Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Optimization of Intra-Node Communication in HPC Systems: Development and Implementation of a Zero-Copy API

Nils Imhoff

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Dr. Martin Bernreuther, Dr. Christian Simmendinger
Commenced:	February 1, 2024
Completed:	April 30, 2024

Abstract

The landscape of High Performance Computing (HPC) is dynamic and intra-node communication efficiency has emerged as a critical factor in system performance. This thesis presents the Zero-Copy Application Programming Interface (ZCom), which utilizes cross-partition memory (XPMEM) technology to improve data transfer within shared memory environments. As such ZCom has also minimised the data replication which is usual associated with Message Passing Interface (MPI) operations, reducing the communication overhead, hence, leading to an improved computational efficiency.

An extensive performance test with microbenchmarks as well as the MiniGhost benchmark suite shows that ZCom significantly improves communication efficacy especially in weak and strong scaling cases with respect to other MPI-based approaches. The approach taken by ZCom, which facilitates direct memory access among processes represents a paradigm move towards minimized data movement and thus makes it an innovative solution in HPC communications.

The potential of ZCom is clear from the performance improvements observed; however, this thesis also identifies the current deficiencies with the evaluation of ZCom which is performed with a small set of applications and benchmarks. This fact highlights the need for more studies aimed at the generalization of ZCom and its influence on various HPC systems and architectures. The said work sets a very solid ground for future development that intends to optimize the performance and scalability of intra-node communication in HPC environments.

Das HPC-Umfeld ist dynamisch und die Kommunikationseffizienz innerhalb der Knoten ist zu einem entscheidenden Faktor für die Systemleistung geworden. Die Arbeit präsentiert die Zero-Copy Application Programming Interface (API) ZCom, welche auf der XPMEM-Technologie basiert, um den Datentransfer in Shared-Memory-Umgebungen zu optimieren. Deshalb minimierte ZCom auch die Datenreplikation, die normalerweise mit MPI-Operationen verbunden ist und dadurch wurde der Kommunikationsaufwand reduziert und die Recheneffizienz verbessert. Um die Kommunikationseffizienz in schwachen und starken Skalierungsfällen im Vergleich zu anderen MPI-basierten Ansätzen deutlich zu verbessern, zeigen umfangreiche Leistungstests mit Mikrobenchmarks sowie der MiniGhost-Benchmark-Suite, dass ZCom ausgezeichnet ist. Der Ansatz, den ZCom verfolgt, ermöglicht Prozessen einen direkten Zugriff auf Speicher und führt zu einem Paradigmenwechsel hin zur Minimierung der Datenbewegung. Deshalb ist es eine innovative Lösung für die HPC-Kommunikation. Durch die beobachteten Verbesserungen in der Leistung wird das Potenzial von ZCom deutlich; jedoch werden auch aktuelle Mängel bei der Bewertung von ZCom identifiziert, da nur eine geringe Anzahl von Anwendungen und Benchmarks durchgeführt wurde. Es ist erforderlich, weitere Studien durchzuführen, um die Verallgemeinerung von ZCom und dessen Auswirkungen auf verschiedene HPC-Systeme und -Architekturen zu untersuchen. Die Arbeit, legt eine sehr Grundlage für künftige Entwicklungen dar, welche darauf abzielen, die Leistung und Skalierbarkeit der internen Knotenkommunikation in HPC-Umgebungen zu optimieren.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface. 4

BSP Bulk Synchronous Parallel. 55

ccNUMA cache-coherent Non-Uniform Memory Access. 25

CMA Cross Memory Attach. 27

HPC High Performance Computing. 3

IPC Inter-process communication. 37

KNEM Kernel-assisted Mechanism. 27

LiMIC Linux Memory Interconnect. 27

MESI Modified, Exclusive, Shared, Invalid. 25

MESIF Modified, Exclusive, Shared, Invalid, Forward. 26

MOESI Modified, Owner, Exclusive, Shared, Invalid. 26

MPI Message Passing Interface. 3

PiP Process-in-Process. 9

UCT UCX Communication Transport. 28

UCX Unified Communication X. 27

UMA Uniform Memory Access. 25

XPMEM cross-partition memory. 3

1 Introduction

The area of HPC is going through a major change with the introduction of advanced GPU architectures into HPC systems. This modification has improved node performance greatly and increased computational capability to a huge extent. Though the internal communication structure of such systems has improved significantly, these systems do not have a proper internal communication infrastructure which is a major bottleneck in overall system performance.

MPI continues to be the main communication standard in HPC systems. However, the traditional implementations of MPI are not very well-suited for multi-core and many-core systems of today. The conventional method consists of multiple-copying of data through all the system, OS, and hardware levels, which results in delays and wastes too much CPU resource. This problem becomes more severe in the systems having advanced multicore CPUs and GPUs, as the gap between computational power and data movement speed results in huge bottlenecks.

On the other hand, XPMEM resolves the problem of memory sharing between processes, which makes data transfers more effective in shared memory environments. A process can map to the memory buffer of a remote process, what allows to access directly the remote memory, avoids data duplication. This way eliminates the overhead on data transfers and gets rid of the delays.

This thesis aims to enhance inter-process communication efficacy in shared memory settings by developing and implementing a zero-copy API based on XPMEM technology. Stated differently, this API is designed to seamlessly integrate with the existing MPI software, facilitating a smooth transition to the zero-copy procedure. The efficiency of the API and any potential implications on HPC systems will be clarified by evaluating its performance in terms of microbenchmarks and real-world applications.

Outline

The thesis is structured as follows

Chapter 2 – Background: Examines the common methods for message passing and shared memory mapping.

Chapter 3 – Related Work: Provides an overview of the literature pertaining to the themes addressed in this thesis.

Chapter 4 – Designing a zero copy shared memory API: Presents the design and implementation of the Zero-Copy API.

Chapter 5 – Performance Evaluation: Discusses the results of the implementation and the performance of the Zero-Copy API.

Chapter 6 – Conclusion: Summarizes the thesis and offers a perspective on future research.

2 Background

The chapter offers a comprehensive summary of the key technical aspects of prevalent message forwarding and shared memory mapping techniques. The chapter is partitioned into two segments. The initial segment presents a comprehensive summary of the fundamental technical aspects of prevalent message passing techniques. The second section presents a comprehensive summary of the key technical aspects of shared memory mapping techniques.

2.1 Multiprocessors

Multiprocessor systems utilize a divided memory architecture and employ hardware or software locks to regulate memory access. In these systems, communication commonly takes place through shared memory. Distributed memory multiprocessors possess a distinctive structure in which each processor possesses its own cache and address space, and communication occurs through remote memory operations. These systems are extensively utilized in contemporary HPC contexts because of their cost-efficiency [HB11].

2.2 Distributed Systems

A distributed computer connects several individual computers over a network to solve a problem. The connected nodes (computers) share the computation. The nodes use message passing for communication. Unlike parallel computers, distributed computers do not share a storage pool [TV17].

2.3 Message Passing

Message passing is a method used by parallel processes to communicate and synchronize with each other. A message passing architecture is a system of communication primitives and synchronization primitives. These primitives serve as a pictorial representation of the fundamental parts of the underlying hardware. Memory in systems that have shared memory communication primitives act as an interface for data exchange between processes. Data transference in distributed memory systems is made possible by communication primitives that utilize the Remote Memory's Get and Put operators. The migration of a software to another multiprocessor architecture usually involves only compilation. [SOH+98].

On the contrary, distributed memory system rely on an alternative method. Here, each process has its own private memory, and communication takes place through explicit message passing. Messages are transmitted and received by processes using separate operations that are usually named as “remote memory Get and Put operations”. Memory-to-memory data transfers between processes, across the boundaries of local memories are also performed through such operations. These means involve shifting the information stored in the memory of one process to another, across the limit of local memories. This complexity of the communication architecture is intrinsic, since it does require deliberate coordination and data exchange protocols and makes it unlike shared memory systems. Yet, it has benefits in scalability and is particularly suitable for systems of large-scale, distributed computing where processes are distributed between several physical machines.

One of the main advantages of message conveyance is flexibility. It allows for a smooth migration of applications between various multiprocessor architectures with minimal changes. In the realm of HPC efficient functioning of applications with many hardware configurations is a necessity. Specifically, message passing is a form of inter-process communication that abstracts the concrete characteristics of the hardware that is being used. It provides a portable and effective way for processes to communicate with one another even if they are using shared or distributed memory systems.

2.4 Message Passing Interface (MPI)

The MPI-Forum was founded in 1992 after a workshop titled SStandards for Message Passing in Distributed Memory Environments,"which was organized by the Center for Research in Parallel Computation. The forum originally consisted of eighty attendees from forty organizations [Wal92]. The objective of this initiative is to create a widely agreed and publicly accessible standard for exchanging messages. The first MPI-1 standard was issued by the MPI-Forum in August 1994. This standard covers a wide range of topics, such as point-to-point communication, collective group operations, process groups, communication domains, process topologies, environment management, and contains bindings for Fortran and C. The document does not include detailed information on how to carry out the implementation. Instead, it expects the implementers to modify the standard according to their particular hardware and consider its distinctive characteristics [For94].

The introduction of this standard resulted in the development of several implementations, including MPICH, created by Argonne National Laboratory [MPICH], and Open MPI [Open MPI], which is a collaborative project involving FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory, and LAM/MPI from Indiana University, as well as contributions from the PACX-MPI team at the University of Stuttgart. The MPI-1.1 standard, which aimed to resolve uncertainties and mistakes in the MPI-1.0 specification, was published in 1995 [MPI-1.1-95]. Subsequently, in 1997, the MPI-2 standard was created. This standard included functionalities such as dynamic process management, input/output operations, one-sided communications, and C++ bindings [MPI-2-97].

The MPI-3.0 standard was finalized by the MPI-Forum in 2012 [MPI-3-12]. This version presented issues about backward compatibility for the first time. The update improved the ability to communicate in one direction using shared memory models and added non-blocking functions and neighborhood collectives to the collective communication options, which are designed for certain process topologies. Additionally, it added the MPI_Count data type, which is used for large contiguous derived data types. It also included Fortran 2008 bindings, replacing the previous C++ bindings with C bindings. Several sophisticated features of MPI-2 were marked as obsolete in the MPI-3.0 release.

The MPI-3.1 standard was completed in 2015, including small improvements and changes to MPI-3.0 [MPI-3.1-15]. The standardization of MPI-4.0 was finalized and made available in June 2021 [MPI-4-21].

Presently, a substantial amount of parallel applications rely extensively on the MPI Continual progress in the HPC industry is expected to lead to improvements in MPI designs and parallel programming paradigms, which will further advance and optimize HPC applications.

2.4.1 Basic concepts of MPI

MPI is grounded in the concept that parallel processes operate within separate address spaces, necessitating explicit communication for data transfer. The core of this communication is a bilateral operation comprising a 'send' action from the source process and a 'receive' action by the destination process [SOH+98].

Sending and Receiving: In MPI, the sending process needs to specify the data it intends to transmit, including its start address and length (usually in bytes), and identify the target process. Conversely, the receiving process must be prepared with a local memory area to accommodate incoming data. It also needs information about the size of this area and the identity of the sending process. This mutual understanding ensures seamless and efficient data transfer between processes [PF12].

Message Matching: Another level of functionality in MPI is message 'matching'. Utilizing an integer identifier or label is one method for doing this. This guarantees that the delivery of a message with a matching label is the sole condition for a receive operation to be deemed successful. For this reason, this label has to be included as a parameter in the operations of both the sender and the recipient. Furthermore, it might be advantageous in a receive operation to explicitly indicate the sender [PF12]. Hence, the protocols for transmitting and receiving data are often specified as `send(address, length, destination, tag)` and `receive(address, length, source, tag, actlen)`. The variables ßourceänd "tag"may be used to filter the incoming messages or as placeholders to accept messages from any sender or with any label. The field äctlen"represents the actual length of the received message [PF12].

Message Buffer: An MPI message buffer is defined by a triplet (`address, count, datatype`), which specifies the beginning address, the number of elements (count), and the datatype of the message. This architecture enables adaptability and accuracy in specifying the content and dimensions of the message to be sent or received [SOH+98].

Process Designation: MPI operates using the concept of process groups. Each process inside a group is assigned a distinct identifier known as a process number or rank. By default, all processes in a MPI program are part of a main group and are assigned sequential numbers inside this group [PF12].

Communicators: In MPI, a communicator is a combination of the context and group ideas, defining the communication environment for processes. Communicators have a crucial role in many MPI activities, including both individual and group communications. The parameters ”destination” or ”source” in these operations always correspond to the process number inside the group connected with the particular communicator [SOH+98].

The evolution of MPI, from its initial version MPI-1 to the latest MPI-4.1, mirrors its adaptability and responsiveness to the dynamic landscape of HPC. This evolution reflects in the way MPI has influenced parallel programming paradigms and architectures [SOH+98]. Importantly, it also impacts the implementation and efficiency of advanced technologies like the Zero-Copy API, which is pivotal in optimizing data transfer methods in HPC systems. MPI's widespread adoption in parallel computing and its continuous development remain essential to the progress and efficiency of data management and communication in HPC environments [SOH+98].

2.4.2 Two-Sided Communication in MPI

Two-sided communication is a fundamental concept in the MPI that underpins many parallel computing operations. The sender and the recipient are two processes that explicitly interact in this communication model, which is also referred to as point-to-point communication. This contrasts with one-sided communication, where one process can access or modify the data of another without the explicit involvement of the second process [PF12].

Basic Mechanism: In two-sided communication, the process initiating the communication is known as the sender, and the process receiving the data is the receiver. The sender explicitly specifies the destination process for the message, while the receiver explicitly specifies the source from which it expects to receive a message (see Figure 2.1). This paradigm necessitates the active involvement of both the sender and the recipient in the process of communication [PF12].

