messages to the increasing number of receivers. In reverse of *many-to-one*, by having multiple receivers to extract the messages at the same time, this communication pattern reduces the possibility of congestion on the receiver and focuses on the capability of a single sender to inject the messages into the network. Generally, with many receivers to absorb the communication, the message rate should keep increasing until the sender reaches its peak injection rate and then flattens out beyond that point.

Figures 3.9 and 3.10 show the message rate of 1,024 and 4,096 byte messages when increasing the number of receivers. This section focuses only on the performance of the process mode (solid lines). For 1,024-byte messages, all three MPI implementations show better performance with the increasing number of receivers until they reach a peak—an anticipated behavior. The result suggests that a single receiver cannot efficiently extract the

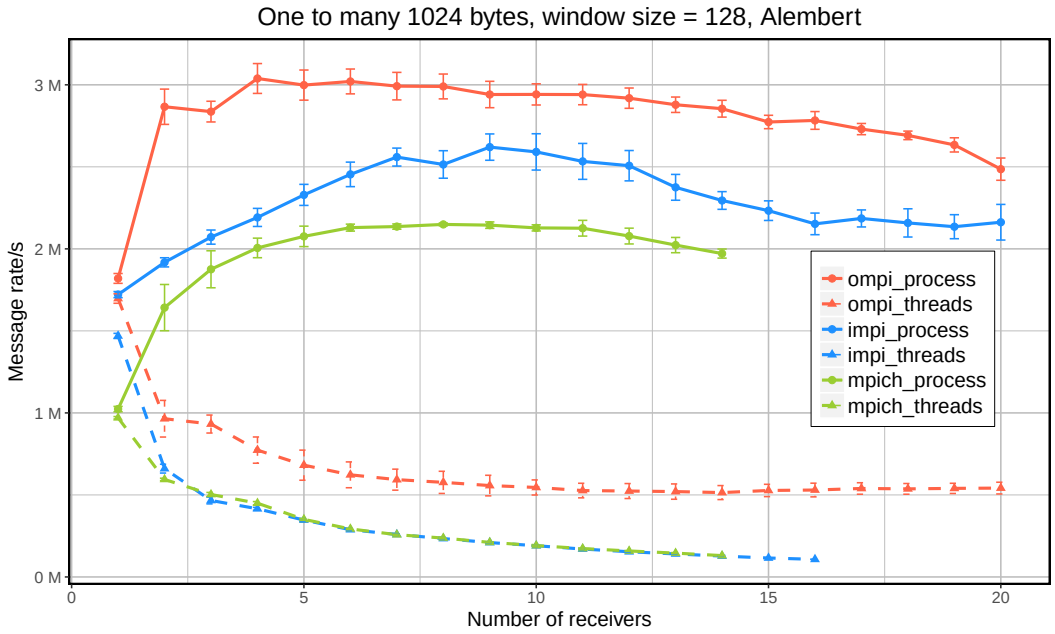


Figure 3.9: The *one-to-many* message rate with a message size of 1,024 bytes, w = 128.

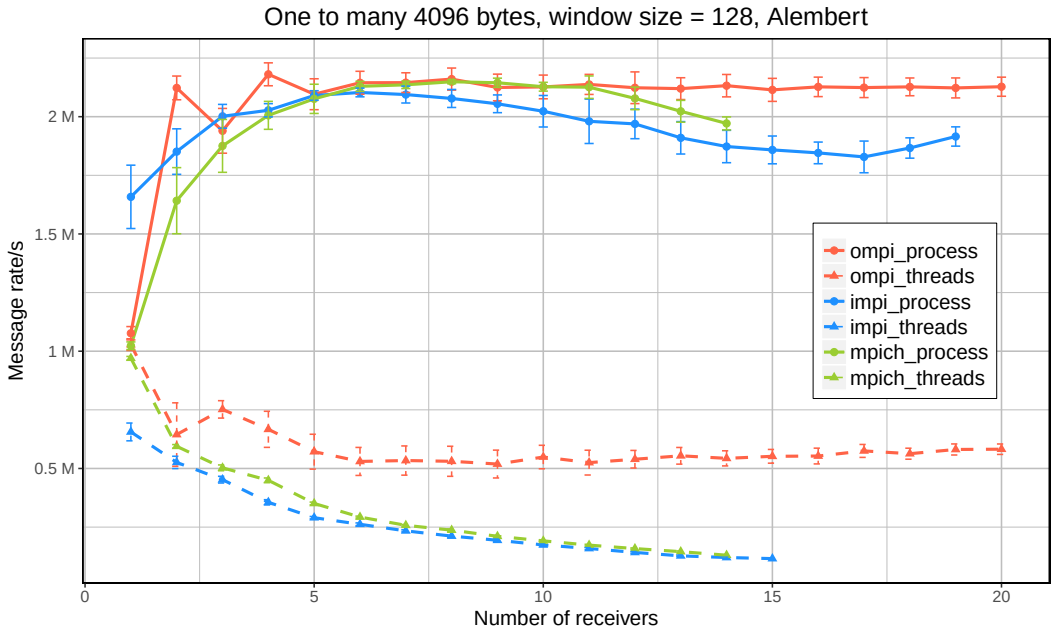


Figure 3.10: The *one-to-many* message rate with a message size of 4,096 bytes, w = 128.

messages injected from a single sender, indicating that every MPI implementation in this experiment exhibits some level of imbalance between the message injection and extraction.

Compared to other implementations, Open MPI gives the best performance in this communication pattern. The best performance point of Open MPI for this setting is around 4–7 receivers at 3M msg/s, and the calculated bandwidth is 24.57 Gbps, only 1/4th of the theoretical network bandwidth. Comparing this result to the pairwise communication pattern, which can achieve up to 90% of the theoretical bandwidth, this experiment also shows that a single sender cannot achieve the optimal injection rate to satisfy the peak network bandwidth for 1,024 byte messages. The result showed an unexpected small performance drop-off for all three MPI implementations after increasing the number of receivers, a behavior that is not presented when increasing the message size to 4,096 bytes, which has to be investigated further.

At the message size of 4,096 bytes, every MPI implementation can reach the plateau around 5 receivers. The Intel MPI gives better performance early on with a lower number of receivers. With larger messages, the message rate required to achieve the peak network bandwidth becomes lower. For example, in this experiment, Open MPI's message rate plateaued out around 2.5M msg/s. The calculated bandwidth for the message size of 4,096 bytes is 81.92 Gbps, or 81.92% of the theoretical network bandwidth of the hardware used.

Together, the experiment results of the *one-to-many* and *many-to-one* demonstrated that (1) In most MPI implementations, the single receiver cannot extract the incoming messages fast enough. The experiment shows that a single sender can satisfy at least 2 receivers in some workloads. (2) Increasing number of concurrent messages from the senders' side might not translate into higher message rate, as the capacity of the extraction process is limited by the receivers.

Many to Many

Moving towards a more dynamic relationship between the number of senders and receivers, this experiment illustrates the message rate of the *many-to-many* communication pattern (every sender to every receiver) from a different perspective. Figure 3.11 shows the performance when increasing the number of senders against fixed sets of receivers, while Figure 3.12 shows the opposite. This section only focuses on the process mode result (top half of the figure), as further discussion on the threading performance is listed separately in Section 3.6.3.

The experiment results illustrated the different behavior of each MPI implementation. Figure 3.11 shows that Intel MPI can perform better with more than one receiver, but increasing the receivers will not give much benefit, as it seems to already reach top performance at around 5 receivers; while Open MPI and MPICH reach top performance at some specific point, the message rate drops off with an increasing number of senders. Unlike Intel MPI, increasing the number of receivers provides some performance impact for both Open MPI and MPICH. From a different perspective, Figure 3.12 also shows that Intel MPI is performing well regardless of the number of senders or receivers, while Open MPI and Open MPI provide a similar result. The two figures show the most efficient performance point of each MPI implementation. For example, Open MPI seems to run optimally around 20 receivers with 4–5 receivers for this particular workload. The result also illustrated the similarity between Open MPI and MPICH, where both implementations utilize Open UCX as their underlying network library.

This experiment presented the versatility of the Multirate benchmark and its ability to expose the optimal performance point of each MPI implementation, which can be beneficial to application developers to design their communication workload. The MPI developer can also use this information to get the better understanding of their MPI implementation under a specific workload for better optimization.

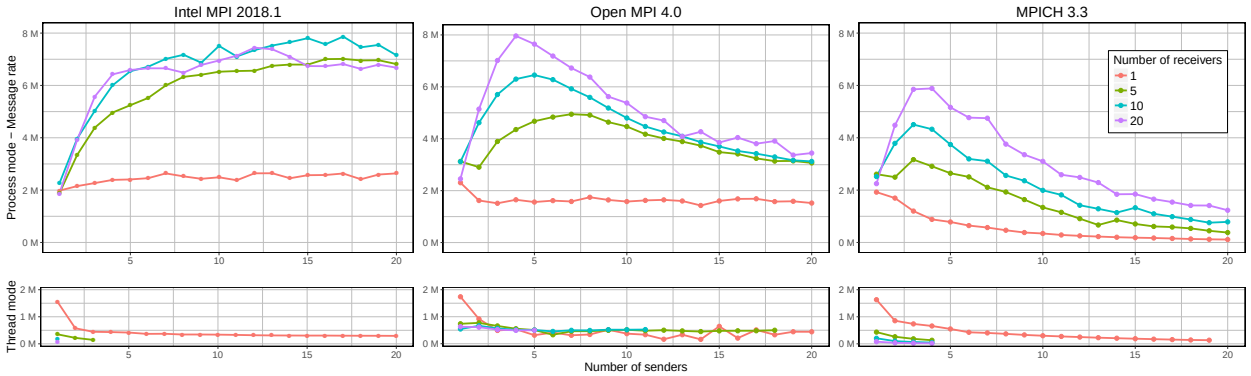


Figure 3.11: The *many-to-many* communication performance with a fixed number of receivers.

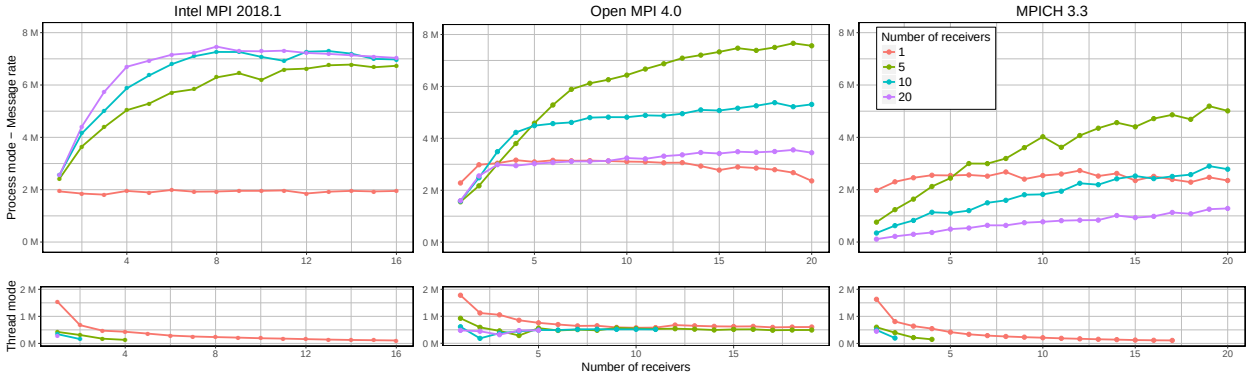


Figure 3.12: The *many-to-many* communication performance with a fixed number of senders.

3.6.2 Variable workload

While the earlier experiments demonstrated the different behaviors of MPI implementations with respect to different message sizes, this section performs the experiments on the different communication patterns with different window sizes (number of messages per iteration) and observes how MPI implementations react to different workloads.

For *one-to-many* communication (middle row), Intel MPI still shows similar scaling to the *many-to-one* experiment with a small drop-off at the end. Open MPI performs well in this communication pattern with a higher message rate overall for every window size, while MPICH demonstrates good performance, but also with a small drop-off later on.

It has been learned from the earlier experiment that a pairwise communication pattern gives the best performance and scaling with increasing number of pairs. For this experiment, Figure 3.13 (bottom row) indicates that the window size also affected the overall communication performance. Generally speaking, for this particular message size, a larger window size allows more messages to be injected per iteration and increases the message rate until it reaches the limit of the network device and plateaus out. The performance of each MPI implementation is slightly different but still follows a general trend.

In this experiment, the Multirate benchmark shows that it can expose the behaviors of the different MPI optimization. For example, it can be concluded that Intel MPI is very sensitive to the window size and will perform well with a larger one. Open MPI is not as sensitive, as the results show similar performance across all window sizes but in *many-to-one*, the window size does not affect the performance at all. This behavior indicates that there is some limitation in its message extraction capability. MPICH is struggling in *many-to-one* communication, especially with higher numbers of messages, consistent with the findings from the earlier experiment. MPICH developers can use this information to pinpoint the origin of the performance bottleneck in their implementation.

While the result of this experiment is measured from a single message size in process mode, it demonstrates that Multirate is capable of exposing the optimal points of each MPI implementation. Users can simply tweak the parameters to perform measurement on their desired workload and mode operation.

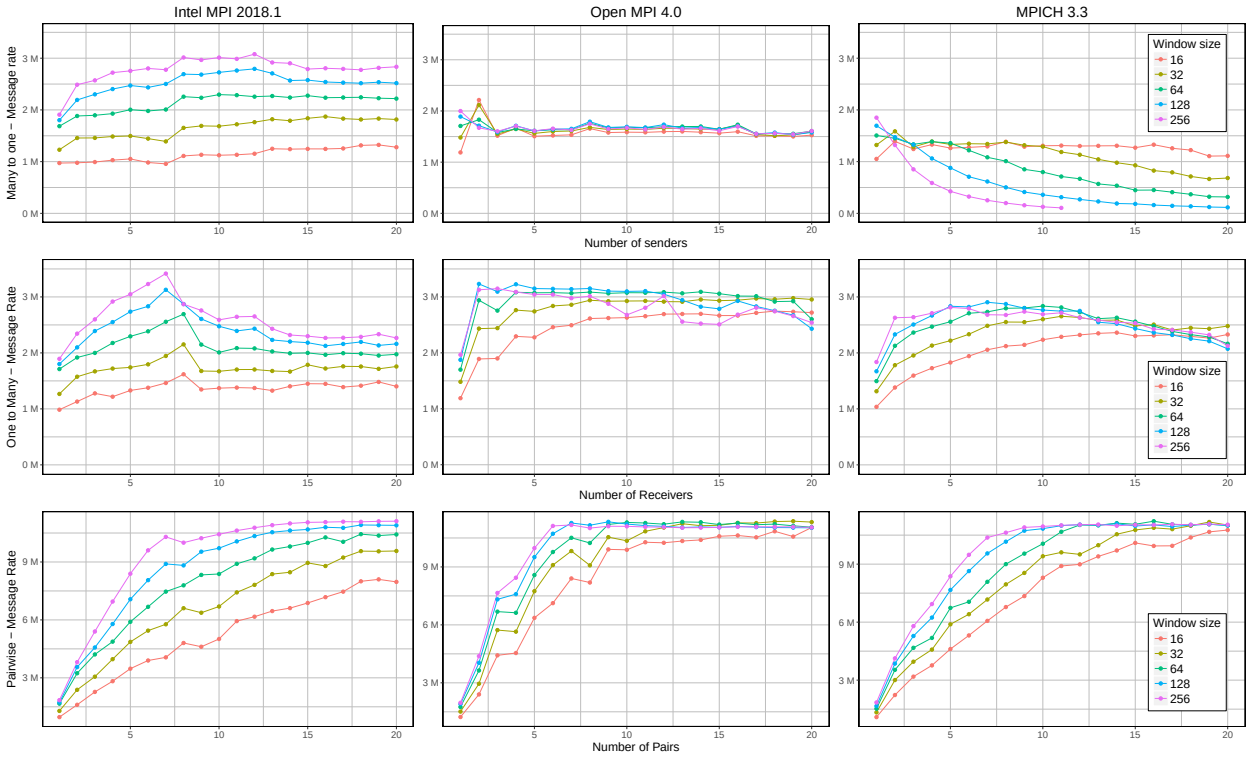


Figure 3.13: Message rate (1,024 bytes) on different communication patterns on multiple window sizes.

Case study

Other than using Multirate to compare the performance between MPI implementations, it can also be used to expose the performance issues in a single implementation. The case study is the comparison between the two stable releases of Open MPI, 4.0 and 3.0.

This experiment only uses one pair of communication entities and increases the window size. Since the window size per communication pair in the early experiment is 128, the window size is increased in multiples of 128 in 20 steps. Internally, Open MPI 4.0 uses a pml/ucx network module as default, while Open MPI 3.0 uses btl/openib. Though the two modules have significant differences, both of them are designed to efficiently operate with Mellanox's Infiniband hardware.

