DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design of a Throughput Oriented Network Transport Layer Based on MPI for the Data Acquisition System of the CMS Detector at CERN

Michael Lettrich

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design of a Throughput Oriented Network Transport Layer Based on MPI for the Data Acquisition System of the CMS Detector at CERN

Entwicklung einer durchsatzorientierten Netzwerktransportschicht auf Basis von MPI für das Datenerfassungssystem des CMS Detektors des CERN

Author:	Michael Lettrich
Supervisor:	Prof. Dr. Michael Bader
Advisor:	Steffen Seckler (TUM), Luciano Orsini (CERN)
Submission Date:	15.07.2018

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.07.2018 Michael Lettrich

Acknowledgments

This thesis represents part of the results from a 14 month work placement as a *Technical Student* at the TriDAS Group of the CMS experiment at the European Organization for Nuclear Research (CERN). I am grateful to Frans Meijers (CERN) and Prof. Michael Bader (TUM) who officially supported my efforts to turn parts of the placement into a thesis topic. I would also like to express my deepest gratitude towards my advisors Steffen Seckler and Luciano Orsini (CERN). With their experience and level of expertise they set me up on the right path and helped me to overcome difficult obstacles along the way.

Furthermore I would like to thank my colleagues Emilio Meschi, Remigius Mommsen and André Holzner for their explanations on the current CMS Systems. Without a team of committed system administrators and DAQ on-calls who supported me in creating and running an environment where I can benchmark my applications, none of the graphs and studies would have been possible. Further I would like to thank Andrea Petrucci who helped me with many organizational issues and helped me to refocus the project after unsuccessful attempts. Finally I would like to thank friends, flatmates and family for their tremendous support as well as all of those diverse encounters with people from so many different countries I made through CERN that helped me to grow professionally and as a person.

Abstract

The Large Hadron Collider (LHC) at CERN allows observation of unstable sub-atomic particles by colliding two opposed particle beams with up to 14TeV in predefined interaction points. As one of four LHC experiments, the Compact Muon Solenoid (CMS) experiment registers them in its set of different sub-detectors. With collisions producing 1MB of data at a rate of 40MHz, the data rate must be reduced by the data acquisition system (DAQ) before sent to mass storage. After reduction by hardware filtering to 100kHz, a second, high level software filter reduces the rate even more. Between these steps, an event-building cluster merges data from the same collision at 100kHz using an Infiniband FDR RDMA interconnect programmed at driver level. This thesis investigates how the Message Passing Interface (MPI), mostly used in high performance computing, can be leveraged to implement an efficient, scalable and portable network transport that can handle the high bandwidths required for the CMS Event-Builder.

Contents

1 Introduction

The question how the world around us came to be and what it is made of, has probably been one of the oldest questions of mankind. Throughout the centuries, scholars and scientists have improved our knowledge, giving us more precise models and theories on how our universe and the matter it consists of came to be. With the rise of particle physics, complex collider experiments have helped us to build models of matter on the scale of subatomic particles. The largest and most powerful of them, the Large Hadron Collider at CERN is the latest effort of particle physicists to observe sub-atomic particles in a controlled environment and trying to find answers to open questions in our current models.

Capturing particle collisions at a nanosecond rate, the detectors are forced to reject interactions that are not relevant for current studies using multi-stage distributed filtering systems based on electronics and software. At the accumulated data rates of an order of a hundred Gigabytes per second between electronics and software filter, efficient use of network resources is required.

The cluster system connected by a high-performance RDMA network used for gathering data of the same particle collision from various subsystems is interacting with network equipment on driver level, following design paradigms of real-time systems and distributed applications. These kind of clusters are also the backbone of High-Performance Computing (HPC) with the difference that in HPC problems are usually driven by compute intensive tasks instead of network throughput.

Over the years, the HPC community put significant efforts into development of tools, programming frameworks and libraries that simplify the creation, tuning and debugging applications. One of them is the Message Passing Interface (MPI) library, that simplifies the creation of distributed applications and enables processes to communicate with each other through high-performance networks. Following the logic, that the hardware used by data acquisition systems for high-energy physics experiment are the same as HPC clusters, we can now ask the question if data acquisition can also be reinterpreted as a HPC task and benefit from HPC technology such as MPI.

As there are no known published investigations of MPI in DAQ systems, we will study the Event Builder at the CMS detector, a distributed application driven by network throughput. Our goal is to understand if MPI is suitable for solving these kind of tasks and evaluate if MPI is capable of exploiting network hardware in an adequate way.

1.1 Outline

The thesis is structured as follows. The first part will provide a brief introduction to CERN and its current flagship project, the Large Hadron Collider. It is followed by the description on why and how observed collisions of accelerated particles are filtered in the CMS experiment by a two stage triggering and data acquisition system. We will then describe the process of event building in broader depth, detailing both the Event Builder Protocol and the software system used to run the large distributed system.

We will then change our focus to the key concepts of RDMA network technologies used to handle the high-bandwidths of the CMS Event Builder and how the Message Passing Interface (MPI) uses those technologies efficiently.

Having provided the necessary background, we will formulate the problem statement of designing a messaging service on top of MPI for the CMS Event Builder and name the requirements a solution should obey to. We then will present a system design based on a layered architecture and the Pipes and Filters architectural patterns that proposes a solution to the problem aligned with the requirements. We will then detail the lgorithmic and implementation details of that solution before evaluating different aspects of out messaging service through a series of benchmarks.

2 Work Environment

This chapter provides the necessary context about the technical aspects of the environment this thesis was created in. It particularly gives an overview of the data acquisition process at the CMS detector on the LHC ring and explains the necessity for high-throughput networking in the CMS Event Builder.

2.1 The Large Hadron Collider at CERN

The *Conseil Européen pour la Recherche Nucléaire* (CERN) or *European Council for Nuclear Research* was formed in 1952 with the mandate to establish a European physics laboratory [1]. It resulted in the European Organization for Nuclear Research, that was founded in 1954 as an "organization, [that] shall provide for collaboration among European States in nuclear research of a pure scientific and fundamental character, and in research essentially related thereto" [2] by 12 European member states. It was decided to locate its facilities in Meyrin on the Franco-Swiss boarder close to Geneva, Switzerland.

Over the past decades, CERN attracted new member states and formed collaborations with research institutes around the world. New experimental facilities in the form of particle accelerators and detectors were built in order to investigate the fundamental constituents of matter and the forces acting between them [3] [4].

According to Einstein, mass and energy are equivalent and can be converted into each other. This is exploited by particle accelerators, where the impulse of particles colliding is converted into energy and back into mass, giving a glimpse of unstable sub-atomic particles. Higher beam energies and intensities increase the chances to observe heavier sub-atomic particles [5].

Today, CERN is home to the *Large Hadron Collider* (LHC) [6], the world's largest and most powerful particle accelerator. Passing through smaller accelerators, two beams of protons or lead ions traveling in opposite directions are injected into the 27km circuit of the LHC where the beams reach full energy of around 6.5TeV. The beams, chopped up in 2808 *bunches*, containing about 1.15×10^{11} particles each are then brought into collision in predefined locations at center of mass collision energies up to 13TeV.

The four LHC experiments ATLAS, CMS, ALICE and LHCb are located around these *interaction points*, where the beams cross and particles collide, registering the

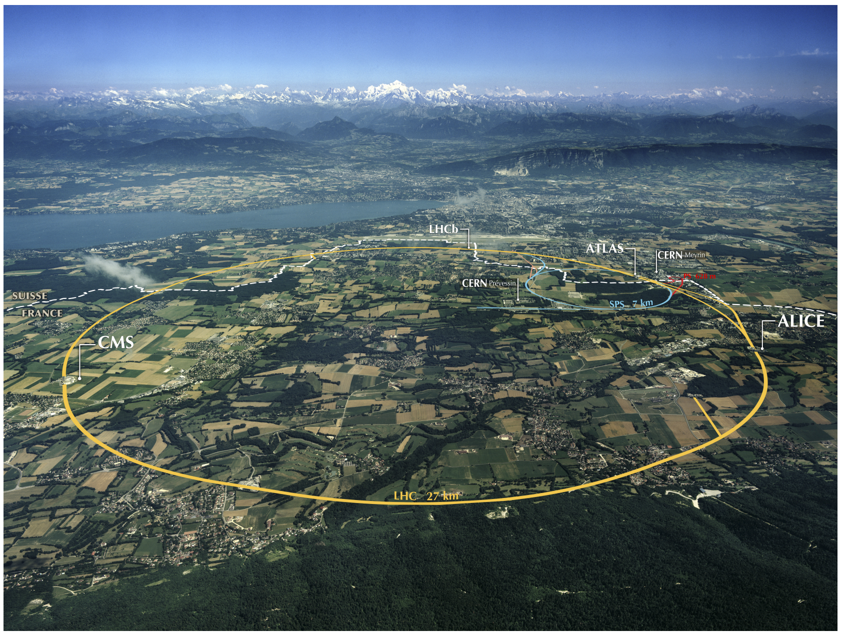


Figure 2.1: Aerial view of the underground CERN accelerator complex and the four experiments located on the LHC ring. (Photo: CERN)

collision and the generated sub-atomic particles through direct or indirect measurement methods by specialized detectors.

2.2 The CMS Experiment and its Data Acquisition System

The *Compact Muon Solenoid* (CMS) [7] detector is one of the four detectors on the LHC ring, installed around 100 meters underground on French territory between Lake Geneva and the Jura mountains. It is a general-purpose detector, consisting of different sub-detectors designed to measure all relevant parameters required to study phenomena at the high energy domains the LHC opens.

The technological challenges that had to be overcome to build and reliably operate this detector are manifold. After the particle bunches have reached full energy, they cross at a rate of 40MHz (or 25ns time intervals), leading to an order of 10^9 collision events a second that are registered in the detector. At a total data rate in the order of 1MB per event, it is impossible to store all events. However, protons are composite objects that consist of quarks and gluons transmitting the forces that holds them together. A hard collision of quarks or gluons is statistically rare with other, soft collisions occurring much more frequently. As these soft events are of limited scientific

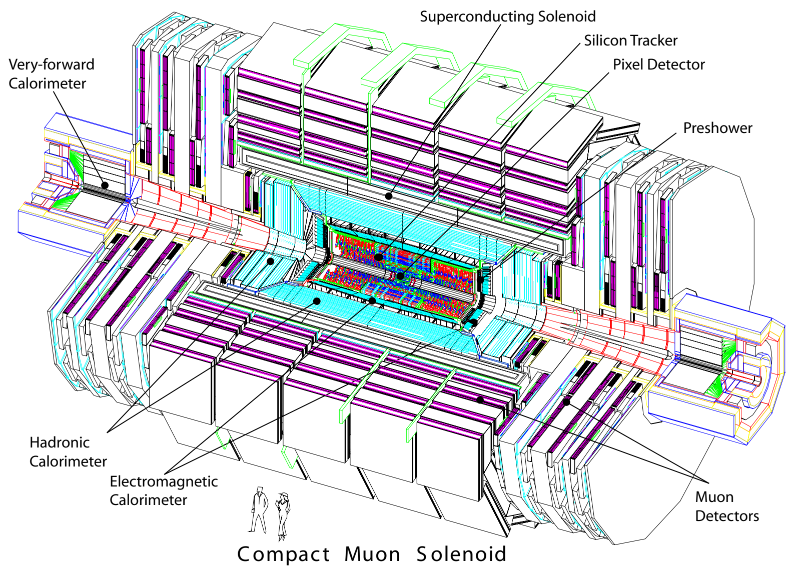


Figure 2.2: A schematic of the CMS detector showing the subsystems that allow the detection of different types of particles (Graphic: CERN, for the benefit of the CMS Collaboration)

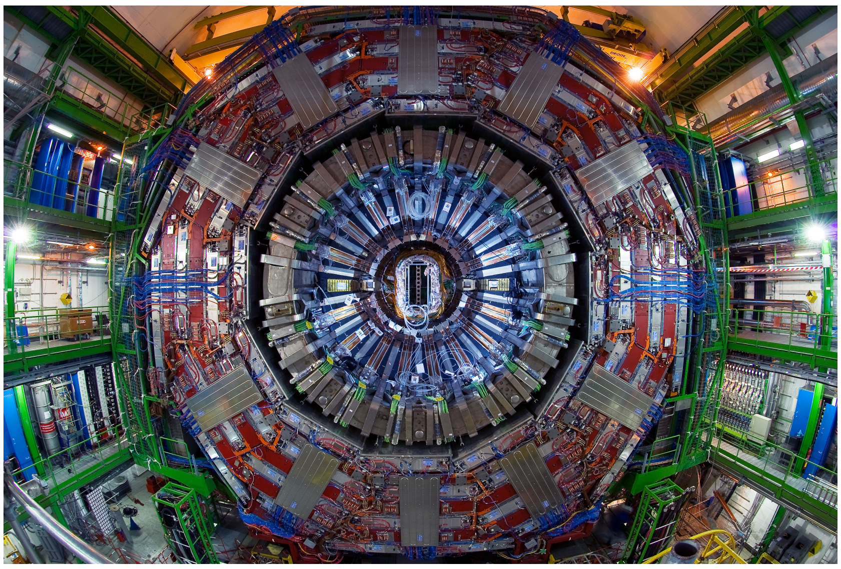


Figure 2.3: Front view of the CMS detector during its assembly in its experimental cavern in Cessy, France. (Photo: CERN, for the benefit of the CMS Collaboration)

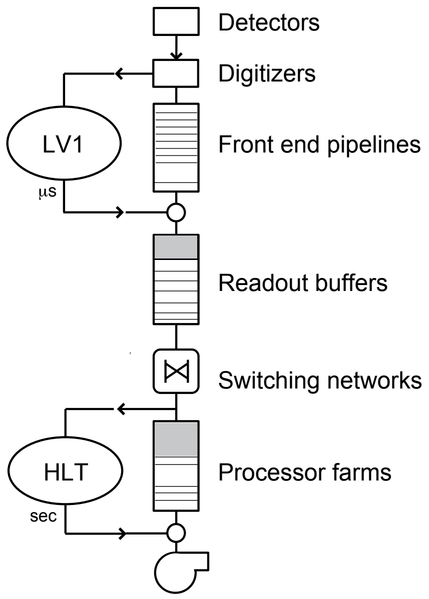


Figure 2.4: Dataflow in the CMS Trigger/DAQ system. Detector data is filtered in a two stage process by a hardware Level-1-Trigger and a software High Level Trigger. Graphic: [9]

interests, a trigger system has to distinguish between events that are worth storing and those that are not.

2.2.1 The CMS TriDAS System

The CMS *Trigger and Data Acquisition System* (TriDAS) [8], is designed to perform this distinction and reduce the data rate to an order of 100MB/s for archiving a later analysis. In order to achieve this rejection rate, the system is divided into a *Level-1 Trigger* and a *High-Level-Trigger* (HLT). The Level-1 Trigger is a custom-built hardware-based system, which reduces the event rate to 100kHz. The HLT on the other hand is a software system operating on a farm built from commodity of the shelf hardware and performs the last rejection step to an order of 1kHz.

Operating on a file basis, the HLT takes its rejection decisions based on the full information of the event. Therefore, all read-out information from the subsystems that make up the CMS detector need to be merged into a single file. This is the main purpose of the *Event-Builder* [9], a distributed system, that is the link between the hardware-based readout buffers of the detector and the software-based HLT.

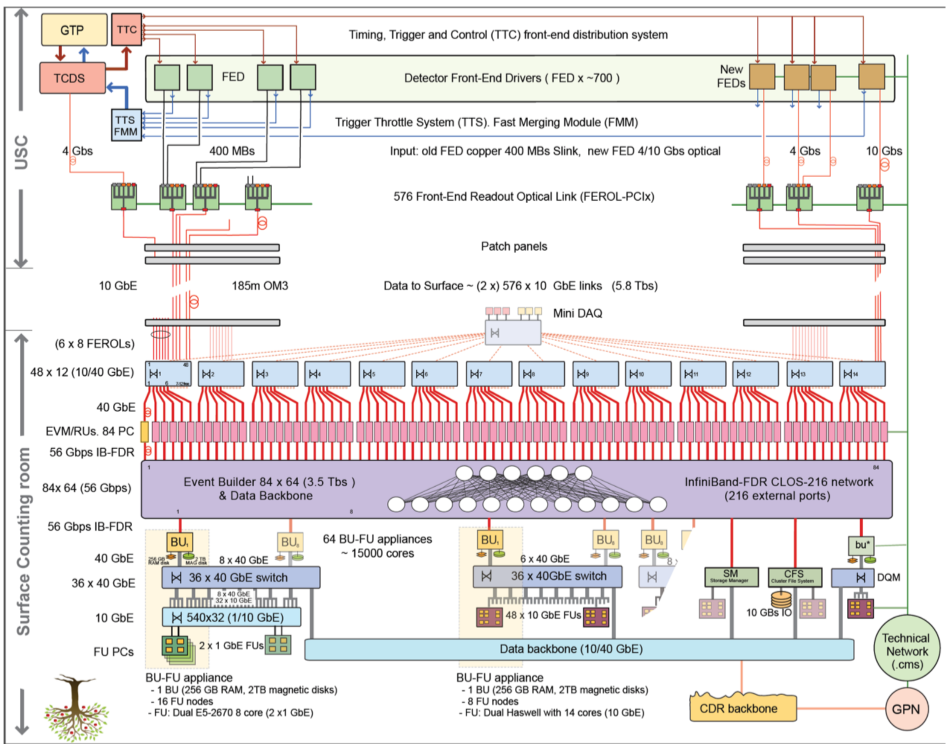

GTP
TTC
TCDS
TTS FMM
Timing, Trigger and Control (TTC) front-end distribution system
FED
Detector Front-End Drivers (FED x ~700)
New FEDs
Trigger Throttle System (TTS). Fast Merging Module (FMM)
USC
4 Gbs
400 MBs
Input: old FED copper 400 MBs Slink, new FED 4/10 Gbs optical
4 Gbs
10 Gbs
576 Front-End Readout Optical Link (FEROL-PCIx)
Patch panels
10 GbE
185m OM3
Data to Surface ~ (2 x) 576 x 10 GbE links (5.8 Tbs)
Mini DAQ
(6 x 8 FEROLs)
48 x 12 (10/40 GbE)
40 GbE
EVM/RUs. 84 PC
56 Gbps IB-FDR
84x 64 (56 Gbps)
Event Builder 84 x 64 (3.5 Tbs) & Data Backbone
InfiniBand-FDR CLOS-216 network (216 external ports)
Surface Counting room
56 Gbps IB-FDR
64 BU-FU appliances ~ 15000 cores
40 GbE
8 x 40 GbE
6 x 40 GbE
36 x 40 GbE
36 x 40 GbE switch
36 x 40GbE switch
SM
CFS
DQM
10 GbE
540x32 (1/10 GbE)
48 x 10 GbE FUs
10 GBs IO
FU PCs
2 x 1 GbE FUs
Data backbone (10/40 GbE)
Technical Network (.cms)
BU-FU appliance
- 1 BU (256 GB RAM, 2TB magnetic disks)
- 16 FU nodes
- FU: Dual E5-2670 8 core (2 x1 GbE)
BU-FU appliance
- 1 BU (256 GB RAM, 2TB magnetic disks)
- 8 FU nodes
- FU: Dual Haswell with 14 cores (10 GbE)
CDR backbone
GPN

Figure 2.5: Full schematic of the CMS DAQ2 system depicting the technical infrastructure and subsystems involved in the DAQ process. Graphic: [10]

2.2.2 Data Flow inside the CMS DAQ

The full TriDAS System in its current shape is depicted in Figure 2.5. The processing starts at about 740 custom-made *Front-End Drivers* (FEDs). The FEDs deliver for each L1 trigger their event fragments of 0.1-0.8kB to *Front-End Readout Optical Links* (FEROLs) to which they are connected over point-to-point links using a custom protocol. The 576 custom-designed FEROL boards translate the custom protocols to TCP/IP and send data from each connected FED as a TCP stream over 10GBit Ethernet to a predefined *Readout Unit* (RU) computer. On the way, up to 18 streams are concentrated by Ethernet switches from 10GBit Ethernet to 40GBit Ethernet, connecting the 576 FEROLs to the 108 RUs.