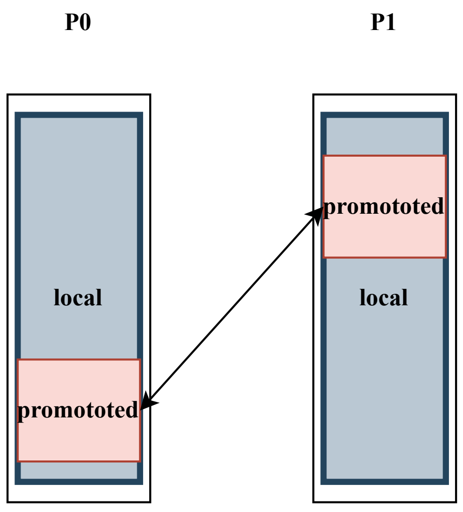

Figure 2.1: Two-Sided Communication in MPI

Send Operations: MPI provides various send operations, each catering to different requirements and scenarios. These include [PF12]:

- `MPI_Send`: A standard blocking send operation. The sender may block until the MPI system has enough information to ensure that the message can be sent. It does not necessarily mean the message is synchronously sent, as the operation may complete before the receiver has started to receive the message.
- `MPI_Bsend`: A buffered send operation that returns immediately, allowing the sender to reuse the buffer. Although the message is held in an internal buffer until it is ready for reception, this does not ensure that it was sent at the time of return.
- `MPI_Ssend`: A synchronous send where the sender blocks until the receiver starts receiving the message, ensuring that both parties are synchronized in the communication process.
- `MPI_Isend`: An immediate, non-blocking send operation in which the transmission process commences before the recipient begins receiving the message, and the function returns instantaneously.
- `MPI_Ibsend`: An immediate, non-blocking buffered send, combining the characteristics of `MPI_Bsend` with the non-blocking nature of `MPI_Isend`.
- `MPI_Issend`: An immediate, non-blocking synchronous send, allowing the sender to proceed without waiting for the receiver to begin receiving, yet ensuring synchronization as in `MPI_Ssend`.
- `MPI_Rsend`: A ready send operation where the sender assumes the receiver is ready to receive the message immediately. This requires careful coordination as the receiver must be prepared to handle the incoming message.
- `MPI_Sendrecv`: A combined operation that allows simultaneous send and receive actions, facilitating direct communication exchanges between a pair of processes.
- `MPI_Send_init`: A send operation that is initialized but not started, designed for later execution, providing flexibility in managing the communication timeline.

Receive Operations: Similarly, MPI provides various receive operations [SOH+98]:

- `MPI_Recv`: A standard receive operation, blocking until the specified message is received.
- `MPI_Irecv`: An immediate (or non-blocking) receive, where the function returns immediately, and the actual receive happens asynchronously.
- `MPI_Recv_init`: A receive operation that is initialized but not started, allowing for later execution.

Matching Send and Receive: A critical aspect of two-sided communication is the matching of send and receive operations. For a successful data transfer, the sender's and receiver's tags, communicators, and message sizes must match. This ensures that messages are correctly routed to their intended destinations and that the right messages are received by each process [SOH+98].

Synchronization: Two-sided communication inherently provides synchronization mechanisms. When a process performs a blocking send or receive operation, it waits for the corresponding receive or send operation to start or complete. This synchronization can be crucial for ensuring data integrity and proper sequencing of operations in parallel applications [SOH+98].

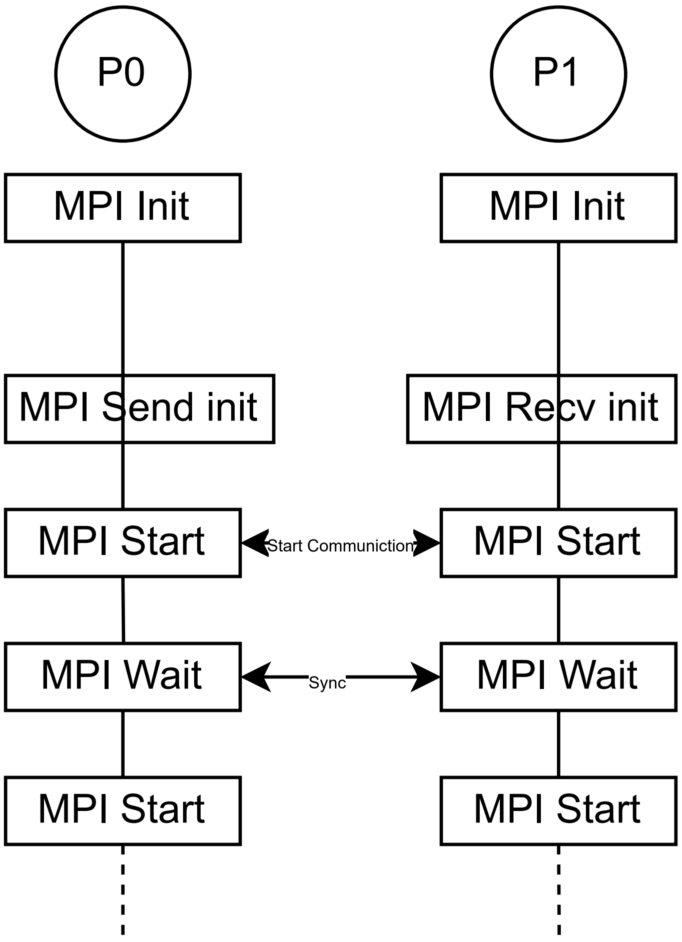

Figure 2.2: Persistent Communication in MPI

Use Cases: Two-sided communication is widely used in scenarios where communication patterns are well-defined and predictable. It is particularly suited for algorithms where processes need to exchange data in a coordinated manner, such as in matrix multiplication, sorting algorithms, and other data-parallel tasks [SOH+98].

In summary, two-sided communication in MPI is a powerful and versatile model for data exchange in parallel computing. It provides a clear and structured approach to message passing, with various options for sending and receiving operations to cater to different needs and scenarios.

2.4.3 Persistent Communication

Persistent Communication Requests in MPI framework is designed to optimize and facilitate repeated communication patterns among processes. It achieves efficiency by allowing the setup of communication parameters to be done once and reused multiple times.

Persistent Communication Requests in MPI is centered around the concept of persistent requests. These requests encapsulate all the necessary information for a communication operation, such as the sender and receiver information, the buffer location, and the data size and type. Here is a detailed breakdown of the mechanisms [Mes23]:

1. **Initialization**: Persistent requests are initialized using specific MPI functions that correspond to different types of communication operations. For example, `MPI_Send_init` and `MPI_Recv_init` are used to initialize persistent requests for send and receive operations, respectively. During this phase, all the parameters required for the communication are specified, including the source and destination ranks, the communication buffer, and the message tag.
2. **Activation**: Once a persistent request is initialized, it is not immediately active. It needs to be activated using the `MPI_Start` or `MPI_Startall` function. This step signals MPI to begin the communication operation based on the parameters defined during the initialization phase. Importantly, a persistent request can be activated multiple times, allowing for repeated communication without re-specifying the communication parameters.
3. **Completion Checking**: Similar to other MPI communication operations, the status of a persistent communication request operation may be verified using test or wait functions, such as `MPI_Test`, `MPI_Wait`, and their variations. Before accessing or altering the data in the buffer, these procedures ensure that the communication has been completed. .
4. **Deallocation**: Once a persistent request is no longer necessary, it should be released using the `MPI_Request_free` function. It is essential to do this step in order to prevent resource leaks and guarantee that MPI can recover any resources linked to the request.

The primary benefit of persistent communication requests lies in its efficiency and performance optimization for applications with repetitive communication patterns. By reducing the overhead associated with repeatedly specifying communication parameters, applications can achieve significant performance improvements. Additionally, persistent communication requests simplifies the implementation of complex communication patterns, making code easier to read and maintain.

In summary, persistent communication requests in MPI provides an efficient mechanism for managing repeated communication operations, offering both performance improvements and coding simplicity for suitable applications.

2.4.4 One-sided Communication

One-sided communication represents a significant advancement in MPI, introducing a Remote Memory Access (RMA) model. This communication paradigm enables a process to directly access memory segments of another process for reading, writing, or updating, without requiring active participation from the target process. For these remote accesses to be effective, they must be coupled with appropriate synchronization operations. This model is applicable to both shared-memory multiprocessors and distributed memory systems [TRH00].

In MPI's one-sided communication, the initiating process, known as the source, solely undertakes RMA operations, supplying all necessary parameters for the communication. This contrasts with two-sided, point-to-point exchanges where both processes are actively involved in the communication. The target process, whose memory is accessed remotely, plays a passive role in this setup [TRH00].

Explicit synchronization is required in one-sided communication, allowing for optimization by spreading the overhead across multiple operations. Processes must allocate a portion of their memory, termed 'window memory', for access by other processes. MPI facilitates this through communication windows, optimizing the placement of window memory in shared memory systems for efficient access [TRH00].

2.5 Remote Memory Access

In the context of MPI, the memory allocated for remote access by a process is structured as a 'window'. The properties of each window include the location of the memory segment (whether in local or global memory), its size, and a displacement unit that aids in calculating offsets inside the window [OS1].

Global memory for remote access is often allocated using the `MPI_Alloc_mem` function, whereas local memory allocation may be done using normal allocation techniques. A group of memory windows linked to a MPI communicator forms a window object, which is created using the `MPI_Win_create` function. This object contains information on the specific windows, their current locking state, and the group of processes that are permitted to access them [OS1].

One-sided communication within MPI is organized into phases referred to as epochs, specifically termed exposure epochs or access epochs [OS2]. In an exposure epoch, a process makes its window accessible to one or several other processes. Conversely, during an access epoch, a process may access windows that others have made available. The transition between these epochs is regulated by synchronization operations, with MPI offering three main types of synchronization: Fence, Dedicated synchronization, and Locks [OS2].

2.5.1 Fence

The Fence action serves as a synchronized mechanism for all processes linked to a window object. It begins a window sharing period that is started by the `MPI_Win_fence` function so that the processes can see each other's windows. The period is finished by performing another Fence operation, which guarantees the completion of all RMA procedures [JRS16]. This procedure is especially valuable in situations when processes interact with numerous windows, use `MPI_Allreduce` to determine the number of operations directed at their window, while local synchronization verifies the completion and coherence of the operations [OS2].

2.5.2 Dedicated Synchronization

This synchronization strategy is designed to target a certain group of processes. The process initiates an exposure epoch by using the `MPI_Win_post` function. This allows other processes to access its memory during this phase using the `MPI_Win_start` function. The actions of the other processes may be finalized using the `MPI_Win_complete` function. Prior to progressing, it is crucial to fulfill any Return Merchandise Authorization (RMA) requests that are aimed at a certain timeframe [Mes23].

2.5.3 Locks

Locks offer a passive target synchronization method. A remote process can open an access epoch with an individual or shared lock, and it can be closed with an unlock call. This ensures the completion of all RMA requests in both the source and target. Locks are granted based on the current status of the target window, balancing the needs for shared and exclusive access [Mes23].

2.6 Shared Memory

Shared memory is a crucial element in the design of multiprocessor systems, which have a unified physical address space that can be accessed by several CPUs. There are two principal types of architectures distinguished by their memory management techniques: systems with uniform memory access (Uniform Memory Access (UMA)) and systems with cache-coherent non-uniform memory access (cache-coherent Non-Uniform Memory Access (ccNUMA)).

2.6.1 Cache Coherence

In computing architectures where multiple CPU caches are operating simultaneously, cache coherence is essential. In the UMA and ccNUMA systems, it is crucial. Inadequate coherence strategies can lead to data inconsistencies and computational errors because separate processors' caches might not update appropriately in response to actions from other processors. To guarantee that every cache in the system presents the same view of the memory, cache coherence protocols are put into place [Hag11].

A fundamental protocol used for enforcing cache coherence is the Modified, Exclusive, Shared, Invalid (MESI) protocol, which organizes cache lines into four distinct states to reflect their condition within the cache system [Hag11]:

- **Modified**: Indicates that the cache line is solely present in the current cache and has been modified from its original condition in the main memory.
- **Exclusive**: The cache line has not been changed and is only present in the current cache.
- **Shared**: The cache line is found in at least one other cache and remains unmodified.
- **Invalid**: The current cache contains an invalid cache line.

The MESI protocol manages transitions between these states efficiently, ensuring data integrity and minimizing the risk of data staleness. Transitions between these states are triggered by specific actions such as reads, writes, or synchronization activities, which are all monitored by the coherence protocol [Hag11].

Additional protocols like Modified, Owner, Exclusive, Shared, Invalid (MOESI), Modified, Exclusive, Shared, Invalid, Forward (MESIF), and the Illinois protocol, which are tailored to meet specific system requirements and architectural features, introduce further optimizations and improvements. To reduce the load on main memory, the MOESI protocol, for instance, adds a ”Owned” state that permits a cache line to be modified and shared simultaneously. This allows the cache that owns the line to fulfill read requests from other caches [Hag11].

Effective implementation of cache coherence significantly reduces the overhead associated with maintaining consistency across multiple caches, thereby optimizing system performance. This ensures that all processors operate with the most up-to-date data, enhancing the scalability of multiprocessor systems and facilitating efficient data sharing among numerous processors [Hag11].

As computational architectures advance, especially with the proliferation of multi-core and many-core architectures, the role of sophisticated cache coherence protocols becomes increasingly crucial. These protocols are key to optimizing the performance and efficiency of contemporary computing environments, affecting both application throughput and system responsiveness [Hag11].

2.6.2 UMA Systems

The fundamental architecture of UMA systems involves processors interconnected via a common bus, such as the Front-Side Bus (FSB), through which they also access memory. Chipsets, responsible for memory module control and interfacing with other node components, play a vital role in these systems. However, UMA systems face inherent bandwidth limitations as the number of FSBs or sockets increases. To alleviate this, non-blocking crossbar switches providing point-to-point connections between sockets and memory modules are often incorporated [Hag11].