The result is demonstrated in Figure 3.14. For the default UCX module, the message rate does not increase with the window size, thereby confirming the finding that a single entity cannot satisfy the network bandwidth. For the btl/openib module, this experiment exposes a sub-optimal implementation of the network module, as the result shows the performance drop when increasing the message size with a high variation between runs. After some investigation, I find that btl/openib has a poor credit management system, which leads to starvation of network send credits under a heavy workload. The behavior is non-deterministic, as the starvation and the recovery occur at different points for every run. I indirectly adjust the number of send credits in btl/openib module the via Modular Component Architecture (MCA) Squyres parameters offered by Open MPI and can mitigate the credit problem from the module (marked with * in the figure).

3.6.3 Multithread MPI

Overhead of threading

This experiment uses the pairwise process mode when initializing MPI with MPI_Init and MPI_Init_thread with MPI_THREAD_MULTIPLE, without spawning any other thread, to show the minimum overhead from each MPI thread safety implementation. Figure 3.15 compares the performance between MPI implementations. We can see only a 3–5% performance decrease from MPI implementations for this communication pattern. However,

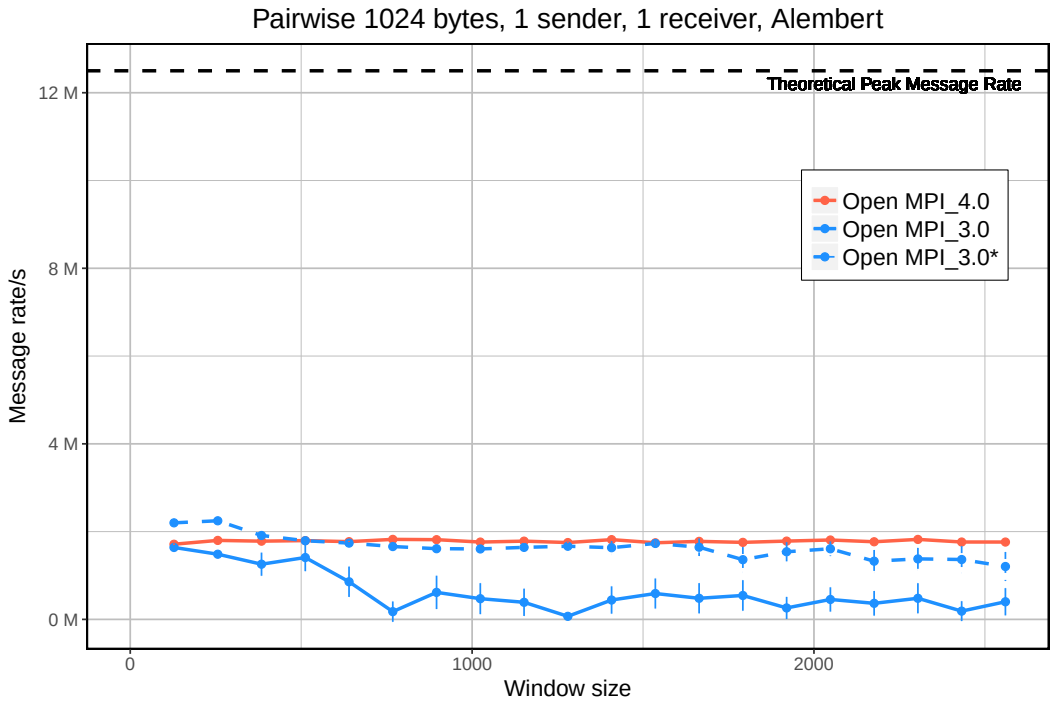


Figure 3.14: Performance difference between two Open MPI release versions; 1,024 bytes pairwise, process to process mode.

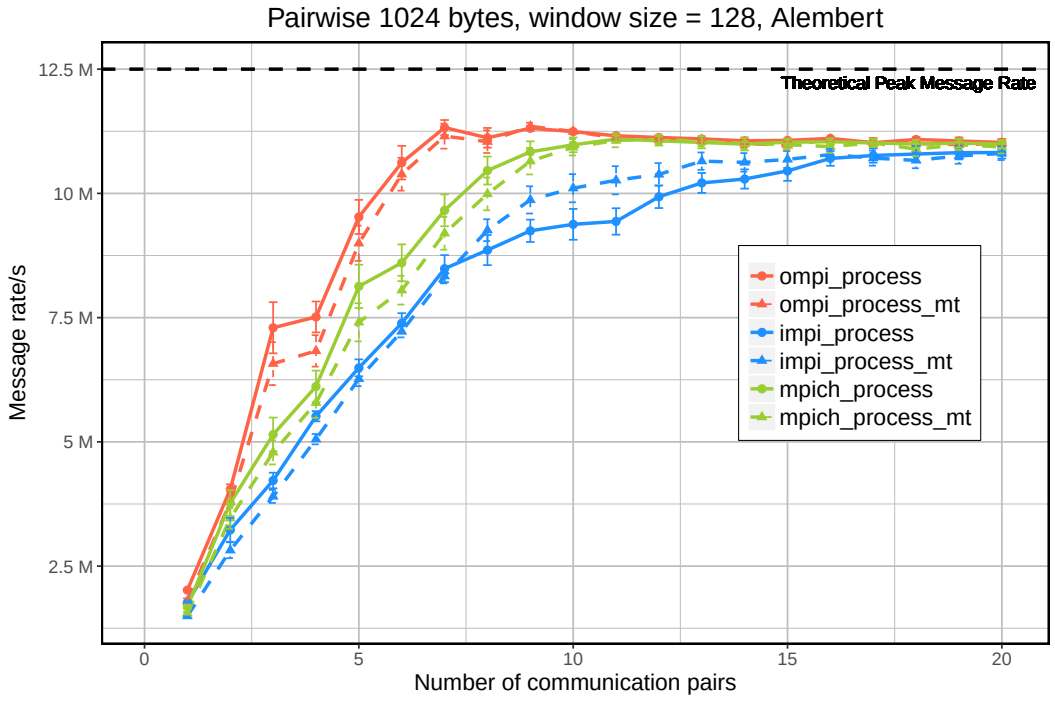


Figure 3.15: Minimum cost of thread safety.

It has been shown in the earlier experiments that the performance will drop drastically with the increasing number of threads (Figure 3.5), indicating that most of the performance degradation originated from the thread contention.

Different communication patterns

The dashed lines in Figures 3.5 and 3.6 show the message rate of the increasing number of communication pairs with a pairwise pattern in thread mode. For both message sizes, Intel MPI and MPICH have similar performance, but the message rate of Open MPI drops at first and then bounces back with more communication pairs. However, all of them suffer significant performance loss compared to the results in process mode. The dashed lines in Figures 3.7–3.10 show the performance of *many-to-one* and *one-to-many* patterns in thread mode.

For *many-to-one*, every MPI implementation seems to suffer a performance loss when introducing more sender threads. Open MPI shows a performance drop early on, but performance recovers after increasing the number of senders, while Intel MPI and MPICH performance gradually decreases.

In *one-to-many* communication, the performance drops with more communication pairs for all three implementations. Open MPI is the best among the tested MPI implementations, but none of them shows comparable performance with process mode.

many-to-many communication shows the significant difference between process mode and thread mode. While process mode can handle this type of communication easily, thread mode—in some cases—cannot run to completion before the 30 second timeout.

Ideally, in thread mode, running on the same hardware with the same communication pattern should provide comparable performance to process mode. However, the result shows that MPI in thread mode is significantly slower than its process mode counterpart—indicating that the designs of current state-of-the-art MPI implementations are not well optimized for threading.

To further analyze the threading performance, we have to look deeply into the design of each MPI implementation to identify the bottleneck. Some of the MPI implementations evaluated in this experiment make use of a global message matching queue. Using multiple

threads in communication can increase the contention on the matching queue which has to be accessed sequentially. Usually, MPI implementations make use of a mutual execution (mutex) lock to serialize the access to critical parts of the communication. The cost of securing a mutex lock increases with the number of threads. The threading result from Figure 3.5 is isolated and shown in Figure 3.16, displaying performance degradation with the increasing number of communication pairs and suggests the serialization bottlenecks.

Communicator's effect

For Open MPI, btl/uct is a non-default communication module that does not use global matching but separates the message matching by the communicator. In Figure 3.16, the btl/uct is manually selected to perform the experiments with single and multiple communicators. The result demonstrated that using multiple communicators to allow multiple threads to perform matching concurrently yields a better message rate. It also suggests that matching can be one of the bottlenecks for MPI in thread mode, and an MPI developer should try to reduce the serialization in the process.

While Multirate offers a communicator's effect test for every communication pattern, only the experiment for pairwise communication is selected for this study, as the default network module for every MPI implementation makes use of global matching and shows similar performance.

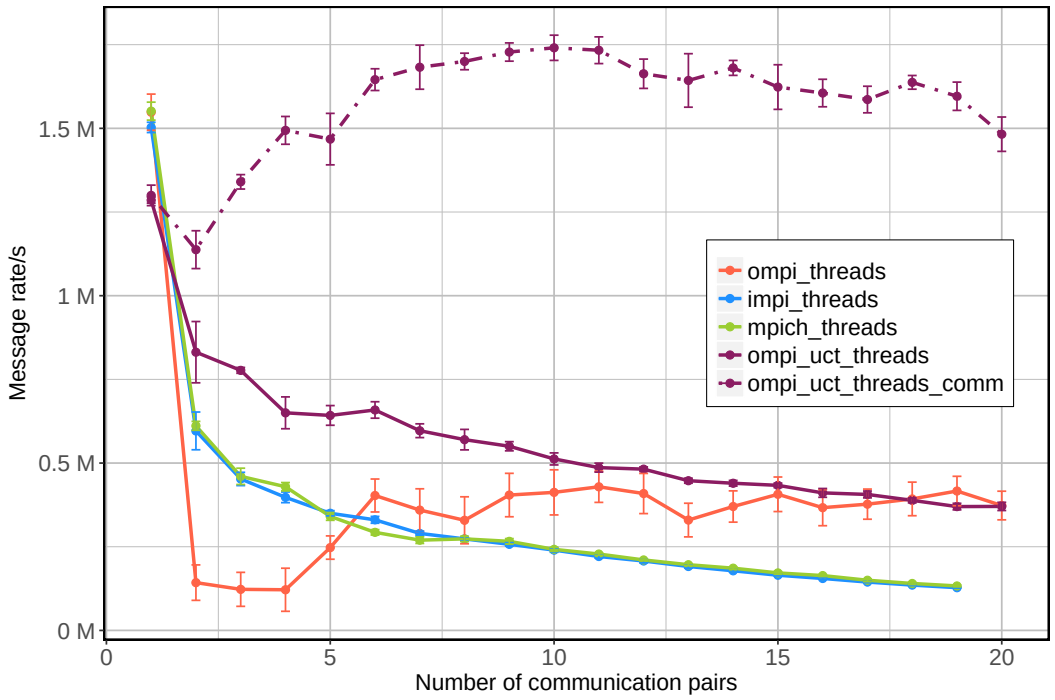


Figure 3.16: Zoomed-in graph for pairwise message rate for thread mode with btl/uct to demonstrate the effect of the communicator.

3.7 Conclusion

This chapter presented the Multirate benchmark, a novel multi-threaded MPI performance measurement tool. The main contributions of this chapter concerning Multirate are: (1) the benchmark offers multiple communication patterns that can be used in conjunction with different modes of operations, allowing quick comparison in various settings between MPI in threading and non-threading environment; and (2) Utilizing the benchmark for performance assessments of current state-of-the-art MPI implementations.

The experiments show the potential of the Multirate benchmark for evaluating, quantifying and understanding the performance of MPI implementations under realistic, application-provided workloads. The Multirate benchmark can benefit PI developers as one of the evaluation tools used to identify bottlenecks in their implementations, or as a regression testing tool—and also to users when making the decision on what MPI implementation has the potential of maximizing the performance of their application. Moreover, Multirate can be used as an optimization tool allowing quick testing of different sets of configuration parameters for the different implementation protocols, and assessing which set provide the best overall performance on a specific architecture and/or platform.

In this study, the Multirate benchmark significantly contributes to the discovery of multi-threaded environment bottleneck for Open MPI. The bottlenecks, along with the proposed solutions, are discussed and presented in Chapter 5. The Multirate benchmarks is hosted on Github under a BSD 3-Clause 'New' or 'Revised' License, and will be soon make available [1]. The user guide for the benchmark is presented with this dissertation in Appendix A.

[1] https://github.com/ICLDisco/multirate

Chapter 4

Advance Thread Synchronization

4.1 Overview

This chapter describes the current state and limitations of MPI_Wait and MPI_Test and all of its variances (such as MPI_Waitall, MPI_Waitsome, and MPI_Waitany) in the multi-threaded environment from the standpoint of an MPI implementation. I introduce the thread synchronization object design, along with its API to equip the MPI implementation with the means to control and redirect threads for better optimization. I demonstrate the potential of my design by implementing MPI_Wait* with the thread synchronization object to reduce the lock contention of the MPI progress engine and show that my implementation can achieve up to 7× performance in shared memory communication and up to 3× inter-node communication.

I further discuss the efforts, ongoing collaboration, and preliminary results regarding the utilization of my thread synchronization object design to achieve more from thread parallelism when initializing MPI with MPI_THREAD_MULTIPLE, including my proposal of the MPI extension to bring the thread synchronization object up to the user level.

4.2 Introduction

One of the major roadblocks for MPI implementations to optimize for threading performance is the lack of threading information from the user level. Currently, there is no standardized

way for the user to notify MPI with information such as how many threads they are expecting to be performing communication, or even a piece of simple information such as the thread identification to let the MPI implementation know which thread is calling the MPI routine. There are several studies focused on establishing the infrastructure for sharing thread information (or 'interoperability') between threading frameworks such as POSIX thread (pthread) Lewis and Berg (1998) or OpenMP Dagum and Menon (1998) and MPI implementations, but the idea has not been carried out by the MPI forum. Such information, if obtained, would a positive impact on the MPI implementation as they can design better algorithms to navigate through threads and utilize them properly, along with allocating proper resources for their uses.

Rather than relying on the threading framework to provide the information, several studies suggested a new API for the MPI standard to let the user manually manage the threads through MPI. One of the suggestions is to allow the user to create endpoints under the MPI rank. With multiple endpoints within an MPI rank, the user can map the endpoints with threads, allowing them to address a specific thread at the target rank to perform thread-to-thread communication while also providing the crucial threading information to the MPI implementation for better threading optimization Dinan et al. (2014). However, the suggestion comes with difficulties, such as the problem with collective operations and more. The MPI forum has not yet approved the endpoint proposal but the work is still in discussion Mpi-Forum (2016).

To address the lack of interoperability problem without relying on the support from the MPI forum, I propose the design of thread synchronization object, an abstraction layer to provide the MPI implementation more control over the user-level threads and redirect them for better utilization without requiring any change from the application level. In this chapter, I demonstrate the great benefit of my design for multi-threaded MPI_Wait operation and further discuss additional possibilities in utilizing the thread synchronization object to harness the full power of thread parallelism.

4.3 Progress Engine Serialization

From a high-level perspective, the MPI progress engine is the component that ensures communication progress, either by moving bytes across the hardware, ensuring the expected message matching, or guaranteeing MPI's FIFO message order requirement. From an implementation perspective, the progress engine is the central place where every component in an MPI implementation registers its progressing routine such as polling for incoming messages, processing pending outgoing messages, including messages for collective operations, or reporting completion to the user level. The design is illustrated in Figure 2.3 from chapter 2.

As the MPI standard does not provide an API for explicitly progressing messaging, calls into the MPI progress engine occur under the hood during calls to other MPI routines. The decision to enter the progress engine or not on a given MPI function call is up to the MPI implementation, with the exception of blocking routines such as MPI_Send, MPI_Recv or MPI_Wait where message progression, at least related to the operation itself, is mandatory. That being said, the main purpose of the progress engine is to give the MPI implementation the opportunity to check for message completion events from the network and to ensure timely progress on non-blocking communications. MPI usually reads entries from the completion queues (CQs) for completion events on a particular network endpoint. Completion events can be from both incoming and outgoing messages. In the case of outgoing message completion, MPI marks the corresponding send request as completed and doing so might release the user from a blocking call such as MPI_Send.

In a multi-threaded scenario, the MPI implementation has to ensure thread safety. Since the progress engine is a centralized part where many other components register the progress of their operations, and it is not guaranteed that every registered component will be thread safe, the MPI implementation often casts a wide blanket by creating a large critical section for the entire progress engine. While the coarse-grain approach is proven sufficient in providing thread safety to the user, the contention on lock often hinders the overall performance of the progress engine, especially with an increasing number of concurrent threads trying to gain access to that critical section. The major cost from the lock semantic usually increases

with the contention on the lock itself. I perform an experiment to demonstrate the cost of securing a lock when increasing the number of threads and presents the result in Figure 4.1. The result suggested that the cost of a lock operation increases in polynomial order with the number of threads, consistent with earlier studies on lock-less data structures such as Amer et al. (2016) and Amer et al. (2015).