The RUs decode streams, checksum the data and match fragments belonging to the same event into a *super fragment*. These super fragments then are sent to *Builder Unit* (BU) computers, where the full event is put together and cached. RUs and BUs are connected by a high-throughput Infiniband FDR network at a rated of 56Gbit/s and arranged in a balanced, *Folded Clos* network topology consisting of 12 leaf- and 6 spine

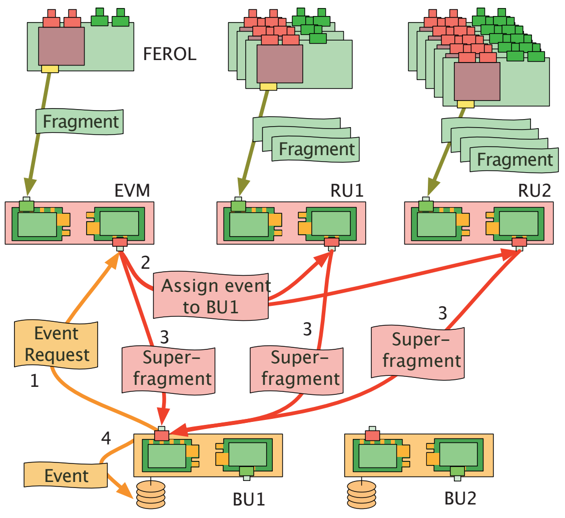


Figure 2.6: Visualization of the CMS event building protocol. BUs with available resources subscribe to the EVM (1). After choosing a BU all RUs then are notified by the EVM to send their super fragments for the assigned BU (2). Once all super fragments arrive at the same BU (3) the event can be built (4). Graphic: [9]

switches, providing 216 ports and 12Tbit/s bi-sectional bandwidth.

Filter Units (FUs) from the HLT farm then pick up the cached files from the BUs to perform the final selection step, that decide which events are sent to mass storage for later data analysis.

In the near future the end of life for most commercial equipment will be reached and replacements will be needed. A jump in CPU and network technology could allow RU and BU to be merged into a RUBU system, saving a significant amount of nodes in the Event Builder as well as expensive switch ports while using available network links in both directions compared to the unidirectional traffic of the current system.

2.2.3 The Event Builder Protocol

The Event Builder is a distributed system involving three types of actors: RUs, BUs and the *Event Manager* (EVM). RUs decode streams from multiple FEDs, extract event fragments and assemble those belonging to same event into a super fragment. BUs perform the assembly of a full event from all super fragments belonging to the same event. This means a reduction of all super fragments belonging the same event needs to take place from all RUs into a single BU. Furthermore, the BUs cache their assembled events on a RAM disk and make them available to the HLT farm. The EVM is a special kind of RU, receiving additional information from the Level-1-Trigger Control System and managing the interplay between RUs and BUs, acting as a load balancer.

Since each event has a different size, the time till the full event is built differs. Furthermore, these differently sized events result in differing processing times on the HLT. Without the presence of load balancing in the system, BUs that both build and serve completed events can easily get overwhelmed. As a result, the event builder protocol (Figure 2.6) between RUs, BUs and EVM loosely resembles a publisher subscriber pattern allowing load-balancing and a certain degree of fault tolerance.

BUs subscribe to the EVM (1), asking for events to build. The EVM then notifies the RUs about which super fragments for which events to send to which BU (2). If a BU either has no resources or experiences a hardware or software failure, it no longer asks the EVM for jobs and the Event Builder can continue without interruption. On the other hand, a failure of a RU or the EVM will cause the event builder to enter a erroneous state and block further data taking . To lower communication overhead, BUs will request a block of events to build which allows RUs to pack multiple super fragments and send larger messages over the Infiniband network.

2.3 The CMS DAQ software

Writing software that powers the data acquisition system of a high-energy physics experiment like CMS is a non-trivial task. High data rates force us to distribute the task to a cluster of machines, establish efficient and reliable communication between different actors and allow real time monitoring of the system. Furthermore, interaction with custom built hardware, upgrades of the detector, and changes in research interests create a dynamic environment with changing requirements and require a flexible software solution that will adapt to the environment.

XDAQ [8], [11]–[13] (pronounced cross-DAQ) developed at CMS reflects these requirements by providing a highly modular software framework of run time pluggable components for building a distributed data acquisition system for high-energy physics experiments.

2.3.1 System Architecture

XDAQ follows the layered middleware architecture pattern for distributed systems. An XDAQ application consists of a basic executable, launched on all processes that is configured and extended at run time using dynamic libraries specified in configuration files (see Figure 2.7). Around a *middleware executable*, *core* and *application* plugins extend the system. While application plug-ins provide application logic, core plugins manage system and hardware related functionalities like memory, inter-process communication and block device access. Application- and core interfaces provide middleware functionality to the plugins.

This strict separation of application logic from hardware and system specific tasks as well as the preferential usage of industry standard protocols allow portable application code and quick adaptation to new hardware.

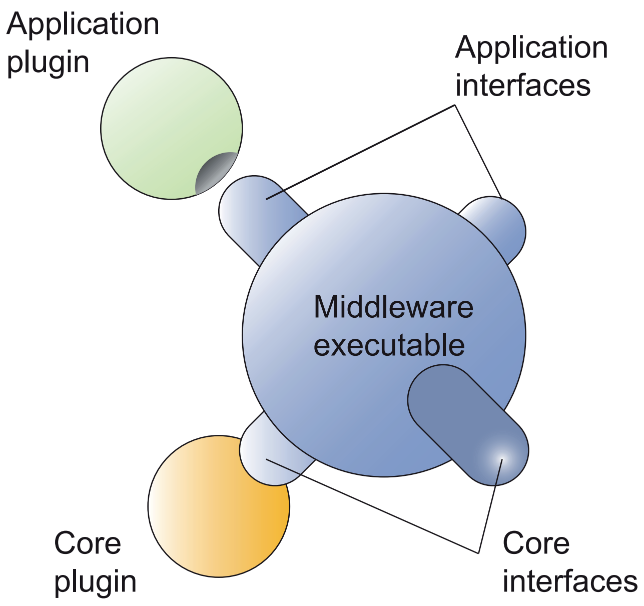


Figure 2.7: Illustration of the plugin-based architecture of XDAQ.Graphic: [8]

Amongst others, the middleware executable manages data transmission, address resolution and information dispatching to applications. Applications use these services to communicate on the basis of a peer-to-peer message passing model with an event driven processing scheme using user-supplied callback functions.

2.3.2 Data Transmission

Data transmissions in XDAQ are performed by application components referred to as *Peer Transports*. A Peer Transport registers itself with the executable declaring it can resolve addresses and exchanging data. Application plug-ins use the messaging services provided by the middleware executable which re-directs messages to a suitable peer-transport. This again promotes portability of applications and the integration of new communication technologies and protocols without changing applications. Furthermore it allows simultaneous use of different communication protocols e.g. TCP/IP over Ethernet for monitoring traffic and Infiniband RDMA messaging for Event Building traffic without having to use different APIs.

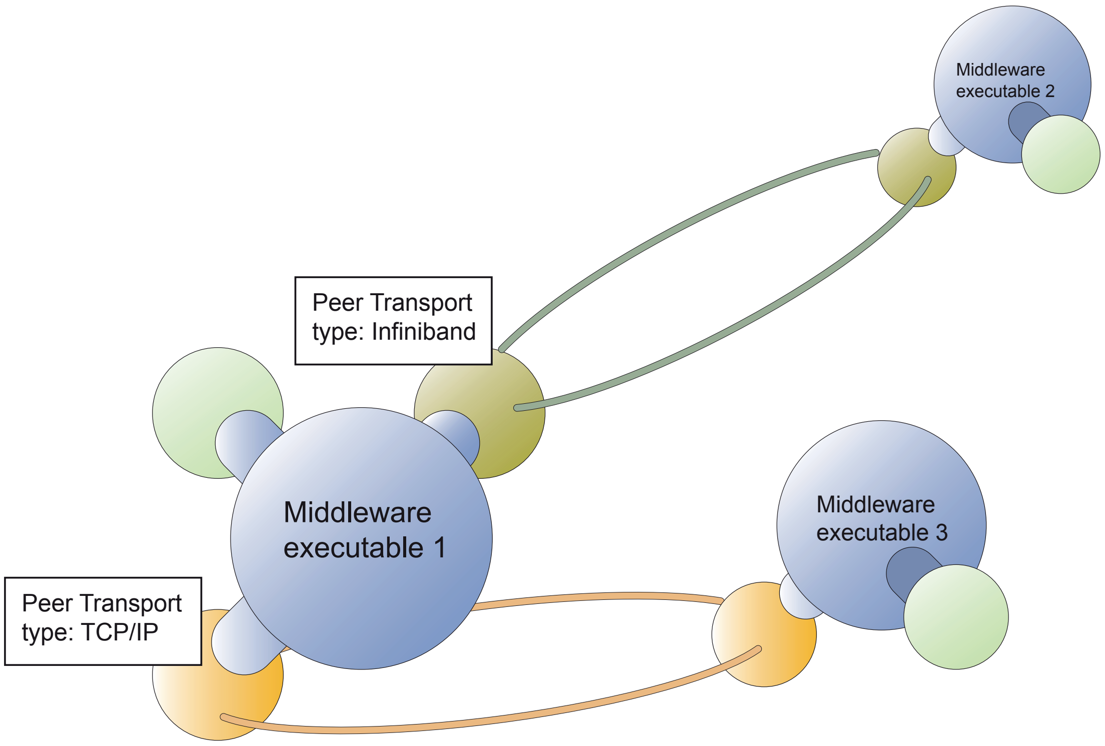


Figure 2.8: Communication of applications using Peer Transports. Graphic: [8]

3 Background on Used Technologies

With the high network throughput required for event building at 100KHz specialized RDMA interconnects such as Infiniband are an alternative to classical sockets programming. In this chapter we will therefore explain the fundamentals of RDMA network technology. Afterwards we briefly introduce the message passing interface standard MPI and one of its implementations named Open MPI, designed as a framework to simplify messaging in the context of distributed memory applications across various interconnects. The rest of the thesis will then build upon this technology stack investigating where and how MPI can be used in the CMS event builder.

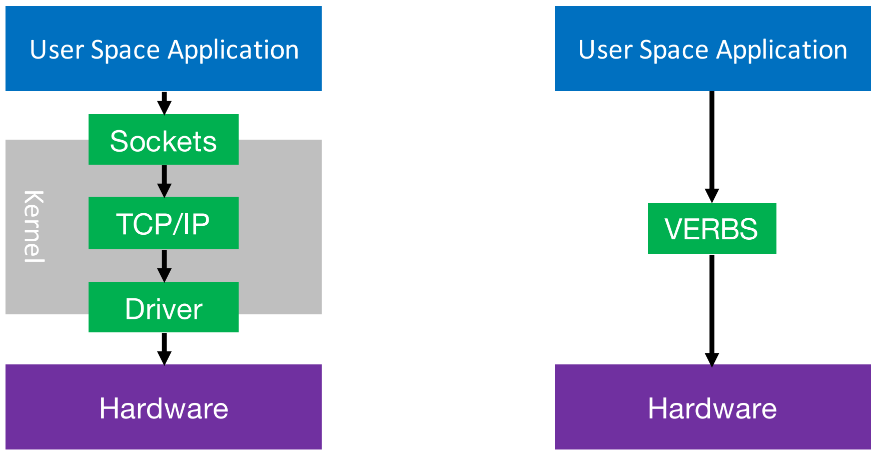
3.1 High-Throughput RDMA networks

Network controllers, as an I/O device are a shared resource. As such they are usually owned by the operating system which manages resource and communication with the device. Network I/O functionality is exposed as a service of the operating system. Thus the communication between applications over a network will happen indirectly through the mediating layer of the operating system on the sender and receiver side [14].

While this approach provides ease of use as well as hardware and operating system independence, involvement of the OS is a source of inefficiency caused by frequent context switches and may involve multiple memory copies of buffers on their way from source to destination as it is the case in TCP socket programming [15].

An alternative to this system-centric view is an application-centric view. Instead of seeing the network mainly as an operating system resource, it can be seen as a mean to move data from the address space of one application to the address space of another application. The main benefit of this view is that it does not actively involve the operating system. Instead messaging systems can be made available to the application directly. An application centric view and operating system bypassing are the common denominators of *Remote Direct Memory Access* (RDMA) hardware [16].

While there are several vendors of RDMA hardware, they are members of the *OpenFabrics Alliance* [17], that provides open source tools and drivers in the *OpenFabrics Enterprise Distribution* (OFED). The purpose of the alliance is a common, low level programming interface for RDMA enabled applications to promote sales of RDMA



(a) TCP sockets approach (b) Kernel bypassing of RDMA

Figure 3.1: Comparing the Kernel bypassing technology of RDMA to sockets programming

hardware by ensuring interoperability of applications. Currently all commodity of-the-shelf RDMA network technologies like Infiniband, RoCE, iWARP and Omnipath are part of the OpenFabrics Alliance.

We will describe the main concepts behind RDMA networks at the examples of Infiniband, as it is the most popular on amongst the RDMA networks on clusters of this scale [18] and currently is in active use at the CMS Event Builder Network.

3.1.1 Connecting Processes with Channels

Since processes live on disjoint address spaces, a pipe in the shape o a communication *channel* needs to be established [16]. On each end of the channel sits a `Queues Pair` (QP)containing a `Send Queue` (SQ) and a `Receive Queue` (RQ). These `Queue Pairs` are directly exposed to the the user application to avoid involvement of the OS. If communication with multiple applications is desired, a separate channel needs to be established to each of them. Thus applications will have a `Queue Pair` per communication partner. Furthermore, channels opened by different applications on the same network controller need to be isolated from each other.

To achieve both security and performance, a transparent multi layer architecture is used with both *kernel* and *userspace drivers*. Tasks associated with the management of the network interface adapter as a system wide shared resource, e.g. establishing of isolated, secured channels is managed by the kernel space drivers. Once resources are assigned, performance critical operations are handed over to userspace drivers that

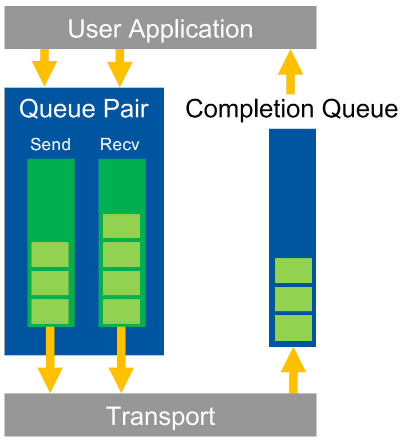


Figure 3.2: Interaction of user code with network hardware through a a queueing model in OFED VERBS.

directly access the underlying hardware without having to go through kernel space drivers.

3.1.2 Channel I/O

Once a channel is established, data can be transfered from sender to receiver using `SEND/RECEIVE` semantics. The sender application requests the transmission of a memory segment to a receiver who expects a message and pre-posts an adequate memory segment for receiving data. Both sending and receiving is initiated by submitting a `Work Request` to the `Send Queue` or `Receive Queue` of the respective `Queue Pair`. Memory segments from the sender are transmitted as messages directly into the indicated memory segment on the receivers side. Therefore a send needs to be expected by the receiving side. This is the only way a true zero-copy transmission without buffering can be ensured.

Another type of supported operations are `RDMA READ`/ `RDMA WRITE`. Instead of active involvement of both sender and receiver in the transmission process, one side, the *initiator* accesses a remote memory location on the *target* to read or write data. The benefit of `RDMA READ`/ `WRITE` is that the target application and CPU are not involved and the transmission is completely managed by the network interface adapters. However, the initiator needs to be made aware of the remote addresses to access and the *target* will need to be manually informed about completed RDMA operation by the initiator. This programming model can be a better choice if a sender cannot ensure the existence of a matching pre-posted buffer to receive into or if frequent memory transfers into fixed, predefined buffers occur.

In case of Infiniband, Mellanox claims close to equal performance of both `RDMA READ/WRITE` and `SEND/RECEIVE` semantics [19], thus developers are free to choose the approach which maps most naturally onto the problem.

Once a `Work Request` is posted into one of the `Send` - or `Receive Queues` the task is offloaded to the network hardware. On completion of an operation, an `Event` is pushed to a `Completion Queue`, indicating to the application that the network hardware finished processing the `Work Request`. Meanwhile the application was able to use CPU resources for different tasks. Nevertheless, `Completion Queues` need to be checked periodically for new `Events`, i.e. *polled*. Only after the reception of an `Completion Event` can the application safely reuse memory segments used for sending or assert that data was fully received.

The necessary functionality to establish channels, interact with Send/Receive and Completion queues is possible through the *VERBS API* that is used to interact with OFED devices. It represents the lowest level layer an application can use to interact with the messaging services provided by RDMA hardware. The VERBS API provides high flexibility and allows building of complex and high throughput, low latency distributed memory applications portable across all OFED devices at the cost of programming complexity [20]. For a set of problems a higher level network API that uses VERBS internally may largely reduce implementation complexity for users at an acceptable performance penalty.