2.6.3 ccNUMA Systems

In ccNUMA architectures, processor cores are grouped into Locality Domains (LDs), each with efficient access to its local memory. These LDs are interconnected, facilitating transparent memory access across different processors. The scalable bandwidth of ccNUMA systems makes them particularly suited for multi-processor configurations [Hag11].

Despite their advantages, ccNUMA systems face challenges such as the localization problem, where performance can be impacted by non-local memory accesses, and potential conflicts arising from concurrent memory accesses by processors from different LDs. Addressing these issues involves careful consideration of data access patterns and limiting each processor's data access predominantly to its respective LD [Hag11].

In summary, shared memory architectures, encompassing both UMA and ccNUMA systems, play a pivotal role in modern multiprocessor systems. Their effective utilization hinges on sophisticated cache coherence mechanisms and architectural considerations to ensure efficient, consistent memory access across multiple processors [Hag11].

2.7 Overview of Address Space Mapping Mechanisms

In distributed systems and high-performance computing, memory allocation is critical. Address space mapping techniques improve performance by making it easier for processes to allocate memory efficiently, particularly when sending large amounts of messages back and forth. Linux Memory Interconnect (LiMIC) [JSCP05], Kernel-assisted Mechanism (KNEM) [GM13], and Cross Memory Attach (CMA) [Vie14] are three prominent methods for mapping address spaces with the kernel. Every one of these solutions uses a different operational methodology and has unique properties. Additionally, XPMEM, a technique for sharing memory space between partitions, provides a distinct and different approach.

2.7.1 LiMIC (Linux Memory Interconnect)

LiMIC is designed to efficiently map segments of a communicating process's memory into the kernel space. This approach allows the receiver process to directly copy the sender's pages, which are mapped into the kernel, into its local address space. The primary advantage of LiMIC is its ability to reduce the need for intra-node, large message memory copies, which can be a significant bottleneck in distributed systems. However, one downside of LiMIC is the overhead associated with system calls for each user-level data transfer [JSCP05].

2.7.2 KNEM (Kernel-assisted Mechanism)

KNEM, another kernel-assisted mechanism, enhances direct memory transfers between application processes. While not detailed in the context provided, KNEM operates similarly to LiMIC and CMA by facilitating efficient memory sharing and reducing the overhead in large message transfers. Like LiMIC, it also incurs system call overheads, impacting performance at scale [GM13].

2.7.3 CMA (Cross Memory Attach)

Linux kernel 3.2 established CMA, which provides similar functionality as LiMIC via two new system functions. These calls facilitate the sharing of memory regions across processes, hence greatly lowering the need for memory duplication in communications inside a single node. Nevertheless, like LiMIC, CMA's dependence on system calls might result in additional costs. In addition, both LiMIC and CMA conduct mapping and unmapping of pages at the level of individual pages, which might result in reduced speed when transferring big messages [Vie14].

2.7.4 UCX (Unified Communication X)

The communication framework called Unified Communication X (UCX) is created to give distributed applications and glshpc systems high-performance and scalable communication capabilities. It offers a unified API that abstracts the details of various underlying communication technologies, such as shared memory, RDMA (Remote Direct Memory Access), and network sockets [SVL+15].

2.7.5 UCT (UCX Communication Transport)

UCX Communication Transport (UCT) is a component of the UCX framework that provides a low-level API for direct communication between processes. It is designed to facilitate efficient data transfer mechanisms across different types of hardware, including GPUs and networking devices. UCT supports a variety of communication methods, such as active messages, tag matching, and RDMA operations, allowing developers to optimize the communication according to the needs of their specific applications.

One of the strengths of UCT is its ability to offer direct access to the communication hardware, thereby reducing the overheads typically associated with high-level abstractions and system calls in traditional communication methods. This direct access capability enables high throughput and low latency communication, which is crucial for performance-critical applications in HPC and distributed systems [SVL+15].

2.7.6 XPMEM

XPMEM [Hje], which stands for "cross-partition memory,ïs a more sophisticated technique in this field. It enables processes to allocate certain sections of their memory for the purpose of sharing with other processes. The architecture of XPMEM consists of a library that operates at the user level and a kernel module. This combination provides a smoother and more adaptable method for mapping address spaces.

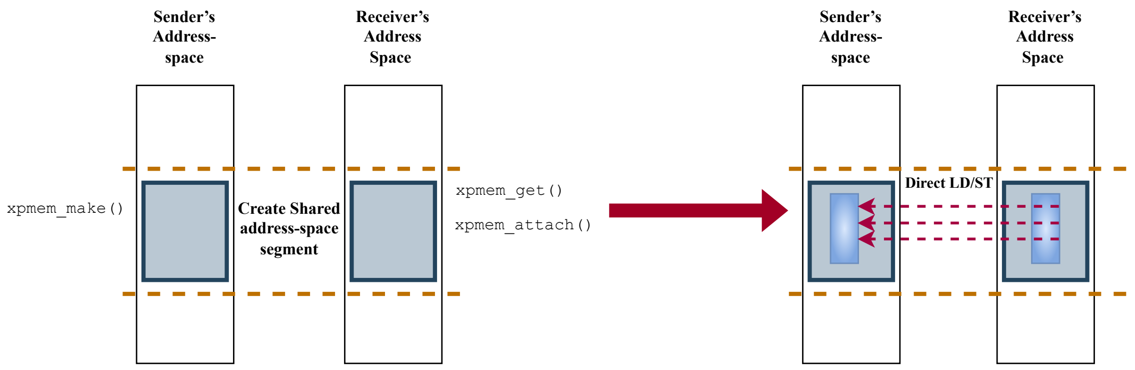

Figure 2.3: XPMEM – Cross-Partition Memory [Has19]

Kernel Module At the core of XPMEM is its kernel module, which is responsible for the low-level operations of memory sharing. This module manages the interactions with the system's memory management unit (MMU) and handles the necessary permissions and security checks for shared memory access [Hje].

User-Level Library: Complementing the kernel module is the user-level library. This library provides a set of APIs that applications can use to interact with XPMEM, simplifying the process of memory sharing. The user-level abstraction allows application developers to utilize XPMEM functionalities without delving into the complexities of kernel-level operations [Hje].

XPMEM's operation revolves around the concept of creating 'segments' of memory that can be shared between processes. These segments are portions of a process's address space that it makes available to other processes. The sharing mechanism involves a series of steps (illustrated in Figure 2.3 [Hje]):

1. Creating a Memory Segment (xpmem_make): A process that intends to share a part of its memory calls the xpmem_make() function. This function registers a specified memory region with XPMEM and returns a unique handle. This handle represents the shared memory segment and is used for subsequent operations.

2. Obtaining Access to a Memory Segment (xpmem_get): A different process, which needs access to the shared memory, uses the handle obtained from xpmem_make() and calls xpmem_get(). This function returns another handle, specific to the receiving process, which it will use for accessing the shared memory.

3. Connecting the Memory Segment (xpmem_attach): In order to receive access to the shared memory, the receiving process must use the xpmem_attach() function, providing the handle acquired by xpmem_get(). The shared memory segment can be mapped more easily into the process's address space thanks to this function, allowing the process to access the memory as if it were a part of its own address space. The operating system specifies the specific virtual address to which the segment is mapped.

4. Direct Load and Store Operations: After the memory segment is connected, the receiving process may immediately access and modify the shared memory.

5. Detaching and Releasing Memory Segments: Processes can detach from shared memory segments when they no longer need access, and the original process can revoke the shared memory segment when it is no longer needed.

6. Efficiency: XPMEM enables direct memory sharing without the need for copying data between processes, significantly reducing overhead in large-scale data transfers.

7. Flexibility: Its user-level library provides a simple and flexible interface for applications to share memory, making it easier to integrate into various software architectures.

8. Scalability: XPMEM [Hje] is particularly beneficial in distributed systems and high-performance computing environments, where efficient resource utilization is crucial.

In summary, XPMEM is a powerful tool for shared memory operations in distributed computing environments. Its combination of kernel-level efficiency and user-level simplicity makes it an invaluable resource for applications requiring fast and flexible memory sharing capabilities, particularly in high-performance and parallel computing contexts [Hje].

In conclusion, while LiMIC [JSCP05], KNEM [GM13], and CMA [Vie14] provide foundational mechanisms for kernel-assisted address space mapping, XPMEM stands out with its user-friendly approach and efficient memory sharing capabilities. These mechanisms are pivotal in optimizing performance in distributed computing systems, particularly where large-scale data handling and inter-process communication are involved.

2.8 Shared Memory in MPI

MPI is widely recognized for its distributed memory model, but it also supports shared memory programming. This capability allows processes (ranks) on the same node to communicate more efficiently by accessing a common memory region, which is typically within a compute node [Rab].

2.8.1 The Role of MPI_Comm_split_type

To aid in shared memory programming, MPI provides the `MPI_Comm_split_type` function. This function groups ranks that share a physical memory domain into the same communicator, using the shared memory type, `MPI_COMM_TYPE_SHARED`, during the communicator split process [Rab].

Practical Steps [Rab]

1. **Initial Setup:** Initialize the MPI environment and ensure all necessary modules are loaded.
2. **Using MPI_Comm_split_type:** Begin by splitting the global communicator into smaller, shared memory communicators.

   ```
   MPI_Comm shared_comm;
   MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shared_comm
   ↪ );
   ```

3. **Working Within the Shared Memory Communicator:** Perform shared memory optimizations and operations within the `shared_comm` communicator.
4. **Synchronizing and Managing Shared Memory:** Use MPI and/or other shared memory constructs for effective data management and synchronization.
5. **Finalizing:** Ensure a clean shutdown by finalizing the shared memory communicator and the MPI environment.

   ```
   MPI_Comm_free(&shared_comm);
   MPI_Finalize();
   ```

Advantages and Use Cases

Shared memory grouping in MPI is beneficial for hybrid programming, optimizing intra-node data sharing and communication. It is essential for HPC applications and large-scale simulations that require efficient shared data access [Rab].

3 Related Work

An overview of the current state of research on shared memory communication, zero-copy communication, and XPMEM application in HPC systems is given in this chapter. Three sections make up the literature review, each of which focuses on a different paper related to the research question.

3.1 Developing Optimal Shared Address Space Reduction Collectives for Multi-/Many-core Architectures

The article *"Designing Efficient Shared Address Space Reduction Collectives for Multi-/Many-cores"* [HCB+18] presents an advanced exploration into optimizing MPI collective operations—specifically MPI Reduce and MPI Allreduce—on modern multi-core and many-core architectures. Utilizing the shared address space model on XPMEM, the study introduces novel, zero-copy designs that significantly enhance the efficiency of data transfer and computation within nodes by avoiding intermediate data copies.

3.1.1 Zero-Copy Design Strategy

The shared memory segment can be mapped more easily into the process's address space thanks to this function, allowing the process to access the memory as if it were a part of its own address space. Through the use of XPMEM for direct memory access, the designs allow processes to directly access and modify data in peer processes' address spaces on the same node, thereby establishing a zero-copy environment. This method drastically reduces the latency associated with data movement and minimizes the overhead introduced by traditional memory copying techniques [HCB+18].

3.1.2 Performance Benchmarking and Analysis

To validate the effectiveness of these new designs, extensive benchmarking was conducted across several architectures, including Intel Broadwell, Knights Landing, and IBM OpenPOWER. The performance improvements were quantified through latency measurements in Reduce and Allreduce operations, which showed up to a threefold decrease. Additionally, runtime improvements in scientific and deep learning applications, such as MiniAMR and the AlexNet neural network, were documented, demonstrating up to 37% and 19% reductions, respectively.

3.1.3 Theoretical Modeling

The research [HCB+18] includes a theoretical model to assess the impact of the proposed zero-copy designs on collective communication operations. This model offers a quantitative framework to analyze the balance between communication and computation, aiding in the prediction of performance gains under various system configurations. To predict the scalability and efficiency of the collective operations, the model makes use of parameters like message size, number of processes per node, and system architecture.

3.2 Contributions and Future Work

The paper [HCB+18] not only showcases significant improvements in MPI collective operations but also sets the groundwork for future research in scalable and efficient HPC systems. It proposes a shift in the traditional approach to designing collective operations by emphasizing a shared address space model that can dynamically adapt to the evolving landscape of high-performance architectures.

The ongoing research aims to further these initial findings by scaling the proposed designs to larger clusters and exploring their effectiveness on additional computing architectures and interconnects. This work is critical in advancing the state-of-the-art in HPC middleware, with implications for future exascale computing systems where efficiency in both computation and communication is critical.

3.3 Process-in-Process

The paper *"Process-in-process: techniques for practical address-space sharing"* introduces a novel method called PiP, which facilitates shared virtual address space (VAS) environments in parallel computing. Unlike traditional multiprocess and multithread models that rely on message passing or operating system support, PiP employs Position-Independent Executables (PIE) and the `dlmopen()` function from Glibc to dynamically load program binaries into the same VAS, thus enhancing address-space sharing.

3.3.1 Technical Details of Process-in-Process (PiP)

PiP [HSG+18] enhances shared-memory communication within MPI runtimes, reduces MPI multithreading overhead, and supports data-sharing in scientific computing. It offers improvements in memory efficiency, reduced performance overhead, and scalability. The fact that PiP only runs in user space increases its portability for use in massive supercomputing applications and makes it easier to integrate with other runtime systems like MPI and OpenMP. Improved communication performance is supported by this integration for applications involving HPC.