4.4 Synchronization Object

In this study, I propose a novel approach for managing multiple threads from inside the MPI implementation with the synchronization object (sync object). Traditionally, when multiple threads are waiting for the completion of MPI requests, they race against one another to execute the progress engine, which is protected by a lock, as described earlier in this chapter. The race creates lock contention and degrades overall performance from the progress engine, while under-utilizing thread parallelism as the lock acts as a funnel for only a single thread to pass through. The sync object provides the MPI implementation with a mechanism to redirect threads for other tasks and a work-tracking capability to release them back to the

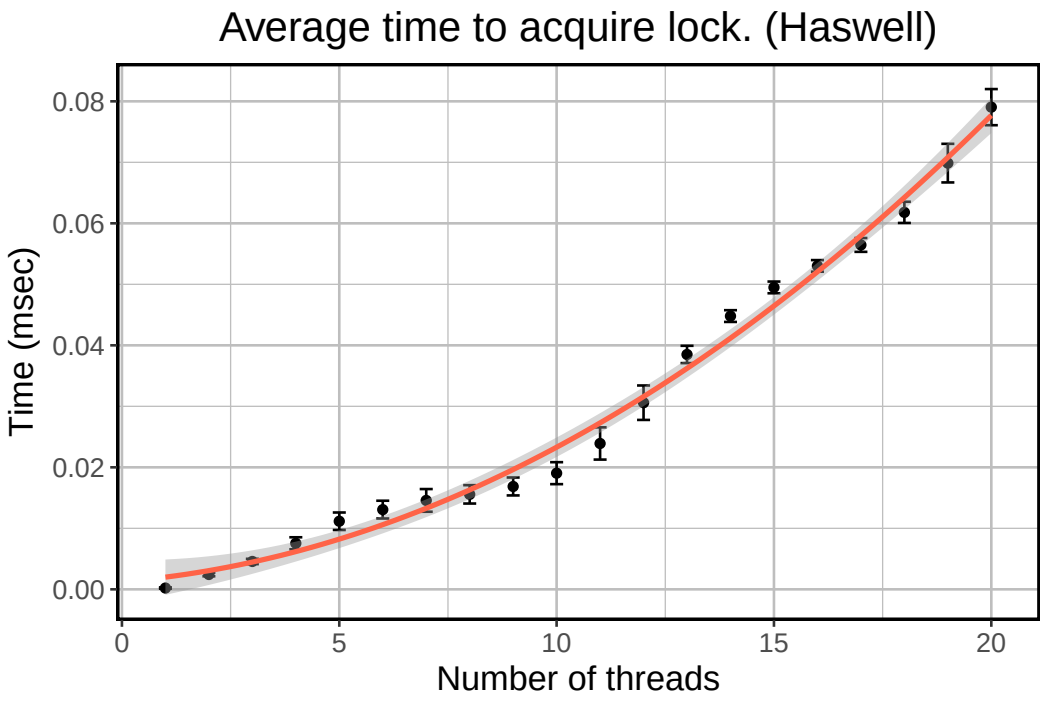


Figure 4.1: Cost of lock acquiring on Intel Xeon E5-2650 v3 (Haswell)

user as soon as their waited requests are completed, providing a better opportunity for thread utilization.

A sync object is an object with a simple reference counter that allows MPI requests to attach to it. For every request attached, the sync counter increases its reference counter; once a request completes, the MPI implementation can notify the associated sync object and decrease its reference counter. This process allows MPI to notify only the thread involved in the operation without involving other threads.

Synchronization Object API

For Open MPI's internal use, I create the sync object API and utilize it to redesign the Open MPI progress engine. The API provides 4 methods to interact with the sync object (INIT, WAIT, SIGNAL, and UPDATE). The accurate C API is located in the appendix of this dissertation.

- **SYNC_INIT**: Initialize the synchronization object.
- **SYNC_WAIT**: Blocking call, wait until signaled or the counter becomes zero.
- **SYNC_SIGNAL**: Release the synchronization object from waiting.
- **SYNC_UPDATE**: Add/subtract the number from the object's counter

Implementing Wait Operation

In asynchronous (non-blocking) communication, MPI returns an MPI request as a handle for the user to track the status of the operation later with MPI_Wait or MPI_Test. Generally, we can categorize the request into two groups: send requests and receive requests. As their name suggests, the send request is the request that is associated with a send operation and the receive request is associated with a receive operation. Usually, an MPI request is marked as completed when the MPI implementation receives the completion event from the network by reading its completion queue (generally, through the progress engine). However, for the receive requests, they can also be completed at posting; the message can arrive from the

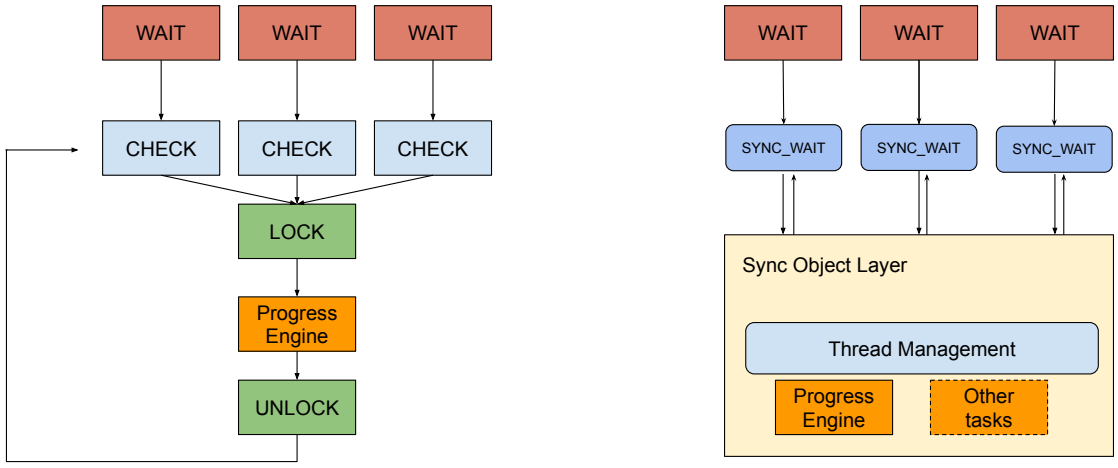
network before the user posts a corresponding receive for it, and the MPI implementation matches the message with the request as soon as it is posted.

Traditional MPI_Wait* (waitall, waitsome, waitany) implementation involves a loop over every request given at the user level, constantly checking for their completion, and simply counting the number of completed requests in the loop. Once the number of completed requests satisfies the wait condition (variants such as all, some, or any), the wait operation is successful and returns to the user from the blocking call. If the condition is not satisfied, the wait routine usually executes the progress engine to look for completion. In a multi-threaded scenario, access to the progress engine is protected by a coarse-grain lock (Figure 4.2a and Algorithm 1). As discussed earlier in this chapter, this creates a bottleneck and increases the overall operation cost with the number of threads.

With the synchronization object API as a management layer, the MPI implementation can become more efficient in redirecting threads for other purposes, and return it to the user as soon as it needs to be returned (Figure 4.2b). I present the algorithm of the new MPI_Wait* in Algorithm 2. In this implementation, MPI_Wait* relinquishes the authority of waiting to the synchronization object API, which can redirect the threads for other tasks. The MPI_Wait* will get notified from the synchronization object API (by returning from SYNC_WAIT) when the waited requests are complete.

SYNC_WAIT can redirect the threads to different tasks. First, my implementation aims to reduce the stress on the progress engine lock. I built a queue system which allows only a single thread to execute the progress engine while the other threads wait peacefully, yielding the usage of CPU core back to the user. Since all but one thread is yielding and not actively trying to secure the lock, the contention on the progress engine lock is minimal, allowing a single thread full access to the progress engine. The executing thread is referred to as a 'progress owner'.

When executing the progress engine, the progress owner can complete any pending request from any thread. Once a request completes, the sync object associated with it gets updated directly via SYNC_UPDATE. If a sync object's counter reaches zero, the progress owner issues a signal via SYNC_SIGNAL to the corresponding sync. Since a sync object is directly associated with a thread performing wait, when signaled, the thread stops yielding



(a) Original (b) Proposed – with sync object

Figure 4.2: MPI_Wait* operation implementation in multi-threaded scenario.

Algorithm 1 Original MPI_Waitall implementation

```
1:  function MPI_Waitall(n,requests)
2:      while true do
3:          c = 0
4:          for each requests do
5:              if request is complete then
6:                  c ← c + 1
7:          if c is equal n then
8:              break;
9:          lock Progress Engine Lock
10:         call Progress Engine
11:         unlock Progress Engine Lock
12:     return
```

Algorithm 2 New MPI_Waitall implementation with synchronization object API.

```
1: function MPI_WAITALL(n,requests)
2:     call SYNC_INIT (sync)
3:     c = 0
4:     for each requests do
5:         if request is complete then
6:             c ← c + 1
7:         else
8:             attach request to sync
9:     call SYNC_UPDATE (sync,c)
10:    call SYNC_WAIT (sync)
11:    return
```

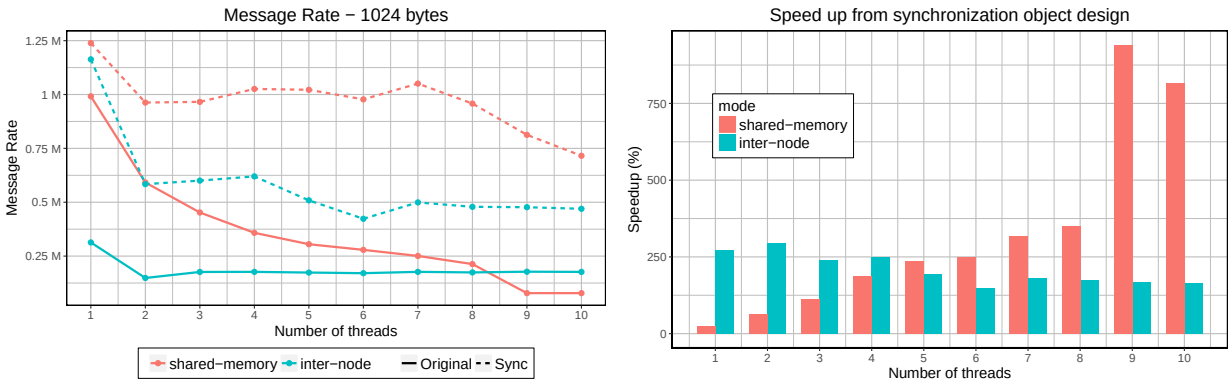
and reschedules itself for execution, removing itself from the queue. In the case where the progress owner's sync object counter becomes zero, it passes on the progress ownership to the next sync object in the queue to take its place. This design guarantees that if there are multiple threads calling MPI_Wait, there will always be one thread executing the progress engine.

4.5 Experimental Evaluation

For evaluation of my design, I measure the message rate by using the Ohio State University (OSU) microbenchmark OSU and the Multirate benchmark on the University of Tennessee's Alembert cluster in both shared-memory and inter-node communication via a high-speed InfiniBand network. The performance result is illustrated in Figure 4.3a and the speedup in Figure 4.3b.

For shared-memory intra-node communication where the communication is expected to be very fast through a simple memory copy operation, we can see a significant speedup from the original design, especially with a higher number of threads. The synchronization object design greatly reduces the lock contention on the progress engine and we can see up to 8× performance improvement. On the other hand, when the communication is inter node via InfiniBand, high-performance network hardware, the performance gain is up to 2.5× and slightly drops off after increasing the number of threads. Although the overall performance is increasing, we can still see that using a single thread to perform communication yields a better result than multiple threads. From my design, using a single thread to execute the progress engine, the performance should at least flatten out around a single thread performance. This result suggests other bottlenecks in the multi-threaded MPI implementation. Further in this study, I identified and addressed the discovered bottlenecks. The details are discussed thoroughly in chapter 5.

In the case of thread over-subscription (binding multiple threads to the same physical CPU core), the original design suffers from multiple unnecessary context switches as the threads blindly race to take the control of the progress engine, then perform the check on each MPI request associated with it. The synchronization object design, with a proper



(a) (b)

Figure 4.3: Performance gain from utilizing thread synchronization object in MPI_Wait implementation.

notification system, only performs context switching when it is necessary. Figure 4.4 shows that with the thread synchronization object, the design can minimize the context switching and achieve up to 250× performance for shared-memory communication and 60× for inter-node communication. While the performance gain is massive, it is unlikely that the modern HPC application is designed to operate in an over-subscription environment.

4.6 Ongoing Research

So far, the current usage of the synchronization object is only for serializing the progress engine execution and reducing the lock contention to the progress engine. Despite contributing to better threading performance in most cases, only the thread with progress ownership gets to work while the others are yielding and get de-scheduled. There is potential for more thread parallelism with the synchronization object design. This section explores some of the potential use cases for thread parallelism from synchronization object with small prototypes and proofs of concept.

4.6.1 User-Level Extension

The drawback of the synchronization object design is the cost of attaching and detaching the request to the sync object. Since the scenario is multi-threaded, the attach and detach

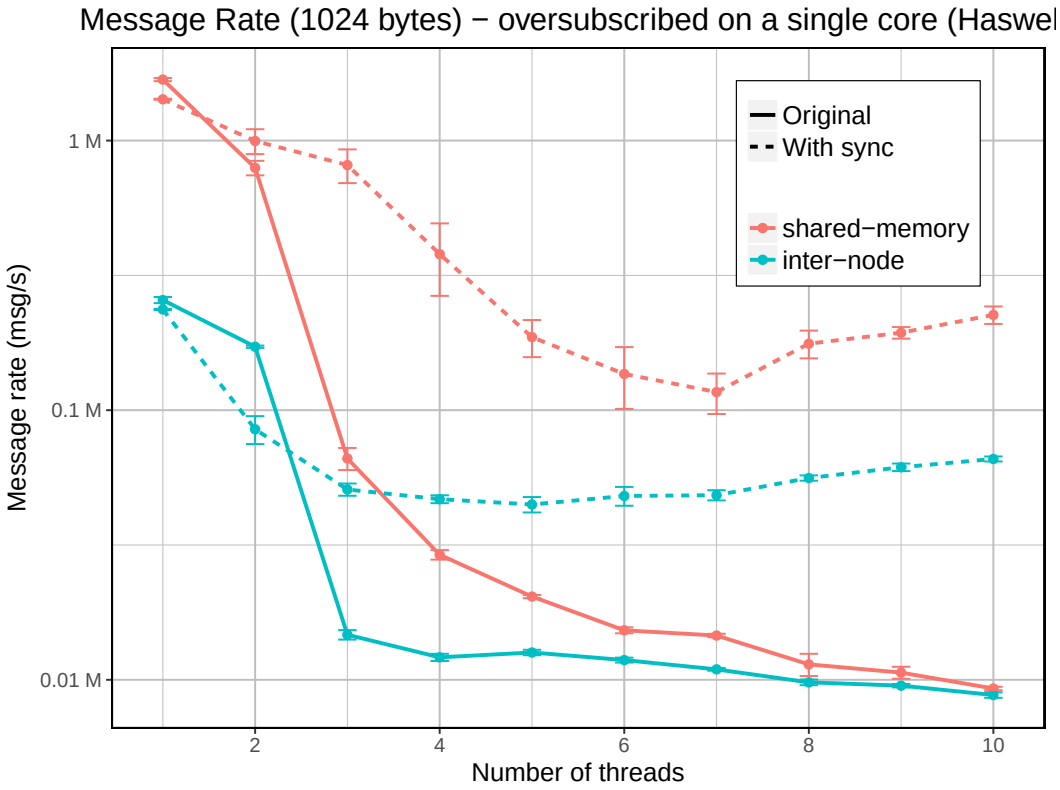


Figure 4.4: Message Rate in thread over-subscription scenario.

cannot be a simple assignment operation as it is prone to the race conditions (in the case where the request becomes complete while attaching). The solution is an atomic compare and swap operation, which is significantly more costly than a simple assignment operation. The performance impact from the atomic operations is not severe for MPI_Waitall, as every request has to be completed before returning, limiting the chance of unnecessary detaching. However, for MPI_Waitany and MPI_Waitsome, when the user usually calls with the same set of requests over and over, the performance might not be optimal, as the requests have to be attached and detached before returning them to the user for every call. The impact prevents this design from becoming feasible for MPI_Test operations where the completion is not required.

This section presents the extension to the MPI API to allow user-level usage of the synchronization object. With the user-level API, the synchronization object can further provide more flexibility and functionality for a multi-threaded MPI environment, including avoiding unnecessary detach operations. I propose an extension of the synchronization object to the user level through the MPIX notation to demonstrate the potential of my design.