3.2 The Message Passing Interface Standard Specification and its Implementations

The rise of distributed memory computing entailed the creation of different frameworks following a message-passing parallel model where parallel processes communicate by moving data from the address space of one process to the address space of another process through cooperative operations on each process [21, p. 1]. In 1992, a consortium of research institutes, universities, vendors and industry began standardization efforts leading to a *message-passing library interface specification* called Message Passing Interface (MPI) [21]. The goal was to design a language independent application programming Interface (API), that allows efficient communication of processes in a heterogeneous environment and can be implemented on many vendors platforms.

Over the years the standard was extended to include different types of communication, datatypes, dynamic creation and management of processes and parallel file I/O. The API is suitable for developing applications where multiple-processes operate asynchronously on different sets of data. This paradigm is referred to as *multiple instruction,* *multiple data* abbreviated MIMD [22]. Today, MPI is mainly used for programming applications on large clusters and super computers. The predominant type of parallel applications used on these kind of machines is a single program being launched on multiple processes operating on a subset of a large problem for work sharing to obtain results faster. This approach is called *single program, multiple data* (SPMD), a more restricted subset of MIMD.

3.2.1 Inter-Process Communication in MPI

MPI is centered around communication of parallel processes. These can either be located on the same system or on a remote machine connected via a network. Each processes is assigned an integer in ascending order, called *rank* by which they are addressed. One or multiple processes can be organized in a group called *communicator*, with the *default communicator* containing all processes available at startup time.

The MPI standard specification further defines a set of communication operations suitable for a large variety of use cases. In general, these communication operations can be subdivided in three different groups:

Point-to-Point Communication allows exchanging data between two ranks using explicit send and receive operations. It is the most basic type of communication in the MPI standard, see [21, p. 23ff].

Collective Communication procedures involving all ranks in a communicator simplifying frequent communication patterns like broadcasting data to all other ranks in a communicator or gathering computation results from all ranks in a communicator to a single rank. All collective communication routines can be written entirely using MPI point-to-point procedures. For an in depth overview see [21, p. 141ff].

One Sided Communication also referred to as Remote Memory Access (RMA) allows an *origin* process to directly operate on remote buffers previously advertised by another process (*target*) without any communication calls being issued by the target. This differs from point-to-point communication, where both sides actively need to participate. See [21, p. 401ff].

3.2.2 Implementation of the Standard in Open MPI

MPI is a *message-passing library interface specification* where all parts of the definition are significant [21, p. 1]. The existence of multiple implementations desired and necessary to meet the different requirements of the community. One of these implementations is *Open MPI* [23]. It is an open-source implementation of the standard, backed by vendors, research institutes and universities with the goal of a high-performance, scalable,

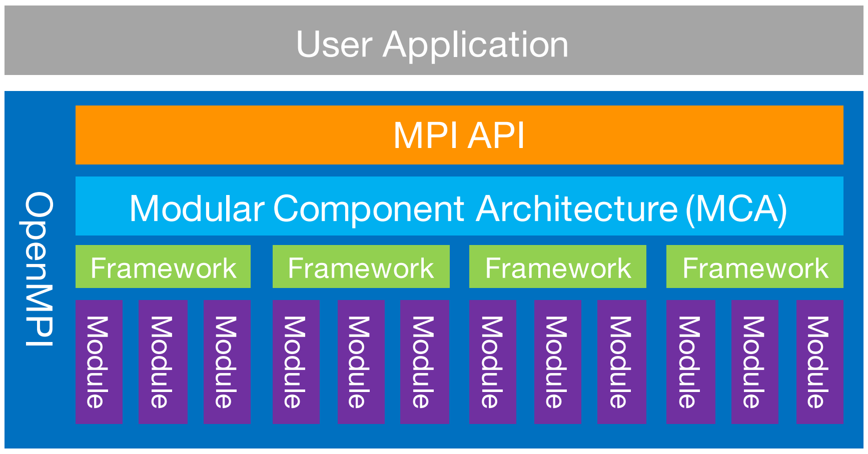


Figure 3.3: A plugin high-level view of Open MPI showing the Modular Component Architecture

production-quality MPI implementation [24]. Open MPI does not only provide an implementation of the MPI standard in a library with C and Fortran bindings but also a compiler wrapper that sets up the environment to compile MPI applications as well as tools to efficiently launch distributed applications with a built-in system to manage hardware locality.

The MPI standard defines the MPI procedures and their expected behavior. It is up to the implementations to decide which hardware to support and how to implement the functionality as long as they adhere to the definitions in the standard. For maximal flexibility Open MPI is designed around a *Modular Component Architecture* (MCA) [24]. The architecture has three main building blocks:

MCA : The MCA serves as a backbone component that sets up ,connects, manages all *component frameworks*. This includes forwarding runtime parameters from the environment.

Component Frameworks : Major Functional areas are divided into component frameworks (e.g. network transport or memory management) which house and manage modules. Each framework has different rules about how many modules can be simultaneously loaded and how they interact internally.

Modules : software units exporting well defined interfaces that can be deployed and composed with other modules at run-time, adhering to the interfaces prescribed by the component frameworks.

This architecture provides a large flexibility as it allows the addition and removal of components or even whole component frameworks isolated from other parts of the software, while maintaining a stable API towards the user. To further increase flexibility, most modules expose a large set of parameters that can be configured at run time towards the users needs. Open MPI thus often acts as a basis for vendor customized MPI implementations. Specific modules targeted at users of the vendor hardware are supplied without having to provide a full implementation of the MPI standard.

3.2.3 Open MPI Transport over Open Fabrics Devices

As discussed in Chapter 3.1, all todays commercial off-the-shelf RDMA capable interconnects are programmable using the OFED software stack. They share the concepts of bypassing the OS kernel and Channel I/O.

Since over the past years, Infiniband has been the predominant RDMA technology, most resources about RDMA and Open MPI are centered around Infiniband but are equally relevant to all other Open Fabrics RDMA devices.

Memory Registration for RDMA

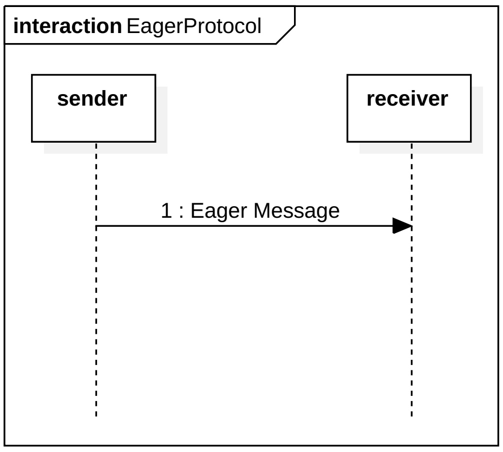
In order to establish which memory segment should be transfered over over the network, a common ground between the RDMA hardware and the user application needs to be established. This is achieved by communicating the physical memory addresses of the segment in question. To prevent the operating system of interfering by page swapping, the concerning virtual memory page must be stay mapped to the same physical location during communication. This operation is called memory registration. Once the segment is no longer needed for communication between the network interface and the user application, it can be unregistered.

The registration and unregistrarion processes are slow since they involve the operating system and context switches. Open MPI thus caches memory once registered in the hope to amortize for the registration costs, as long as the memory is not returned to the operating system. Therefore Open MPI needs to both remember registered memory and introduce hooks to deregister deallocated memory [25].

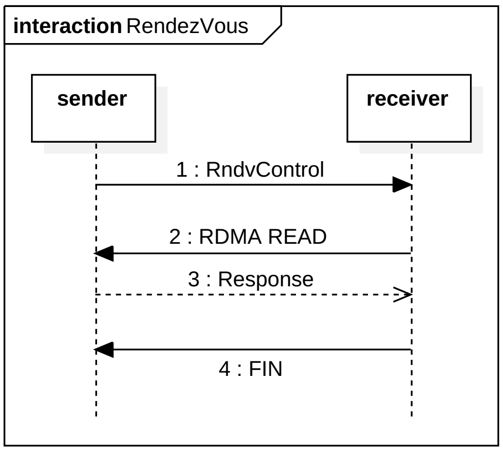
Since the MPI Standard abstracts from interconnects to be more universal, all of these processes are transparent for users of the MPI API, but behavior can be adjusted using the Open MPI module parameters at run-time.

Communication Algorithms for RDMA Architectures

One of the goals of the MPI standard is to allow high-performance communication. Performance in a network can be characterized in different ways. Two important



(a) Eager Protocol



(b) RDMA Protocol

Figure 3.4: Structure of eager and rendez-vous protocols in MPI implementations .

parameters are:

Throughput the amount of messages that were successfully transfered over the network in a fixed amount of time, often measured in bits or packages per second.

Latency the time required to transfer a fixed amount of data from source to destination, measured in seconds.

Where an optimal system has high throughput and low latency. However often it is difficult to achieve both goals at the same time and depending on the tasks, requirements are different.

The assumptions made in Open MPI is that for small messages that may occur e.g. when broadcasting a single integer to a set of nodes, low latency will be more important than throughput, whereas in cases where many large messages are transfered between nodes, the available network bandwidth has to be used efficiently.

As discussed in Chapter 3.1.2 on RDMA principles, the receiving side always must provide buffers for incoming messages, an unexpected message at the receiver is erroneous and will require a resend from the sender side. For the case of small messages Open MPI uses eager sending semantics, where the whole message is transmitted from sender to the receiver without prior announcement, expecting the receiver to provide a buffer and taking the risk of a resend see Figure 3.4a. For the case of large messages, a rendez-vous protocol is used (Figure 3.4b). First the initiator sends a setup message communicating the location of its data. The target side then fetches the data using an `RDMA READ` to its destination buffer once it is available and acknowledges the end of transmission to the initiator [26]. If additionally memory registration of the send buffer is required, more complex pipelining system is used in order to hide latency of the registration process [27].

For small to medium sized messages furthermore, the overhead of registering a memory segment takes very long compared to transmission size. For these cases,some vendor optimized modules perform copies of a memory segment that should be transmitted into a preregistered bounce buffer [28]. While this behavior will perform well for small to medium sized messages, the scalability of buffered copies is limited by system resources.

Vendor Implementations

The current RDMA hardware for the EVB network is an Infiniband FDR system by Mellanox. With their involvement in Open MPI, Mellanox offer a vendor optimized version of Open MPI [29] containing custom modules optimized for their hardware and OFED driver version.

Currently Mellanox proposes the use of these modules for optimized transport over Infiniband build on top of the OFED VERBS stack:

mxm [30] is a vendor proprietary implementation of a communication library by Mellanox for Infiniband hardware.

ucx [31], [32], is a new, open source development backed by a consortium of industry and research - including Mellanox - with the goal to form a communication middleware supporting various interconnects and is suitable for MPI and other distributed memory approaches such as openSHMEM.

Configured properly, both libraries performed nearly identical in benchmarks and applications.

4 Requirements Analysis

The main purpose of this thesis will be to study if and how MPI can be used for Event Building at CMS instead of OFED VERBs. One of the major concerns of the responsibles is to understand how much a high level framework like MPI will affect the maximal bandwidth of the system. Furthermore our investigations should answer the question how well the procedures defined by the MPI standard map onto the throughput driven task of event building and if possible trade-offs in performance can be compensated by a more programmer friendly API.

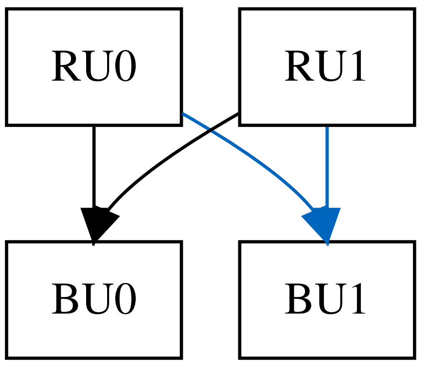
As the whole EVB system is too complex to consider for such an investigation, simple and reproducible benchmark applications that depict the communication patterns of the current, two-stage RU to BU Event Builder as well as a possible, future single-stage Event Builder with RUBUs should be created:

nton_MPIstreamIO simulates the communication pattern of the two-stage Event Builder on a fully balanced system with an equal amount of RUs and BUs where BU needs to gather fragments from all RUs, see Figure 4.1a

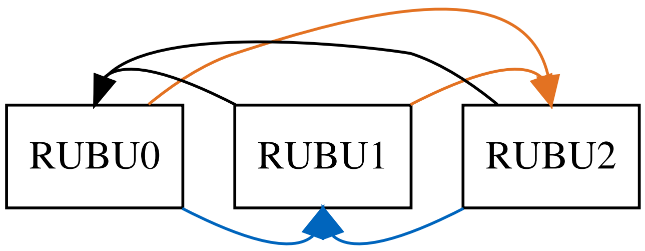
nxn_MPIstreamIO simulates the communication pattern of the single-stage Event Builder where each RUBU needs to gather fragments from all other RUBUs, see Figure 4.1b

Both benchmarks should provide an upper bound for throughput for both communication patterns in a simplistic and reproducible manner. Therefore the following simplifications are made:

1. Senders never run out of fragments.
2. No scheduling is required.
3. There is no notion of events nor any distinction of fragments and super-fragments.
4. Receivers discard buffers instantly without having to wait for all remaining data belonging to the same event.
5. All fragments have the same size.



(a) Two stage EVB: A BU gathers fragments of the same event from all RUs.



(b) Single stage EVB: A RUBU gathers fragments of the same event from all other RUBUs.

Figure 4.1: Communication Patterns for different styles of event builders

While the benchmarks applications are an idealization, the transport algorithms should be fully functional and able to act as a drop in replacement for other transports inside XDAQ applications. Therefore the following requirements apply:

1. Both benchmark applications nton_MPIstreamIO and nxn_MPIstreamIO should scale to the size of the current CMS DAQ2 event builder system of ~180 nodes.
2. The MPI transport has to cope with slightly varying fragment sizes during runtime. Neither the fragments within one event nor the fragments generated by the same subsytems over time can be assumed to have constant sizes.
3. The transport algorithm should be versatile, handling both unidirectional and bi-directional traffic for benchmarking applications and suitable as a drop in replacement for an XDAQ transport.
4. Transport algorithms should not have any predetermination on the network structure.
5. There is no target buffer size. The MPI transport therefore should show performance close to OFED VERBs for any super-fragment size larger than the Infiniband MTU size of 4kB.

5 Designing a Throughput-Oriented Messaging Service on Top of MPI

With requirements, technology stack and context in place, we can start analyzing our requirements and propose a solution to the posed problem.

5.1 Design Study: MPI Calls for Event Building Traffic

Over the different versions of the MPI standard specification [21] a significant amount of functionality has been added. Today, it defines a large set of communication operations suitable for a large variety of use cases. As discussed in see Chapter 3.2.1, these communication operations can be subdivided in three different groups: Point-to-point communication, collectives and one-sided communication. Each group contains procedures adhering to the same design concepts. As the choice of communication strategy is at the core of our task, thorough investigation is required.

MPI Collectives

When we look at the communication patters of the two-stage event builder and the single stage event-builder, we quickly find suitable MPI collectives that match the tasks well.

In the two stage event-builder, each bu BU gathers event fragments belonging to the same event from all RUs. This maps onto the different variations of gather collective procedures in MPI, where buffers from a group of ranks are concatenated into a buffer at a single rank, referred to as the *root* [21, p. 149ff and p. 200ff]. The pattern is illustrated in Figure 5.1.

In the single stage event builder, a RUBU receives fragments from all other RUBUs including itself for the events it was scheduled to build. All other fragments are sent to the respective RUBUs. While again, this can be modeled by a set of gather procedures, the MPI standard specifies separate procedures for *complete exchange* [21, p. 168ff and p. 206ff] that fits the task very well. The pattern is illustrated in Figure 5.2.

If all communication for event building traffic could be covered by a single MPI collective procedure, this could largely simplify the transport layers of an event builder.

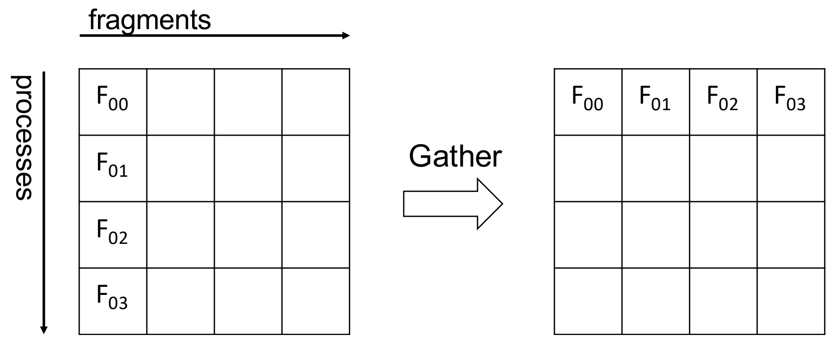


Figure 5.1: Visualization of the MPI gather collective pattern. Buffers from different processes are gathered into the memory of a single process

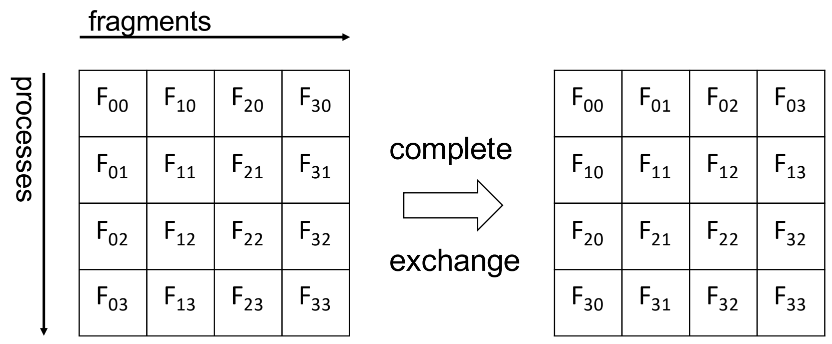


Figure 5.2: Visualization of the MPI complete exchange collective pattern. Buffers are scattered and gathered at the same time amongst all participants. The effect is similar to that of a matrix transposition.