3.3.2 Architecture and Implementation

The PiP technology enables the allocation of many processes to a unified Virtual Address Space (VAS), giving each process the ability to access both its own exclusive storage and the storage of other processes within the same space. This architecture integrates the advantages of multiprocess and multithread models by offering process isolation in addition to shared-memory benefits. The execution of PiP entails a ”root process” that has the Virtual Address Space (VAS) and has the ability to generate several "PiP tasks". These jobs have the ability to run distinct applications inside the same Virtual Address Space (VAS), which is not a common feature in ordinary thread architectures.

3.3.3 Performance Evaluation

Performance assessments demonstrate that PiP reduces the synchronization overhead typical of other parallel execution models and minimizes page faults associated with traditional memory-sharing techniques. The use of PIE and `dlmopen()` facilitates the independent yet shared loading of tasks and their dependencies into the VAS, optimizing the use of computational resources and reducing the setup time and memory footprint compared to methods requiring multiple independent processes.

3.4 Casper

The paper, titled "Casper: An Asynchronous Progress Solution for MPI One-Sided Communication,ïntroduces a method called "Casper"that addresses the issue of asynchronous progress in MPI one-sided communication on systems with multiple cores. In order to enable software participation in RMA operations and enable progress on the target side without interfering with hardware-based RMA operations, Casper employs background ghost processes. The study provides a comparison between Casper and conventional thread- and interrupt-based asynchronous progress models. The performance improvements of Casper are shown using microbenchmarks and a practical chemical application. The research evaluates Casper’s performance on several platforms and provides improvements in load balancing. Furthermore, the study examines how Casper ensures both precision and effectiveness in complex systems and provides valuable observations on the impact of asynchronous progress in various scenarios.

3.4.1 Casper’s Efficiency and Scalability

The efficacy of Casper is showcased through microbenchmarks and the utilization of a production chemistry application, NWChem, to exhibit substantial enhancements in asynchronous progress. The paper also investigates the effects of various RMA implementations, load balancing optimizations, and the scalability of Casper. Moreover, the research offers valuable insights into the effects of asynchronous progress in intricate applications, emphasizing its importance, particularly in situations where communication occurs more often and computation time decreases. The paper examines the comparison between Casper and different thread-based approaches, emphasizing the benefits of Casper in enhancing communication overlap and performance [SPH+15].

3.5 OpenSHMEM

OpenSHMEM, an open standard for SHMEM (SHared MEMory) libraries, emerged from the need for a standardized approach to programming in PGAS (Partitioned Global Address Space) environments. Originally developed for supercomputers to exploit the high-speed network interfaces and shared memory architectures, SHMEM libraries provided a set of APIs for efficient data transfer and synchronization among parallel processes. OpenSHMEM's inception was motivated by the desire to unify these libraries under a common standard, enhancing portability and scalability across various hardware and software platforms [CCP+10].

The adoption of the PGAS model, which OpenSHMEM exemplifies, represents a significant shift in parallel computing. PGAS offers a global memory space accessible by all processes, simplifying the programming model compared to traditional message-passing interfaces. This model facilitates a more intuitive development process, enabling programmers to focus on the computational aspects of their applications rather than the intricacies of data communication [CCP+10].

3.5.1 Key Features and Advantages

OpenSHMEM's design is characterized by several key features that contribute to its effectiveness in parallel computing. Its symmetric memory model allows for direct access to global shared memory, reducing the overhead associated with data movement. Communication primitives, such as put and get operations, along with collective operations, provide a robust mechanism for data exchange and synchronization among processes. These features are complemented by OpenSHMEM's synchronization mechanisms, including barriers and locks, which ensure correct program execution in concurrent environments.

Its API simplifies the development of parallel applications by abstracting the complexities of direct memory access and communication. This simplicity, coupled with the efficiency of its communication primitives, can lead to significant improvements in application performance. Additionally, programs created with OpenSHMEM can be readily adapted to run on a range of hardware architectures, from multicore processors to large-scale supercomputers, thanks to the model's support for scalability and portability [CCP+10].

3.5.2 Technical Architecture and Components

OpenSHMEM leverages a highly optimized technical architecture to maximize performance in parallel computing environments. At its core, OpenSHMEM provides a library of functions that facilitate memory management and communication between processes running on distributed memory systems. This section delves deeper into the technical components and the functionality of the OpenSHMEM library.

Memory Management

OpenSHMEM implements a symmetric heap memory model, where each process allocates and accesses memory in a global address space. This allows for direct memory access (DMA) capabilities, which bypass the operating system to accelerate data transfer speeds. The symmetric heap is dynamically managed, allowing runtime allocation and deallocation, which is critical for adapting to varied workload sizes.

Communication Primitives

The library offers an extensive array of communication primitives that are fine-tuned for enhanced performance:

- **Put and Get Operations:** Employed for one-sided communication, these operations enable a process to write data to (put) or read data from (get) the memory of a remote process without requiring the remote process's active participation.
- **Atomic Operations:** In concurrent access scenarios, these operations are essential to maintaining data integrity. Building counters, locks, and other synchronization mechanisms requires the ability to perform atomic operations like `fetch-and-add`, `swap`, and `compare-and-swap`, which OpenSHMEM makes possible.
- **Collective Operations:** OpenSHMEM includes a variety of collective functions like `shmem_barrier_all()`, `shmem_broadcast()`, and `shmem_reduce()`. These functions are designed to synchronize data uniformly across all processes or to execute tasks such as reductions and broadcasts effectively.

Synchronization Mechanisms

Synchronization in OpenSHMEM is designed to ensure consistency and coordination among processes:

- **Barriers and Locks:** Barriers are used to coordinate and synchronize all processes at certain places in the program, guaranteeing that no process advances until all others have reached the barrier. Locks enforce mutual exclusion, guaranteeing that only a single process may access a critical region at any one moment. This is crucial for preventing race situations.
- **Fence and Quiet Operations:** These operations are used to ensure the ordering and completion of memory operations. `shmem_fence()` guarantees the ordering of delivery of put and get operations, while `shmem_quiet()` ensures that all outstanding puts and gets are completed before proceeding.

Programming Model Enhancements

OpenSHMEM also includes extensions to support more complex data structures and programming paradigms:

- **Derived datatypes:** Similar to MPI, OpenSHMEM can define complex datatypes to better map to the memory layout of user-defined structures, enabling more efficient data transfers.
- **Dynamic Tasking:** Recent extensions to OpenSHMEM propose the inclusion of dynamic tasking capabilities, allowing more flexibility in distributing workloads among processes dynamically based on runtime decisions.

4 Designing a zero copy shared memory API

Inter-process communication (IPC) is critical in enhancing the performance of distributed computer systems. The core of scalability is data movement efficiency. Improved intra-node communication between processes is the goal of Thw Zero Copy API ZCom, which is discussed in the following section. Zero-copy communication occurs via shared memory. The API is intended to address the trade-offs and overhead that characterize bilateral communication protocols such as MPI. The ZCom API utilizes XPMEM extension (Cross-Partition Memory) to facilitate the direct memory access between processes thus reducing data transfer overhead and latency. The supplied text provides a presentation of structure, contents, and main features of the ZCom API. It highlights the capability of the API to speed up the performance of parallel applications in HPC environments. The resultant API includes a total of six methods related to communication channel initialization, read-write synchronization, and cleanup. The ZCom API has been optimized to be easily used within the currently popular parallel programming models such as MPI with the goal of improving the efficiency and scalability of inter-node data sharing. The API is designed with simplicity, effectiveness, and flexibility as the driving principles; the greatest effort here is directed at reducing communication overhead and maximizing efficiency in HPC environments.

4.1 Address Space Mapping Mechanisms

The choice of using XPMEM [Hje] in the zero-copy API as opposed to other shared memory solutions such as LiMIC, KNEM, or CMA, is grounded on several considerations. Mapping a memory region in LiMIC, offered by the LiMIC library, directly into the kernel of a communicating process is allowed. Due to this manipulation, the data transfer between the processes is facilitated, since the receiving process gets the access to the pages, which are being mapped in the kernel. Although, this way, there is less need to copy large amount of data inside a node, every system call that is associated with each transaction impacts the overall performance. KNEM provides such approach by allowing shared memory access between processes that can show performance gains in some cases. CMA, which was only introduced with Linux kernel 3.2, further possibilitize these features by two new system calls which enable efficient transfer of large areas of memory. Despite the fact that CMA helps to increase the performance by decreasing the number of particular operations, the overhead of system calls is a major problem, especially for frequent or big data transfer. Furthermore, the LiMIC, KNEM, and CMA approaches all suffer from the granularity of the page size when mapping and unmapping, which may have negative impact on performance for large message transfers.

On the contrary, XPMEM offers a better approach to shared memory access through permitting a more direct and versatile manner of sharing and accessing memory areas between processes. XPMEM allows programs to map remote pages directly to their address space avoiding numerous system calls employed through a user library and a kernel module. The main XPMEM functions, xpmem_make(), xpmem_get(), and xpmem_attach(), are the way of how to share and access memory segments, which dramatically increase the transfer efficiency and minimize overhead. This dominance in simplicity, efficiency, and flexibility positions XPMEM as the desirable solution for developing a zero-copy API that aims to minimize data transfer overhead and maximize performance in HPC environments.

4.2 ZCom API Architecture

Many HPC applications rely on the MPI point-to-point communication techniques, which are covered in chapter 2. These mechanisms require additional memory copies, which add to the superfluous traffic on the memory bus. We have created a straightforward communication API by utilizing the shared address space design philosophy seen in XPMEM. The fundamental concept is that all MPI processes inside a node can directly access each other's application vectors when they are engaged in communication. Without additional transmissions via point-to-point channels, each participant process is able to execute computation and communication on both local (own data) and remote rank applications vectors.

Three steps make up communication using our zero-copy API. The first step is necessary to establish the exchange information. The real communication operation takes place in the second phase. The data structures utilized for information exchange are to be deleted in the third stage.

1. **Initialization:** The first step is to initialize the communication channels. This is achieved through the `ZCom_init_c` function that creates the shared memory segments, synchronization counters, and communication channels. In addition to that, the function also selects the location of the victim processes and makes communication channels for them. The input is a set of data buffers, their sizes, MPI communicator, target process ranks, and the number of target processes. It provides the same with related addresses and channels IDs to perform operations of communication. The linked addresses are the shared memory segment addresses of the target processes. They work as receiving buffers for the source process. The channel IDs are used to make the channel identification process.

2. **Communication:** The second step is the communication step. This is done through the `ZCom_can wrote`, `ZCom_have_written` and `ZCom_can_read`, `ZCom_have_read` operation. The function `ZCom_can_write` tests whether the source process has the write permission on the target process shared memory buffer, and the function `ZCom_can_read` looks for the read permission on the source process shared memory buffer. Since the shared buffer of the target process has been written by the source process, it is up to the `ZCom_have_written` function to be called; then, after the target process has read from the shared buffer of the source process, the `ZCom_have_read` function is called. The assemblies in the functions contain synchronization of read and write operations between the processes using counter techniques to ensure data consistency while blocking false sharing. These canals take as input parameters id of channels as input variable. `ZCom_can_write` is used to hold the incrementing target process's read counter, and `ZCom_can_read` is used to increment the target process's write counter because the source process's read counter is being incremented. `ZCom_have_written` compares the source process write counter notation to that of the target process read counter, whereas `ZCom_have_read` calculates the target read counter against the source process write

counter. Response 1 shall be considered as a sign that the comparison is correct and the operation of communication is to be accomplished one by one. The comparison function returns 0 when communication is not possible either because end device is unknown, the remote unit does not accept the communication request or the remote unit settings does not match with the communication parameters.

3. **Cleanup:** The third step is to clean up of the resources associated with the communication channels. This is achieved by invoking the function `ZCom_cleanup`, which detaches from the shared memory segments and discards the segment IDs. This functionality also takes advantage of synchronization counters that see to it that all operations has been done and there is no data in flight. The input of the function is the channel IDs and it returns an error code after the success or failure of the cleanup operation.

4.3 ZCom_init

Initialises ZCom channel for IPC through shared memory specifying the destination processes. This method sets up shared memory segments for buffers, organizes read and write operations in terms of sync counters, and controls communication channels among the processes, ensuring that all intended processes have the required shared resources. The system utilizes XPMEM for memory sharing optimization and MPI for inter-process communication so that it can easily adapt to both shared and distributed memory architectures. Here is a step-by-step breakdown of the `ZCom_init_c` function's operation (compare Figure 4.1)

1. **Parameter Reception**: The function commences by accepting a multitude of critical parameters: a list of data buffers (`buffer`), their respective sizes (`buffer_size`), an MPI communicator (`comm`), target process ranks (`target_ranks`), the number of target processes (`num_targets`), and storage locations for attached addresses and channel IDs. The most crucial information about the shared data and the planned communication processes are captured by these parameters..

2. **Process Locality Verification**: A critical step of verifying that all target processes are situated on the same physical host. This verification is imperative for the feasibility of shared memory access. If any target process is on a different host, the function aborts the initialization, signaling an error due to the infeasibility of shared memory usage across different hosts.

3. **Environment Setup**: The function enhances communication efficiency for processes on the same physical host by dividing the processes based on shared memory accessibility by using the MPI communicator. This step is crucial for efficiently harnessing the shared memory capabilities.

4. **Buffer Adjustment for Shared Memory Alignment:** One of the most critical operations performed by the 'ZCom_init_c' function involves adjusting the given buffers to align with the system's memory page boundaries. This adjustment is essential for the efficient use of shared memory, as it ensures that memory accesses are optimized for hardware performance. Here's how this process unfolds:

- For each buffer intended for communication, the function calculates the starting address of the buffer and aligns it to the nearest lower page boundary. This alignment is necessary because shared memory segments must start at page boundaries to be shared effectively across processes.
- After determining the aligned start address, the function calculates the end address of the buffer, ensuring that it includes the entire buffer while also aligning it to the nearest upper page boundary. This step might extend the memory segment slightly beyond the original buffer size, but it is crucial for maintaining alignment constraints.
- The total size of the memory segment that must be shared is then obtained by deducting the aligned start address from the aligned end address, which yields the size of the aligned memory segment.
- A shared memory segment is created for this aligned buffer, allowing for efficient and direct access by the target processes. The alignment to page boundaries ensures that the operating system can efficiently manage these memory segments, leading to improved performance in data sharing and communication.
- Finally, any necessary adjustments to the pointers and sizes in the communication channel structures are made to reflect the aligned addresses and sizes. This ensures that when processes access the shared memory, they are accessing the correct locations and sizes, thereby preventing any potential data corruption or access violations.