The MPIX_Sync API

I propose a new MPI object, MPIX_Sync, with 5 user-level APIs to interact with the synchronization object: INIT, ATTACH, DETACH, QUERY and QUERY_BULK.

- **MPIX_Sync_init**: Initialize the sync object.
- **MPIX_Sync_attach**: Attach a request to the synchronization object with associated callback data. The callback data will be returned as the reference to the user when the request is complete in the query API. The request is detached from the sync object automatically after its completion.
- **MPIX_Sync_detach**: Detach a request from the sync object.
- **MPIX_Sync_query**: Query the sync object for a request completion. Return the callback data of a completed request. Similar to MPI_Testany API.

- **MPIX_Sync_query_bulk**: Query the sync object for multiple request completions. Return the callback data of completed requests. Similar to MPI_Testsome API.

For the implementation of the proposed API, I take the current design of the synchronization object and expand its functionality to be appropriate to use from user level. Each sync object consists of a completion counter and a completion queue to store the callback data. Once the user attaches the request to the sync object with user-specified callback data, the user relinquishes the request to the MPI implementation, and should now only rely on the callback data they associated with the request. Figure 4.5 depicts the general design of the API. The accurate C API, along with the user guide for the MPI extension, can be found in the Appendix B.

When an operation completes, the callback data associated with the operation is added to the completion queue, and the counter gets updated accordingly. The synchronization object keeps track of the number of outstanding completions and the callback data for each completion of the operations. The user can query the completion through QUERY, which will return the callback data from the completion queue in first-come, first-serve manner. This queue is protected for thread safety.

The MPI standard prohibits concurrent wait or test operations on the same MPI request. For example, the user cannot perform MPI_Test on the same MPI request simultaneously from multiple threads. However, it is a common practice for some categories of application such as the runtime scheduler or the work-stealing programming model, which relies on

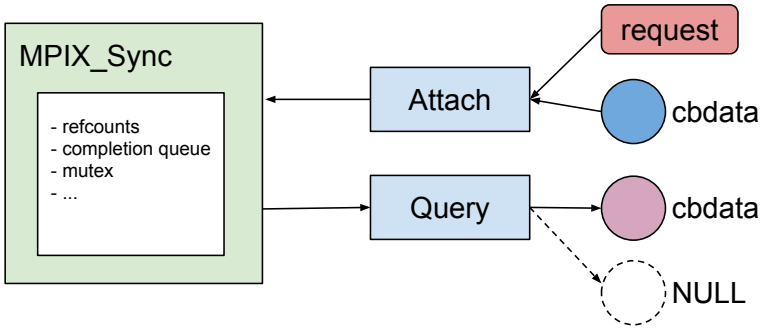


Figure 4.5: The MPIX_Sync API design.

posting persistent wildcard receive requests in anticipation of a message or "task" from other peers, and constantly performs MPI_Test on the requests to detect the incoming message. The applications that make communication decisions at the runtime such as PaRSEC Bosilca et al. (2013), rootsim Pellegrini et al. (2011) and Graph500 Ang et al. (2010) only use a single thread for communication due to this limitation. The MPIX_Sync API, with proper thread protection, allows for multiple threads to check for completion on the same set of requests simultaneously through QUERY and QUERY_BULK, providing the opportunity for more flexible message completion routines with thread parallelism, increasing the usability of MPI_THREAD_MULTIPLE.

As the synchronization object's counter always keep tracks of the number of outstanding completions, it eliminates the need for the loop over every request to check for completion status. The user can check the number of completions just by reading the value of the counter, complexity: O(1) instead of O(n). Additionally, with user-level control, the user can query the same synchronization object again for more completion of attached requests without having to reattach them, circumventing the cost of atomic operation associated with attaching/detaching procedure.

With the user-defined callback data, my proposed API relieves the burden of bookkeeping from the applications, as they no longer need to keep track of MPI requests. The user can define their own completion scheme, or use the callback data in the same manner as the "Active Message" approach to direct the flow of their application. The current API is still evolving. I plan to explore the possibility of a user-level callback function where the MPI will execute the user-provided function as soon as the request is completed.

For the evaluation, first I demonstrate the improvement from the proposed API by timing each call of MPIX_Sync_query comparing to MPI_Testsome by varying the number of requests given to the API. The result is illustrated in Figure 4.6. The MPIX_Sync API has the advantage of using the counter to check for completion instead of looping over every request. In the case of no completion, the MPIX_Sync_query gives optimal performance. However, with a higher number of completed requests, the benefit starts to drop off and becomes comparable to the original MPI_Testsome.

For real-world application evaluation, as a collaboration with Reazul Hoque, a graduate student from the University of Tennessee, we take PaRSEC Bosilca et al. (2013), a task-based runtime, and modified its communication engine to use MPIX_Sync API instead of the original MPI_Testsome for request completion. PaRSEC relies heavily on persistent requests and only uses a single dedicated communication thread in MPI non-threading mode. We perform the experiment on two different PaRSEC subroutines and demonstrate the result in Table 4.1. First, ping-pong, which involves only the communication workload. We can see the performance improvement of 13% for the small message, and the performance benefit diminished as the message size increases. This is expected behavior as the larger the message, the more execution time that will be spent in the actual communication—thus, less impact from MPI overhead. Second, we tested with PaRSEC DPLASMA, linear algebra operation. We cannot observe a significant difference between the two APIs. However, it should be noted that the PaRSEC communication engine is already highly optimized by utilizing techniques such as re-packing the MPI requests to match the completion order. Thus, the performance benefit from this design is expected to be minimal. At this stage, we have not yet altered the PaRSEC communication engine to allow multiple threads for communication.

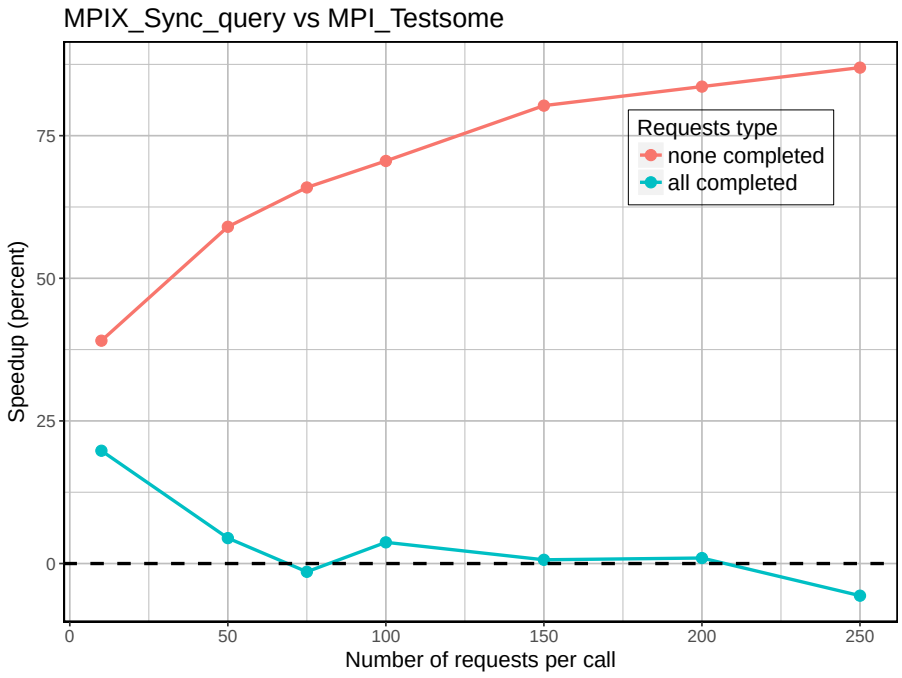


Figure 4.6: MPIX_Sync_query performance comparing to MPI_Testsome.

Table 4.1: PaRSEC performance speedup from MPIX_Sync API.

Pingpong Message Size (Bytes)	Speedup (%)
400	13.73
4000	7.15
40000	3.1
400000	2.55

Kernel	Speedup (%)
dpotrf	∼0*
dgeqrf	∼0*

4.6.2 Thread Pool

Yielding the CPU core back to the user might be great in the case of thread over-subscription, as it reduces the chance of context switching between threads. However, with the current hardware trend being more cores per chip, it is very likely that each thread will have a one-to-one mapping to the physical CPU core. With this in mind, it is better to utilize the CPU core while waiting instead of having them in the idling state.

The thread pool design enables the utilization of waiting threads. Instead of de-scheduling the threads, the threads are constantly looking for tasks to execute. I implemented a task-stealing model for the waiting threads. The task can be generated from any component of the MPI implementation, including the progress engine. For example, the matching process for a message can be passed off as a task. Generating tasks for other threads to execute might shave off execution time of the critical path and gives a better overall performance (Figure 4.7).

For demonstration, in an ongoing collaboration with Yicheng Li, at the University of Tennessee, Knoxville, on his research of Open MPI datatype engine optimization, we utilize the thread pool, task-stealing design to parallelize the packing operation of MPI vector datatype messages. In this experiment, the packing operation (via MPI_Pack) is split into several tasks while several threads are actively waiting to execute tasks in SYNC_WAIT. Once the tasks are created and added to the queue, the waiting threads pick them up and execute them in parallel. Figure 4.8 illustrates the achieved bandwidth with parallel packing via thread pool design. We observe the speedup when the buffer size is beyond 100KB and see the most benefit when the buffer size is around 10 MB. This is proof of concept that the thread pool design is one of the approaches that can extract more thread parallelism from threads performing the wait operation. Currently, we are experimenting with different task types from inside the MPI implementation.

From the early evaluation, MPIX_Sync API is the worst case, performing on the same level as MPI_Test API but provides more benefits in some cases. We have not yet evaluated the performance impact in a multi-threaded environment.

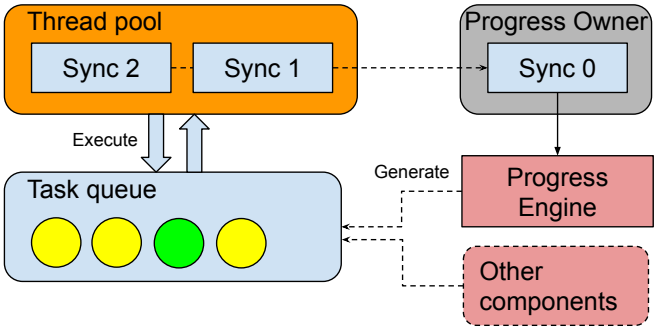


Figure 4.7: Thread pool design utilizing the synchronization object.

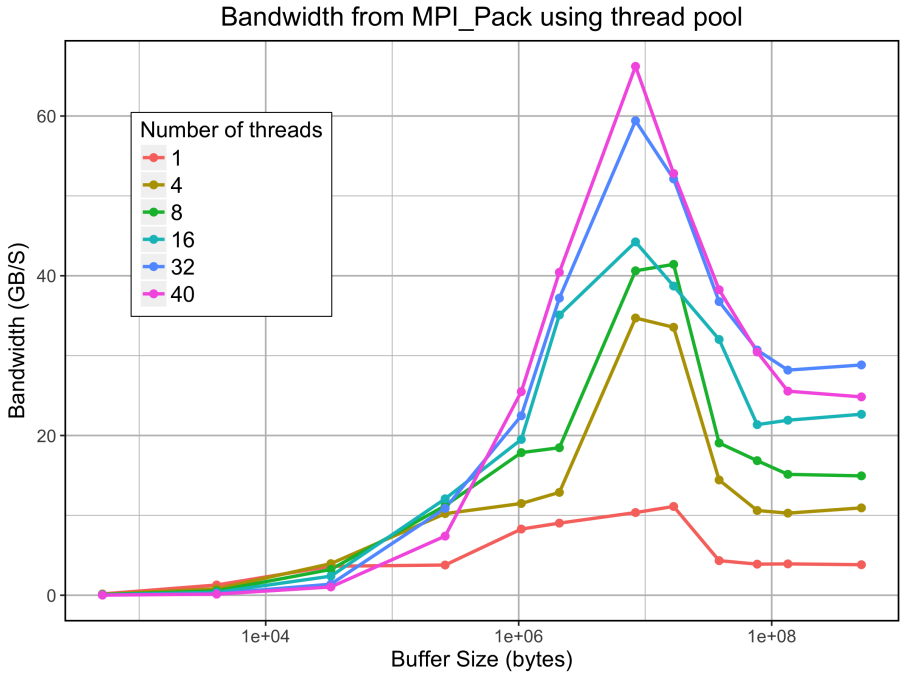


Figure 4.8: MPI_Pack performance when utilizing threads in the threadpool design.

4.6.3 Multi-Threaded Progress Engine

The progress engine is a crucial part of MPI communication that still remains serial. Another approach in utilizing thread parallelism is to execute the progress engine in parallel. The synchronization object design as a thread management layer can be modified to allow concurrent access to the progress engine, while still maintaining the capability of the thread pool design and user-level design, increasing the opportunity for more thread optimization. That being said, in order to attain the parallel progress engine, each component registering itself to the progress engine has to become thread-safe, which imposes a burden onto the component owner. Nonetheless, in this study, I focus mainly on this approach—to investigate the potential of concurrent execution of the progress engine for true thread parallelism. I discuss my design and implementation in detail in chapter 5.

4.7 Conclusion

This chapter introduces the thread synchronization object, a novel approach which provides more nuance in thread management for MPI implementation. I utilize the synchronization object to mitigate the known bottleneck at the MPI progress engine, allowing for better performance for both normal and thread over-subscribed cases in MPI_Wait variants. I showed the performance gain of 7× for shared memory and 3× for inter-node threading communication.

I explore other potentials of the thread synchronization object to further harness the power of thread parallelism in MPI. I presented several prototypes and proofs of concept for my designs, including the extension of the concept to the user level, which will provide more flexibility for MPI programming paradigm with better threading support. While the research of these possibilities is ongoing, I showed significant benefits of the synchronization object. Moving forward, I focus mainly on utilizing multiple threads inside the MPI progress engine to speed up overall communication. The topic is discussed in detail in chapter 5.

Chapter 5

Design of True Thread Concurrency in MPI

5.1 Overview

In this chapter, I propose a design that enables true thread concurrency for the MPI implementation. My design addresses two problems: (1) the resource contention when multiple threads are accessing the same network resources to perform communication; and (2) that only a single thread is allowed to execute the progress engine at a time, ultimately eliminating the opportunity to utilize thread parallelism in communication. The two intertwining problems are the remnants from the original bulk synchronization design where only a single resource is available while multiple threads race to access it, creating massive lock contentions.

I introduce the concept of Communication Resource Instances (CRIs), objects that encompass every required resource to perform the communication, which can be allocated multiple times. Next, I expand on the thread synchronization object from my work in chapter 4 to propose a design that allows multiple threads to access the resource instances in parallel, enabling them to perform multiple communication operations simultaneously. I discuss the design in detail along with its benefits and shortcomings. I evaluate my design with Multirate benchmark (chapter 3, Patinyasakdikul et al. (2019)) while obtaining the internal information from Open MPI via the built-in software counters Eberius et al.

(2017) for two-sided communication, and multi-threaded one-sided communication, RMA-MT Dosanjh et al. (2016) benchmark.

The evaluation results show that my approach can achieve up to 2× performance gain from the CRI design alone. Furthermore, I show that the parallel matching process is the key to achieve better performance (up to 10×) for multi-threaded MPI. Lastly, the results of my design in one-sided multi-threaded communication illustrated that, without the matching process, it can achieve up to 200× the performance of the original design.

5.2 Background

There are multiple challenges that need to be addressed in order to improve the performance of the multi-threaded environment in MPI. This section presents a high-level background of the communication process and the challenges in multi-threaded optimization from the MPI implementation's perspective.

Communication Resources

In order to perform communication, the MPI implementations utilize the low-level networking library such as socket, or verbs to interact with the network hardware. Recently, there are efforts to unify the network library under a single standard such as the Open Fabric Interface (OFI) Grun et al. (2015) or Open UCX Shamis et al. (2015) which provide the high-level abstraction for the high-performance network devices with HPC capabilities such as RMA and hardware tag-matching.

Generally, the MPI implementation interacts with the network hardware through the allocated network resources with the associated network library. The resources such as 'network endpoint' are the handle to the network hardware. In order to perform the communication, the MPI implementation has to issue the send or receive command to the underlying network library with an endpoint. The other critical resource is the completion queue (CQ) which is usually attached to an endpoint. When an operation completes, the network library generates a completion event and put it in the completion queue. The MPI

implementation has to read the completion queue and report the completion back to the user-level appropriately.

Since the MPI implementation has to associate the completion event with the issued operation from the user level, it has to store the information to look up later. However, it is not optimal to allocate and free the memory for every operation. Most of the MPI implementations usually utilize a buffer pool, a common technique which allocates a chunk of memory in advance, and provide a mechanism to request and return the memory to the pool to enable the reuse of the memory, and avoid the costly memory allocation in time-critical operations.