On top of that, the BU in charge of the event assembly knows upon completion of the collective operation, that all data belonging to the same event has been successfully received. Unfortunately the MPI standard for gather procedures is very restrictive demanding that " the amount of data sent must be equal to the amount of data received, pairwise between each process and the root." [21, p. 150 and p. 152]. This means that either fragment sizes must be fixed or fragment sizes have to be communicated explicitly to the root before the gather procedure is initiated. Fixed fragment sizes collide with requirement 2 from 4. The problem could be resolved by padding bytes to achieve constant sizes, however this option is not acceptable since there is no predefined message or fragment sizes. This could lead to absurd cases where more padding than data is sent. The communication of variable sizes increases latency by preceding each gather procedure with another one communicating the send sizes to the root process. The same holds for complete exchange procedures [21, p. 169 and p. 171ff]. Further it is not possible to cancel an ongoing collective operation in case of failure, e.g. when fragments for subsystems do not arrive at a RU because of a hardware fault upstream. All these considerations lead us to believe that it is neither safe nor efficient to use collective procedures for event building.

Point-to-Point Communication

All collective operations can be written entirely using MPI point-to-point procedures. Thus a gather collective can be broken down into the RUs each sending a buffer and the BU as the root of the gather calling a matching number of receives. For the single stage event builder, A gather operation executed by each RUBU then has the same effect as a complete exchange.

At first glance this approach seems to be inferior to the collective operation. Not only does it drastically increase implementation and testing efforts during development but we also have to explicitly implement functionality to verify that all fragments arrived at their destination and the BU can proceed building the full event. However the use of point-to-point communication removes severe limitations of collective operations. While the standard requires collective procedures to pairwise match send size and receive size at the root, receiving of messages in point-to-point communication much less restrictive. The only explicit restriction by the standard is that "the length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer" [21, p. 29]. This allows us to receive variable sized messages into sufficiently large buffers without knowing the exact size of the data we receive in advance. Furthermore the standard allows cancellation of receive operations, increasing fault tolerance .

We therefore can conclude that the point-to-point communication capabilities of MPI offer the required flexibility to design and implement a transport that is aligned with the requirements specified in chapter 4 at the cost of a more complicated system design and higher implementation efforts.

MPI One Sided Communication

One sided communication semantics is among the more recent extensions of MPI standard definition. It is based on the idea of Remote Memory Access (RMA) where processes - called *origin* - can operate on memory made accessible by another process - referred to as *target* - without intervention of the latter. While Message-passing communication achieves both communication of data from sender to receiver and synchronization of sender with receiver, RMA design separates these two functions [21, p. 401]. This results in communication that is separated into phases. In an initialization phase, buffers are declared by the targets which should be made accessible to remote hosts. Synchronization calls open up and close a so called *access epoch*. During an epoch zero or more RMA calls to targets can be issued. This amortized the synchronization costs with multiple data transfers allowing the implementation of complex communication patterns and is especially useful in applications with distinct communication and computation phases.

With a more real time natured style of our application where communication does not happen in phases but in a constant stream, one-sided procedures do not map well onto the concept of the CMS event builder and therefore should not be further considered. Instead we recommend an implementation of the EVB communication patterns using point-to-point procedures.

5.2 Defining a Messaging Service on Top of MPI

Our findings from the analysis of MPI procedures conclude that the only efficient way to mapping the communication patterns of the CMS Event Builder to MPI while adhering to the requirements is point-to-point communication. The cost of not being able to use collective MPI operations is that the BU has to manually understand when all super-fragments belonging to the same event have arrived so that the event can be built.

On the other hand, the use of traditional message passing semantics allows us to clearly separate the network transport from the actual task of event building and design a throughput-oriented messaging service on top of MPI for multiple destination. In a resulting distributed application, processes can communicate with each other based on the specific task without having to use MPI directly.

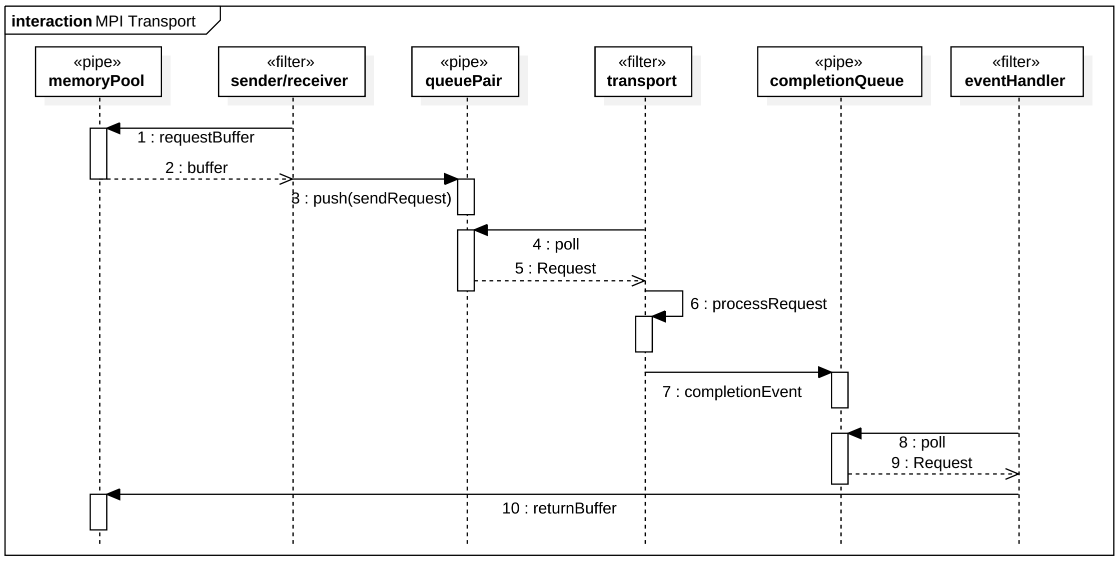


Figure 5.3: Pipes and Filters architectural style for a messaging service based on MPI that allows flexible replacements of components and enables functional parallelism

5.2.1 Queuing and Multithreading using a Pipes and Filters Architecture

At the high frequency the detector readouts provide data to the Event Builder we can eventually interpret the problem as a stream processing problem. A highly flexible architectural pattern that is suitable for systems processing streams of data is *Pipes and Filters* [33, p. 53ff]. It breaks down the stream processing task into subtasks (*filters*), connected by objects (*pipes*) that buffer and transport data between two filters. With a filter being only dependent on the products of its preceding filter, the pattern allows filters to be swapped out easily and with pipes acting as buffers, each filter can be its own process or thread. Furthermore this architectural style is aligned with the concepts of low level RDMA network programming.

We therefore propose the architecture depicted in Figure 5.3. It is based on three filters and connected by three pipes.

Memory Pool is a pipe that provides memory buffers for sending and receiving.

Sender is a filter requesting memory buffers from the pool into which the event fragment to be transfered is put. Based on its destination, the buffer then is passed to the appropriate Queue Pair pipe.

Receiver drop in replacement for the sender. A filter requesting memory buffers from

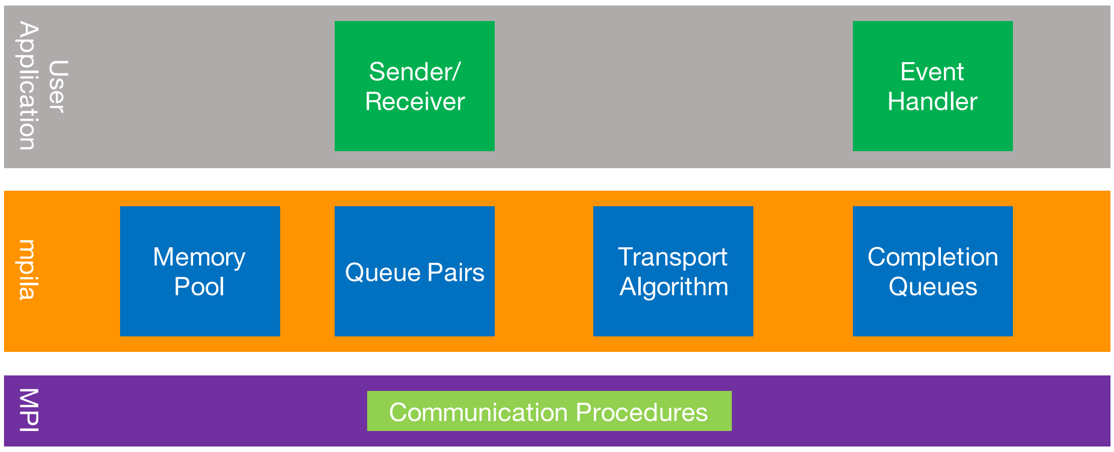


Figure 5.4: Layers view of mpila.

the pool into which fragments will be received. It also leaves a callback to be executed upon completion of the receive operation.

Queue Pair For each possible destination there is a Queue Pair. A queue pair holds both requests to send and requests to receive buffers in separate queues.

Transport The transport filters picks up send and receive requests from all Queue Pairs pipes and performs the transmission over the network fabric. Upon completion of a request an event is generated and passed to the Completion Queue pipe.

Completion Queue pipe that holds events for all completed send and receive request.

Event Handler filter picks up completion events executes callbacks if required before returning buffers back to the resource pool.

5.2.2 Reusability and Reduction of Complexity by Layering

The architecture proposed above includes pipes and filters that will be reusable in any application using this transport. On the other hand, the Sender, Receiver and Event Handler filter will differ in each application.This allows us to package the reusable components into a library usable by other applications which we call *mpila*. As a mediator between user applications and MPI, mpila can be seen as a classical representative of a *layered architecture style*.

The layered architecture style [33, p. 31–51] has been widely applied in the design of network protocols such as the *OSI-model* [34]. The pattern structures subtasks based on their level of abstraction with higher level operations relying on services provided by lower layers (see Figure 5.4). Communication usually traverses layers from high to low as requests with notifications about events traveling in the opposite direction. As each individual layer shields all lower layers from direct access by higher ones, the coupling between layers is very loose promoting reusability, exchangeability and portability at the cost of overhead being generated by each individual layer.

This design has multiple benefits. The clean separation of the application domain from the networking domain makes mpila a reusable library for our various benchmarking programs or for third parties that need the flexibility of an MPI based transport for existing applications inside the XDAQ environment without any knowledge of MPI or changes in their user code. Finally, the request oriented API of the library allows us to keep a stable interface towards user applications, while components inside mpila could be swapped out. This was especially helpful for evaluating different communication strategies during development while ensuring that all algorithms are benchmarked under the same conditions aiding reproducibility and comparability of results.

5.3 Deliverables

In accord with the requirements specified in 4 the following deliverables will be supplied:

mpila communication library based on MPI providing all reusable elements of a network transport for event builder traffic.

nton_MPIstreamIO executable benchmark application to simulate the communication pattern of the two stage event builder.

nxn_MPIstreamIO executable benchmark application simulating the communication pattern of the single stage event builder.

ptmpi library as a prototype usable as a peer transport for the XDAQ framework.

6 Implementation

This chapter discusses the particularities of implementing the individual components in the system architecture. The first part discusses the transport library mpila, while the second part puts it's focus how mpila is used inside the benchmarking applications.

6.1 Memory Pools for Efficient Memory Handling

The task of a memory pool is to manage buffers used for sending and receiving. Efficient handling of memory buffers is one of the corner stones in a high performance RDMA transport. As discussed in 3.2.3, memory buffers that should be used by RDMA hardware require prior registration with the operating system to preserve the virtual/ physical mapping of the memory page the buffer is located on. Since memory registration and deregistration is a slow process involving context switches and requires MPI to do bookkeeping, it is desirable to reduce these operations to a bare minimum. This can be achieved by using a resource pool that holds a predefined amount of memory buffers which will be registered with the operating system by MPI upon first use and reused in subsequent operations.

Another benefit of a resource pool is flow control: If there is an imbalance between sender and receiver where the receiving side is overwhelmed by the sender, limited resources inside the pool are used up and no new receive operations can be initiated until memory is returned. As each send operation requires a matching receive, the sender is thus automatically restrained.

A simple yet efficient implementation of a pool is to internally enqueue references to memory buffers in a bounded FIFO-queue. If a buffer is requested, it is removed from the front of the queue, while returned buffers will be enqueued at the back. In case of an empty queue, resources are depleted. If multiple threads are accessing the same memory pool, the FIFO queue needs to be thread safe.

6.2 Queue Pairs and Completion Queues for Request and Event Handling

Another crucial part of the architecture are the pipes that forward requests to the transports and buffer the events generated upon completion of a request. Taking inspiration directly from the design concepts behind Infiniband – see 3.1.2 for more details – we propose a queuing model. The idea is to introduce a pair of bounded FIFO-queues per MPI rank, one for requests to send data, another one to receive data. The separation is sensible for several reasons. The design study in 5.1 yielded that the most suitable way to employ MPI for our use case is to use point-to-point communication routines consisting of data exchange via explicit sends and receive procedures. This makes subdivision into separate send and receive queues a logical choice. Another important aspect is the enforcement of fairness between (possibly threaded) senders/ receivers and different ranks. By granting each rank the same, limited resources we can assert using a petri net that the system will remain deadlock free if the transport algorithm performs a fair scheduling, that avoids starvation. By bounding the queues we again introduce flow control, preventing a sender or receiver from flooding the transport with requests at a rate they cannot be processed.

Once the transport processed these requests, it pushes a notification about event completion into the FIFIO `Completion Queue` referenced by the `Queue Pair`. This gives end user applications the flexibility to use one or multiple completion queues for different event types or destinations. As the network hardware causes all our sends and receives to be inherently serialized with one send and one receive operation at a time, a single, large `Completion Queue` for all queue-pairs should be adequate, if the filter that processes the completion events has a sufficient throughput.

6.3 Work Requests and Completion Events

As a logical follow-up the `Request` and `Event` structure should be described. `Requests` no matter if used for sending or receiving require a quasi identical set of parameters:

Target rank is the rank that should be sent to or received from. This information is implicitly given by the `Queue Pair` the request was submitted to.

Pointer to a buffer the buffer to be sent from or received into by the interface adapter.

Buffer size in bytes. For the sender it indicates the size of the buffer to be sent, for the receiver it indicates the maximal size of the receive buffer.

Tag optional for matching sends and receives belonging to the same event fragment.

Memory Pool reference to the `Memory Pool` the buffer was taken from, in order to return the buffer to the correct `Memory Pool`.

Context an optional pointer to a callback that should be executed once the request is completed.

These parameters can be put into a structure that is submitted to the Queue Pair.

Once a request has been completed by the transport filter, status information about the completion of the request will will have to be appended, providing feedback whether the operation was completed successfully or if communication was erroneous. Furthermore, in case of a receive operation, the size should be updated such that the amount of bytes received is correctly communicated to an `Event Handler`. Other information such a reference to the memory pool and a potential callback function are forwarded appropriately from the `Request`.

6.4 Duplex Transport Based on MPI Point-to-Point Communication

The `Transport` filter is the core component of mpila. Processing `Request` that have been posted to the `Queue Pairs` it performs the sending and receiving of data over the network using MPI point-to-point procedures. Based on its architectural specification in Chapter 5 resulting from the requirements in Chapter 4 the following components can be identified :

- Polling of all `Queue Pairs` for `Requests`
- Sending to all other destinations.
- Receiving from all other senders.
- Notifying event handlers of finished operations by generating a `Completion Event`

All of theses components need to be arranged such that they provide a correct and efficient transport algorithm.

6.4.1 Processing of Requests

Our design foresees `Queue Pairs` as the endpoint of individual channels established with every other process in our distributed application. Without any predetermination on a communication pattern or network layout, the transport algorithm has to treat requests from all queue pairs equally. In our primary use-case of EVB communication this assumption is ideal, since the amount of requests to all communication partners will be uniformly distributed. A simple, yet effective solutions to fair processing of requests is a round-robin algorithm that iterates over all `Qeue Pairs` processing an equal amount of requests at a time, skipping a `Queue Pair` if no requests are present. We now have to map our multi-destination, bidirectional network protocol with a round-robin scheduling strategy onto MPI procedures.

MPI point-to-point communication can be conceptually divided into two blocks: *blocking* and *non-blocking* communication procedures [21, p. 47ff]. Both types consider a send operation complete, if the send buffer can be reused and a receive operation as complete, if the receive buffer contains the data of its matching send operation. The difference is that a blocking procedure will block program execution until the operation is completed. A non-blocking procedure on the other hand will return immediately with a handle to a `MPI_Request` object, completing the operation in the background. The application then can use this object to either test for completion or issue a blocking wait at a later point in time. Both blocking and non-blocking procedures can be mixed in an application.

The use of blocking operations is disqualified by the inability to flexibly start bidirectional communication in parallel. Additionally the transport algorithm would have to ensure that two processes never open a blocking send operation to each other at the same time. This would cause a deadlock as both processes in the absence of a matching receive operation would block infinitely, waiting for the opposite side to start the necessary receive. The same holds for the case of blocking receives.

Algorithm 1 describes our solution to a bi-directional MPI transport. It is based on non-blocking point-to-point send/ receive procedures and a polling mechanism, that will check for completion of pending MPI operations. For each destination, we allow multiple sends and receives to be opened at a time forming a pipeline to ensure that the MPI library is kept busy processing requests. The length of this pipeline, which will be restricted in size will be referred to as *pipeline-depth*. The `Work Requests` and the `MPI_Request` of pending operations have to be remembered, since they are needed to post a `Completion Event`. We therefore introduce a *look-aside structure* for each `Send Queue` and each `Receive Queue`. We then check if any of the pending *Send Requests/Receive Requests* were completed by MPI. To avoid deadlocks, we perform a non-blocking test for completion, instead of a blocking wait. Completed requests are removed from the look-aside structure and pushed to the respective `Completion Queue`. Then we proceed to the next `Queue Pair`.

```
while true do
    foreach Queue Pair do
        while Pipeline not full do
            if Recv Queue not empty then
                non blocking recv;
                remember Request as pending recv;
                pop from Recv Queue;
            end
            if Send Queue not empty then
                non blocking send;
                remember Request as pending send;
                pop from Recv Queue;
            end
        end
        poll pending recvs;
        foreach finished recv do
            pop from look asside structure push to Completion Queue;
        end
        poll pending sends;
        foreach finished send do
            pop from pending list push to Completion Queue;
        end
    end
end
```

Algorithm 1: A high level description of the mpila full duplex transport algorithm.

6.4.2 Buffering Behavior of MPI Send Modes

An MPI send operation has completed once the send buffer can be reused. This kind of behavior helps to decouple send and receive operations, since data from a send buffer can be copied into an intermediate buffer provided by MPI. Transmission over the network is then performed once a matching receive has been posted by the receiver [21, p. 37ff]. The `MPI_Isend` procedure, which executes a non-blocking send decides internally if the message should be buffered. As for the algorithm we propose, buffering is not desired.