This buffer adjustment step is pivotal in the initialization of communication channels as it directly impacts the efficiency and reliability of data sharing between processes. By aligning buffers to memory page boundaries, the 'ZCom_init_c' function maximizes the performance benefits of shared memory communication, thereby enhancing the overall performance of the high-performance computing application.

5. **Shared Memory Segmentation**: For each buffer designated for communication, the function creates a shared memory segment for the aligned buffer, accessible to the target processes. This shared memory mechanism is facilitated via XPMEM technology, allowing direct memory segment access across processes on the same node.
6. **Channel and Segment Information Distribution**: Through MPI's collective communication capabilities, particularly the `MPI_Alltoallv` function, the function distributes the segment IDs created in the shared memory segmentation step among all target processes. This ensures each process is aware of the segment IDs necessary for establishing direct memory access to the buffers of its communication partners.
7. **Attach shared memory segments to target processes**: The function links the shared memory segments to the target processes, enabling them to directly access the shared memory. The enclosed addresses are kept for future communication actions, guaranteeing that the processes are prepared for effective data exchange and synchronization.
8. **Synchronization Mechanism Initialization**: The function sets up synchronization counters for each communication channel, essential for coordinating read and write operations between the processes, ensuring data consistency, and preventing race conditions.

9. **Communication Channel Establishment**: For each target process, the function finalizes the creation of a communication channel, involving assigning a unique channel ID and attaching the process to the shared memory segment corresponding to its buffer. The attached address and channel ID are stored for later communication operations.
10. **Error Handling and Cleanup**: Throughout the initialization process, the function meticulously checks for errors. In case of failure, such as memory allocation errors, issues in shared memory segment creation, or process locality problems, the function gracefully aborts the initialization, cleans up any partially allocated resources, and returns a specific error code indicating the failure type encountered.
11. **Success Indication**: Upon successful completion of all steps, the function sets the error code to indicate success, signifying that the communication channels have been correctly established and the processes are ready for efficient data sharing and synchronization operations.

4.4 Synchronization Mechanism

ZCom API for managing shared-memory communication between two processes, denoted as Rank A and Rank B. At the core of this mechanism is the `SharedData.sync_counters` array, which consists of four elements used to synchronize read and write operations between the ranks.

Here is what each element represents in the synchronization process:

1. **Rank A written (`sync_counters[0]`)**: This counter is incremented by Rank A after it completes a write operation. It serves as an indicator for Rank B that new data has been written and is available for reading.
2. **Rank B read (`sync_counters[1]`)**: This counter is incremented by Rank B once it finishes reading the data written by Rank A. It acts as an acknowledgment to Rank A that the data has been read, allowing Rank A to perform subsequent write operations if needed.
3. **Rank B written (`sync_counters[2]`)**: Similar to `sync_counters[0]`, this counter is used by Rank B to indicate that it has completed a write operation, notifying Rank A that there is new data to read.
4. **Rank A read (`sync_counters[3]`)**: This functions like `sync_counters[1]`, where Rank A increments the counter after reading the data written by Rank B, signaling to Rank B that the data has been successfully read.

The arrows in the diagram show the flow of operations: Rank A writes data and waits for Rank B to read it (indicated by the interaction with `sync_counters[0]` and `sync_counters[1]`), while Rank B writes data and waits for Rank A to read it (indicated by `sync_counters[2]` and `sync_counters[3]`). This synchronization mechanism ensures that both ranks can communicate efficiently, with a clear indication of when data is written and read, thus preventing data race conditions and ensuring consistency in shared-memory communication.

Example Usage of the ZCom API

The example (see 4.1) illustrates the initialization and usage of ZCom API in a distributed system controlled by MPI. The main operations described are setting up communication channels, writing to these channels and cleanup activities. During the setup phase, each process identifies its communication targets, excluding itself, and buffers for message passing. Then, the `ZCom_init_c` function is used to initialize channels for communication with each target, depending on the MPI communication world. In particular, the communication example illustrates a case where one process (rank 0) writes data to another process (rank 1) using the ZCom API. The function `ZCom_can_write_c` checks if the channel can be written to followed by the actual write, then a confirmation with `ZCom_have_written_c`. At the end, the example is followed by clean-up stage when all the allocated resources are deallocated and MPI is finalized, providing a nice shutdown of the application.

Code Example

```c
#include "ZCom_comm.h"
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define SHARE_SIZE 1024

int main(int argc, char *argv[]) {
    int err, rank, size, *channel_ids, **send_buffers;
    size_t *buffer_sizes;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Setup communication targets and buffers
    int num_targets = size - 1, *target_ranks = malloc(num_targets * sizeof(int));
    send_buffers = malloc(num_targets * sizeof(int *));
    buffer_sizes = malloc(num_targets * sizeof(size_t));
    for (int i = 0, j = 0; i < size; i++) {
        if (i != rank) {
            target_ranks[j] = i;
            send_buffers[j] = malloc(SHARE_SIZE);
            buffer_sizes[j] = SHARE_SIZE;
            j++;
        }
    }

    // Initialize ZCom channels
    void **attached_addrs = malloc(num_targets * sizeof(void *));
    channel_ids = malloc(num_targets * sizeof(int));
    ZCom_init_c((void **)send_buffers, buffer_sizes, MPI_COMM_WORLD, target_ranks, num_targets
↪ , attached_addrs, channel_ids, &err);

    // Example communication (rank 0 writes to rank 1)
    if (rank == 0) {
        for (int i = 0; i < num_targets; i++) {
            if (target_ranks[i] == 1 && ZCom_can_write_c(channel_ids[i], &err)) {
                *send_buffers[i] = rank * 100;
                ZCom_have_written_c(channel_ids[i], &err);
            }
        }
    }

    // Cleanup and finalize
    ZCom_cleanup_c(channel_ids, num_targets, MPI_COMM_WORLD, &err);
    for (int i = 0; i < num_targets; i++) free(send_buffers[i]);
    free(send_buffers);
    free(buffer_sizes);
    free(target_ranks);
    free(channel_ids);
    free(attached_addrs);
    MPI_Finalize();
    return 0;
}
```

Listing 4.1: Example Usage of the ZCom API

4.5 Fortran Interface

The ZCom API is made to work with a variety of programming languages and to fit in well with pre-existing parallel programming paradigms, such as MPI. To do this, we offer Fortran interface to the ZCom API, enabling Fortran applications to leverage the zero-copy communication features of the API. The Fortran interface contains the required module and subroutine definitions that allow for Fortran programs to invoke the ZCom API functions and conduct zero-copy communication. The interface is straightforward and user-friendly enabling Fortran programmers to easily use the ZCom API and its effective shared memory communication facilities. For the Fortran support we have implemented two wrapper functions for the ZCom API, which are in C one for the initializing and one for the cleaning of the ZCom API. This wrappers are required as Fortran does not handle MPI_Comm type, which is needed for ZCom API. Thus the wrapper functions are utilized to map the MPI_Comm type into an integer type that is compatible with Fortran.

4.6 Comparative analysis

In this section of the comparative analysis, the ZCom API is carefully compared to technologies that are commonly used in the field of HPC communication. In order to comprehend the complex performance, integration capabilities, scalability, memory management efficiencies, technological foundations, error handling mechanisms, and validation practices of these communication technologies, this assessment is organized around a set of carefully selected criteria 4.1.

4.6.1 Evaluation Criteria

The following criteria are used to assess the performance and integration capabilities of HPC communication technologies: The results are shown in Table 4.1.

- **Performance Enhancement and Efficiency:** This criterion assesses the capacity of each technology to mitigate communication latency and enhance the overall performance metrics, which is pivotal in HPC settings.
- **Integration with Existing Systems:** This evaluates the ease and compatibility with which these technologies can be integrated into prevailing HPC ecosystems, ensuring seamless interaction with established protocols.
- **Scalability:** The ability of the technology to sustain or improve performance as the system scales up in size and complexity is critically analyzed under this parameter.
- **Memory Management:** This involves a thorough examination of how each technology approaches memory operations, with a focus on optimizing shared memory interactions and direct memory access strategies.
- **Underlying Technology and Design Approach:** The foundational technological choices and their design rationale are scrutinized, highlighting how these decisions are suited to meet the demands of contemporary HPC architectures.
- **Error Handling and Fault Tolerance:** This criterion looks into the robustness of the technology in managing communication errors and maintaining data integrity under failure conditions.
- **Testing and Validation:** The methods and practices employed to empirically validate the effectiveness and reliability of the technology in real-world HPC applications are considered here.

4.6.2 Comparative Analysis

In this section, we compare the ZCom API with other prominent HPC communication technologies, including MPI, PiP, Casper, and OpenSHMEM, based on the evaluation criteria outlined above.

MPI

The traditional MPI uses the send-and-receive model which, though suitable for distributed computing, forces multiple copies of data between the application buffers and system buffers. This adds large overhead, especially to large data transfers as it uses both time and system resources, thus leading to increased latency. Unlike, ZCom uses a zero-copy technique supported by shared memory that allows direct memory access between processes. This approach cuts the requirement of intermediate data copies, thereby, decreasing latency and improving utilization of bandwidth.

The efficiency improvement is most pronounced in intra-node communications when data locality can be completely leveraged. ZCom also works on improving scalability by eliminating unnecessary copies of the memory as well as through the shared memory that allows direct access. Its architecture facilitates communication continuing indefinitely, unlike the approach of the MPI which usually checks communication progress using specific functions. This property enables ZCom to provide faster communication in shared memory environments, thus showcasing its abilities to enhance performance and resource utilization in high-performance computing applications.

Efficient Shared Address Space

Hashmi et al. propose an approach that improves co-operative work in multi-/many-core systems. The ZCom API system containing the zero-copy mechanism significantly decreases the intranode communication overhead, which gains big improvement especially in the processing of the large workloads, thus satisfying the performance improvement requirement. Hashmi et al. methodology, which integrates with MVAPICH2 MPI library, satisfies the integration criterion in terms of the compatibility with popular HPC platforms. They also pay attention to scalable communication primitives and efficient caching mechanisms that cover scalability and memory management criteria, proving a deep understanding of shared memory in HPC settings.

Process-in-Process (PiP) Techniques

Shared address space utilization is improved by a flexible execution model in PiP. The design philosophy of ZCom is compatible with the approach of PiP in the optimization of shared memory, which addresses the memory management criterion by minimizing the data transfers and improving performance efficiency. Although PiP is made flexible through user-level design and portability across different computing environments, ZCom is focused at enhancing the intra-node communication, especially in shared memory systems, hence delivering targeted performance improvements in HPC applications.

Casper's Asynchronous Progress Model

Casper addresses the issue of asynchronous MPI Remote Memory Access progress, which impacts performance scalability in many-core architectures. This matches the scalability and underlying technology requirement. Even though ZCom optimizes intra-node data transfers, it follows the same aim of communication efficiency as Casper does just through a different conceptive framework. Casper's design is capable of handling the integration with MPI and compatibility and demonstrates a strong architecture that can handle the challenges of asynchronous operations handling in the current HPC settings.

OpenSHMEM

One of the principal roles of OpenSHMEM [SVP+14], as an open-source interface, is to support one-sided communications which is essential in HPC. This model allows the processes to access the memory in a remote manner without the peer processor. It uses RDMA (Remote Direct Memory Access) for better data transfer performance, decrease of CPU intervention, scalability, and reduced large-scale computing latency.

Library supports different communication modalities, like point-to-point, collective communications, and synchronization operations. The API of it is complete and includes the operations put, get, and atomic memory on different nodes. These operations are vital for ensuring integrity and synchronization of processes in a distributed system. Finally, OpenSHMEM offers memory barriers and collective synchronization for all operations on shared data to be done before any process continues, which is necessary to maintain the integrity and correctness of parallel computations [SVP+14].

Also, OpenSHMEM is dedicated to memory management, which is evident in its symmetric heap and such objects, support. In this model, each process can allocate and manage memory which is shared with all other processes, thereby making the programming and execution of distributed applications simpler by enforcing the global view of memory, but at the same time having the ability to take advantage of local optimizations by the individual process [SVP+14].

The standardization of OpenSHMEM has helped to bring interworking between OpenSHMEM libraries, making it easy the user to use any OpenSHMEM. The Committee for the OpenSHMEM Specification is very active in creating new versions and extensions of the standard that would be able to run on new hardware architectures and emergent memory models, thus enabling it to follow the rapid progression of the computing technology [SVP+14].

ZCom compared with OpenSHMEM Unlike OpenSHMEM's [SVP+14] comprehensive concept, ZCom brings more specialized solutions to zero-copy shared memory communication diseases that improve communication within the node by the best use of shared memory space. While within a single node data can be exchanged between the running processes without the need to copy it between buffers, this feature is especially useful for applications that need to achieve high throughput and low-latency communication between threads or processes within a same physical machine [SVP+14].

Comparative Analysis The comparative analysis brings out how the specialized approach of ZCom is in line with and departs from the existing and emerging technologies such as OpenSHMEM. Each technology has its unique strengths, and the operational focuses, ZCom being just one of them, as a result, bring variety into design qand communication strategies of the HPC. The above detailed analysis highlights the position ZCom occupies within this eco-system, demonstrating its ability to enhance performance efficiency in specialized intra-node communication situations. The inter-operation of OpenSHMEM's global memory model and ZCom's intra-node optimizations shows a complete solution for inter-node and intra-node communications issues in the modern HPC systems.