Resource Allocation

One major difference between using multiple MPI processes versus a single MPI process with multiple threads is the resources allocated for MPI operations. Resources such as buffer pools, network contexts and endpoints, or CQs are generally created per MPI process. In the process-to-process communication model, with this single producer–single consumer relationship, resource contention is limited. In the case of multiple threads in the same MPI process, these resources have to be protected, as concurrent access to a resource may not be supported, or might create race conditions that could compromise the correctness of the communication or even corrupt the state of the MPI library. At the same time, this protection adds an extra cost to the operation, and the cost often increases with the number of concurrent threads. Moreover, the protection effectively eliminates any opportunity for performing network operation in parallel.

Matching Process

The matching engine is another important piece of an MPI implementation for handling incoming messages, as it is responsible for the correct matching of sends and receives.

For single-threaded applications, the MPI standard offers the guarantee that all messages between a source and destination pair on the same MPI communicator are matched in FIFO order, ensuring that the send order is the same as the matching order. This simplifies the

semantics for the MPI users, as it ensures that, in single-threaded applications, with the same peer, messages are always delivered in each communicator in a deterministic order.

However, at the network level, the story is different. For performance and routing optimization reasons, networks do not provide any ordering guarantee by default and the messages might be delivered in an arbitrary order. This requires the MPI library to implement a software solution to provide users with the required message ordering guarantee. For multi-threaded usage, the MPI standard only guarantees message ordering within a single thread. Messages sent from different threads are only guaranteed to happen in some serialized order, as MPI communications, even blocking, are not synchronizing.

The algorithms to provide message ordering may be different for each MPI implementation, but they share a common approach: generate a sequence number for each message and pack it within the message header. For simplicity, this sequence number is generally per peer per communicator. The receiver extracts the sequence number from the incoming header and uses it to ensure messages are processed in the same order they were sent. Any message arriving out of sequence needs to be saved for matching at a later time when that message sequence number is called for. The implementation has to allocate the necessary memory to store the out-of-sequence messages, adding an extra overhead to the operation. The out-of-sequence messages can occur in a single-threaded scenario, where sometimes the network device determines to switch the sending order of messages for optimization reasons—but the occurrence is usually very rare, and therefore the overhead is negligible. However, this is not the case for multi-threaded MPI. In the scenario with multiple threads concurrently sending messages on the same communicator to the same destination MPI process, given the nature of their non-deterministic behavior, threads can easily compete and send the messages out of order. With more likelihood of out-of-sequence messages, multi-threaded MPI could suffer significant performance degradation with an increasing number of threads from the out-of-sequence message handling overhead.

After the MPI implementation successfully validates the sequence number of an incoming message, the message is matched against a queue of the user's posted receives. This code region is a critical section and must be protected with a lock in a multi-thread scenario to prevent concurrent access to the queue. For example, races can occur when threads are

simultaneously posting receives; or when a thread adds a request to the posted receive queue while another thread is in the progress engine trying to match an incoming message with a request on the same queue. The matching lock is mandatory for the correctness of MPI operation in a multi-threaded environment, but it also serves as a huge critical section that prevents the MPI implementation from achieving more thread parallelism.

The matching process plays a significant role in determining the latency of the two-sided communication, especially in a multi-threaded environments. Currently, there are many efforts to improve the matching process. Network hardware vendors such as Mellanox, Intel, and Cray incorporate the matching process into the hardware itself. The recent network hardware with tag-matching capabilities relieves stress from the software stack, such as MPI, from implementing their own solution. However, this approach moves the burden onto the hardware, which might not have enough software-level information to make the right optimization decision. On the other hand, researchers are studying multiple techniques to speed up the entire matching process, ranging from utilizing the vector instruction for fuzzy matching Schonbein et al. (2018) to the algorithmic approaches such as Flajslik et al. (2016). Nonetheless, there is still no working implementation to utilize multiple threads to perform message matching simultaneously.

Remote Memory Access

In addition to two-sided communication, the MPI-3.1 standard provides support for one-sided (RMA) communication. This support allows an MPI implementation to directly expose hardware RDMA, a feature which is present on most high-performance networks (e.g., Infiniband and Cray Aries). This allows the MPI implementation to offload communication directly to the hardware. In addition, the one-sided model separates communication (data movement) from the synchronization (completion). There is no need for any explicit matching for one-sided communication, removing a potential multi-threaded bottleneck. This makes RMA well suited for multi-threaded applications, but at the same time, it moves the burden of the synchronization to the user, and potentially increases the complexity and readability of the application's code.

With the current MPI standard there is support for three different types of communication operations: put (remote write), get (remote read), and accumulate (remote atomic); and for two classes of synchronization: active-target (fence, post-start-complete-wait), and passive-target (lock, flush). Active-target requires the target MPI process of an RMA operation to participate in the synchronization of the window. It is not well suited for multi-threaded applications, as all synchronization needs to be funneled through a single thread. Passive-target flush, on the other hand, does not require the target of an RMA operation to participate in either the communication or synchronization and allows for concurrent synchronization.

5.3 Design and Implementation

This section presents the designs to allow true thread concurrency in the MPI implementation. The goal is to optimize for maximum thread parallelism by giving them proper resources, removing any unnecessary critical sections to create more opportunity for threads to collaborate instead of racing against one another.

5.3.1 Communication Resources Instance

There are a variety of critical internal MPI resources that must be protected in a multi-threaded environment, such as the network endpoints, network contexts, and CQs. In existing MPI implementations, a single network context is typically created per MPI process and a single network endpoint per source/destination pair. The CQ is usually attached to the network context to store completion events. For multi-threaded MPI, access to both network contexts and their CQs may have to be protected, thus creating a potential bottleneck.

To give multi-threaded MPI a fair chance, more resources have to be allocated for the entire MPI process. I introduce the concept of a Communication Resources Instance (CRI) to encompass resources such as network contexts, network endpoints, and CQs with a per-instance level of protection to perform communication operations. The MPI implementation can allocate multiple CRIs internally for multi-threaded needs.

Currently, there is no interoperability between threading frameworks such as POSIX threads and MPI; therefore, the MPI implementation does not have a standardized way to get the number of threads that will be used for MPI communication from the application. Thus, it is challenging for the implementation to assess the proper number of CRIs to allocate. That being said, an implementation can provide the user with a way to give a hint via environment variable(s), MPI info key(s), or other means (MCA parameters Squyres for Open MPI Gabriel et al. (2004) or the new MPI control variables MPI_T_cvar) to let the implementation know how many threads the application will use for concurrent MPI operations. The implementation can then allocate the CRIs accordingly. In my implementation, MPI allocates a set of CRIs into a resource pool and creates a centralized body to assign the allocated instances to threads.

Ideally, there should be a one-to-one thread to CRI mapping to completely eliminate the potential for lock contention. However, in some cases, there might be a limit to the resources available for creating CRIs. Some network devices, such as Cray Aries, might have a hardware limitation on the number of network contexts the user can create, so the design must also accommodate cases where the number of CRIs is less than the number of threads.

Giving more resources to the threads might not be sufficient to increase communication performance for two-sided communication, as the MPI implementation still serializes the calls to both the send operation and progress engine to prevent any potential race conditions. In order to benefit from more allocated resources, both the send and receive paths have to be redesigned to allow for more parallelism while maintaining thread safety and continuing to ensure the expected matching semantic.

5.3.2 Try-lock Semantics

Using locks to protect critical resources is one of the popular approaches to ensure thread safety. These locks also act as a funnel when multiple threads are going through the same code path as lock contention will cause threads to block. We can mitigate the funneling effect by using try-lock semantics, which is a non-blocking version of lock, where it will return immediately after it fails to acquire the lock.

Try-lock semantics provide more opportunities for parallelism. When the lock is already taken, we can be certain that a thread is progressing that particular code path, and, therefore, the current thread can move on and try to pick up another code path to execute or become a helper thread and complete other menial work.

The following subsections describe how to leverage the try-lock semantics with the communication resources instances (I will further refer to them as CRIs or "instance" in the following sections), to alleviate resource contention from MPI's internal message extraction process.

5.3.3 Concurrent Sends

For the MPI implementation to perform a send operation, it needs access to a network endpoint. In the multi-threaded case, the implementation usually protects the network context with a lock. In this new design, the network context is associated with a CRI along with other resources. The protection is changed from per-endpoint level to per-instance level, allowing the threads to perform send operations simultaneously on different instances. To assign a CRI to a thread, I propose two strategies: round-robin and dedicated (Algorithm 3).

Round-Robin Assignment

In this strategy, every time a thread needs to communicate it first acquires a CRI. The MPI implementation assigns an instance for single use on a first-come, first-served manner. Once the last available instance is assigned, the implementation will recycle the instances and then give out the first instance again. This approach reduces the possibility of lock contention by assigning a different instance for every call. It also improves load balancing by giving a fair share of work among the allocated instances.

Dedicated Assignment

To permanently assign a CRI to a thread, Message Passing Interface (MPI) can utilize Thread-Local Storage (TLS), provided either by the threading library (e.g., POSIX threads)

Algorithm 3 Utilizing multiple CRIs to allow concurrent sends.

```
1: function INIT
2:     for i ← 1, NumInstances do
3:         instance[i] ←CREATE-INSTANCE()

4: function SEND(msg)
5:     k ← GET-INSTANCE-ID()
6:     LOCK(instance[k] → lock)
7:     NETWORKSEND(instance[k], msg)
8:     UNLOCK(instance[k] → lock)

9: function GET-INSTANCE-ID–ROUND-ROBIN
10:    static current_id ← 0
11:    ret = current_id
12:    current_id ← current_id + 1
13:    return (ret  mod numInstances )

14: function GET-INSTANCE-ID–DEDICATED
15:    static thread_local my_id ← undefined
16:    if my_id is defined then
17:        return my_id
18:    else
19:        my_id ← GET-INSTANCE-ID()–ROUND-ROBIN
20:        return my_id
```

or the programming language (e.g., C11, C++11). This approach can only be implemented when the system or the compiler supports TLS, a pretty standard feature nowadays. My implementation uses the native compiler support either from C11 or GCC. When checking for a CRI to use, the implementation can check if instance information is stored in TLS. If not, it can assign an instance with a round-robin assignment and save the instance information in the TLS. With a dedicated assignment strategy, there is no possibility of lock contention on the instance as long as the number of threads is lower or equal to the number of instances allocated. If not, some communicating threads might share the same instance and might even introduce some lock contention if they simultaneously communicate.

5.3.4 Concurrent Progress

Traditionally, Open MPI serializes calls into the progress engine, allowing only a single thread to progress communications. Such coarse-grained protection under-utilizes the available thread parallelism and limits the speed of message extraction to the power of a single thread. To allow threads to extract messages concurrently, the serialization is removed from the progress engine. The design exploits the instance-level protection to provide the required thread safety instead.

The progress engine also suffers from the lack of threading information in MPI. When a thread makes a call into the progress engine, it requires an instance to progress. This design utilizes the same centralized body as concurrent sends to assign an instance to a thread. The strategies to choose which instance to progress are similar to how the instance is chosen for the send path, namely, *Round-robin* and *Dedicated* (Section 5.3.3).

For the *Dedicated* strategy, with a permanent instance assigned to each thread, a few issues need to be addressed. First, the MPI implementation has to make sure that it progresses every allocated CRI to prevent a deadlock scenario where message completion is generated in an instance that is not progressed by the associated thread. Second, the user might destroy the thread and create orphaned CRI that cannot be reused by other threads. To overcome this limitation, each thread is mandated to try progressing their dedicated instance first, and if there is no completion event, move on to try progressing other instances. This design provides the guarantee that every instance will eventually get progressed while still maintaining the optimization benefit from TLS.

Furthermore, the try-lock semantics on the instances become a valuable weapon to the efficiency of concurrent progress design (Algorithm 4). If a thread fails to acquire the lock for an instance, it assumes that another thread is progressing that particular instance, and the current thread can try to pick up another instance to progress or return.

5.3.5 Concurrent Matching

The matching process is still, largely, a serial operation. By changing from serial progress to a concurrent progress engine, the design effectively moves the bottleneck to the matching

Algorithm 4 Dedicated instance assignment to give priority to the thread assigned instance before trying to progress others, ensuring eventual progress for every instance.

```
1: function COMMUNICATION PROGRESS
2:     count ← 0
3:     k ←GET-INSTANCE-ID()–DEDICATED

4:     if trylock → instance[k].lock = success then
5:         progress instance[k]
6:         count ← number of completions
7:         unlock → instance[k].lock

8:     if count = 0 then
9:         for i ← 1, NumInstances do
10:            k ← GET-INSTANCE-ID()–ROUND-ROBIN
11:            if trylock → instance[k].lock = success then
12:                progress instance[k]
13:                count ← number of completions
14:                unlock → instance[k].lock
15:            if count > 0 then
16:                return
```

process. As long as this process still cannot be performed in parallel, it will be challenging to get the optimal performance from multi-threaded MPI (Figure 5.1).

The current message matching design from state-of-the-art open-source MPI implementations such as MPICH and Open MPI drastically differ. Even in the context of the same MPI implementation, the matching infrastructure can be different depending on the network used (Portals provides hardware matching), the hardware capabilities (AVX provides opportunities for vector matching), and the configured software stack. As an example, Open MPI supports multiple methods for matching, going from hardware matching when the Portals library is used, to a single global queue when using the UCX PML, to a vector fuzzy-matching single global queue Schonbein et al. (2018) and finally to the default, more decentralized matching in the OB1 PML (with a matching queue per process per communicator with special arrangements for MPI_ANY_SOURCE).

A study of optimized or parallel matching is not within the scope of this study. However, the potential of concurrent matching can still be shown by utilizing OB1, a point-to-point matching layer (PML) component designed to perform the matching process per MPI

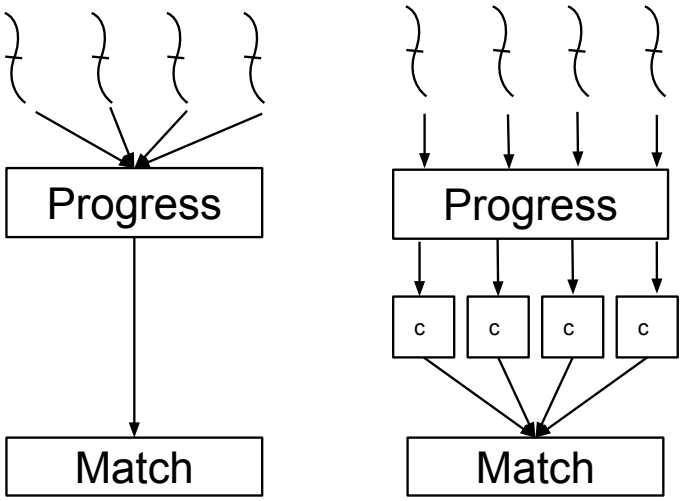


Figure 5.1: Matching process with serial and concurrent progress engine.

communicator instead of globally. We can then simulate the concurrent matching process by creating multiple communicators and allowing threads to perform matching in parallel, unhindered. While this approach might not be practical for some real-world applications, it is sufficient to demonstrate the potential of multi-threaded MPI.

5.4 Experimental Evaluation

I implemented the strategies presented in this chapter by taking advantage of the modular design in Open MPI, utilizing the OB1 point-to-point messaging component (pml/OB1) in conjunction with the *uct* and *ugni* Byte Transfer Layer (BTL) components (btl/uct) which were updated to use multiple CRIs. I modified the Open MPI progress engine (opal_progress) to allow multiple threads in the progress engine.

To gain low-level insights into the different statistics related to the communication engine, I took advantage of Open MPI's built-in Software-based Performance Counters (SPCs) Eberius et al. (2017) expose internal MPI information with low overhead. SPCs offer a variety of measurements from the MPI level such as the number of messages sent/received as well as MPI internal information such as the number of unexpected or out-of-sequence messages, the cost of matching, or the length of the matching queues. This study only focuses

on two of these counters: the number of out-of-sequence messages and the total matching time.

To evaluate the impact of each strategy presented in this study, the message rate is measured by the Multirate benchmark Patinyasakdikul et al. (2019) in pairwise pattern for two-sided communication, and RMA-MT benchmark RMAMT for one-sided communication. Several hundred experiments are performed; the mean and the standard deviation are reported within the figures, which should be noted is consistently very small.

Multirate–pairwise spawns pairs of communication entities which can be mapped to either an MPI process or a single thread to perform communication simultaneously (Figure 3.3). The two-sided communication experiments use the message size of zero byte, as it allows us to capture only the cost of message movement as MPI sends necessary matching information to be matched on the receiver side without the user-level message (the size of this matching header is small in Open MPI, around 28 bytes).

RMA-MT is a benchmark developed at Sandia National Lab (SNL) and Los Alamos National Lab (LANL) to stress-test an MPI implementation under a heavy multi-threaded MPI RMA workload. The experiment's results are from the University of Tennessee's Alembert and LANL's Trinitite cluster. The specifications of the systems used are presented in Table 5.1.