Our entire system architecture works on the concept of *back pressure*. If a consumer of requests is slower than the producer, bounded queues between theses objects will implicitly synchronize the two components. If the coupling between sender and receiver is broken and sends are deferred, there is no mechanism to propagate the back pressure produced by an overwhelmed receiver back to the sender resulting in an ever growing pile-up of deferred sends that at some point will cause the process to run out of physical memory. In earlier implementations of our algorithm using `MPI_Isend`, we have been able to reconstruct this exact behavior.

Fortunately the standard defines a set of send procedures, that refrain from deferred sending and buffering [21, p. 37ff]. Ready sends perform eager sending of data assuming that a matching receive has already been posted at the destination. Otherwise the operation is erroneous and results in undefined behavior. Since we cannot assert availability of preposted receives in all cases, we have to fall back to synchronous sends via `MPI_Issend` which on completion will not only will guarantee that the send buffer can be reused but also that a matching receive has been posted and started receiving the sent data.

6.4.3 Polling Strategies

As described in Algorithm 1, we submit multiple MPI send/receives in a non-blocking way per queue pair, which allows us to pipeline multiple sends. In modern MPI implementations, non-blocking point-to-point procedures are completed by threads in the MPI library, such that progress is made in the background. Pending operations are remembered in separate look-aside data structures for each `Send Queue`/ `Receive Queue` within a `Queue Pair`. The completion status of these pending operations has to be checked without a blocking wait operation. The MPI standard [21, p. 52ff and 57ff] defines four procedures out of which three are of interest for us:

MPI_Test checks for the completion of a single `MPI_Request` object returned by a non-blocking send/ receive operation. A boolean informs about the completion status of the posted operation. In case of completion a `MPI_Status` structure contains information about the completed send/ receive.

MPI_TestSome checks an array of `MPI_Request` objects for completion, returning the amount of completed operations, their indices in the array of requests and `MPI_Status` objects of completed operations.

MPI_TestAll also checks an array of `MPI_Request` objects for completion returning true only if all requests have been completed. In that case, an array of `MPI_Status` objects provides more information about the completed operations.

During the implementation phase it was unclear on which procedure would yield the highest throughput polling algorithm, thus algorithms for each procedure have been implemented and will be evaluated in Chapter 8. Each of the procedures has slightly different semantics and input parameters. As we expect polling to occur more frequently then posting of sends/ receives from the `Queue Pairs` or the creation of `Completion Events` we have to ensure the highest execution efficiency in this component of the transport algorithm. An important aspect will be an implementation of a look aside data structure that maps to the individual MPI testing procedures without additional copies or transformations.

MPI_Test

One of the premises of MPI point-to-point communication is the ordered, non-overtaking nature of messages. The standard demands that "if a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending " [21, p. 41]. Together with the guarantees of a synchronous send mode (see 6.4.2) this ensures completions of sends in a FIFO order. The very same holds for receive operations. We therefore can use a bounded FIFO queue with the length of our pipeline as a look aside data structure, containing the work request removed from the `Queue Pair` and the `MPI_Request` returned by the non-blocking MPI send/ receive operation. Since `MPI_Test` checks for the completion of a single `MPI_Request` at a time we always operate on the first pending request in the FIFO queue, and dequeue it in case of completion. In the case `MPI_Test` finds a pending, incomplete request, the ordering properties allow us to skip the verification of all remaining requests that may follow.

A schematic overview of the polling algorithm is depicted in Figure 6.1.To facilitate communication between the send/ receive, polling and completion components in the transport module, a system of two queues is used. Before the first send is issued, the `Free Requests Queue` holds references to `Pending Request Structures`. The sender

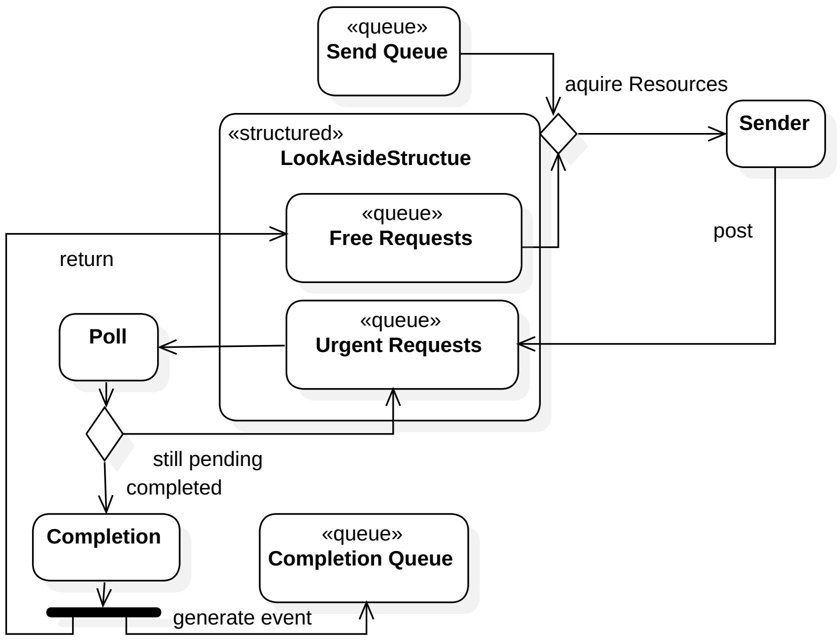


Figure 6.1: Polling strategy using `MPI_Test`.

needs to obtain both a `Pending Request Structure` from the `Free Requests Queue` and a `Work Request` from the `Send Queue` before an send can be submitted to MPI. Thereafter the send is considered pending and a filled `Pending Request Structure` is pushed to the `Urgent Requests Queue` which is checked by the polling mechanism. Once a pending request is completed, a `Completion Event` is pushed to the associated `Completion Queue` and the `Pending Request Structure` is returned to the `Free Requests Queue`. The same applies to receiving.

MPI_Testall

`MPI_Testall` operates on an array of `MPI_Requests` which allows us to check multiple pending operations for completion. This however requires us to arrange the `MPI_Request` objects in an array structure. We thus have to change the data layout of our look-aside data structures.

A schematic of the algorithm can be found in Figure 6.2. Again we can use a double queue system with a `Free Requests Queue` and an `Urgent Requests Queue` to limit the pipeline depth and remember pending `Work Requests`. The `MPI_Requests` however which are generated from the non-blocking MPI send/recv operations, will be arranged in an array, that is kept in sync with the position of the corresponding work request in

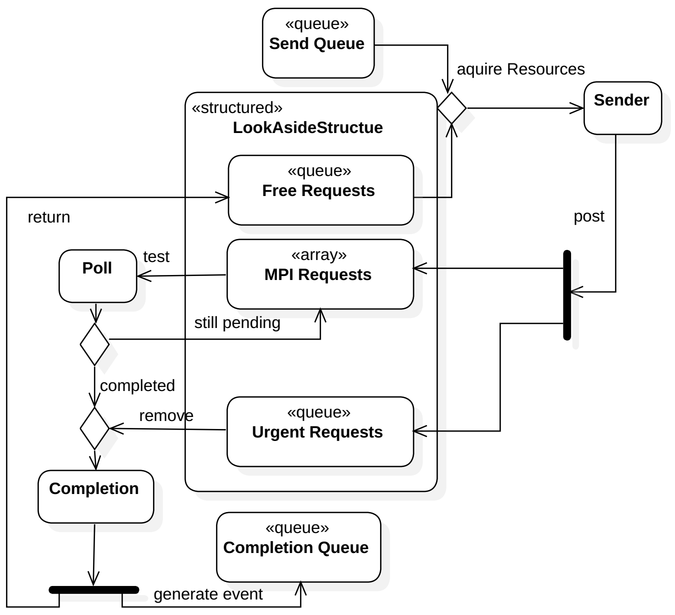


Figure 6.2: Polling strategy using `MPI_Testall`.

the `Urgent Requests Queue`.

This allows us to use `MPI_Testall` on all currently pipelined `MPI_Request` objects in the pipeline. We then can use the fact that `MPI_Testall` will only return true if all requests in an array are completed and clear out the entire pipeline at once, which keeps `MPI_Requests` and `Work Requests` in sync. Again completed `Requests` will be returned to the *Free Requests Queue* after a `Completion Event` has been emitted.

MPI_Testsome

Like `MPI_Testall`, `MPI_Testsome` operates on an array of `MPI_Requests`. Out of a set of `MPI_Requests`, the indices of those that are completed are returned in an array, while those that are not are kept pending. The effect is therefore exactly the same as looping over an array of request with `MPI_Test`.

When adapting the look-aside structures use with `MPI_Testsome`, we could rely implicitly on the ordered, non-overtaking properties of message passing in MPI. However the MPI standard does not specify any ordering for the indices returned by `MPI_Testsome` indicating which requests completed. We therefore have to find a more universal solution for the look-aside structure.

Our solution is outlined in Figure 6.3. Both `Work Requests` and `MPI_Requests`, are stored in their own array sized according to the maximal pipeline depth. Their positions

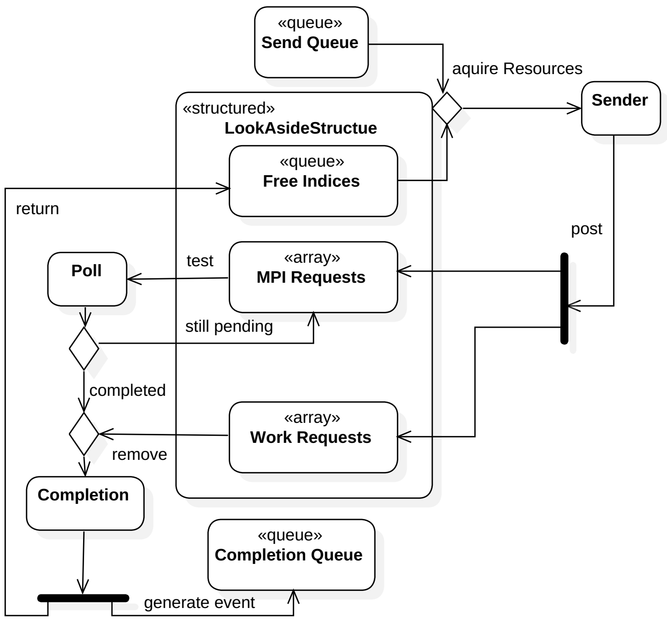


Figure 6.3: Polling strategy using `MPI_Testsome`.

are kept in sync. A FIFO queue contains the indices of currently inactive spots in those arrays, so that new `Work Requests` can be inserted without overwriting pending `Work Requests`.

`MPI_Testsome` has to always check the full array of `MPI_Requests`. If the array contains inactive `MPI_Requests` (i.e. those without any associated send or receive operation), they will be skipped. If the array contains no active handles, the amount of requests completed returns *MPI_UNDEFINED*, distinguishing it from the case where no `MPI_Request` has been completed yet.

6.5 Benchmark Applications

As described in Chapter 4 our primary goal is the implementation of benchmarking applications for simulating the communication patterns of the single-stage and two-stage event builder. Both applications will use the mpila library as their backbone, implementing only the `Sender/ Receiver` and `Event Handler` components (see Figure 5.4) which are individual to the specific user applications. The processing of `Requests` will follow the data flow described in Chapter 5.2.1.

Both applications `nton_mpistreamio` and `nxn_mpistreamio` will have the same basic structure. Each process creates a `Queue Pair` for all other MPI ranks. Each `Send`

`Queue` and `Receive Queue` within a `Queue Pair` is fed from a separate `Memory Pool`. All `Events` about terminated operations from all `Queue Pairs` will be pushed into the same `Completion Queue`. At the beginning, all receiver processes will prefill their `Receive Queues` with requests. Then individual threads for `Sender`, `Event Handler` and `Transport` are created. If the process is a sender, the `Sender` component will iterate over all `Queue Pairs`, trying to push new `Work Requests` in a round-robin way. The `Transport` will process them as described in Chapter 6.4. Finally the `Event Handler` treats `Completion Events` submitted to the `Completion Queue` returning `Send Buffers` to the appropriate `Memory Pool` and replacing closed `Receive Requests` by new ones in the corresponding `Receive Queues` of the `Queue Pair`. Additionally a separate Sampler thread is responsible for calculating the bandwidth out of the number and size of processed buffers, see Chapter 7.2.

6.6 A XDAQ Peer-Transport: ptmpi

XDAQ applications use Peer-Transport plugins to provide address resolution and messaging services, see Chapter 2.3.2. In order to include mpila into XDAQ, the respective Interfaces of a Peer Transport had to be implemented accordingly. Furthermore the buffers circulated by mpila `Work Requests` are no longer provided by the mpila `Memory Pool` but by XDAQ. The resulting XDAQ application plugin is called *ptmpi*.

In order to launch an XDAQ application with ptmpi, the Peer Transport has to be registered to the Middleware Executable using the XDAQ XML configuration files so that it can be loaded dynamically at application startup. To pass the necessary initialization information to the MPI library, the distributed XDAQ application is launched using `mpirun`.

7 Test Environment and Methodology

Before we can start analyzing the performance of our implementation and compare it to the current VERBS based system, we need to introduce our measurement environment.

7.1 Hardware

For the benchmarks, two kinds of systems have been used, DAQ2VAL and cDAQ. DAQ2VAL is the development and testing system, whereas cDAQ runs the CMS event builder. As the cDAQ system is only available for short periods of time during technical stops of the LHC, we have been taking most data on the development system and only performed scaling tests on cDAQ.

7.1.1 cDAQ Production System

The cDAQ system is the production cluster running the CMS event builder. It consists of 106 RUs and 73 BUs. Appart from BUs having significantly more RAM and a slightly adapted OS configuration the systems are identical:

- Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
- RAM : 32 GB for RUs, 297GB for BUs
- Mellanox Infiniband MT27500 Connect-X3 (FDR: 56 Gb/s)
- Mellanox OpenMPI 1.10.5a1 with Mellanox MXM
- GCC 4.8.5
- Centos 7.3.1611

The nodes are connected via an Infiniband FDR switched fabric (56 Gb/s) in a fat tree configuration. It consists of 18 switches with 36 ports, out of which 12 are the leafs and 6 are spines. To each leaf switch 15 nodes are attached, while 18 switch ports are used as uplinks to the spine switches. This results in each leaf switch being connected to each spine switch by 3 links.

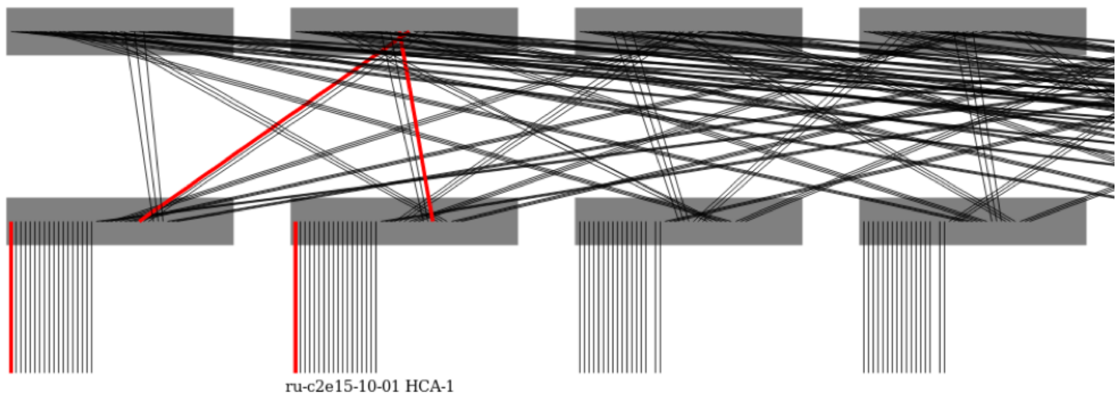


Figure 7.1: Topology of the cDAQ system.

7.1.2 DAQ2VAL Development System

DAQ2VAL is a system used for development and testing purposes. It consists of 4 RUs, 4 BUs and 15 dvRUBUs on a single Infiniband FDR switch. The main system specifications of the dvRUBUs correspond to the one of the RUs as listed above, however built by a different vendor and thus show slight differences to the production system in the firmware used. Development and all two node benchmarks have been run on the dvRUBU systems.

7.2 Measurement Strategy

To get more accurate bandwidth measurements, data will be taken using a sampling based approach. In both `nton_mpistreamio` and `nxn_mpistreamio`, the thread polling and processing the completion queue will increase counters for completed send and reveive operations. In intervals of 5 seconds, a sampling thread will read out these counters and calculate the current bandwidth. The measurements will last for multiples of the sampling interval with the first and the last measurement being discarded and the remaining measurements being saved to a file for further processing.

For unidirectional measurements, data will be taken at the receiver only, while bidirectional measurements will measure at all nodes.

All performance measures coming from the unidirectional bandwidth benchmark `nton_mpistreamio` and the bidirectional `nxn_mpistreamio` benchmark are mean values based on the samples of either all receivers (unidirectional) or all nodes (bidirectional).

8 Performance Analysis and Tuning

Having designed and implemented our messaging algorithms, we can now measure the network bandwidth of our solutions and understand if the bandwidth requirements of Chapter 4 can be fulfilled.

8.1 Upper Bandwidth Bounds on Infiniband FDR using Microbenchmarks

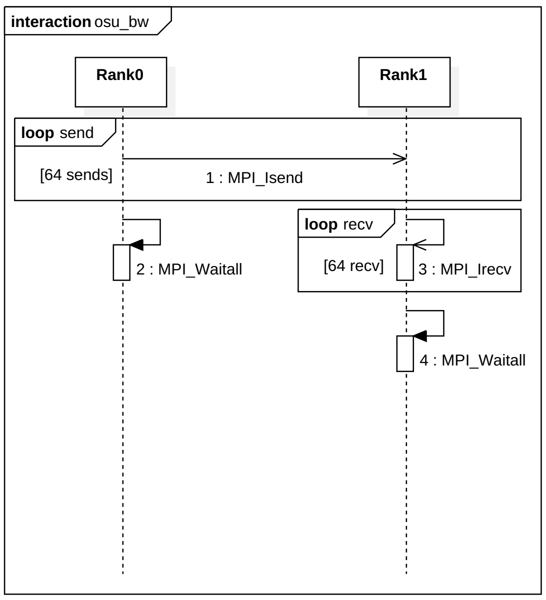
As an entry point for performance engineering, reference measurements are required that mark the upper performance bound on the available hardware. Usually, micro-benchmarks that measure a well defined set of functionality and are easily reproducible are suited for that task.

A suite that focuses on measuring the performance of an MPI implementation on the provided hardware are the OSU Micro-Benchmarks developed by the Ohio State University. Amongst others, the suit includes both uni- and bidirectional MPI bandwidth measurement benchmarks , `osu_bw` and `osu_bibw` measuring the bandwidth between two nodes.