Table 4.1: Comparison of ZCom API with MPI, PiP, Casper, and OpenSHMEM

Feature	ZCom	MPI	PiP	Casper	OpenSHMEM
Communication Model	◐	◑	◐	◐	●
Memory Management	◐	◑	◑	◐	●
Scalability	◐	●	◑	◑	●
Target Application	◐	●	◐	◐	◑
Distinct Features	◐	●	◐	◐	◑

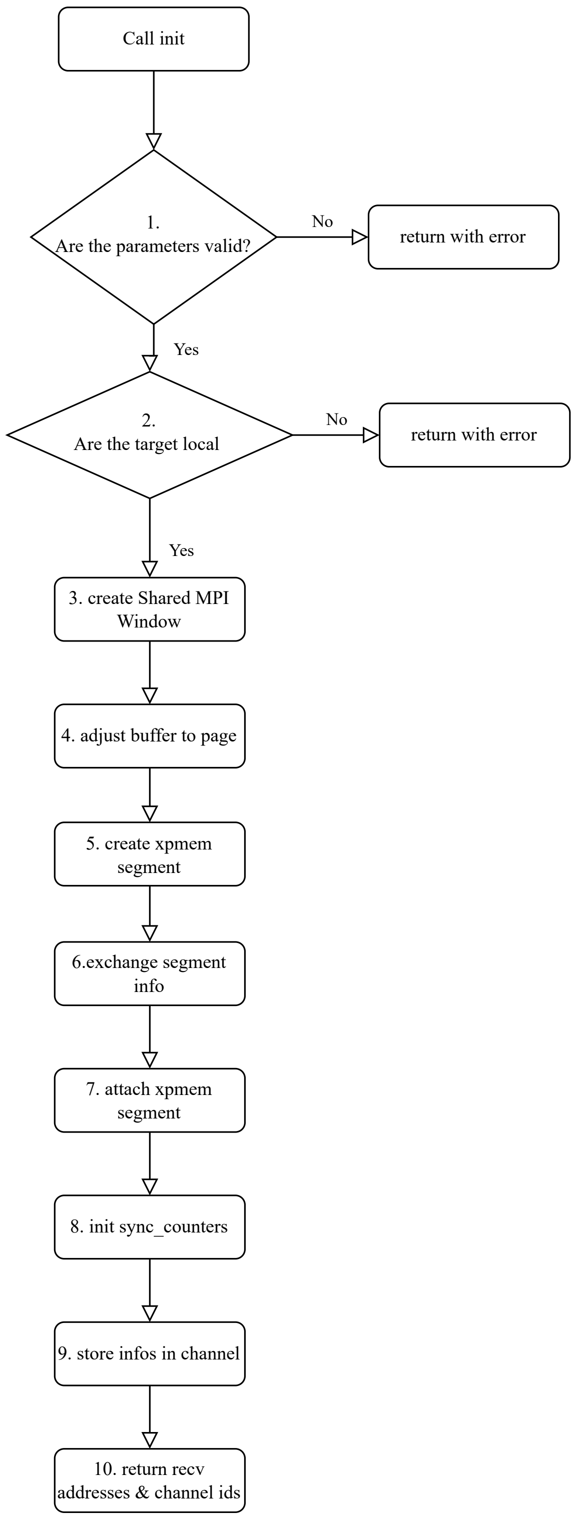

Figure 4.1: Flowchart of the ZCom_init function.

5 Performance Evaluation

The chapter is focused on the performance evaluation of the ZCom API, using microbenchmarks and real-world applications. The assessment is done on a multi-core architecture and the results, show the effectiveness and scalability of the ZCom API in different situations. The performance measures include the latency, throughputs, and scalability. These two metrics are then contrasted against the standard communication modes used by traditional methods so as to underline the benefits of zero-copy communication in HPC environments. The latter also considers the impact of the ZCom API in the performance of real-world applications, such as scientific simulations and data analytics to demonstrate its practical applicability in the area of high-performance computing.

5.1 Experimental Setup

The performance test is carried out on Hawk supercomputer, an HPC machine designed in a multi-core model. The system is built on the AMD EPYC Rome 7742 processor, which has 64 cores per CPU and runs at a base 2.25 GHz. A 5632-node system, with each node having 256 GB RAM.

5.2 Microbenchmarks

The microbenchmarks are designed for evaluating latency and throughput of the ZCom API in various communication contexts. Another set of benchmarked items are point-to-point communication, collective communication and synchronization operations. The results demonstrate the efficiency and scalability of the ZCom API in different communication patterns and sizes of the messages.

5.2.1 Ping-Pong Latency

The complete round trip time of a small message communication between two processes is what the ping-pong latency benchmark evaluates. The results reveal that the ZCom API has very low overhead and low latency for small message sizes which is why it is suitable for low-latency communication in HPC applications. So, the table data provides a performance comparison of different communication methods where an average time and acceleration factor are determined using the standard MPI Two-Sided approach as a baseline.

- **Standard MPI Two-Sided** gives the average execution time of *0.000692 milliseconds* and is used as the benchmark with the acceleration factor of *1.00*. This approach is typical for MPI applications where the communication is both sending and receiving messages between processes.

- **MPI One-Sided** achieves a slightly longer average time of *0.001048 milliseconds* with an acceleration factor of *0.66* against the baseline. It signals that MPI One-Sided is slower in this particular scenario. The One-Sided MPI, also known as Remote Memory Access (RMA), has been designed specifically for cases when one process can access the memory of another process directly, without involving this process in communication. This approach is developed to provide the more transparent and even quicker data transfer.
- **ZCom** is observed to take an average time of 0.000029 milliseconds and therefore, a speeding factor of 23.86 and hence, is the superior of MPI that is standard. The note "shared memory, significantly faster", indicates that ZCom applies the shared memory method to accelerate communication, and that is more effective for those sorts of tasks.

The results revealed a very big contrast between the performances of these communication approaches. Despite the fact that a well-established MPI Two-Sided communication is a common practice, some alternatives such as ZCom, can provide a significant performance boost, especially in cases where fast communication is crucial. The type of communication that is being used can severely impact the overall efficiency and speed of data processing in a distributed computing environment.

Communication Method	Average Time (ms)	Acceleration Factor (Compared with Standard MPI (MPT))
Standard MPI Two-Sided	0.000692	1.00
MPI One-Sided	0.001048	0.66
ZCom	0.000029	23.86

Table 5.1: Communication methods comparison

5.2.2 Sync_p2p kernel

PRK, or Parallel Research Kernels suite [Lab], is a set of parallel benchmarks aimed at testing the performance of parallel computing systems. Pipeline kernel is an integral part of Parallel Research Kernels (PRK) suite and it intends to evaluate and benchmark the performance of point-to-point synchronization mechanisms in distributed computing frameworks. This kernel represents a situation of the pipelined algorithm execution across a 2D grid, with the main emphasis on performance behavior of communication and synchronization in parallel computing environments.

Methodology

For HPC, the kernel is implemented using MPI, in accordance with best practices for parallel programming. It is intended to assess the efficiency of data synchronization among the distributed processes, an important part of the parallel computation applications that need iterative updates based on the adjacent data points [Lab].

Grid Partitioning and Communication

With respect to the Pipeline kernel, the two-dimensional grid is distributed between MPI processes through a strip-wise decomposition in the first dimension. This separation makes it possible to use a distributed processing strategy where each process is in charge of a certain area of the grid. The kernel permits grid line grouping as an option, which optimizes communication by limiting the number of times that inter-process messages have to be exchanged. Synchronization is carried out by exchanging boundary values between the adjacent processes, which provides data integrity in the distributed grid [Lab].

Computational Model

Computational core of the kernel performs iterative update of grid values, using computational stencil that includes the values of the adjacent points. This model typifies a diversity of scientific computing applications in which local computations are affected by the data points located near them [Lab].

Performance Measurement and Analysis

The performance metrics of the Pipeline kernel include the total run time and the throughput of point-to-point synchronizations. In these metrics you can find information about the effectiveness of synchronization and communication overhead in the distributed processes. In addition, the kernel performs the computation result's correctness validation by comparing certain grid values with predetermined verification values, which contributes to the correctness of the algorithm [Lab].

The informal description of the Pipeline kernel is augmented with a graphical representation depicting the synchronization mechanism among distributed threads in a computational grid [Lab]. The graph represents the stripwise decomposition of the grid across threads (Thread 0 to Thread 3), and displays the data flow direction to sync. This visualization underlines the kernel's core principle: point-to-point communication between neighbouring threads for updating grid values reflecting the essence of the algorithms distributed compute and synchronization strategy.

The Pipeline kernel brings an important angle to the performance effects of point-to-point synchronization in distributed systems. Through concentrating on communication and synchronization, this allows for a frame to be created for the assessment and tuning of parallel algorithms and system, focusing on the role of attaining effective communication mechanisms in achieving glshpc environments [Lab].

Messurements and Results

Rate vs. Number of Ranks

The performance analysis involves comparing two methods: MPI (initial) and Zcom (updated), based on their execution rates (in MFlops/s) and average processing time, against the number of ranks.

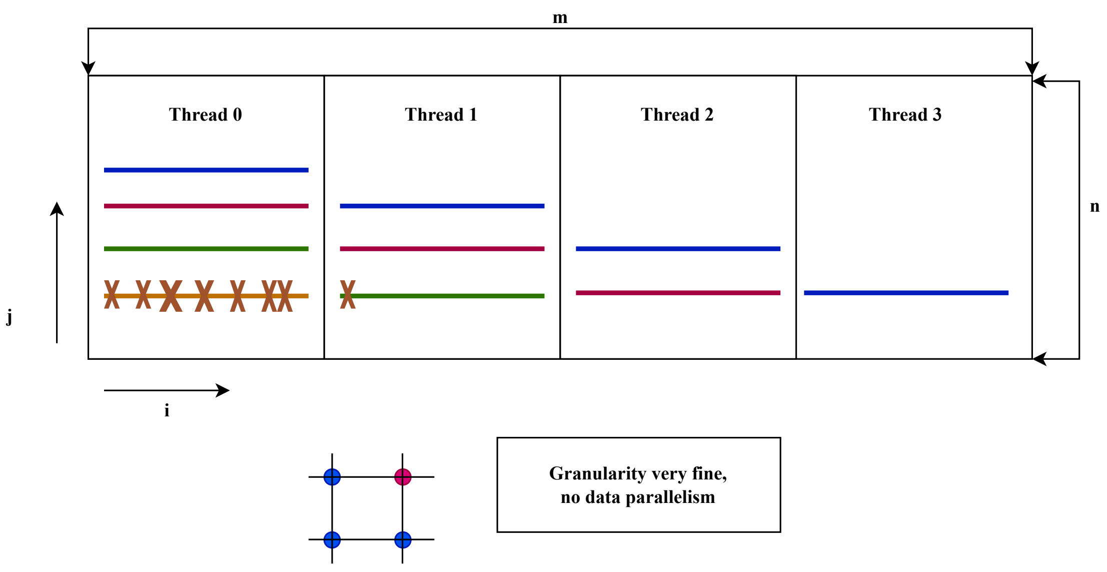

Figure 5.1: Point-to-point synchronization [Van16]

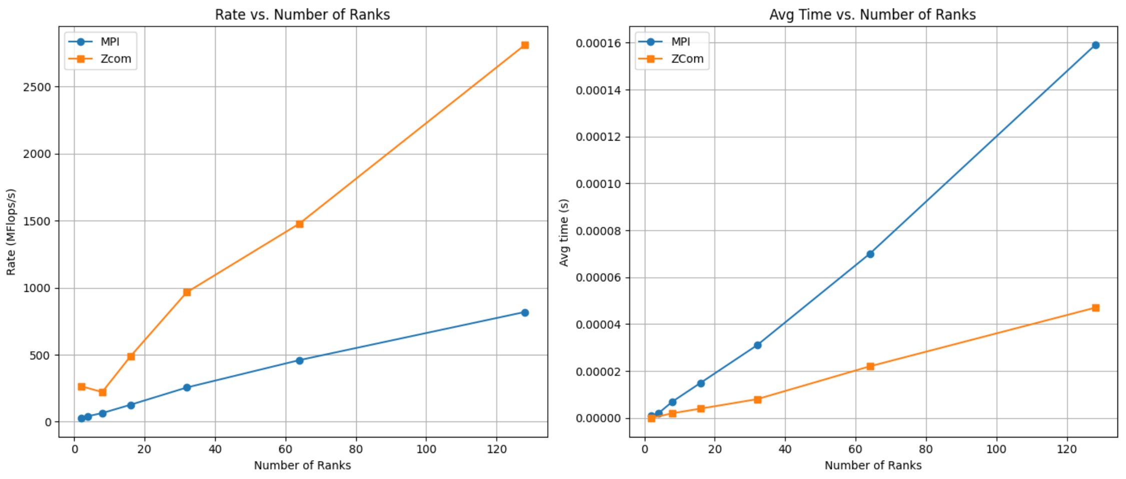

Figure 5.2: Results of the Sync_p2p kernel benchmark

- **Initial Observation:** The MPI method shows a steady increase in the rate as the number of ranks increases, indicating linear scaling behavior. This suggests that the method scales well with the increased computational load.
- **Updated Method:** The Zcom method displays an inconsistent pattern initially, with a decrease in performance at a lower rank count but then showing a significant increase at higher ranks. It surpasses MPI considerably in performance, particularly beyond 64 ranks, indicating superior scaling and efficiency in processing.

Average Time vs. Number of Ranks

- **Initial Trends:** The MPI method shows a gradual increase in average processing time as the ranks increase, which is an expected trend as the computation becomes more demanding.
- **Updated Performance:** Zcom outperforms MPI consistently across all rank counts in terms of average time, starting off with a lower average time and maintaining better efficiency as the ranks increase.