Table 5.1: Configuration of the testing systems, *Alembert* and *Trinitite*.

Property	Alembert Configuration	Trinitite Configuration
Processor	Dual 10-core Intel Xeon E5-2650 v3 @2.3 Ghz	Dual 16-core Intel Xeon E5-2698 v3 @2.3 Ghz
Microarchitecture	Haswell	Haswell
Main Memory	64GB DDR4 2,133 MHz	128GB DDR4 2,133 MHz
Interconnect	InfiniBand EDR (100 Gbps)	Cray Aries (100 Gbps)
Operating System	Scientific Linux 7.3	Cray Suse Linux
Compiler	GCC 8.3.0	GCC 8.3.0

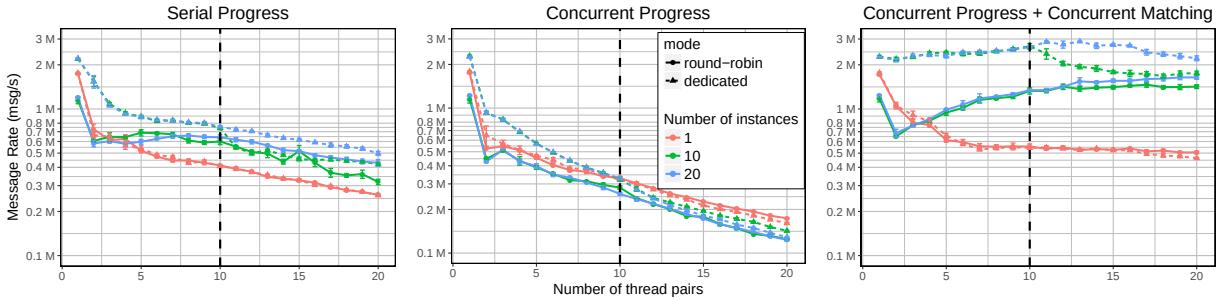
5.4.1 Concurrent Sends

Figure 5.2a demonstrates the effect of allocating additional internal resources, CRI. The original design of serial progress (only allowing a single thread to perform the network extraction at a time) is used for the experiment. By only introducing changes on the sender side, these experiments demonstrate the impact of increasing resources availability, thus decreasing contention on the send path. This allows multiple threads to reach the lowest network level simultaneously, each in different contexts, and technically performing send operations concurrently. I employ the two strategies described in Section 5.3.3 to assign an instance to a thread: *round-robin* and *dedicated* presented by solid and dashed lines, respectively. Each color represents a different number of instances allocated for the experiment.

The red lines represent the base performance, the original multi-threading support in Open MPI, with a single instance shared between all threads. The contention impact is visible very early, basically starting from 2 threads. This scenario is very demanding, as all threads sharing the same instance will fight for the same protection lock, and the lock will therefore always be contested.

Ideally, a one-to-one mapping from a thread to an instance should give the best performance, as there is no contention on the instances. The scenario is achieved by employing the *Dedicated* strategy for this experiment, represented in blue-dashed lines (with 20 threads, 20 instances). Just by increasing the number of instances, we can see a performance gain up to 100% compared to the original case. When the number of instances is reduced to 10, the performance drop-off begins to appear after going over 10 threads, as the threads start sharing the instances and thus introducing some congestion (green-dashed line).

Although the *round-robin* strategy (solid lines) does not give the best performance, it softens the effect of the congestion significantly by spreading the instance evenly among threads, thus reducing the lock congestion. It is still a viable strategy when *Dedicated* cannot be implemented due to the lack of compiler support on the platform.



(a) (b) (c)

Figure 5.2: Zero byte message rate on different strategies.

Table 5.2: Software Performance Counters information from last data point of the experiment

	Serial Progress			Concurrent Progress			Concurrent Progress + Matching		
Number of instances	1	10	20	1	10	20	1	10	20
Out-of-sequence messages	2,154,493	2,323,003	2,225,190	2,375,922	2,425,818	2,420,660	15,188	45	0
Out-of-sequence (%)	83.32%	89.98%	86.08%	91.89%	93.82%	93.62%	0.59%	≈ 0%	0%
Match time (ms)	2,732	2,622	2,738	8,553	7,944	8,069	476	430	389

The performance metrics obtained from the SPC is presented along with Figure 5.2 in Table 5.2. For clarity and conciseness, I only present the information from the last data point from the best result of each figure, at 20 thread pairs, 20 instances with the *Dedicated* assignment strategy. In general, for serial progress, the SPCs show similar numbers of out-of-sequence messages (up to 90%) with similar time spent in matching.

5.4.2 Concurrent Progress

Figure 5.2b presents the performance impact from concurrent progress. The difference with the above experiment is the concurrent progress which basically allows multiple threads to execute the progress engine simultaneously.

Concurrent progress hinders the performance instead of boosting it, even with increased parallelism (Figure 5.2b). The results show a funneling effect as the number of threads increases, regardless of the number of instances or the assignment strategy, just as expected. The potential parallelism from concurrent progress is restricted and cannot give a performance boost as long as the matching process remains a serial operation; the approach effectively moves the bottleneck from the progress engine to the matching process (Figure 5.1).

The SPC information from Table 5.2 reveals that the MPI implementation is spending up to 300% more time in matching comparing to the earlier experiment, which is consistent with my expectation.

5.4.3 Concurrent Matching

This experiment relaxes the constraints of the matching, in hopes of improving upon the previous experiments. To simulate a concurrent matching process, this experiment creates multiple communicators, taking advantage of the matching logic in the OB1 PML, with matching queues private to communicators. Since the pml/OB1 component in Open MPI performs matching per-communicator, this effectively provides us with support for concurrent matching.

Multirate–pairwise provides an option to assign a communicator per each pair of communicating threads. With a unique communicator per thread pair along with concurrent sends and concurrent progress, this part of the experiment represents the multi-threaded performance when the contention in the matching process is minimal.

The results are demonstrated in Figure 5.2c. Even *Round-robin* assignment (solid lines) shows performance improvement with the number of threads, a completely different outcome from the earlier experiments. The instance assignment strategy seems to perform well even after the number of threads is greater than the number of instances. For this strategy, messages from the same communicator can be sent out from different instances. There are chances that the receiver, as their threads extract the messages simultaneously from multiple instances, will perform matching on the messages from the same communicator and introduce some congestion (Figure 5.1).

Dedicated assignment gives the best performance as each thread always uses the same network instance in addition to using the same communicator (dashed-lines). The blue dashed line represents an ideal scenario with one-to-one mapping from thread to CRI to a communicator. The performance is scaling with the number of threads but drops back down with a large number of threads, suggesting other possible bottlenecks. The green-dashed line shows the same performance scaling until the threads have to share instances (at 11 threads and over) before dropping off similarly to the blue-dashed line.

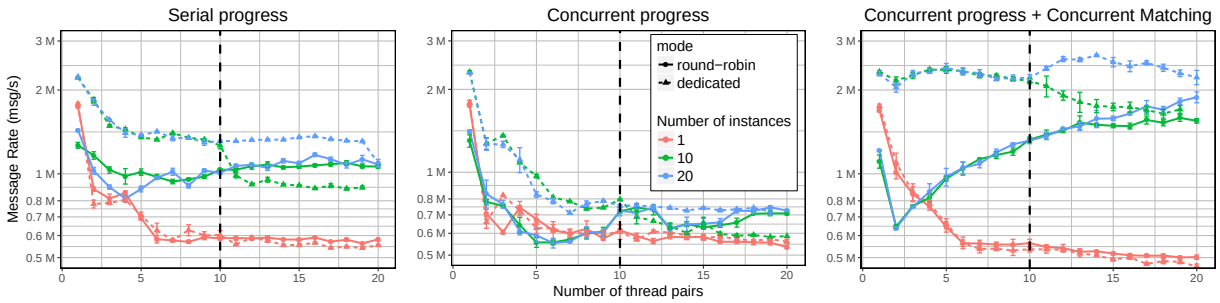
The information from the SPCs also shows drastic improvement over earlier experiments as the number of out-of-sequence messages drops significantly after introducing more instances. The match time is minimal as there is a guarantee for no contention on both the instance and the matching process. However, using dedicated communicators for each communication thread pair might not be practical for every application. Nonetheless, the experiment successfully shows that the major bottleneck for multi-threaded MPI is the matching process contention.

5.4.4 Message Overtaking

We can break the matching process into two parts: sequence number validation, and the queue search to match messages with MPI requests. As described in Figure 2.5, out-of-sequence messages force the MPI implementation to allocate memory to buffer the message for processing later which is a costly operation in the critical path. This experiment allows MPI to ignore the sequence number validation by providing the MPI info key: *mpi_assert_allow_overtaking* to the communicator, allowing MPI to ignore the sequence number and therefore to immediately match every incoming message. This info key is not novel: it has been intensely discussed in the MPI Forum and has been approved for inclusion in the next version of the MPI standard. This study can serve as validation for the usefulness of this info key in threaded scenarios.

Allowing the MPI implementation to match every incoming message immediately will lead to high stress for the queue search. When using multiple tags, the queue search is a linear operation where the cost increases with the queue length. When a message is matched out of sequence, the average time to search the queue is increased as the request associated with the message might be at the end of the queue. To fully reap the benefits of message overtaking, the Multirate–pairwise is modified to post the receive with a wildcard tag (MPI_ANY_TAG) to force the implementation to always match the incoming message with the first posted receive request, skipping the queue search entirely.

This experiment represents the multi-threaded MPI performance when the cost of the matching process is minimal. The same set of experiments from earlier are performed with the tweak. The result is demonstrated in Figure 5.3.



(a) (b) (c)

Figure 5.3: Zero byte message rate when the message ordering is not enforced

If we take a look at the serial progress performance (Figure 5.3a), for a single instance (red lines), we can still see that increasing the number of instances helps in giving some performance boost from the sender side. the message rate flattens out around 500K msg/s and it does not drop with the increasing number of threads as the earlier experiment (Figure 5.2a). This suggests that the source of performance degradation in multi-thread MPI is mostly from the matching process.

Although concurrent progress still shows the same performance drop from matching congestion where multiple threads try to acquire the matching lock, the message rate still flattens out around the same point as serial progress (Figure 5.3b). While in the last case with both concurrent progress and concurrent matching (Figure 5.3c), removing the ordering does not affect the performance because the matching process for this strategy is already optimal.

5.4.5 Current State of MPI Threading

In this section, I take the improved performance from my proposed strategies and compare with the different state-of-the-art MPI implementations on the same configuration of Multirate–pairwise. To get a better understanding on where the threaded performance is overall, I also compare with the process-based mode, where communications instead of happening between threads now happen between processes placed on the same nodes as the original threads.

Ideally, running on the same hardware with the same communication pattern should yield similar performance, regardless of whether processes or threads are used. Unfortunately, as demonstrated in Figure 5.4, at the current stage of threading support in MPI implementations, we are far from this ideal scenario.

The MPI implementations presented in this experiment are Intel MPI 2018.1 IMPI, MPICH 3.3 Gropp (2002) and Open MPI 4.0.0 Gabriel et al. (2004) with and without my modification. Each MPI implementation was compiled with GCC 8.3.0 with proper optimization flags (except for Intel MPI which is only available as a pre-compiled binary from the vendor).

Figure 5.4 highlights using a log-scale Y axis that from multi-thread standpoint, there is little difference between MPI implementations (dashed lines), they all perform similarly poorly. We can see a roughly 100% performance boost from the base implementation by employing try-lock semantics with multiple CRIs (dark red), but these results should be put in a larger context and compared with the process-to-process performance. The black dotted line represents the CRI injunction with concurrent progress and concurrent matching, the most optimistic scenario for communicating threads. While the design does give a significant boost in performance, up to 10x compared with the base implementation, it still cannot reach the same level of performance as the non-threading mode, potentially suggesting not yet understood bottlenecks for multi-thread MPI.

5.4.6 RMA Performance

To test the performance of my implementation with one-sided MPI, the RMA-MT benchmark is used for the measuring the performance. The experiments were run on the Trinitite system at LANL using both Intel Knights Landing (KNL) and Haswell compute nodes. Open MPI was configured to use the ugni Byte Transport Layer (BTL) and the rdma osc components. The ugni btl provides support for multiple CRIs for one-sided communication only. By default, the ugni btl will try to detect the number of cores available to the MPI process and will attempt to create one instance per available core. In the case of the RMA-MT benchmark, this creates 32 instances on Haswell nodes and 72 instances on KNL nodes.

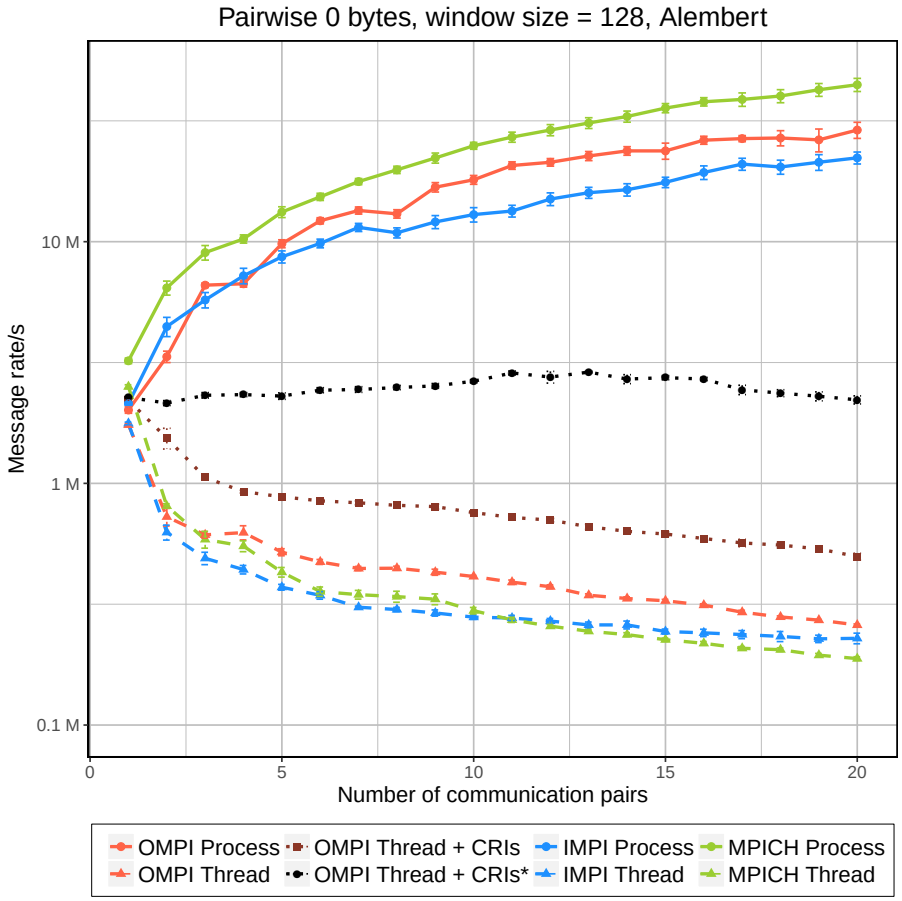


Figure 5.4: Zero byte message rate from different state-of-the-art MPI implementations.

All tests were configured to bind each benchmark thread to a dedicated CPU core (-x option), running from 1 to 32 threads on Haswell nodes, and to 64 threads on KNL nodes, using the MPI_Put operation (-o put) and MPI_Win_flush synchronization (-s flush) with both round-robin and dedicated assignment strategy. This benchmark spawns a user-specified number of threads that, for each message size, performs 1000 put operations. The first thread then calls MPI_Win_sync to synchronize the window. The results for both Haswell and KNL architectures appear in Figures 5.5a and 5.5b where the black horizontal line in each sub-figure represents the theoretical peak message rate for that particular message size.

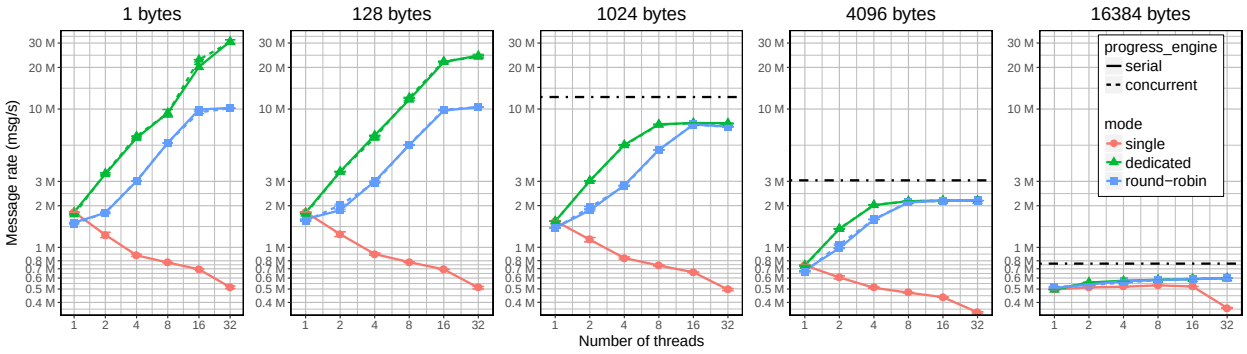
The results show that the performance when using dedicated instances for threads (triangles) significantly outperforms round-robin (square). The performance difference is similar on both KNL and Haswell nodes. When using a dedicated thread instance the performance of the RMA-MT benchmark scales almost perfectly with the number of threads. The single instance performance (red) represents the performance before support was added for multiple network instances where the performance drops with the increasing number of threads due to the lock contention on a single instance.