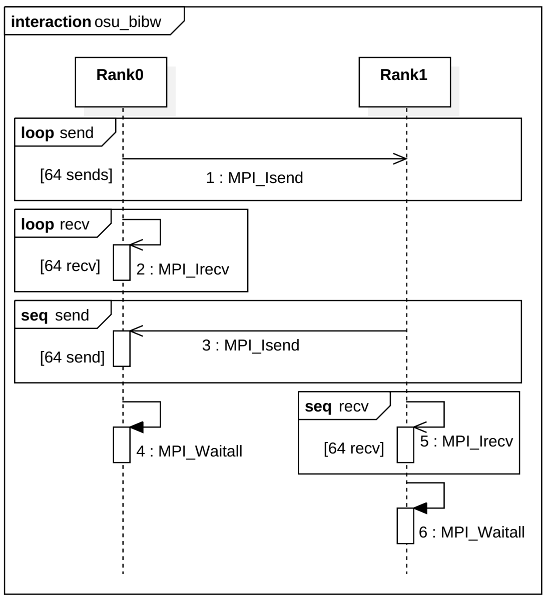
Both benchmarks use MPI point-to-point procedures, issuing 64 non-blocking sends and receives at a time before waiting blockingly for the completion of all operations. After the receiving side completed, it notifies the sender using a control message. This triggers another burst of messages. After gathering sufficient statistics, the mean bandwidth is calculated. The procedure is repeated for all message sizes from 1 Byte to 4MB, doubling the size in each run.

To decouple measurements of network bandwidth from the rest of the system, both benchmarks use the very same message buffer for all sends and for all receives, allowing the network adapter to cache message buffers if possible and prevents memory registration. Thus the benchmarks represent an upper limit for peer-to-peer bandwidth that can be obtained by two nodes.

Using our test environment on DAQ2VAL, we obtain results for `osu_bw`, showing the uni-directinal bandwidth and accumulated bidirectional bandwidth for `osu_bibw` in Figure 8.3. Both benchmarks show a smooth curve, saturating close to the line speed of the Infiniband fabric. Comparing uni- and bidirectional bandwidth we can observe



(a) Pattern of the `osu_bw` benchmark.



(b) Pattern of the `osu_bbiw` benchmark.

Figure 8.1: Communication patterns of the OSU bandwidth benchmarks.

that the accumulated bandwidth is not doubled when using both channels.

8.2 Study of Open MPI Transmission Algorithms

The OSU Micro-Benchmarks also lend themselves very well for studying the different transmission algorithms used by Open MPI. As described in Chapter 3.2.3 MPI uses both eager and rendez-vous protocols for sending data. Furthermore, as mentioned in Chapter 3.2.3, RDMA transmission of data requires regions in virtual memory to be pinned down to a physical address by a registration procedure with the operating system. For small to medium sized messages this results in a latency hit if not done beforehand. Thus copies to bounce-buffers pre-registered by the MPI implementation can increase performance for these messages. With growing message sizes however, the copy overhead will become larger than registration costs.

Normally, a MPI implementation will switch transport algorithms based on message size. The thresholds for switching are determined empirically by implementors to yield reasonable performance for most use cases. This does not necessarily have to result in the best results for a particular application. Thus implementations give users the flexibility to manipulate thresholds at runtime. For our experiments we force the

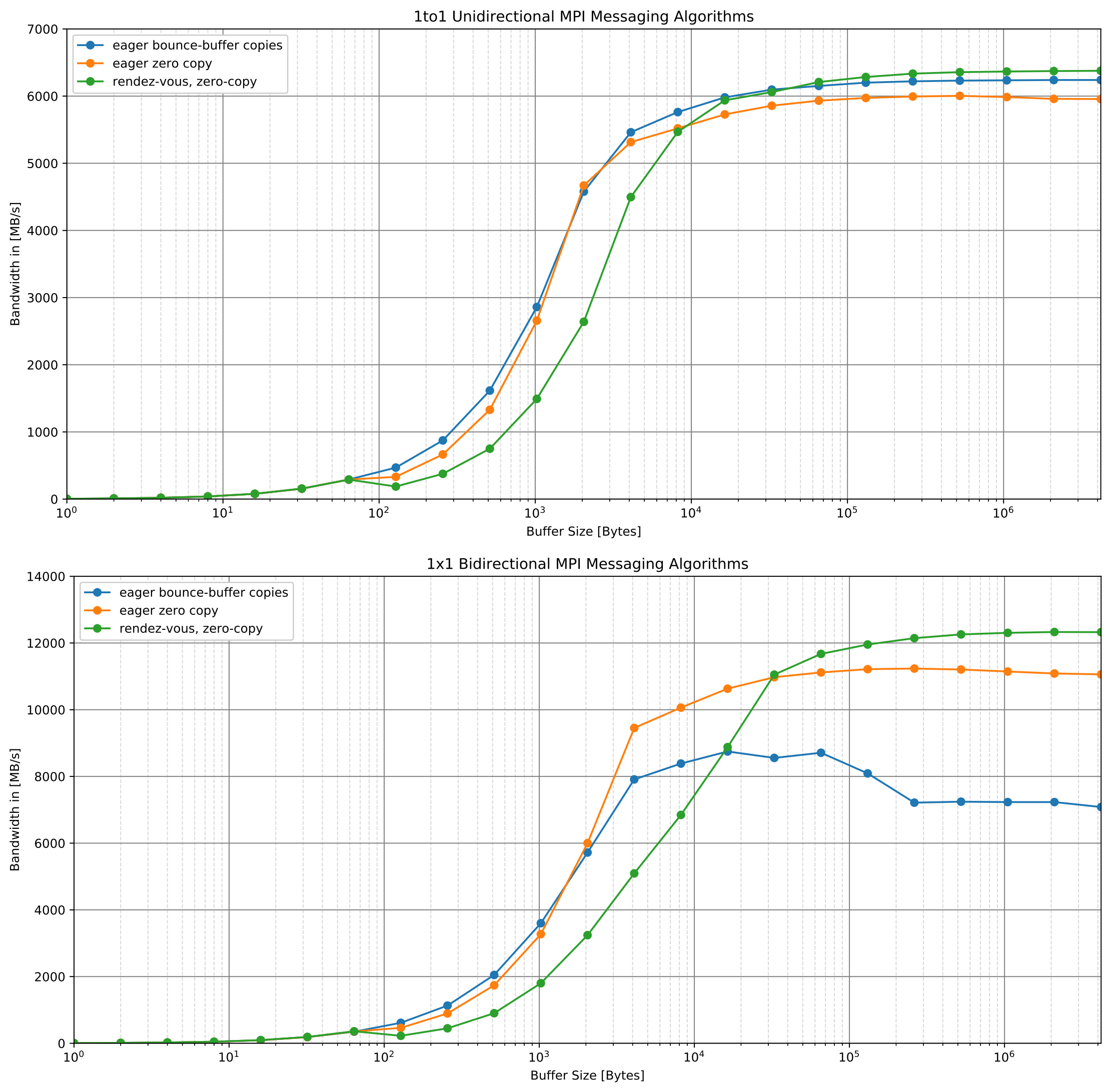


Figure 8.2: Testing MPI messaging algorithms using the OSU bandwidth benchmarks.

Open MPI transport module to stick with one algorithm for all message sizes. For both benchmark programs we conducted three measurements:

1. eager protocol with bounce-buffer copies
2. eager protocol with zero-copy semantics
3. rendez-vous protocol with zero copy semantics

The results are depicted in Figure 8.2. For both benchmarks we can see, that each strategy is optimized for different message sizes. Eager transmission with bounce-buffer copies will mainly perform for small to medium sized messages. We can also observe that bounce-buffer copies on this system will reach their performance limits with bidirectional communication. We are also able to observe the superiority of a rendez-vous protocol for large messages compared to the eager protocol as the costs of network retransmissions increase significantly with message size.

8.3 Study of Thread and Memory Affinity

The last run-time specific setting we want to study using the OSU Micro-Benchmarks is memory and CPU affinity. Both DAQ2VAL and the cDAQ clusters are built from dual socket machines working on a NUMA principle. Each CPU has its own memory controller and is directly connected to the slots for a portion of the available expansion card slots. Components and memory attached to the other CPU can be accessed by a processor interconnect at the cost of a latency hit and bandwidth limited to that of the interconnect. With an attached RDMA NIC, we not only have to ensure that all threads interacting with the same memory buffers are preferably kept on the same CPU, but also ensure that they are assigned to the CPU that is directly connected to the Infiniband card. In our case, we use the Open MPI launcher `mpirun` to pin down processes to the correct socket and the program `numactl` to control the memory allocation policy.

We then repeat both `osu_bw` and `osu_bibw` with an optimized configuration where both processes, memory and network adapter are on the same NUMA domain and one, where we force processes and memory on the other NUMA domain as the NIC. The measurements depicted in Figure 8.3 show a clear degradation in network bandwidth, if the interface adapter has to fetch its memory buffers from a remote NUMA domain.

8.4 Performance Analysis of mpila

The previous measurements based on the OSU Micro-Benchmarks the helped us to understand the hardware and software stack of the systems we are working on and

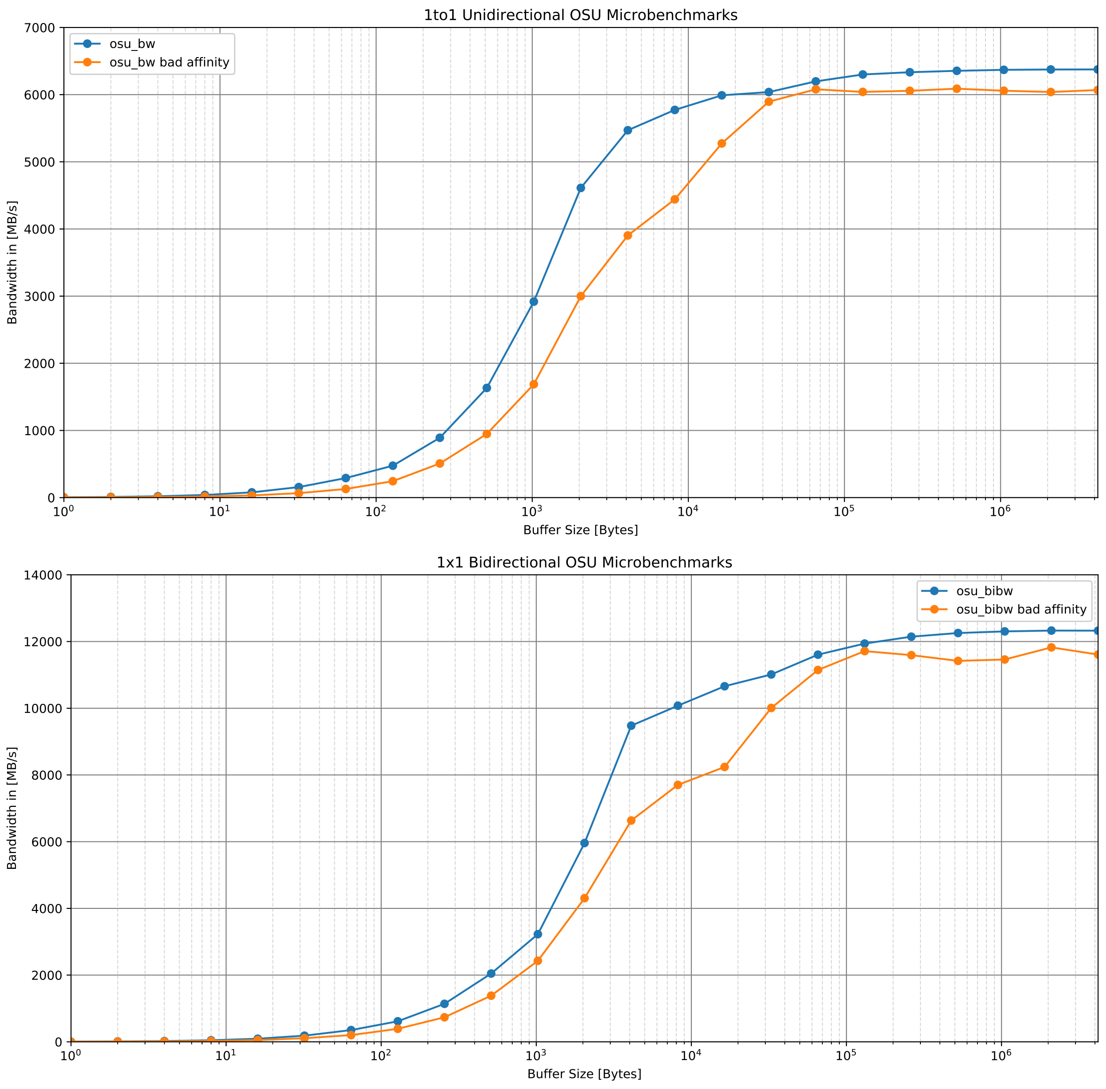


Figure 8.3: OSU bandwidth benchmarks executed with best case and worst case NUMA pinning.

created a point of reference to comprehend the performance of our own application. Using these points of reference we now conduct our own set of experiments that should help us understand the performance of the individual components of mpila and identify where our implementation, design decisions or requirements become performance limiting factors.

8.4.1 Measuring Queuing Overhead

As a first step we need to ensure that the software layer on top of MPI will not have an intolerably large impact on performance.

For this purpose we will run `MPIstreamIO_nton` and `MPIstreamIO_nxn` on two nodes and compare results with `osu_bw` and `osu_bibw`. To ensure we only measure the overhead introduced by the architecture, we modify mpila to get as close as possible to `osu_bw` and `osu_bibw`:

- modify the memory pool to assign the same memory buffer to all `Work Requests` to allow caching of memory segments inside the interface adapter and reduce latencies that might be introduced by memory access as in `osu_bw` and `osu_bibw` .
- modify the transmission algorithms discussed in Chapter 6.4.1 to resemble the ones used in both `osu_bw` and `osu_bibw` (see Chapter 8.1). The non-blocking synchronous sends will be replaced by `MPI_Isend` and the non-blocking polling polling procedures by a blocking `MPI_Waitall`. As in the OSU benchmarks, we pipeline of 64 send and receive operations. The overall structure of `Queue Pairs` and `Completion Queues` is preserved.

The modified applications will be referred to as `osu_nton` and `osu_nxn`. All measurements are taken with optimized thread affinity and manually tweaked transition points between the Open MPI transmission algorithms described in 8.2.

The results are depicted in Figure 8.4. With unidirectional transmission, bandwidth curves deviate only very slightly, indicating extremely low overhead at buffer sizes even bellow the MTU size of 4KB. The bidirectional bandwidth measurements cannot repeat the excellent results of the unidirectional test case. Only for packages above 8KB, the differences in bandwidth become acceptable reaching amortization after 128KB. Investigations showed that mainly components that are required for a multi-purpose messaging service are responsible for the lowered bandwidth. We therefore have to accept the results as an upper bandwidth limit for the sake of fulfilling the given requirements of Chapter 4.

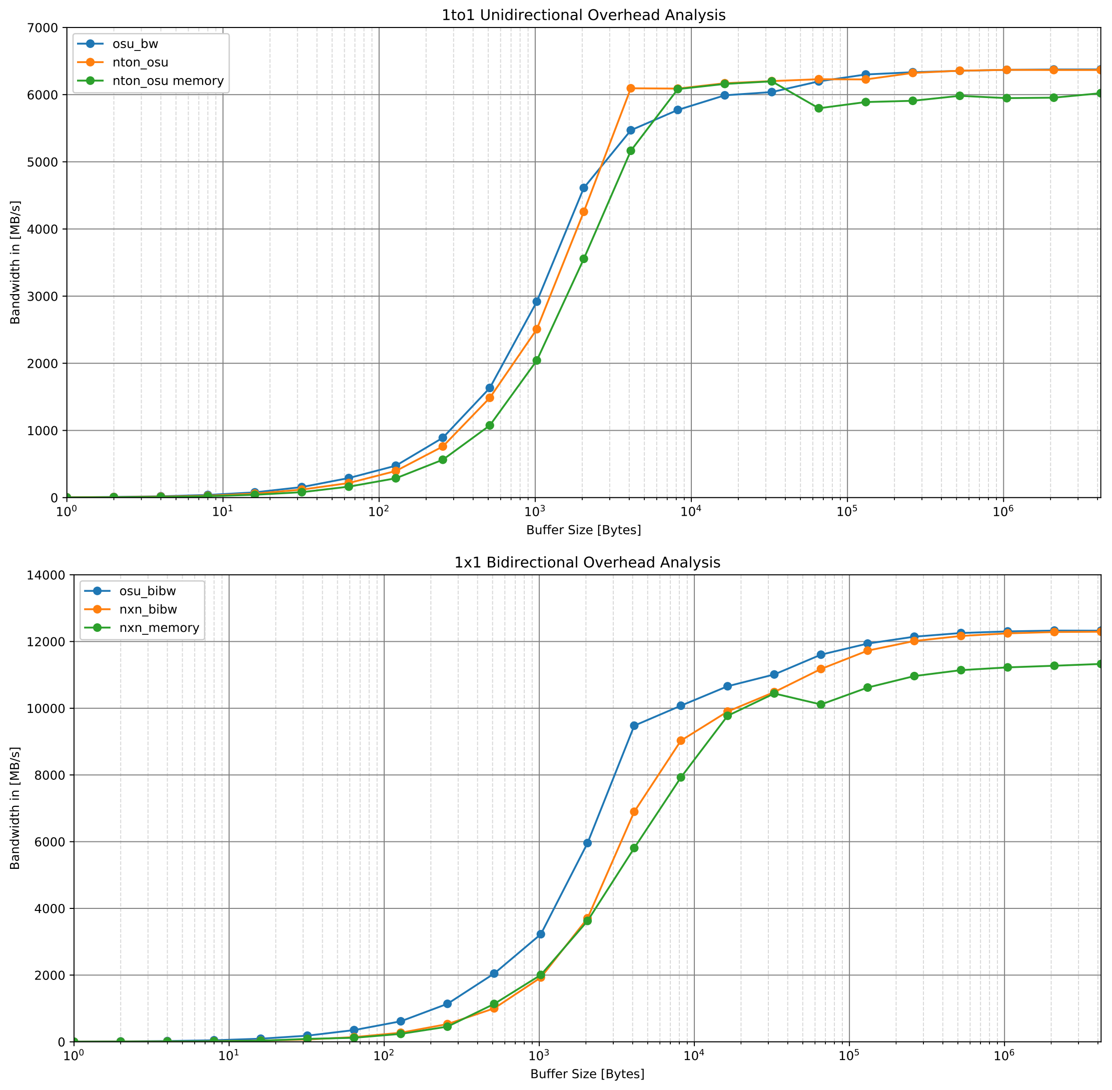


Figure 8.4: Measuring queuing overhead and the effect of memory handling with modified versions of `nton_mpistreamio` and `nxn_mpistreamio`.

8.4.2 Optimizing the Memory Pool

As discussed in 5.2.1, the task of the `Memory Pool` component is to manage buffers used for sending and receiving event fragments. It was introduced for flow control and as a performance optimization to avoid frequent memory registrations.

As described in [25], MPI performs lazy registration of memory buffers. However the MPI standard defines the `MPI_alloc_mem` procedure which allows allocation of *special memory* that can have performance benefits for specific hardware [21, p. 337]. The kind of memory to be allocated is defined by an `MPI_Info` object giving all freedom to the implementors. Open MPI unfortunately does not define any keywords for requesting registered memory for Infiniband or other RDMA interconnects . Without any keywords, there was no measurable performance benefit of using `MPI_alloc_mem`.

Testing different memory layouts, the highest performance was achieved by allocation of memory in a single, contiguous block, aligned with the page boundary. This block is then cut into equally sized buffers. Introducing individual pools for the `Send Queue` and `Receive Queue` within a `Queue Pair`, buffers will circulate in the same order, since messages to the same destination are always ordered, non-overtaking and events are built one after another. Infiniband network adapters thus can exploit this data locality by prefetching.

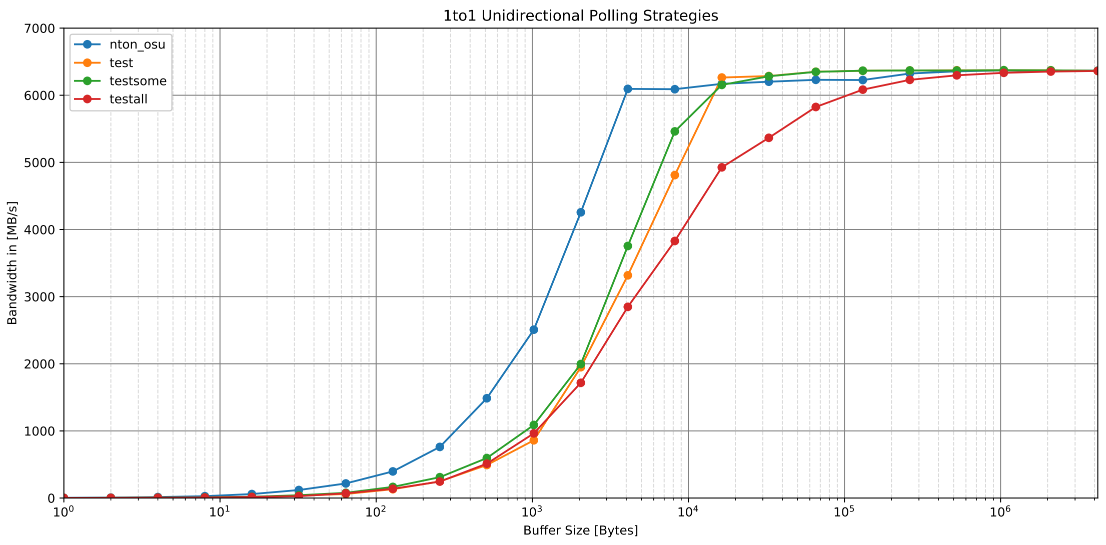
8.4.3 Multiple Memory Buffers

With an optimized memory pool we can quantify the performance impact of using more than one, cached memory buffer. We will run `osu_nton` and `osu_nxn` with single buffer and multiple buffers to obtain a realistic upper performance bound for a functional transport algorithm.

As we can see in Figure 8.4, the addition of memory handling comes at a bandwidth penalty throughout the whole range of memory buffers. Bounce buffer copies can compensate it for small to medium sized messages, however as soon as copies become inefficient, the degradation is clearly visible. Again, bidirectional communication suffers an additional bandwidth penalty.

8.4.4 Benchmarking Polling Strategies

Until now, we have evaluated the impact of different components on the network bandwidth of our messaging service pipeline with a simplified transport algorithm for the sake of comparability with the OSU micro-benchmarks. Using more than two nodes for an asynchronous network transport however cannot be achieved with an algorithm using a blocking MPI wait, see Chapter 6.4.1. We therefore have designed



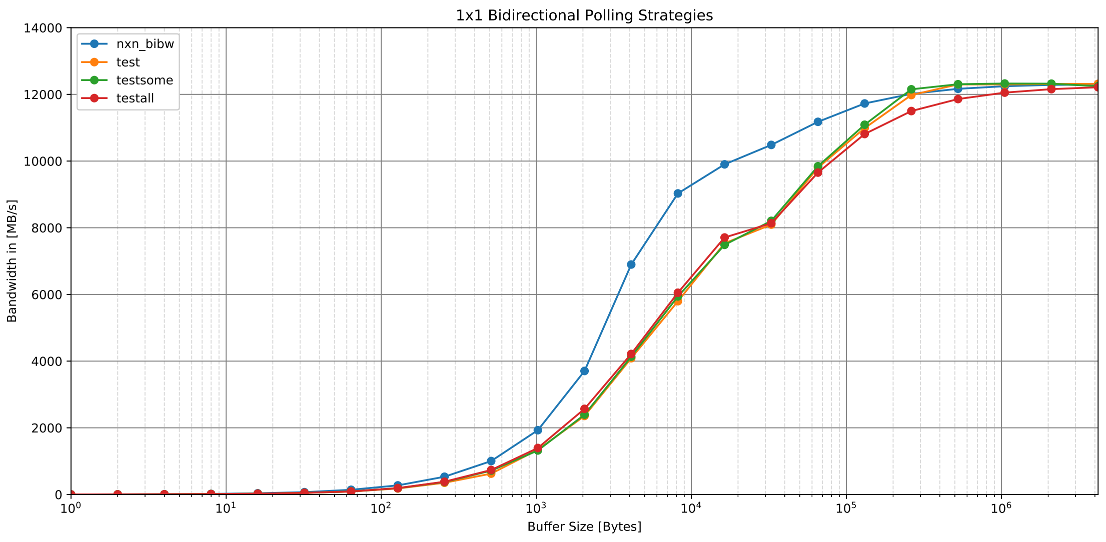


Figure 8.5: Benchmark of polling strategies

.

and implemented transport algorithms based on different polling strategies based on MPI test, which we want to evaluate now.

For these results we will use all components as described Chapter 6.4.3, the transport algorithms based on `MPI_Test`, `MPI_Testsome` and `MPI_Testall` as well as multi buffer memory pools on two nodes and compare the results to the ones obtained in the last experiment in Chapter 8.4.3 using two processes. The pipeline-depth of our polling based transport algorithms has been set to 8 as it generally yielded the best results.

The results are depicted in Figure 8.5. For all polling based algorithms we can clearly observe another degradation in network-bandwidth compared to the MPI wait based algorithms in `osu_nton` and `osu_nxn`. It is difficult to explain the exact circumstances of this performance drop. First of all we want to point out that the pipeline depth from wait to polling based approaches had to be decreased drastically from 64 to 8, as it yielded better performance. For additional measurements see Chapter 8.4.5. Furthermore, profiling runs showed that there is an order of 10 times more executions of polling than actual posting of sends or receives. Increasing the polling intervals using different strategies decreased network-bandwidth even further as it simply increases the latency between MPI completing the operation and our application being made aware of it. Also a mixture of polling and waiting did not show any changes on network bandwidth. As polling is absolutely necessary for a deadlock free algorithm that allows communication between more than two processes, we have to again accept the bandwidth degradation for the sake of designing a an algorithm that is in line with the requirements of Chapter 4.

Comparing the results for unidirectional and bidirectional messaging, we can see that the former clearly favors polling based on `MPI_Testsome` and `MPI_Test` to `MPI_Testall`, while no significant preference of a polling algorithm for the later could be observed. There are two possibilities to interpret these results:

- The overhead of an additional stream of messages that is processed by the same thread may reduce the polling frequency such that strategies that performed better in a unidirectional case are limited to the same baseline. Plausibility of this will be tested by scaling up unidirectional communication to more than two processes.
- Processing both send and receive operations simultaneously may saturate either the MPI library or the network interface adapter to such a level that the full clear-out of one pipeline occurring after a successful `MPI_Testall` does not lead to a penalty where as in the case of unidirectional traffic, both the MPI library and the interface adapter idle until new communication requests are processed. Since however the OSU bandwidth benchmarks use blocking `MPI_Waitall` procedures that clear out the full pipeline of the preposted 64 sends and receives in the

bidirectional case while still outperforming our algorithm this seems unlikely to us. Nevertheless, it is worth to have a look at the sensitivity of the transport algorithms towards the pipeline depth.

8.4.5 Sensitivity of Polling Strategies Towards Pipelining

As discussed in Chapter 6.4.3, our transport algorithms keep multiple sends and receives open at the same time for both `Send Queue` and `Receive Queue`, trying to keep the underlying driver based `Queue Pairs` busy. How many operations are ideally enqueued at the same time is a matter of empirical tests, as both underlying hardware and MPI internal flow control may differ based on the setup.

We therefore conducted a series of measurements ranging from 1 to 64 operation per queue that can be pending at the same time. As results were similar for all polling strategies, we picked the polling strategy based on `MPI_Testsome` as a representative and plotted them against the memory enabled version of `nton_osu` resp. `nxn_osu` for comparison. The results are depicted in Figure 8.6.

For both unidirectional and bidirectional messaging we observe that pipelining is an important measure to increase throughput for most message sizes. The second general trend we can read from the results is that the optimal size of the pipeline depends on the size of the buffers that are being transmitted. With small messages usually come high message rates, that quickly can consume the pipelined messages. On the other side, transmission of large buffers will lower the message rate, requiring less pipelined sends/receives. Especially in the bidirectional case we can clearly see that increasing the pipeline depth can result in significant bandwidth hits. Furthermore results mostly indicate the existence of a bandwidth plateau at message sizes between 16kB and 512kB for unidirectional transport and 64kB to 512kB for bidirectional transport. Large messages at the size of 1MB and above seem to not be processed as efficiently as the smaller ones.

To conclude, we have to recommend to adjust the pipeline depth according to the mean expected message size, since the influence on bandwidth is non-negligible. Nevertheless, a pipeline depth of 8 seems to be a reasonable preset that will yield good performance across the entire range of buffer sizes.

8.5 Performance at Scale

So far we have studied properties of mpila throughly on a small scale using two nodes on the DAQ2VAL system. One of the goals is a messaging service that is capable of delivering VERBS like bandwidths at scale of the CMS event builder. We therefore devised another measurement series for both unidirectional and bidirectional traffic on

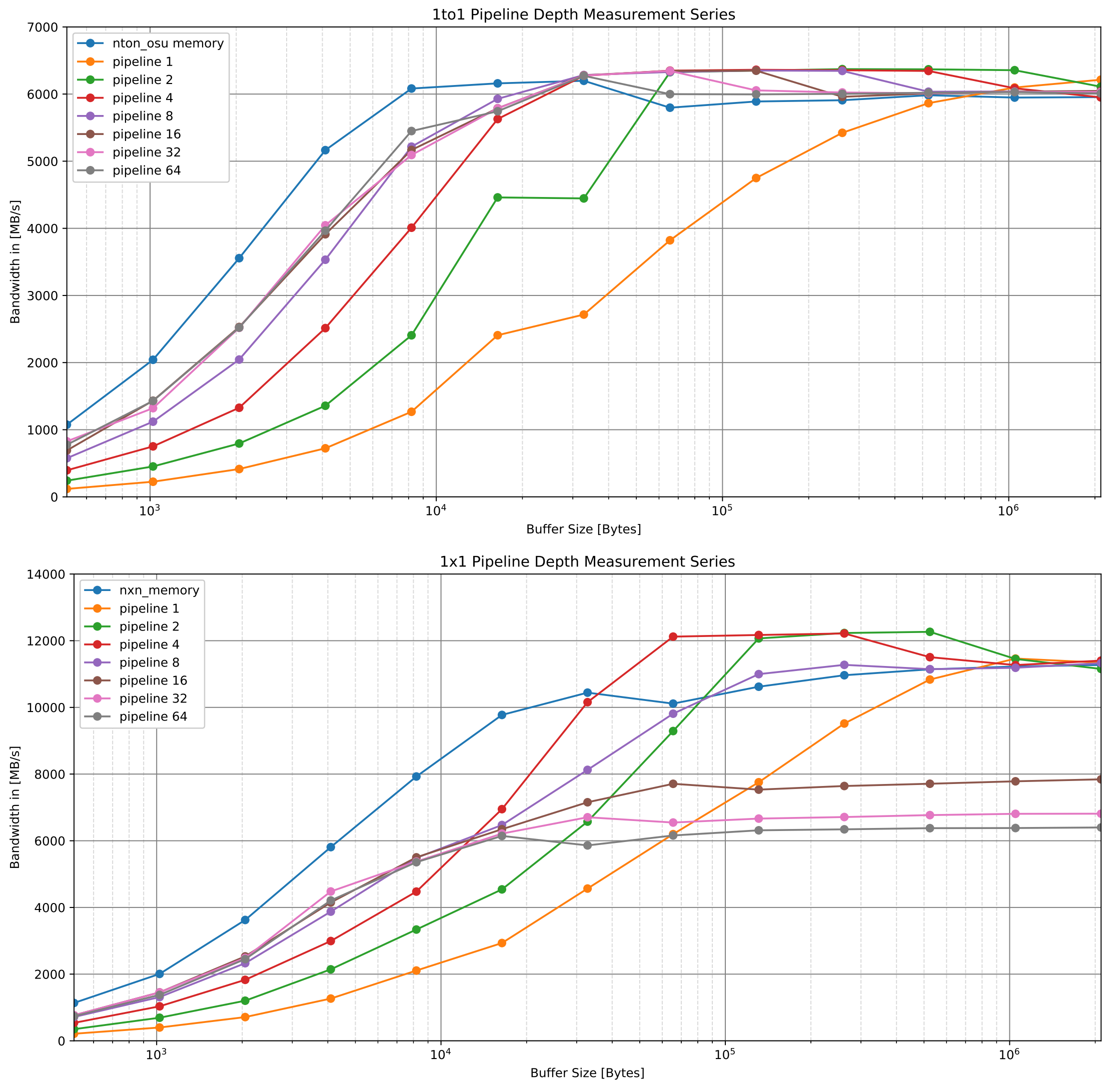
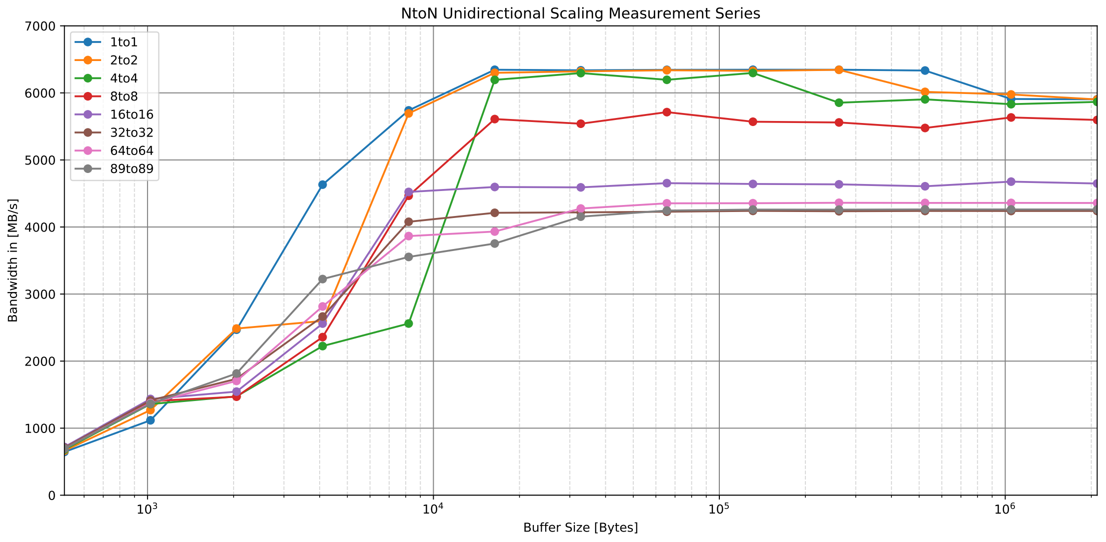


Figure 8.6: Sensitivity of polling strategies towards pipelining.



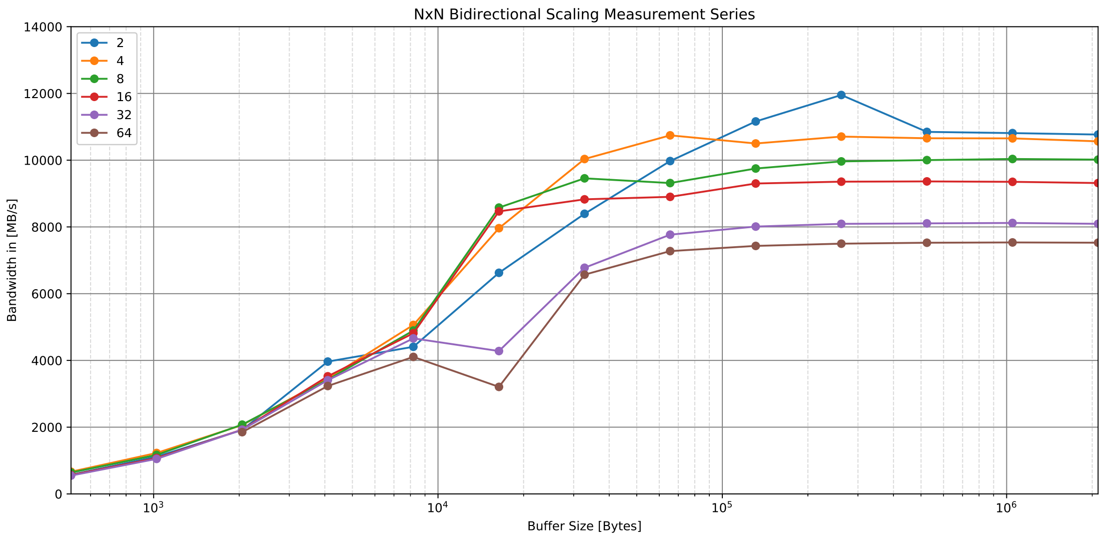


Figure 8.7: Scaling behavior of mpila

Nodes	Leaf Switches
2	1
4	1
8	2
16	3
32	6
64	11
128	12
178	12

Table 8.1: Distribution of nodes onto leaf switches for scaling tests

the cDAQ production system increasing scale from 2 nodes 178 nodes, doubling the amount of nodes each time. Implementing the results from smaller scale measurements on the DAQ2VAL test system, we decided to use a pipeline depth of 8 and the `MPI_Testsome` based polling algorithm for both types of measurements.

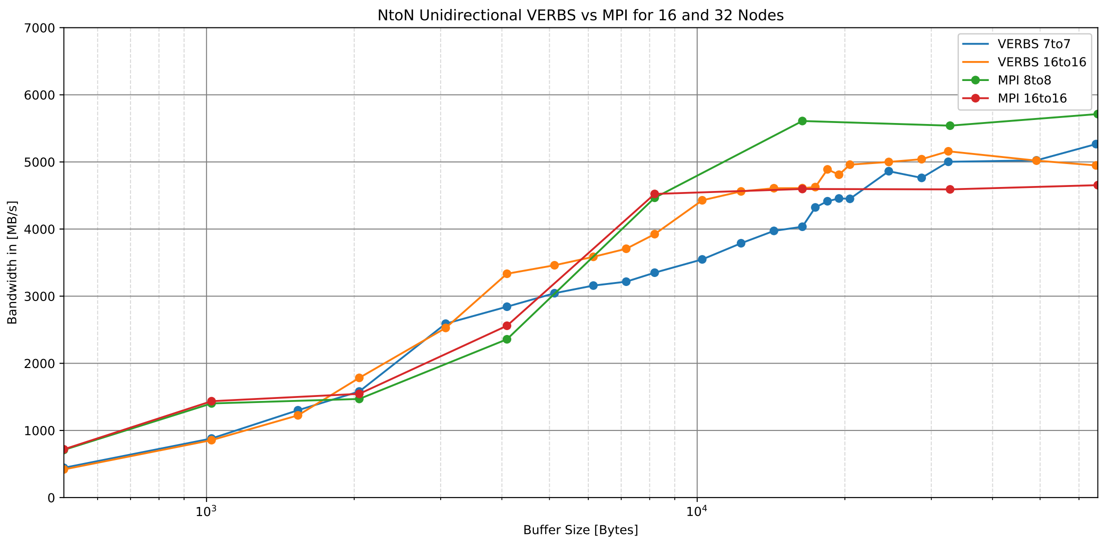
As the cDAQ cluster is a production system that is only available for short periods of time during technical stops of the LHC, we were not able to fine-tune the MPI runtime to a comparable extent as the test system and refrained from adjusting any parameters apart from NUMA pinning. Furthermore we doubled the runtime of the tests to obtain more samples to account for possible larger deviations during the runs.

Unfortunately our measurements were accompanied by technical problems with the cDAQ cluster with the Infiniband Interface adapters. As a consequence that data could in most cases only be taken on BUs as they remained stable during our measurements. This limited data taking to 64 nodes and required us to spread out to multiple switches starting 8 nodes which measurably lowered the achievable performance. Only in a second run we have been able to gather at least measurements for 128 and 178 nodes for the unidirectional case.

8.5.1 Interpretation of Measurement Results

Results of the measurement series are depicted in Figure 8.7. For both unidirectional and bidirectional traffic we can observe curves where bandwidth is increasing with buffer sizes until a plateau is reached. With an increasing amount of nodes, the plateau is reached at smaller buffer sizes, as the link at the receivers can be saturated faster. Furthermore we can see that increasing the amount of nodes will decrease the maximal bandwidth observed at the plateau.

This can be partially explained by bandwidth bottlenecks created by the uplink of switches. The more nodes are in use, the higher the probability of having to



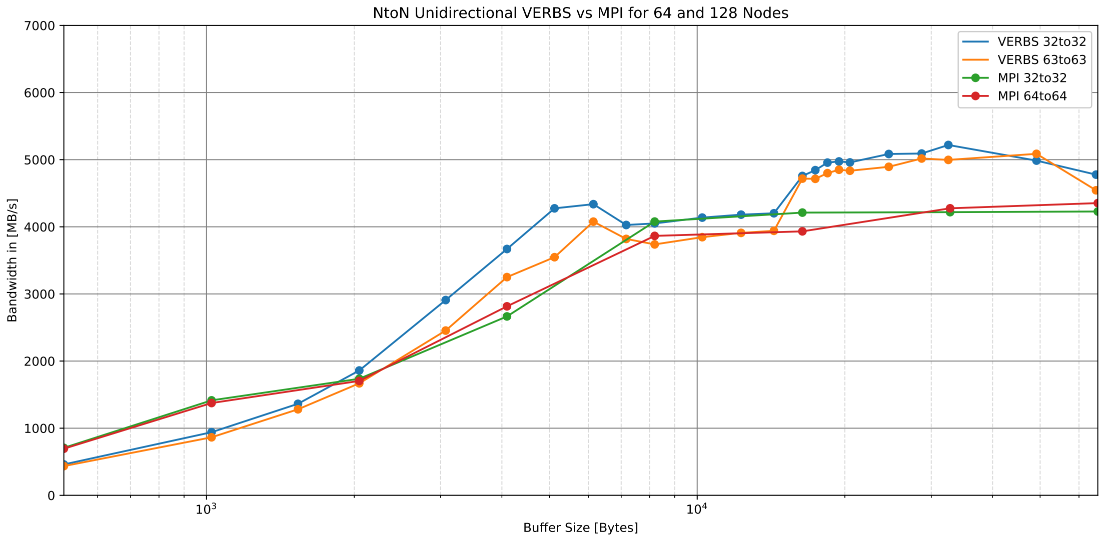


Figure 8.8: Comparing performance of MPI and VERBS algorithms for unidirectional messaging.

communicate over an uplink that has to be shared by multiple nodes. Looking at Table 8.1, it is easy to observe the bandwidth restrictions imposed by the switches. Especially the unidirectional results for 64,128 and 178 nodes saturate at very similar bandwidths having to use 11 respectively 12 leaf switches.

In the unidirectional case, unexpectedly low values can be observed for two measurement points at 2kB for the 2to2 and at 8kB for the 4to4 measurements. These can be explained by inadequate switching from eager bounce-buffer copies to eager zero-copy communication algorithms prematurely by Open MPI. The same holds for the bandwidth drop at 16kB for the bidirectional case where eager zero-copy was wrongly switched to a rendez-vous zero-copy transmission algorithm.

8.5.2 Comparison with VERBS

Our `nton_mpistreamio` benchmark follows the same schematic as `mstreamio`, a XDAQ benchmarking application used for measuring the bandwidth of peer transports in the XDAQ framework. This allows us to compare the measurement series performed by us with a similar series performed by the TriDAS group for the VERBS peer transport during its development. The comparisons are depicted in 8.8.

The bandwidth results are obtained in the same way by sampling and averaging, however the VERBS based data has been recored in a higher buffer size resolution. In the upper plot of Figure 8.8, both the VERBS peer transport and mpila show a very comparable level of performance. We seem to even outperform VERBS with 15 nodes on a single switch with 16 nodes spanning over 3 switches. As node count doubles , the VERBS implementation takes the lead, however we are still confident that our results could be on pair with VERBS, if we did not have to use twice the amount of switches.

If we double the amount of nodes again, we can see that VERBS now outperforms our MPI implementation significantly by up to 1GB/s, while starting to show slight fluctuations in bandwidth. Again a certain performance penalty of the MPI implementation can be explained by having to only use half the switches for 64 nodes, however at 128 nodes, both benchmarks use a comparable amount of switches and the performance differences remain the same.

We believe that the major bandwidth differences between MPI and VERBS in this case can be explained by the additional overhead that is being generated by MPI. With the VERBS implementation, the VERBS `SEND/RECEIVE` point-to-point communication semantics are used and with full control over all driver level queues, the VERBS implementation can ensure that there is always a sufficiently large amount of preposted buffers in the receive queue awaiting new sends. With MPI we switch to a rendez-vous protocol (see Chapter 3.2.3) that largely increases the amount of messages that need to be send back and forth between sender and receiver putting more stress on the interface adapter and possibly congesting the network with control messages instead of sending payload.

Since there is no equivalent bidirectional version of `mstreamio`, we are unfortunately not able to do a comparison with VERBS.

Since MPI implementations are built on top of VERBS in their transport modules, we cannot hope to outperform VERBS in all cases. On the other hand we are happy to see that on smaller sized systems our implementation is able to outperform VERBS and still remains competitive for large scales.

9 Conclusion and Outlook

Throughout this thesis we designed, implemented and evaluated a throughput-oriented transport layer based on MPI for the data acquisition system of the CMS experiment at CERN. After an initial evaluation of MPI procedures, we argued why point-to-point communication is the only messaging paradigm suitable for the high-throughput, fault-tolerant and real-time requirements of Event Building. With a sufficient background on low level RDMA concepts we then proposed a system architecture for a messaging library based on MPI point-to-point procedures that works hand in hand with the queue based model of OFA based RDMA devices. The design was followed by outlining a messaging algorithm and proposing different implementations based on non-blocking point-to-point MPI procedures and a non-blocking polling strategy. The proposed solutions then have been analytically evaluated against MPI bandwidth benchmarks,trying to individually isolate the factors that separate an MPI bandwidth benchmark from a throughput driven messaging service, which can be used universally in MPI applications. Finally we evaluated the MPI implementation against the existing solution on VERBS identifying that our proposed MPI based solution is competitive against VERBS on a smaller scale, capable of outperforming the current messaging service. With increasing scale, the additional overhead of MPI algorithms is no longer able to keep up with a purpose made low level implementation based on VERBS while still maintaining reasonable performance for a high level library.

Furthermore a prototype implementation of a XDAQ peer-transport using our messaging service was implemented. It was able to launch XDAQ applications with an MPI launcher integrating flawlessly into the design principles behind XDAQ.

This experiments proved that MPI in general is capable of high-performance throughput oriented messaging, sufficient for the highly demanding task of event building. With the integration into XDAQ we were also capable of demonstrating the enormous flexibility of XDAQ to adapt to changes in requirements and the ability of MPI to be integrated into different types of applications.

However we have also seen the limitations of MPI. A standard that defines an API and leaves its implementation to other interest groups is able to support a large variety of hardware and optimize really well for those platforms. On the other hand this leads to lack of control from users who will have to not only understand the MPI standard, but also the behavior of the implementation which in many cases is sparsely documented as well. Furthermore the tuning of hundreds of more or less documented runtime options of the MPI implementations and understanding which ones are meaningful for your case can be very time consuming if an application fails to perform well with standard parameters.

The large community around MPI and the profiling tools developed analyze performance of MPI applications were not very helpful when it came to analyze inefficiencies of our library. However parallel MPI debuggers largely contributed to finding errors faster.

In the end we can conclude that while it is possible to build a throughput-oriented messaging service on top of MPI for the event-builder of CMS that performs well enough to be brought to production usage, we always had the feeling to be working against the goals of MPI to provide an easy to use messaging library for distributed programming by constantly trying to get as close as possible to driver level control within our application. If the goal is to design a communication library without having to reach down to driver level, other libraries as open ucx [31] seem to be a better alternative.

On the other hand we feel that the proposed architecture for a messaging library can be a beneficial approach for the growing community around hybrid MPI+X applications relying on both MPI and threading libraries such as OpenMP. By allowing users to exchange data across processes in a multi-threaded environment efficiently via a simple interface without having to access MPI directly, but being able to use its powerful and well grown environment we believe to directly address one of the core issues of hybrid applications.

List of Figures

List of Tables

Bibliography

[1] A. Hermann, L. Belloni, J. Krige, and E. O. for Nuclear Research, *History of CERN: Launching the European Organization for Nuclear Research*, ser. History of CERN. North-Holland Physics Pub., 1987, ISBN: 9780444870377.

[2] *Convention for the establishment of a European organization for nuclear research: Paris, 1st July, 1953 : as amended. Convention pour l'établissement d'une Organisation européenne pour la Recherche nucléaire. Paris, le 1er juillet 1953 : telle qu'elle a été modifiée*. Geneva: CERN, 1971.

[3] (Jan. 2012). About CERN, [Online]. Available: `http://cds.cern.ch/record/1997225`.

[4] M. Krause, *CERN: How We Found the Higgs Boson*. World Scientific, 2014, ISBN: 9789814623469.

[5] L. Evans, "Particle accelerators at cern: From the early days to the lhc and beyond," *Technological Forecasting and Social Change*, vol. 112, pp. 4–12, 2016, ISSN: 0040-1625. DOI: `https://doi.org/10.1016/j.techfore.2016.07.028`.

[6] O. S. Bruening, P. Collier, P. Lebrun, S. Myers, R. Ostojic, J. Poole, and P. Proudlock, *LHC Design Report*, ser. CERN Yellow Reports: Monographs. Geneva: CERN, 2004.

[7] S. Chatrchyan, G. Hmayakyan, V. Khachatryan, *et al.*, "The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment," *JINST*, vol. 3, S08004. 361 p, 2008, Also published by CERN Geneva in 2010.

[8] S. Cittolin, A. Rácz, and P. Sphicas, *CMS The TriDAS Project: Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger. CMS trigger and data-acquisition project*, ser. Technical Design Report CMS. Geneva: CERN, 2002.

[9] J.-M. O. Andre, U. Behrens, J. Branson, P. M. Brummer, O. Chaze, S. Cittolin, C. Contescu, B. G. Craigs, G. L. Darlea, C. Deldicque, Z. Demiragli, M. Dobson, N. Doualot, S. Erhan, J. R. Fulcher, D. Gigi, M. S. Gladki, F. Glege, G. Gomez Ceballos, J. G. Hegeman, A. G. Holzner, M. Janulis, R. Jimenez Estupinan, L. Masetti, F. Meijers, E. Meschi, R. Mommsen, S. Morovic, V. O'Dell, L. Orsini, C. M. E. Paus, P. Petrova, M. Pieri, A. Racz, T. Reis, H. Sakulin, C. Schwick, D. Simelevicius, and P. Zejdl, "Performance of the CMS Event Builder," CERN, Geneva, Tech. Rep. CMS-CR-2017-034. 3, Feb. 2017.

[10] T. Bawej, U. Behrens, J. Branson, O. Chaze, S. Cittolin, G. L. Darlea, C. Deldicque, M. Dobson, A. Dupont, S. Erhan, A. Forrest, D. Gigi, F. Glege, G. Gomez-Ceballos, R. Gomez-Reino, J. Hegeman, A. Holzner, L. Masetti, F. Meijers, E. Meschi, R. K. Mommsen, S. Morovic, C. Nunez-Barranco-Fernandez, V. O'Dell, L. Orsini, C. Paus, A. Petrucci, M. Pieri, A. Racz, H. Sakul, C. Schwick, B. Stieger, K. Sumorok, J. Veverka, and P. Zejdl, "The new cms daq system for run-2 of the lhc," in *2014 19th IEEE-NPSS Real Time Conference*, May 2014, pp. 1–1. DOI: `10.1109/RTC.2014.7097437`.

[11] (Jun. 2018). CMS Online Software project page, [Online]. Available: `https://xdaq.web.cern.ch/`.

[12] J. Gutleber, S. Murray, and L. Orsini, "Towards a homogeneous architecture for high-energy physics data acquisition systems," *Computer Physics Communications*, vol. 153, no. 2, pp. 155–163, 2003, ISSN: 0010-4655. DOI: `https://doi.org/10.1016/S0010-4655(03)00161-9`.

[13] J. Gutleber and L. Orsini, "Software architecture for processing clusters based on i2o," *Cluster Computing*, vol. 5, no. 1, pp. 55–64, Jan. 2002, ISSN: 1573-7543. DOI: `10.1023/A:1012744721976`.

[14] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014, ISBN: 9780133591620.

[15] A. Tanenbaum, *Computer Networks*, 4th. Prentice Hall Professional Technical Reference, 2002, ISBN: 0130661023.

[16] P. Grun, "Introduction to infiniband for end users," *White paper, InfiniBand Trade Association*, 2010.

[17] (Jun. 2018). OpenFabrics Alliance project page, [Online]. Available: `https://www.openfabrics.org/`.

[18] M. Feldman. (Dec. 2017). TOP500 Meanderings: InfiniBand Fends Off Supercomputing Challengers, [Online]. Available: `https://www.top500.org/news/top500-meanderings-infiniband-fends-off-supercomputing-challengers/`.

[19] M. T. Inc, "Why compromise? - a discussion on rdma versus send/receive and the difference between interconnect and application semantics," *White paper*, 2006.

[20] G. Kerr, "Dissecting a small infiniband application using the verbs API," *CoRR*, vol. abs/1105.1827, 2011. arXiv: `1105.1827`.

[21] *MPI: A Message-Passing Interface Standard, Version 3.1.* High Performance Computing Center Stuttgart (HLRS), 2015.

[22] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972, ISSN: 0018-9340. DOI: `10.1109/TC.1972.5009071`.

[23] (Jul. 2018). Open mpi web presence, [Online]. Available: `www.open-mpi.org`.

[24] 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*, Budapest, Hungary, Sep. 2004, pp. 97–104.

[25] J. Squyres. (Jul. 2011). Registered Memory (RMA / RDMA) and MPI implementations, [Online]. Available: `https://blogs.cisco.com/performance/registered-memory-rma-rdma-and-mpi-implementations`.

[26] (Feb. 2017). Understanding Tag Matching for Developers, [Online]. Available: `https://community.mellanox.com/docs/DOC-2781`.

[27] "High performance RDMA protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, Bonn, Germany: Springer-Verlag, Sep. 2006.

[28] *Unified Communication X (UCX) - API Standard*, 1.4, UCF Consortium, Jun. 2018.

[29] (Jul. 2018). Mellanox HPC-X Software Toolkit, [Online]. Available: `www.mellanox.com/products/hpcx/`.

[30] (Jul. 2018). Messaging Accelerator (MXM), [Online]. Available: `www.mellanox.com/products/mxm/`.

[31] (Jul. 2018). Unified communication x - an open-source production grade communication framework for data centric and high-performance applications, [Online]. Available: `http://www.openucx.org/`.

[32] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: An open source framework for HPC network APIs and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43. DOI: `10.1109/HOTI.2015.13`.

[33] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons Ltd., 1996, vol. 1.

[34] H. Zimmermann, "Osi reference model - the iso model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, Apr. 1980, ISSN: 0090-6778. DOI: `10.1109/TCOM.1980.1094702`.