Rate vs. Grid Size

Figure 5.3 delineates the computational rate (in MFlops/s) against the increasing grid sizes for both the MPI and ZCom methods. The data plotted on a logarithmic scale shows that both methods exhibit improved performance rates as grid sizes increase. For ZCom, the performance rate starts at approximately 9668 MFlops/s for a grid size of 1000^2 elements and escalates to about 37728 MFlops/s for 32000^2 elements, demonstrating a strong scaling efficiency with increasing data volume. Similarly, the MPI method begins at 3752 MFlops/s and reaches up to 36080 MFlops/s under the same conditions. ZCom consistently outperforms MPI across all grid sizes, highlighting its superior efficiency in handling larger and more complex computational grids in sync_p2p kernel applications.

This analysis provides crucial insights into the scalability of both methods, particularly underscoring ZCom's ability to efficiently manage larger datasets, a key factor in its superior performance in high-demand parallel computing environments.

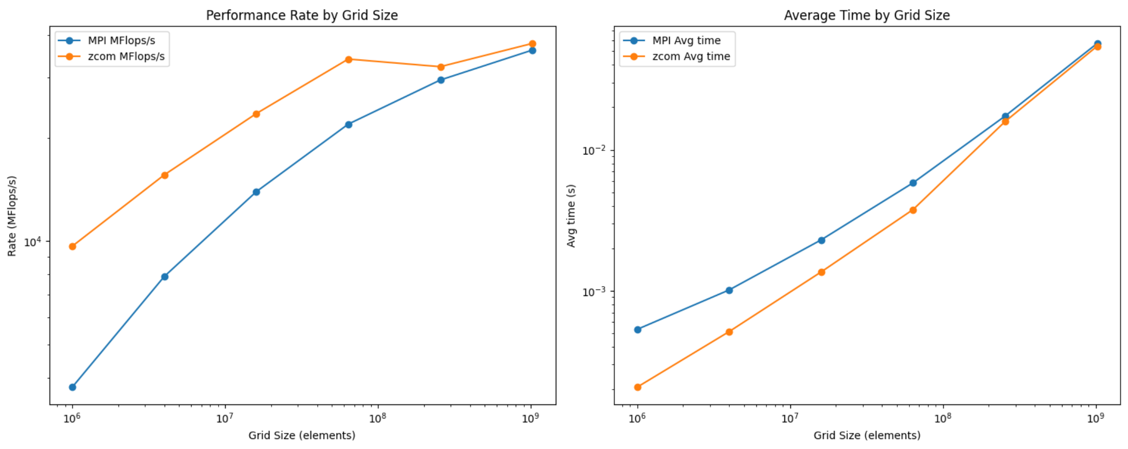

Figure 5.3: Results of the Sync_p2p kernel benchmark with higher Grid Size

Conclusion

From the analysis, Zcom is more efficient and scales better with the number of ranks, particularly notable at higher ranks. Although Zcom's performance at lower ranks starts off inconsistently, it quickly surpasses MPI's, showcasing its capability to handle large-scale parallel processing more effectively.

5.2.3 ZCom vs. OpenMP

Figure 5.4 contrasts the performance of the ZCom method with OpenMP across different numbers of ranks/threads using the Synch_p2p, focusing on computational rates (MFlops/s) and average processing times. The plotted data reveals:

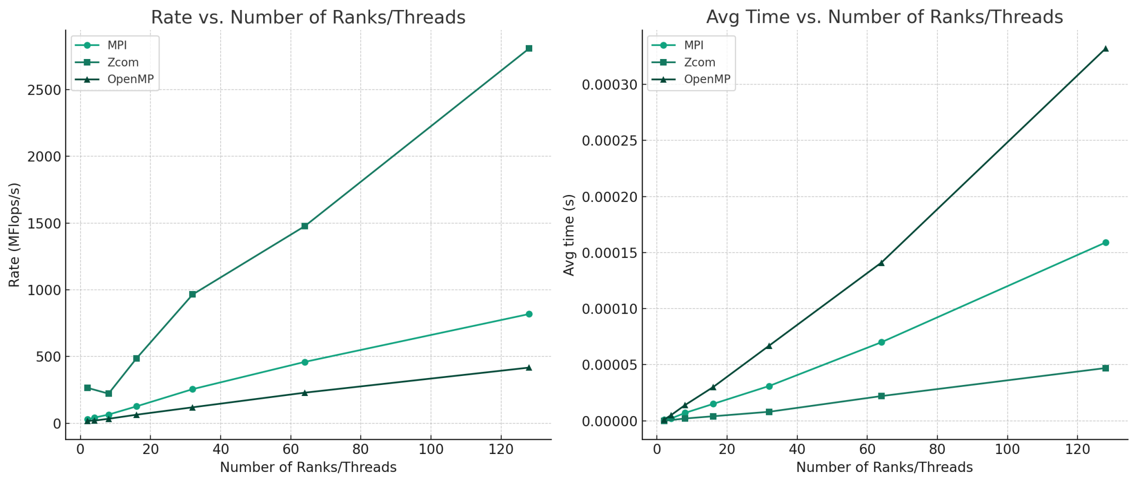

Figure 5.4: Sync_p2p kernel benchmark ZCom compared with OpenMP

- **Rate vs. Number of Ranks/Threads:** Certainly, ZCom and OpenMP exhibit a rise in the performance ratios with the increasing number of ranks/threads. Zcom starts at rank 2 at 265 MFlops/s and scales rapidly to 2808 MFlops/s at 64 ranks which shows that it has a very high scalability. However, OpenMP starts at 18.5 MFlops/s and scales up to 416.9 MFlops/s at 128 threads with more limited scalability compared to ZCom.
- **Average Time vs. Number of Ranks/Threads:** For both approaches, the average computation times reduce with the increase of the ranks/threads, hence, better parallelization efficiency. Despite differences the OpenMP times are in all cases higher than the ZCom times, starting from nearly four seconds at two ranks and rising only a little, to 0.00216 seconds at sixty-four ranks. The average times of the OpenMP, while starting quite close to each other at 2 threads, change more dramatically to 0.000332 seconds at 128 threads.

The ranks' evaluation of performances turns into a comparative ranking analysis of ZCom as such, its scalability and efficiency. ZCom is faster than OpenMP both in terms of computation speed and having low average processing times which facilitates its capability of performing high demand parallel processing tasks.

5.3 MiniGhost Benchmark

5.3.1 Mini Ghost Kernel

The miniGhost benchmark [BVH], that is one of the components of the Mantevo project [HDC+09], is a small-sized application designed exclusively for the study of boundary exchange mechanisms in scientific parallel computing ambits using stencil calculations. The central concern is the computational patterns of difference stencils, which are commonly associated with scientific computations of solving partial differential equations (PDEs) with numerical methods.

In the realm of parallel computing, these calculations often necessitate the division of the geographical area into smaller subdomains which, in turn, mandates the exchange of halo or border data between the subdomains commanded by different processes. This is in line with the Bulk Synchronous Parallel (Bulk Synchronous Parallel (BSP)) model, where communication optimization is achieved by packing the data together to reduce the number of messages sent, taking advantage of the bandwidth and latency of the inter-node communications.

miniGhost is a proxy which is used to analyze the performance and communication patterns of larger scientific applications on current and emerging high-performance computing architectures. It provides an opportunity to study numerous computational and communication models and to create new parallelization strategies and hardware features.

The tool can work in various modes which are: pure communication (interconnect stress testing) and integrated with computation, to emulate the real-world scientific applications' mode of operation. This flexibility is an important feature of miniGhost and makes it useful for performance profiling and investigation of the effects of different hardware and software configurations in a controlled, scalable, and reproducible manner [BVH].

Boundary Exchange Mechanism

The main purpose of the miniGhost is the emulation of boundary exchange, which is a significant part of parallel stencil computations. If the computational domain is split into several processors, every subdomain has to communicate with the edges or boundary data between the neighboring zones. Such a practice, critically important for the simulation's coherence over the whole computational domain, is called halo or ghost cell exchange. miniRADAR helps researchers try different approaches of this data exchange to maximize performance on different hardware configurations [BVH].

Performance Exploration

miniGhost evaluates effects of alternative hardware and communication strategies on stencil computation performance. It emulates the cognitive load of the applications that are more intricate and thus it enables investigators to separate and analyze the consequences of several architectural features and programming models on such computations [BVH].

Scalability Analysis

miniGhost allows users to test the scalability of different parallel processing architectures when carrying out stencil computations. This involves the research of messaging interfaces and communication protocols, how to improve data transfer between nodes and the tradeoff between computation and communication in a variety of settings [BVH].

Proxy for Larger Applications

An example of miniGhost as a proxy app is a quite good illustration of the operation of the more complex scientific code. miniGhost achieves this by modeling the important characteristics of these larger applications, enabling developers and researchers to predict how changes in computing architectures, software designs, or hardware configurations will impact the performance of full-scale applications [BVH].

Development and Testing Platform

This tool provides an environmental probe for fresh programming paradigms, message passing schemes, and architecture-specific enhancements. miniGhost can serve as a tool for developers to investigate various types of parallelism such as task decomposition, data distribution, and synchronization mechanisms, when finding the best solution for specific hardware configurations [BVH].

In general, miniGhost is a powerful toolkit, which provide the scientific computing community an opportunity to analyze and optimize the performance of stencil-based parallel computations. It acts to bridge the gap that exists between the performance models and the implementable performance on current and future computing systems [BVH].

5.3.2 MiniGhost Weak Scaling Benchmark

The weak scaling benchmark tests [BVH] the behavior of the ZCom API in terms of increasing the number of processes while keeping the work-load constant for process. This test is critical in analyzing the response of the ZCom API in terms of the efficiency of the system with respect to computation and communication scaling.

5.3.3 MiniGhost Strong Scaling Benchmark

On the other hand, the strong scaling benchmark citeminighost evaluates the ability of the ZCom API to decrease the execution time with the increase of processes when the total problem size remains constant. The findings show the benefits of ZCom in high-load situations where productive communication is vital for minimizing execution time.

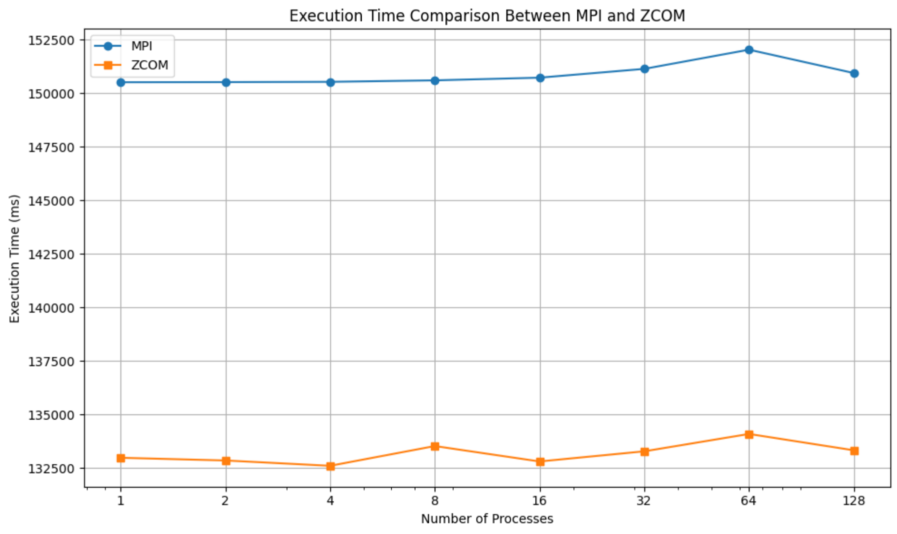

Figure 5.5: Weak scaling benchmark results comparing ZCom and MPI communication methods.

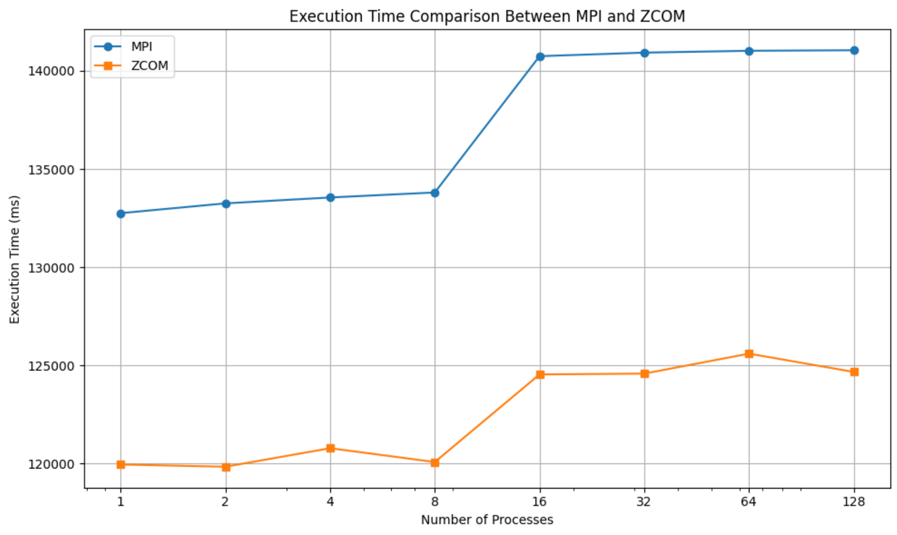

Figure 5.6: Strong scaling benchmark results highlighting the performance improvements of ZCom over standard MPI.

5.3.4 Communication Time and Performance Report

The communication time comparison between MPI miniGhost [BVH] version and ZCom provides ZCom with better performance in all aspects of communication: the operation of packing, shipping, delivery and unpacking data. A comprehensive evaluation of these findings proves the superiority of ZCom zero-copying in intra-node communications.