There appears to be little benefit from concurrent progress in this configuration (dashed lines). It is likely due to the absence of the matching process in one-sided communication.

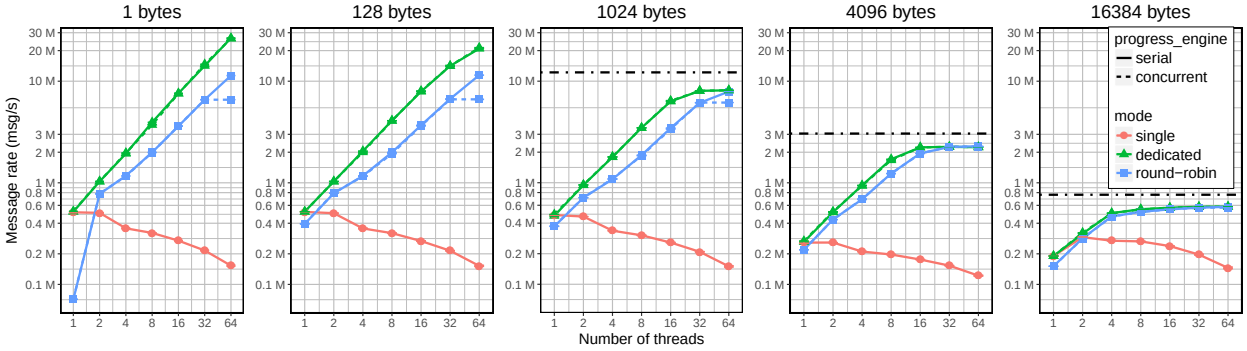
5.5 Optimization Suggestions

In general, the MPI implementations could benefit from allocating more resources to threads to allow them to operate simultaneously. There are several strategies to assign resources to threads. The experiments confirm that the ideal approach is to have a one-to-one mapping from thread to the resource (dedicated assignment), similar to a non-threading environment where each process has exclusive access to the network resources.

For two-sided communication, the likelihood of out-of-sequence messages increases with the number of threads, putting tremendous stress on the receiver side's matching process. Using an MPI info key to allow message overtaking from the application level might help



(a) RMA-MT performance on Haswell architecture



(b) RMA-MT performance on KNL architecture

Figure 5.5: Multi-threaded One-sided communication performance.

in boosting the performance. However, it might only be suitable for some categories of application that do not rely on message ordering, such as a task-based runtime.

The matching process remains one of the major bottlenecks for two-sided communication as it is a critical section that has to be protected. This study further demonstrates the potential of multi-threaded MPI if the matching process is parallelized. Although it is possible to argue that all the protection mechanisms can be optimized, it remains true that the matching, as imposed by the MPI standard, is inherently sequential, and remains a burden and a performance bottleneck. Dropping the matching requirements for messages will either move the programmability of the MPI two-sided communications toward one-sided communications, which come with their own set of constraints, or push in the direction of Active Message, a field that has received little interest from the MPI community.

One-sided communication reaps the greatest benefit from more allocated resources. Since there is no matching process for one-sided communication, the performance does not suffer from the funneling effect on the matching process serialization. The experiment shows good performance scaling with the number of threads. However, one-sided communication imposes the burden of synchronization and programming complexity on the users.

5.6 Conclusion

With the hope to make MPI a more suitable communication infrastructure for mixed programming paradigms (MPI+X), this chapter assessed the performance of two-sided communications on several MPI implementations in a multi-threaded scenario. Confronted with the abysmal performance gap between threads and processes based communications, I proposed several strategies to address this performance gap, and implemented and evaluated them in the Open MPI library, looking at their impact on both one and two-sided communications. In summary, my contributions from this chapter are the following.

- **Communication Resource Instance (CRI)**: Allowing MPI implementation to allocate multiple instances of resources and assign them to threads, enabling them to perform communication operation simultaneously. I proposed two assigning strategies: round-robin and dedicated and demonstrated that the designs are capable of

achieving reasonable performance scaling for the multi-threaded scenario. In two-sided communication, the CRI alone achieved 2× performance of the base implementation.

- **Concurrent Progress Engine**: I expanded on my thread synchronization object work from chapter 4 to allow multiple threads to execute the progress engine concurrently. However, with the concurrent progress engine, I encounter the massive lock contention from the matching process. From the experiment, I showed that with the parallel matching process in a multi-threaded scenario, I can achieve up to 10× performance of the base implementation.

- **Matching Process Impact**: I showed that the proposed CRI design, when used in multi-threaded one-sided communication, can achieve up to 200× the performance, when comparing to the original design on both Haswell and KNL systems. From the experiment results, I suggested that one-sided communication is more suitable to get the benefit from thread parallelism. However, without implicit synchronization, it might be more challenging for the user to switch from the two-sided communication model.

I showed that it is possible to obtain better message exchange rates, but the imposed requirements might weaken the MPI programming model and not be suitable for general purpose programming. I have also proposed a few potential additions to the MPI standard that would allow for better threading support, topics I plan to continue to investigate in the future.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The multi-threaded MPI paradigm is still not widely adopted despite being well defined in MPI standard version 2.1 from 2008. It has been a common understanding that the MPI implementation suffers from performance degradation when operating in MPI_THREAD_MULTIPLE mode, and the evaluations presented in this study reinforced this conclusion. Despite high interest in multi-threaded MPI, the performance issue leads to a hindrance in adopting this paradigm. This study contributes to the efforts in threading performance optimization in multiple steps, from reducing the cost of serialization to fully harnessing the power of thread parallelism. In summary, this study provides the following solutions to the problems in multi-threaded MPI optimization.

- **A Flexible Evaluation Tool:** Currently, there are several trusted multi-threaded MPI benchmarks available. Based on the surveys, most of the studies on multi-threaded MPI utilize these trusted benchmarks as their evaluation tools. However, the existing benchmarks do not capture multiple aspects of the multi-threaded communication. This study observed the shortcomings of the existing benchmarks and proposed a novel flexible benchmark. The Multirate benchmark provides multiple communication patterns such as one-to-one, one-to-many, many-to-one and many-to-many. Each communication pattern stresses the MPI implementation at different

points of execution, allowing the benchmark to expose the bottleneck of the MPI implementation effectively. The benchmark also provides a quick comparison between the process and the thread abstraction on the same hardware resources, to potentially expose the bottlenecks of multi-threaded MPI implementations. I demonstrate the potential of the Multirate benchmark by utilizing it to evaluate the current state-of-the-art MPI implementations in chapter 3 and further use Multirate as one of the evaluation tools for multi-threaded MPI optimization in this study.

- **Efficient Thread Synchronization:** This study proposed and implemented a design to provide the MPI implementations with more control over user-level threads, and allow them to redirect threads to execute useful tasks while waiting for their requests to complete instead of racing against one another in Chapter 4. This portable design can increase threads utilization for any MPI implementation. For demonstration, in this study, the design is utilized to reduce lock contention into the MPI progress engine, and shows up to 7× increased performance in shared memory, and up to 3× increased performance in inter-node communication when comparing to the legacy design. The thread synchronization object design also reduces the unnecessary context switching in thread over-subscription scenarios. Finally, I explore several potential usages of the synchronization object, discussed ongoing collaborations with other researchers, and presented several prototypes and preliminary results.

- **Resources Management for Threads:** I proposed another portable design of Communication Resources Instances (CRIs) to pack the necessary resources for communication into a single object in Chapter 5. The MPI implementation can allocate multiple instances to satisfy the demand from multiple threads. The CRI design allows MPI implementations to mitigate the resource contention that usually comes from the original single resource design. In this study, I demonstrate that increasing resources through CRI can boost multi-threaded communication performance. For two-sided communication, the CRI design improves the performance significantly (40-100% improvement from the original design).

- **Resource Assignment:** This study proposed a centralized design inside the MPI implementation to oversee the resource assignment to threads. I present two strategies, *round-robin* and *dedicated*. The evaluation shows that dedicated assignment yields better performance than the round-robin strategy. However, without the standardized interoperability between MPI and threading frameworks, the implementation has to rely on thread-local storage support from the compiler or the threading framework itself, which might not be always available on the system.

- **Concurrent Matching Process:** There are ongoing efforts in improving the efficiency of the MPI message matching process, but the process still remains a serial operation. In a multi-threaded scenario, the matching process becomes a major bottleneck. This study showed the significant impact of the matching process by simulating a parallel matching process by utilizing multiple MPI communicators in Open MPI. In Chapter 5, the evaluation shows that when the matching process is parallelized, multi-threaded MPI can benefit more from the thread parallelism. The experiment, with my proposed multi-threaded optimization, showed up to 10× improvement over the original design in two-sided communication. Furthermore, in one-sided communication where the matching process is not required, the results showed that my optimization can achieve up to 200× speedup from the original design. Practically, it is still a major challenge to design a parallel matching process for general cases. Nonetheless, this study presented the flaw in the two-sided communication design of MPI when operating in the multi-threaded environment.

This study emphasizes the importance of thread parallelism in MPI by demonstrating that multi-threaded MPI, with my improved design, can perform at up to 10× better than the legacy implementation in inter-node communication. My study strongly suggested that the matching process is one of the remaining roadblocks that still creates performance disparities between MPI in process mode and in thread mode.

The current remedy from the MPI standard is to relax the constraints of the matching process. The standard-provided communicator info key can be used to direct the MPI implementation to ignore the FIFO ordering for the messages—greatly reducing the

complication of the matching process—but the approach might not be suitable for every application. The user can choose to avoid the matching process entirely by switching into the one-sided communication API. However, the one-sided communication comes with its own set of challenges such as platform portability, the burden of synchronization for the user, which can increase the complexity of the application code base, and increasing the complication for large-scale applications.

The contributions from Chapter 3 and 5 have been submitted to ExaMPI 2019 and published at IEEE Cluster 2019 respectively. The optimization proposed in this study has been partially incorporated into Open MPI version 4.0.0, and fully incorporated into Open MPI master branch, expected to be released with Open MPI 5.0.0 in the near future.

6.2 Future Work

The interest in utilizing multi-threaded MPI is likely to increase, due to the rising number of CPU cores per node, and the availability of high-performance threading frameworks such as OpenMP and Argobots. The challenges are likely to be on the MPI developers to deliver the best threading communication performance and maintain the user's interests. The work in this study has established that in order to gain better communication performance for multi-threaded MPI, the MPI implementation has to adopt a design that allows more thread concurrency, and move away from the bulk-synchronization design. While this work implements the solutions for Open MPI, the ideas can be easily extended to other MPI implementations such as Intel MPI, MPICH, and MVAPICH.

This study showed that despite the fact that the MPI message matching design is performing well in non-threading environments, it is seriously flawed when operating in a multi-thread environment. The MPI threading support is defined as an extension to the original MPI standard, forcing the multi-threaded environment, with a different nature, to follow the same set of constraints which can limit its capability. I strongly believe that the efforts to define a better threading interface—such as the endpoint proposal Mpi-Forum (2016), or a standardized method for passing thread information between the MPI and

the threading frameworks—can significantly increase the optimization opportunity for both sides, allowing better overall performance for multi-threaded HPC applications.

Despite the significant performance improvements from the work of this study, the evaluation showed that the multi-threaded MPI is still far from being able to attain the same level of performance as its non-threading counterpart. This indicates more undiscovered bottlenecks in the multi-threaded MPI implementations and leaves room for more optimization in the future. Moving forward, I plan to study several matching designs and incorporate an atomic, lock-free data structure to mitigate the need for unnecessary bulk synchronization, reducing the cost of the matching process in the multi-thread environment. I also plan to continue the efforts in utilizing the thread synchronization object design such as the user-level extension, and to pursue ongoing collaborations to incorporate the design into HPC applications such as PaRSEC.

Bibliography

A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. Mpi+threads: Runtime contention and remedies. *SIGPLAN Not.*, 50(8):239–248, Jan. 2015. ISSN 0362-1340. doi: 10.1145/2858788.2688522. URL http://doi.acm.org/10.1145/2858788.2688522. 24, 61

A. Amer, H. Lu, Y. Wei, J. Hammond, S. Matsuoka, and P. Balaji. Locking aspects in multithreaded mpi implementations. *Argonne National Lab., Tech. Rep. P6005-0516*, 2016. 61

J. Ang, B. Barrett, K. Wheeler, and R. Murphy. Introducing the graph 500. 01 2010. 71

C. Augonnet et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Conc. Comp. Pract. Exper.*, 23:187–198, 2011. 33

P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward efficient support for multithreaded mpi communication. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87474-4. doi: 10.1007/978-3-540-87475-1_20. URL http://dx.doi.org/10.1007/978-3-540-87475-1_20. 24

B. W. Barrett, G. M. Shipman, and A. Lumsdaine. Analysis of implementation options for mpi-2 one-sided. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 242–250. Springer, 2007. 28

M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda. Adaptive and dynamic design for mpi tag matching. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2016. 27

D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. Ecp milestone report a survey of mpi usage in the us exascale computing project wbs 2.3. 1.11 open mpi for exascale (ompi-x)(formerly wbs 1.3. 1.13), milestone stpm13-1/st-pr-13-1000. 1

G. Bosilca et al. PaRSEC: a programming paradigm exploiting heterogeneity for enhancing scalability. *Comp in Sc. and Eng.*, 99, 2013. 33, 71, 72

P. Boyle, A. Yamaguchi, G. Cossu, and A. Portelli. Grid: A next generation data parallel C++ QCD library. 2015. 35

R. Brightwell, S. Goudy, and K. Underwood. A preliminary analysis of the mpi queue characterisitics of several applications. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 175–183. IEEE, 2005. 26

L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 22, 59

R. V. der Wijngaart. Parallel research kernels, a tool for parallel systems investigations - part i. https://www.nas.nasa.gov/assets/pdf/ams/2016/AMS_20161013_VanDerWijngaart.pdf, 2016. [Online; accessed 1-March-2019]. 35

J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible mpi endpoints. *The International Journal of High Performance Computing Applications*, 28(4):390–405, 2014. 23, 59

J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An implementation and evaluation of the mpi 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 28(17):4385–4404, 2016. 28

M. G. F. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges. Rma-mt: A benchmark suite for assessing mpi multi-threaded rma performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 550–559, May 2016. doi: 10.1109/CCGrid.2016.84. 28, 78

G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded mpi communication on multicore petascale systems. In R. Keller, E. Gabriel, M. Resch, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 11–20, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15646-5. 24

D. Eberius, T. Patinyasakdikul, and G. Bosilca. Using software-based performance counters to expose low-level open mpi performance information. In *Proceedings of the 24th European*

MPI Users' Group Meeting, EuroMPI '17, pages 7:1–7:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4849-2. doi: 10.1145/3127024.3127039. URL http://doi.acm.org/10.1145/3127024.3127039. 77, 88

T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schau ser. Active messages: a mechanism for integrated communication and compu tation. In *Computer Architecture, 1992. Proceedings., The 19th Annual Inte rnational Symposium on*, pages 256–266. IEEE, 1992. 3

K. B. Ferreira, S. Levy, K. Pedretti, and R. E. Grant. Characterizing mpi matching via trace-based simulation. *Parallel Computing*, 77:57–83, 2018. 27

M. Flajslik, J. Dinan, and K. D. Underwood. Mitigating mpi message matching misery. In *International conference on high performance computing*, pages 281–299. Springer, 2016. 27, 81

M. P. I. Forum. *MPI: A Message-Passing Interface Standard Version 3.1*, June 2015. http://mpi-forum.org/. 1, 8, 13

E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 14, 40, 83, 95

D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. d. Supinski, and R. Thakur. Minimizing mpi resource contention in multithreaded multicore environments. In *2010 IEEE International Conference on Cluster Computing*, pages 1–8, Sep. 2010. doi: 10.1109/CLUSTER.2010.11. 24

R. Grant, A. Skjellum, and P. V Bangalore. Lightweight threading with mpi using persistent communications semantics. In *Workshop on Exascale MPI 2015 held in conjunction with Supercomputing (SC15)*, 11 2015. 25

W. Gropp. Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44296-0. URL http://dl.acm.org/citation.cfm?id=648139.749473. 39, 95

W. D. Gropp and R. Thakur. Revealing the performance of mpi rma implementations. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 272–280. Springer, 2007. 28

P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. A brief introduction to the openfabrics interfaces-a new network api for maximizing high performance application efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 34–39. IEEE, 2015. 78

A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 989–1000, May 2011. doi: 10.1109/IPDPS.2011.95. 3

R. Gupta and T. Abels. Mpi-aware networking infrastructure, Dec. 14 2006. US Patent App. 11/147,783. 27

K. S. Hemmert, K. D. Underwood, and A. Rodrigues. An architecture to perform nic based mpi matching. In *2007 IEEE International Conference on Cluster Computing*, pages 211–221. IEEE, 2007. 27

N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold. Improving mpi multi-threaded rma communication performance. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 58:1–58:11, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6510-9. doi: 10.1145/3225058.3225114. URL http://doi.acm.org/10.1145/3225058.3225114. 28

T. Hoefler and A. Lumsdaine. Message progression in parallel computing-to thread or not to thread? In *2008 IEEE International Conference on Cluster Computing*, pages 213–222. IEEE, 2008. 26

IMPI. Intel mpi library software. https://software.intel.com/en-us/mpi-library. [Online; accesed 21-March-2019]. 40, 95

S. Kumar and M. Blocksome. Scalable mpi-3.0 rma on the blue gene/q supercomputer. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 7. ACM, 2014. 28

W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. Comb: A portable benchmark suite for assessing mpi overlap. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 472–475. IEEE, 2002. 35

B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. ISBN 0-13-680729-1. 22, 59

H. Lu, S. Seo, and P. Balaji. Mpi+ ult: Overlapping communication and computation with user-level threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 444–454. IEEE, 2015. 26

X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, and J. Dongarra. Adapt: An event-based adaptive collective communication framework. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 118–130, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5785-2. doi: 10.1145/3208040.3208054. URL http://doi.acm.org/10.1145/3208040.3208054. 30

Mpi-Forum. Endpoints issue #56 mpi-forum/mpi-issues, 2016. URL https://github.com/mpi-forum/mpi-issues/issues/56. 59, 104

J. Nakashima and K. Taura. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, pages 222–238. Springer, 2014. 22

OSU. Osu microbenchmark suite. http://mvapich.cse.ohio-state.edu/benchmarks. [Online; accessed 14-Febuary-2019]. 36, 66

T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm. Give mpi threading a fair chance: A study of multithreaded mpi designs. In *2019 IEEE International Conference on Cluster Computing*. IEEE, 2019. 6

T. Patinyasakdikul, X. Lou, D. Eberius, and G. Bosilca. Multirate: A flexible mpi benchmark for fast assessment of multithreaded communication performance. In *Submitted to Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '19, 2019. 5, 77, 89

A. Pellegrini, R. Vitali, and F. Quaglia. The rome optimistic simulator: Core internals and programming model. 01 2011. doi: 10.4108/icst.simutools.2011.245551. 33, 71

C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23 (4):298–298, 2008. 22

S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda. Quantifying performance benefits of overlap using mpi-2 in a seismic modeling application. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 17–25. ACM, 2010. 26

Proxy. Exascale proxy applications. https://proxyapps.exascaleproject.org/. [Online; accessed 1-March-2019]. 36

RMAMT. Rma-mt benchmarks. URL https://github.com/hpc/rma-mt. 89

S. Rumley, M. Bahadori, R. Polster, S. D. Hammond, D. M. Calhoun, K. Wen, A. Rodrigues, and K. B. man. Optical interconnects for extreme scale computing systems. *Parallel Computing*, 64(Supplement C):65 – 80, 2017. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2017.02.001. URL http://www.sciencedirect.com/science/article/pii/S0167819117300170. High-End Computing for Next-Generation Scientific Discovery. 2

W. Schonbein, M. G. F. Dosanjh, R. E. Grant, and P. G. Bridges. Measuring multithreaded message matching misery. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 480–491, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96983-1. 27, 81, 87

S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, et al. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2017. 22

P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015. 40, 78

H. Shan, S. Williams, W. de Jong, and L. Oliker. Thread-level parallelization and optimization of nwchem for the intel mic architecture. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 58–67. ACM, 2015. 35

M. Si, A. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. Mt-mpi: multithreaded mpi for many-core environments. In *Proceedings of the International Conference on Supercomputing*, 06 2014. ISBN 978-1-4503-2642-1. doi: 10.1145/2597652.2597658. 23

Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A network protocol independent performance evaluator. In *IASTED international conference on intelligent information management and systems*, volume 6, page 49. (Washington, DC, USA),, 1996. 35

J. Squyres. Modular component architecture. https://www.open-mpi.org/papers/workshop-2006/mon_06_mca_part_1.pdf. [Online; accesed 21-March-2019]. 52, 83

S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints. In *SC '14: Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis, pages 487–498, Nov 2014. doi: 10.1109/SC.2014.45. 23

R. Thakur and W. Gropp. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 46–55. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-75416-9_13. URL https://doi.org/10.10072F978-3-540-75416-9_13. 36

W. S. Thornton, N. Vence, and R. Harrison. Introducing the madness numerical framework for petascale computing. 33

V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Traff. Investigating high performance rma interfaces for the mpi-3 standard. In *2009 International Conference on Parallel Processing*, pages 293–300. IEEE, 2009. 28

K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A hardware acceleration unit for mpi queue processing. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2005. 27

K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó. Improving concurrency and asynchrony in multithreaded mpi applications using software offloading. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2015. 25

M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, et al. Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010. 35

S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell, et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High*

Performance Computing, Networking, Storage, and Analysis, page 52. IEEE Press, 2018. 15

K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008. 22

M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous mpi for the masses. 02 2013. 26

Appendices

A Multirate Benchmark User Guide

In this section, I present the user guide of the Multirate benchmark and provide the basic information and specification of the benchmark. The benchmark is publicly available through Github.com.

Installation

The Multirate benchmark is available as a git repository on Github.com. [1] To install, simply clone the directory and compile with the provided makefile. Multirate is a standalone application which does not require any additional library, other than the MPI library to compile.

Measuring MPI Performance

The benchmark offers 3 modes of operations; (1) process mode – a process to process communication, (2) thread mode – a thread to thread communication, and (3) hybrid mode – a combination of process and thread. To select the mode of operation, the user simply provide 4 variables through the command-line arguments:

- Number of sender process
- Number of sender thread
- Number of receiver process
- Number of receiver thread

The provided number will be used to create the proper communication entity for communication. The benchmark provides two major communication pattern; pairwise and all-to-all. The user can select the communication pattern through the command line arguments. If not, the default communication pattern is pairwise. For example, if the user desired to measure the MPI performance in thread mode, with 4 thread pairs, in pairwise

[1] https://github.com/ICLDisco/multirate.

communication pattern. The number of send process and receive process should be 1, and the number of send thread and receive thread should be 4.

The benchmark is expecting to be mapped by core. In example, if each node has 4 CPU cores, rank 0-3 should be mapped to the first node, rank 4-7 should be on the second. The evaluation will be invalid if the mapping is incorrect. In case of thread mode, the MPI should spawn each rank on each node and bind all the threads to all available cores. Failure to bind the thread correctly might result in incorrect evaluation. Listing 1 presented all available command-line options. Listing 2 demonstrates the example of a run in thread mode (20 threads) with pairwise communication pattern, using window size of 128 and run for 1000 iterations. The result presented the number of sender and receiver entities in process and threads, window size, bandwidth, latency and message rate.

```
Communication Pattern (pick one):
-p : Operate in Pairwise mode. (default)
-a : Operate in Alltoall mode.

Alltoall mode options:
-n (k) : number of sender processes
-m (k) : number of receiver processes
-x (k) : number of sender threads
-y (k) : number of receiver processes

Workload Adjustment:
-t : num_thread_pair (pairwise only)
-s : message size
-w : window size.
-i : number of iteration

Additional test:
-c : use separated communicator for each pair.
-o : ignore MPI message ordering (allow_overtaking)
```

Listing 1: Available Command-line Options for Multirate Benchmark

```
$> (MPIRUN) ./multirate -s 0 -w 128 -i 1000 -t 20 -p
$> 1   20   1   20   0   128   0.00 Gbps   7.13 usec    140199.26 msg/s
```

Listing 2: Example of command-line arguments.

B Thread Synchronization Object MPI Extension

The proposed MPI extension for the user-level to access the thread synchronization object is presented in this section. The extension allows the user to manipulate the synchronization object directly, giving them more control over the object and enabling them to utilize it to their benefits.

User-level API

```
typedef struct ompi_mpix_sync_s *MPIX_Sync;

OMPI_DECLSPEC int MPIX_Sync_init(MPIX_Sync *sync);
OMPI_DECLSPEC void MPIX_Sync_free(MPIX_Sync *sync);

OMPI_DECLSPEC int MPIX_Sync_attach(MPIX_Sync sync,
                                   MPI_Request *request,
                                   void *completion_data);

OMPI_DECLSPEC int MPIX_Sync_waitall(MPIX_Sync sync);
OMPI_DECLSPEC int MPIX_Sync_size(MPIX_Sync sync);
OMPI_DECLSPEC int MPIX_Sync_probe(MPIX_Sync sync);

OMPI_DECLSPEC void* MPIX_Sync_query(MPIX_Sync sync,
                                    MPI_Status *status);

OMPI_DECLSPEC int MPIX_Sync_query_bulk(int incount,
                                       MPIX_Sync sync,
                                       int *outcount,
                                       void **cbdata,
                                       MPI_Status *status);

OMPI_DECLSPEC void MPIX_Progress(void);
OMPI_DECLSPEC extern void *MPIX_SYNC_EMPTY;
OMPI_DECLSPEC extern void *MPIX_SYNC_NO_COMPLETION_DATA;
```

Listing 3: Proposed MPI extension.

Usage

The synchronization object has to be initialized with MPIX_Sync_init before using and free through MPIX_Sync_free before MPI_Finalized is called. Once initialized, the synchronization object can be reused forever until freed. There are multiple ways to interact with the synchronization object.

- **MPIX_Sync_attach**: The attach API takes 3 arguments; (1) the initialized synchronization object, (2) an MPI request, and (3) the completion data from the user or MPIX_SYNC_NO_COMPLETION_DATA if the user does not want the API to return anything. Once the user attached the MPI request to the synchronization object, the user should only check for the completion of the operation of the request through the query API. Once the request becomes completed, it is detached automatically from the synchronization object. The request can still be cancelled through MPI_Request_cancel. If the request is cancelled before its completion, no completion event will be generated and the request is detached from the associating synchronization object.

- **MPIX_Sync_query**: Query the synchronization object for single completion event. Once an attached request becomes completed, a completion event is generated on the associating synchronization object. The user can query for completion event with this API. If there is completion events, the completion data from the first completed event is returned to the user. In case of no completion, the API returns MPIX_SYNC_EMPTY instead. The MPI status is also provided through the argument. The functionality is similar to MPI_Testany.

- **MPIX_Sync_query_bulk**: Query the synchronization object for multiple completion events. Constantly querying single completion event at a time can become expensive. This API provides the similar functionality to MPI_Testsome where the users can provide a number of completion they need and the API will return at most that number of completion events.

- **MPIX_Sync_waitall**: Similar to MPI_Waitall, this API is a blocking call, wait for every attached request to be completed. The completed operations still generate completion events unless the user provide MPIX_SYNC_NO_COMPLETION_DATA as their completion data.

- **MPIX_Sync_size**: The API returns the number of incomplete requests attached to the synchronization object. The API is useful in the case of quick checking the overall status of the operations. The operation cost is O(1) as it reads the reference count on the synchronization object.

- **MPIX_Sync_probe**: The API returns the current number of completion events associated with the synchronization object. The API is useful in the case of performing a quick check on the overall status of the operations. The operation cost is O(1) as it reads the length of the completion queue associating with the synchronization object.

Example

I present 2 hello world examples for the MPI extension. Listing 4 illustrates MPIX_Sync_query usage between 2 MPI process, the query has the similar usage as MPI_Testany while Listing 5 shows the usage of MPIX_Sync_query_bulk extension in the similar manner to the standard MPI_Testsome, with the printout of the user-provided completion data.

```c
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"
#include "mpi-ext.h"

int main(int argc, char **argv){

    int me;
    char buf[10000];

    MPI_Request request;
    MPI_Status status;

    MPIX_Sync sync;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    MPIX_Sync_init(&sync);

    if (me == 1) {
        MPI_Isend(buf, 10000, MPI_BYTE, 0, 42, MPI_COMM_WORLD, &request);
    } else {
        MPI_Irecv(buf, 10000, MPI_BYTE, 1, 42, MPI_COMM_WORLD, &request);
    }

    MPIX_Sync_attach(sync, &request, (void*)1);
    void *ret = MPIX_SYNC_EMPTY;
    while(ret == MPIX_SYNC_EMPTY) {
        ret = MPIX_Sync_query(sync, &status);

    }

    MPIX_Sync_free(&sync);
    MPI_Finalize();
    return 0;
}
```

Listing 4: MPIX_Sync_query example

```c
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"
#include "mpi-ext.h"

int main(int argc, char **argv){

    int me;
    char buf[10000];

    MPI_Request requests[1000];
    MPI_Status status[1000];

    MPIX_Sync sync;

    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    MPIX_Sync_init(&sync);

    if (me == 1) {
        for(int i=0;i<1000;i++) {
            MPI_Isend(buf, 10000, MPI_BYTE, 0, 42, MPI_COMM_WORLD, &requests[i
                ]);
            MPIX_Sync_attach(sync, &requests[i], (void*)i);
        }
    } else {
        for(int i=0;i<1000;i++) {
            MPI_Irecv(buf, 10000, MPI_BYTE, 1, 42, MPI_COMM_WORLD, &requests[i
                ]);
            MPIX_Sync_attach(sync, &requests[i], (void*)i);
        }
    }

    int n=100, ncomplete, c=0;
    void *cbdata[100];
    while(c != 1000) {
        MPIX_Sync_query_bulk(n, sync, &ncomplete, cbdata, status);
        if (ncomplete!=0) {
            for(int i=0;i<ncomplete;i++)
                printf("%p ",cbdata[i]);
            printf("\n");
        }
        c+=ncomplete;
    }

    MPIX_Sync_free(&sync);
    MPI_Finalize();
    return 0;
}
```

Listing 5: "MPIX_Sync_query_bulk example"

C MPI One-sided Window Operations

```
1  int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
2              MPI_Comm comm, void *baseptr, MPI_Win *win)
3
4  int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
5              MPI_Comm comm, void *baseptr, MPI_Win *win)
6
7  int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
8
9  int MPI_Win_complete(MPI_Win win)
10 int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
11             MPI_Comm comm, MPI_Win *win)
12
13 int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
14 int MPI_Win_detach(MPI_Win win, const void *base)
15 int MPI_Win_fence(int assert, MPI_Win win)
16
17 int MPI_Win_flush(int rank, MPI_Win win)
18 int MPI_Win_flush_all(MPI_Win win)
19 int MPI_Win_flush_local(int rank, MPI_Win win)
20 int MPI_Win_flush_local_all(MPI_Win win)
21
22 int MPI_Win_free(MPI_Win *win)
23
24 int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
25 int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
26
27 int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
28 int MPI_Win_lock_all(int assert, MPI_Win win)
29
30 int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
31
32 int MPI_Win_set_info(MPI_Win win, MPI_Info info)
33
34 int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
```

```
35 int *disp_unit, void *baseptr)
36
37 int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
38 int MPI_Win_sync(MPI_Win win)
39 int MPI_Win_test(MPI_Win win, int *flag)
40
41 int MPI_Win_unlock(int rank, MPI_Win win)
42 int MPI_Win_unlock_all(MPI_Win win)
43
44 int MPI_Win_wait(MPI_Win win)
```

Listing 6: Window operations from the MPI standard 3.1

Vita

Thananon 'Arm' Patinyasakdikul was born in Khonkean, Thailand, on September 12, 1990. After completing his graduation at Suranaree University of Technology in the province of Nakhonratchasima, Thailand, with the bachelor degree of Electrical Engineering in 2012. He enrolled in the Ph.D. program in Computer Science at the University of Tennessee, Knoxville in 2013. He also served the role of graduate research assistant at the Innovative Computing Laboratory (ICL) under the supervision of Dr. Jack Dongarra and Dr. George Bosilca. His research interests are focused on high-performance computing, with the concentration on multi-threaded MPI communication. While pursuing his doctoral degree, Thananon completed several internships with prominents companies in the HPC industries such as Cisco Systems in the summer of 2016 and 2017, and Intel in 2018. He served as the lead student volunteer in ACM/IEEE Supercomputing in 2016 and 2017. Thananon is expected to receive his Doctor of Philosophy degree in August, 2019. After graduation he will be pursuing his career at Cray Inc. as an MPI engineer.