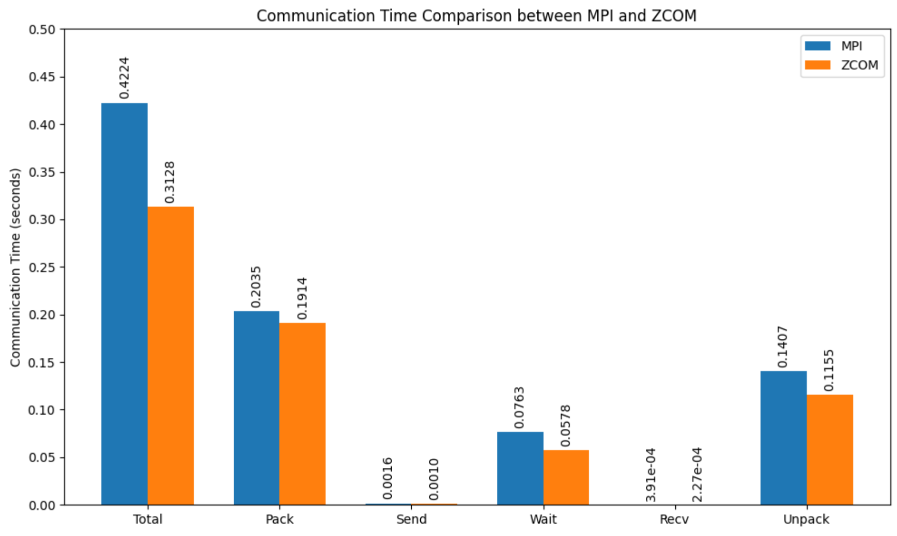

Figure 5.7: Communication time comparison between MPI and ZCom, showing the reduced overhead with ZCom.

In addition, the overall performance time comparison justifies the fact that ZCom is faster that traditional communication means, particularly with the increase in the number of processes. Comparison of execution times under varying numbers of processes shows the best scalability and effectiveness of ZCom.

5.3.5 Conclusion

The comprehensive performance evaluation using the MiniGhost benchmarks underscores ZCom's strengths in handling large-scale parallel processing more effectively than traditional MPI communication methods. ZCom showcases benefits in both weak and strong scaling scenarios, cementing its position as a efficient approach in the realm of high-performance computing, particularly for applications that require frequent and intensive communication among processes.

6 Conclusion

The goal of this was to improve the intra-node communication in HPC systems by introducing a zero copy API. Using XPMEM technology, ZCom provides a significant improvement of data transfer effectiveness in shared memory environments by removing the extensive data replication that is common in the classical MPI-based approaches. Performance assessments made with microbenchmarks and MiniGhost benchmark application indicate that ZCom can resolve communication overhead, enriching scalability and performance of HPC systems.

6.1 Discussion of Limitations

With an evaluation of ZCom API, it is noticed that various technical constraints are identified that may affect its universal usage and functionality in various high-performance computing environments. In particular, ZCom requires some hardware features that could be missing on old systems and make it difficult to use in the legacy network. In addition, there although ZCom has a number of performance benefits in latency reduction and scalability, these benefits are likely to decrease for large number of nodes. The complexity in both operating and deploying ZCom is also a major challenge, especially for simplicity and stability systems. Furthermore, complexity of the approach and the necessity of its constant support can scare away those users who like simple, well-documented solutions. ZCom is not only advanced over communication efficiency in high performance computing but also allows the consideration of research directions in the future. This is the quest for the creation of adaptive algorithms that can sense and adapt to different hardware configurations and development of user interfaces which would facilitate easier integration and wider spread of the applications. Such improvements would compensate for the existing limitations and provide access to a new generation of multifunctional communications systems.

6.2 Counterarguments

Although the ZCom API provides increased benefits over traditional methods like MPI in certain cases, sometimes the traditional methods may still be the preferred choice. For example, in cases where the communication style is intermittent, or does not significantly affect the overall performance, the overhead of introducing a zero-copy API may not be worth the change. Stability and predictability of such environments make traditional methods real winners with their time tested reliability and the broad support ecosystem. Additionally, traditional communication methods have become mature and hence a level of credibility that ZCom and other new technologies are still striving to achieve. Traditional approaches are provided with extensive documentation and community support, which are a treasure trove of resources that ZCom lacks at this moment. In addition, the financial and logistical impacts of new technologies such as ZCom that may need massive capital flows in new hardware or software cannot be also forgotten. These characteristics make traditional methods suitable for many organizations, especially those that are working with very tight budgets. In considering such aspects, it is obvious that ZCom though, offering hopeful prospects to the development of HPC communications, but it is also confronted with serious difficulties and skepticism that has to be overcame. This recognition not only increases the peer-review countenance of the research but also reveals the complexity of the choice of suitable communication technologies in different computing environments. Through considering such limitations and counter-arguments, this study acknowledges the complex compromises HPC development is entwined with, offering a balanced view crucial for the discipline's ongoing evolution.

6.3 Future Research

Directions of research are diverse in the future and provide a large number of possibilities of further improvement of ZCom API. The immediate steps could include enhancing the evaluation of ZCom for a larger number of HPC systems representing various types of computational architectures and application domains. This would give a better perspective of universality of an API and its performance in different computational environments.

Besides that, the interoperability of ZCom with the latest progresses of HPC technologies such as top-of-the-line GPUs and new interconnects will be crucial. Such integration attempts will allow for the assessment of how the API can evolve and function, thus establishing a strong argument on how data-intensive activities will be handled in the upcoming computing environments.

As well, ZCom itself could be enhanced to be more (in)compatible and suitable (employable) for standard HPC workflows. Its compatibility and user-friendliness will be improved, which would also make it more popular among the HPC community. A hybrid communication model which takes advantage of the zero-copy property of ZCom as well as the traditional MPI model is likely to offer a versatile and highly efficient solution which suits the HPC applications of the current era with its diversity.

6.4 Final Remarks

Essentially, the ZCom API represents a qualitative improvement in node communication efficiency in HPC systems. The importance of ZCom can be expected to increase as computational requirements grow and efforts to achieve performance optimization continue. This research success paves the way to a higher computational power which reflects a path that High Performance Computing may take in the future.

Bibliography

[BVH] R. Barrett, C. Vaughan, M. Heroux. *MiniGhost: A Miniapp for exploring boundary exchange*. URL: https://www.spec.org/accel/Docs/miniGhost.v1.0.pdf (visited on 02/18/2024) (cit. on pp. 55, 56, 58).

[CCP+10] B. M. Chapman, T. Curtis, S. S. Pophale, S. W. Poole, J. A. Kuehn, C. Koelbel, L. Smith. “Introducing OpenSHMEM: SHMEM for the PGAS community”. In: *International Conference on Partitioned Global Address Space Programming Models*. 2010. URL: https://api.semanticscholar.org/CorpusID:11871925 (cit. on p. 34).

[For94] M. P. Forum. “MPI: A Message-Passing Interface Standard”. In: (1994) (cit. on p. 18).

[GM13] B. Goglin, S. Moreaud. “KNEM: a Generic and Scalable Kernel-Assisted Intranode MPI Communication Framework”. In: *Journal of Parallel and Distributed Computing* 73 (Feb. 2013), pp. 176–188. DOI: 10.1016/j.jpdc.2012.09.016 (cit. on pp. 27, 29).

[Hag11] G. Hager. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011 (cit. on pp. 25, 26).

[Has19] J. e. a. Hashmi. *Design and Characterization of Shared Address Space MPI Collectives on Modern Architectures*. 2019. URL: https://jahanzeb-hashmi.github.io/files/talks/ccgrid19.pdf (visited on 02/18/2024) (cit. on p. 28).

[HB11] M. Hübner, J. Becker. *Multiprocessor system-on-chip: Hardware design and tool integration*. Springer New York, 2011, pp. 1–270. ISBN: 9781441964595. DOI: 10.1007/978-1-4419-6460-1 (cit. on p. 17).

[HCB+18] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, D. K. Panda. “Designing Efficient Shared Address Space Reduction Collectives for Multi-/Many-cores”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 1020–1029. DOI: 10.1109/IPDPS.2018.00111 (cit. on pp. 31, 32).

[HDC+09] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, R. W. Numrich. *Improving Performance via Mini-applications*. Tech. rep. SAND2009-5574. Sandia National Laboratories, 2009 (cit. on p. 55).

[Hje] N. Hjelm. *xpmem @ github.com*. URL: https://github.com/hjelmn/xpmem (visited on 02/18/2024) (cit. on pp. 28, 29, 37).

[HSG+18] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, Y. Ishikawa. “Process-in-process: techniques for practical address-space sharing”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’18. Tempe, Arizona: Association for Computing Machinery, 2018, pp. 131–143. ISBN: 9781450357852. DOI: 10.1145/3208040.3208045. URL: https://doi.org/10.1145/3208040.3208045 (cit. on p. 32).

[JRS16] J. Jeffers, J. Reinders, A. Sodani. “Chapter 16 - PGAS programming models”. In: *Intel Xeon Phi Processor High Performance Programming (Second Edition)*. Ed. by J. Jeffers, J. Reinders, A. Sodani. Second Edi. Boston: Morgan Kaufmann, 2016, pp. 369–382. ISBN: 978-0-12-809194-4. DOI: https://doi.org/10.1016/B978-0-12-809194-4.00016-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128091944000168 (cit. on p. 24).

[JSCP05] H.-W. Jin, S. Sur, L. Chai, D. Panda. “LiMIC: support for high-performance MPI intra-node communication on Linux cluster”. In: July 2005, pp. 184–191. ISBN: 0-7695-2380-3. DOI: 10.1109/ICPP.2005.48 (cit. on pp. 27, 29).

[Lab] I. Labs. *Parres/kernels: This is a set of simple programs that can be used to explore the features of a parallel platform*. URL: https://github.com/ParRes/Kernels (visited on 02/18/2024) (cit. on pp. 50, 51).

[Mes23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (cit. on pp. 23, 25).

[MPI-1.1-95] *MPI: A Message-Passing Interface Standard*. 1995. URL: https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html (visited on 02/14/2024) (cit. on p. 18).

[MPI-2-97] *MPI-2: Extensions to the Message-Passing Interface*. 1997. URL: https://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/mpi2-report.html (visited on 02/14/2024) (cit. on p. 18).

[MPI-3-12] *MPI: A Message-Passing Interface Standard Version 3.0*. 2012. URL: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (visited on 02/14/2024) (cit. on p. 18).

[MPI-3.1-15] *MPI: A Message-Passing Interface Standard Version 3.1*. 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 02/14/2024) (cit. on p. 19).

[MPI-4-21] *MPI: A Message-Passing Interface Standard Version 4.0*. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (visited on 07/20/2021) (cit. on p. 19).

[MPICH] *MPICH | High-Performance Portable MPI*. URL: https://www.mpich.org/ (visited on 02/14/2024) (cit. on p. 18).

[Open MPI] *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (visited on 02/14/2024) (cit. on p. 18).

[OS1] *MPI topic: One-sided communication*. URL: https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-onesided.html (visited on 07/20/2021) (cit. on p. 24).

[OS2] *One-Sided Communication*. URL: https://docs.oracle.com/cd/E19061-01/hpc.cluster6/819-4134-10/1-sided.html (visited on 02/18/2024) (cit. on p. 24).

[PF12] M. Passing, I. Forum. "MPI : A Message-Passing Interface Standard". In: (2012) (cit. on pp. 19, 20).

[Rab] R. Rabenseifner. *One-sided communication and the MPI shared memory*. Available at: https://fs.hlrs.de/projects/par/mooc/mooc-2/mooc2-week3-4.pdf. (Visited on 02/18/2024) (cit. on p. 30).

[SOH+98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. 2nd. (Revised). Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262692155 (cit. on pp. 17, 19–22).

[SPH+15] M. Si, A. J. Peña, J. R. Hammond, P. Balaji, M. Takagi, Y. Ishikawa. "Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures". In: *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 665–676. URL: https://api.semanticscholar.org/CorpusID:11659715 (cit. on p. 33).

[SVL+15] 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, 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 (cit. on pp. 27, 28).

[SVP+14] P. Shamis, M. G. Venkata, S. Poole, A. Welch, T. Curtis. "Designing a High Performance OpenSHMEM Implementation Using Universal Common Communication Substrate as a Communication Middleware". In: *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*. Ed. by S. Poole, O. Hernandez, P. Shamis. Cham: Springer International Publishing, 2014, pp. 1–13. ISBN: 978-3-319-05215-1 (cit. on pp. 45, 46).

[TRH00] J. Traff, H. Ritzdorf, R. Hempel. "The implementation of MPI-2 one-sided communication for the NEC SX-5". In: *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. Dec. 2000, pp. 1–1. ISBN: 0-7803-9802-5. DOI: 10.1109/SC.2000.10023 (cit. on pp. 23, 24).

[TV17] A. S. Tanenbaum, M. Van Steen. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017. ISBN: 1543057381 (cit. on p. 17).

[Van16] R. Van der Wijngaart. *The parallel research kernels, a tool for parallel systems*. 2016. URL: https://www.nas.nasa.gov/assets/nas/pdf/ams/2016/AMS_20161013_VanDerWijngaart.pdf (visited on 02/18/2024) (cit. on p. 52).

[Vie14] J. Vienne. "Benefits of Cross Memory Attach for MPI libraries on HPC Clusters". In: *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. XSEDE '14. Atlanta, GA, USA: Association for Computing Machinery, 2014. ISBN: 9781450328937. DOI: 10.1145/2616498.2616532. URL: https://doi.org/10.1145/2616498.2616532 (cit. on pp. 27, 29).

[Wal92] D. W. Walker. "Standards for message-passing in a distributed memory environment". In: (Aug. 1992). URL: https://www.osti.gov/biblio/7104668 (visited on 02/14/2024) (cit. on p. 18).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature