An Efficient and Lightweight OpenSHMEM Implementation

George Kalyvianakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

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

Thesis Advisor: *Associate Professor Polyvios Pratikakis*

Thesis Supervisor: *Dr. Nikolaos Kallimanis*

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

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

University of Crete
Computer Science Department

An Efficient and Lightweight OpenSHMEM Implementation

Thesis submitted by
Georgios Kalivianakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Georgios Kalivianakis

Committee approvals: _____
Dr. Nikolaos Kallimanis
Thesis Supervisor

_____
Polyvios Pratikakis
Assistant Professor, Thesis Advisor

_____
Kwnstantinos Magoutis
Associate Professor, Committee Member

_____
Vassilios Dimakopoulos
Associate Professor, Committee Member

Departmental approval: _____
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, March 2022

An Efficient and Lightweight OpenSHMEM Implementation

Abstract

The High Performance Computing (HPC) is rapidly gaining momentum, relying on the benefits of the Partitioned Global Address Space (PGAS) model for optimal results. Numerous languages and libraries have been introduced that leverage the PGAS model, with the most widely known being OpenSHMEM. OpenSHMEM is a standard specification that introduces a one-sided RDMA capable API for extensive use in HPC.

In this thesis we design and implement Gmem, an OpenSHMEM implementation supporting TCP/IP, RoCE and Infiniband networking backed by GSAS, a very lightweight PGAS API allowing processes spawning on a number of nodes to communicate in very similar way to shared memory schemantics. Gmem leverages shared memory for intra-node communications enabling users to fully utilize spacial locality without involvement of the OS or the network adapter. With RDMA we are also able to perform operations on remote nodes with extremely low latency and high throughput.

We evaluate G-Mem with the OpenSHMEM implementation of OpenMPI and MPICH that rely on the Unified Communication X (UCX) framework, for TCP/IP and Infiniband. In our tests we assess the performance of PUT/GET remote memory operations, several atomic memory operations and collectives operations. We find that our implementation is not only on par with our competitors but in some cases we even achieve greater results. In GET operations, for large size transfers we achieve 6x lower latency than OpenMPI, and in Atomic operations 1.25x better latency than OpenMPI.

Τίτλος

Περίληψη

Ευχαριστίες

Στους γονείς μου

Contents

List of Tables

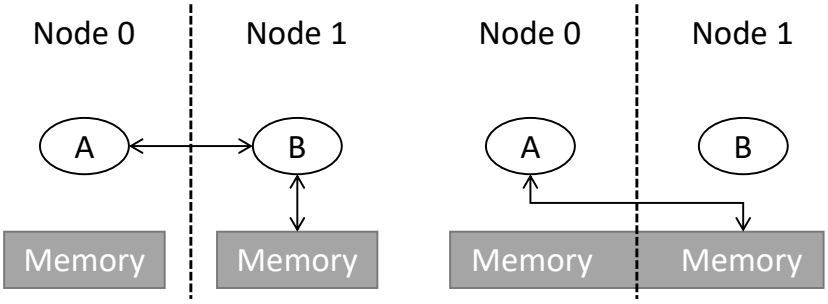
List of Figures

Chapter 1

Introduction

High Performance Computing (HPC) aims to utilize all the computational resources of a cluster to meet the needs of cutting-edge applications. HPC's foundations revolve around three main axis, namely computational power, fast network connectivity and storage. This is mainly achieved by bundling computers together to form a cluster, allowing them to fully utilize their individual computation capabilities by exchanging data between them almost as fast as a computer can access its local resources. Since computational power per processing core has limitations on how fast they can advance such as Moore's law and Dennard's scaling, the focus of HPC has shifted on adding more computational nodes and focusing on efficient and fast networks to exchange data.

Initially, the dominant programming model to utilize an HPC cluster's capabilities was the distributed memory model. By utilizing such a programming model, several processes are spawned on different processing cores of a single or more computational nodes in a network. Additionally, the distributed model follows the Single Process - Multiple Data (SPMD) paradigm, where several processes run the same executable. In the SPMD paradigm, each process receives a unique identifier on start, allowing the execution path of processes that execute the same file to diverge. In the distributed memory model each process has its own set of variables residing in its memory, but has no knowledge of the status of its peer's memory. As depicted in Figure 1.1(a), in order to access or modify data residing on a remote process, it has to communicate with the process in a two-way communication i.e. one remote process needs to transmit and another to receive. This two-way communication is mainly done through message passing protocols. Message Passing Interfaces (MPI), introduce APIs that handle the initialization and the networking aspects in an application, effectively allowing the programmer to focus solely on their application. However, these APIs have some limitations. For example, not all data types and user defined structures can be exchanged directly, requiring some sort of serialization and de-serialization [23] due to the APIs restrictions. Moreover, the two-sided nature of MPI forces users to implement mechanisms in their code to handle requests and data transmissions, as well as to



(a) SPMD two-sided memory accesses

(b) PGAS one-sided memory accesses

Figure 1.1: Differences in one/two sided communications in SPMD and PGAS In Figure 1.1(a) process A can only access Node's 1 memory through process B. In Figure 1.1(b) process A can directly access Node's 1 memory.

ensure proper timing, which introduces a bigger surface for bugs and inefficient code. These limitations led to the introduction of the Partitioned Global Address Space (PGAS) [5] model.

The PGAS programming model is very similar to MPI as a concept, but with key differences being the memory model and that its communications are one-sided. In the PGAS model, a number of processes are spawned on a single or more nodes, running the same application. Again, the SPMD paradigm is utilized but in this case, the processes are aware of remote shared memory segments. Whenever a PGAS application is initialized, each process receives information about its peers shared memory segments locations in the form of remote pointers. By utilizing these pointers, a process is able to interact with remote memory, i.e. Read/Write to it, without intervention from the remote process as depicted in Figure 1.1(b). This is possible because the memory of the application is split in even segments and distributed among every node. Each process is responsible for a segment and benefits from very fast accesses to it. At the same time, the remote pointers that a process possesses, allow for one-sided remote memory operations on its peers memory. This effectively means that with a simple GET and PUT API, programmers are able to write intuitive code that can freely access or modify any shared variable as easily as it would modify a local variable, without special structures and network functions. This leads to the main factor affecting the PGAS performance being the network communications and more specifically the latency of operations. Lower latency means highest throughput of operations. The one-sided nature of PGAS strongly amortizes this cost but for optimal performance the underlying network must be chosen properly.

Currently, numerous PGAS languages [7] and API implementations [1, 2, 10, 17, 19, 21] have been developed, that support different underlying networking protocol and machine architectures. However, this introduces the problem of non-portable code and inconsistency [20] in runtime behavior between the different

implementations. The need for a standardized API arose that bridges the gap between different implementations, leading to the conception of the OpenSHMEM [6] standard. The OpenSHMEM standard aims to designate a general PGAS API, with strict definitions of functions and behaviors per function that will serve as a guideline for any implementation. While an implementation can be tailored to a specific hardware platform or underlying network infrastructure, as long as it follows the OpenSHMEM specification it can execute any program that uses the OpenSHMEM API.

In this thesis we introduce Gmem. Gmem is an OpenSHMEM implementation on top of the GSAS environment [9, 14, 15, 16] a minimal, light-weight PGAS API. GSAS enables processes running on different nodes to allocate and de-allocate memory on any participating node and interact with that memory with a very simple API. Remote processes participating in GSAS are not communicating directly with each other but instead they make use of a service, called atomic service. An instance of the atomic service is present in each node and serves as an end point for requests from either local or remote processes. The atomic service is responsible for serving requests from processes. These requests include but are not limited to managing the memory of a node, i.e. handling allocations and de-allocations, and serving Read and Writes towards that memory. When an atomic service allocates memory, that memory is made available to local processes via shared memory. This allows a local GSAS process to directly interact with local parts of the global address space directly. For remote memory operations, a GSAS process has to communicate its request to the atomic service of a remote compute node. Upon receiving this request, the atomic service will either write or read and return the proper data.

Aside from memory allocation and Reads and Writes, GSAS offers a variety of more sophisticated functionality. Specifically, it allows for forking of remotes processes, as well as performing atomic memory operations and synchronization functions. GSAS allows the forking of processes in a dynamic fashion at any time of an application's life cycle. Additionally, GSAS provides a set of atomic primitives that includes a minimum amount of atomic operations such as, Compare and Swap (CAS), Atomic Swap (SWAP) and Fetch and Add (FAD).

This thesis is split in two parts. The first part is the design and implementation of Gmem. Table 1.1 presents an overview of the supported functionality in our implementation. Based on the OpenSHMEM specification, the memory of a process is split in local and shared memory segments. All global, static and shared allocated variables are part of the shared memory and need to be mirrored in each process and remote accessible in a one-sided communication fashion. Based on the OpenSHMEM specification, each process needs to have two shared memory segments. The first contains all the global variables of a program and is named symmetric memory, while the second is dynamically allocated when a shared allocation operation is called and is named symmetric heap. These segments need to be known to all participating processes at any time. To ensure this, upon start of execution, a process allocates the required amount of memory from its local atomic

service and proceeds to memory map (MMAP) this memory over its initialed and uninitialized global data segments. Once this is completed, processes exchange a pointer to this memory in an all-to-all fashion. By sharing the pointer to this shared memory, any remote process can execute a one-sided Put or Get operation on the shared data (i.e. to a global variable) of any process. In order to deliver the one-sided communication, Put and Gets, we leverage the atomic service of GSAS. When a process wants to perform a Put/Get operation, it communicates its request to the responsible atomic service. Since the shared memory of each process is allocated from the atomic service, upon receiving a request, the atomic service can freely interact with the memory of every local process without interrupting or synchronizing with it. The symmetric heap allocation works in the same manner as the symmetric memory.

OpenSHMEM Functionality Availability		
Categories	Gmem	OSHMPI
Library Setup	Available	Available
Thread Support	N/A	Available
Memory Management Routines	Available	Available
Communication management Routines	N/A	N/A
Remote Memory Access Routines	Available	Available
Non-blocking Remote Memory Access Routines	Available	Available
Atomic Memory Operations	Available	Available
Collective Routines	Available	Available
Point-To-Point Synchronization Routines	Available	Available
Memory Ordering Routines	Available	Available
Distributed Locking Routines	Available	Available
Cache Management	N/A	Available

Table 1.1: OpenSHMEM API Availability in Gmem and OSHMPI

In addition to the shared memory and Put/Get functionality, our implementation also provides more sophisticated OpenSHMEM functionality, namely collective operations point to point synchronization and critical region locking mechanisms. The collective operations include barriers, one-to-all and all-to-all communication functions. For the barriers, we implement our own protocols that leverage the new atomic primitives we introduced in addition to the underlying topology in GSAS, allowing us to deliver a fine tuned solution to our needs. In the case of the rest of the collectives such as broadcast or collect operations, we utilize our Put/Get functionality along with our highly efficient barrier to deliver the requested functionality achieving good performance. Our collective algorithms have been verified via the Synch [13] framework. The Synch framework provides a shared-memory oriented testbed for the implementation and testing of concurrent algorithms. By porting and testing our algorithms in Synch, we are able to validate their functionality, verifying their results as well as clearing them of serious logical errors such as deadlocks. Finally, in regards to the locking mechanisms offered by OpenSHMEM, we have implemented a ticket-lock algorithm that abides by all the requirements of the standard, namely a first-in-first-out property as well as fairness.

In order to accommodate for the requirements of OpenSHMEM atomic operations part of this thesis work has been to extend the existing functionality of GSAS. Specifically, OpenSHMEM supports all atomic bitwise operations, in both fetch i.e. return the result, and non-fetch modes. In addition, for all atomic primitives, both 32b and 64b words are supported. GSAS on the other hand, only supports the Compare and Swap(CAS), Fetch and Add(FAD) and atomic SWAP operations, for 64b words. Due to GSAS fairly minimal code base, we have enriched its functionality to support all aforementioned atomic operations. We have also added support for CAS and SWAP that transmit back the result value of the operation instead of a boolean result status that GSAS provides.

Due to the nature of one-sided communications of the OpenSHMEM, the underlying networking setup plays a major role in its performance. In the past, the Ethernet protocol has been used extensively in HPCs. Ethernet relies on its time tested TCP and UDP transport protocols. TCP provides out-of-the-box support for all sorts of applications while UDP serves as a base for the development of custom network protocols, specifically tailored to an application's needs. Due to its low-cost hardware that is already available on every machine as well as its maturity it has been the go-to solution. However, despite its constant evolution, it still suffers from limitations due to its immense coupling to operating systems and backwards compatibility that hinder both its throughput and latency, two of the most important factors for an efficient cluster. In a typical TCP scenario, the transmitting process has to generate a kernel interrupt, so that data can be copied to the network adapter and transmitted. On the receiving side, the adapter will receive the data, store it and interrupt the kernel. Afterwards, the kernel will notify the proper process, which will copy the data from the adapter to its own

virtual space. The kernel interrupts in combination with the memory copies contribute to increased latency until data from the transmitting process are available to the receiving end. Furthermore, there is a significant cost in CPU cycles to do so. This is why state-of-the-art systems have begun shifting away from TCP/IP to different Remote Direct Memory Access (RDMA) enabled network architectures.

RDMA is a zero-copy technology protocol. Zero copy means that an application is able to communicate directly with the network adapter, without any kernel intervention. This allows it to save and utilize CPU cycles in a more useful manner, minimizing latency and maximizing throughput. Numerous network protocols have emerged that implement RDMA, with the most widely known being: Internet Wide Area RDMA Protocol (iWARP), RDMA Over Converged Ethernet (RoCE), and Infiniband. Both iWARP and RoCE aim to deliver RDMA capabilities over traditional Ethernet through TCP/IP. In order to do so, special NICs are required that enable hardware support for RDMA over Ethernet. The main difference between the two lies on their underlying transport protocol. While both operate on top of Ethernet, iWARP works through TCP while RoCE works through UDP. Infiniband relies on its own hardware and network stack to deliver the RDMA capabilities. Due to its strictly RDMA oriented approach, it is able to mitigate overheads and bottlenecks caused by the TCP/IP affecting the performance. Although it achieves better performance than iWARP and RoCE, it also has its drawbacks. Infiniband code differs from traditional socket programming, meaning that there is a steep initial learning curve. In addition, Infiniband hardware is not readily available like typical TCP/IP NICs and upgrading existing infrastructure can be a costly task.

This brings us to the second part of this thesis work, which is the introduction of Infiniband support in GSAS. The reason behind this is that the GSAS original design only supported RDS sockets on top of the TCP/IP protocol for its communications. The RDS protocol is slowly becoming obsolete and also comes with security flaws making it disabled by default in most production operating systems. In addition to this, the RDS sockets are meant to run over Infiniband and while it is capable of running on top Ethernet, it does so by emulating the Infiniband environment resulting in significant impact on its performance. GSAS was designed to provide very low latency for small messages of fixed size. Based on our observations, supporting only small messages has a negative performance in our OpenSHMEM implementation. For this reason, we also design and implement a mechanism, namely BULK Transfers, that extends the functionality of GSAS with two different transmission functions that are used based on the size of a transfer. More specifically, when a small amount of data need to be transmitted, our mechanism uses inline packets, while for larger sizes we use regular packets that support data transfers of up to the Maximum Transmission Unit(MTU) size.

To conclude, in this thesis we introduce Gmem, an OpenSHMEM compliant implementation. Gmem is built on top of GSAS, a minimal PGAS API in which we introduce Infiniband support. Gmem is based on the version 1.4 of the OpenSHMEM specification. We currently support almost every feature of the standard,

with the exception of communication contexts and multi-threaded operations. We verify our implementation with tests supplied by the official OpenSHMEM github repository [18]. The OpenSHMEM foundation offers a test suite that can be used to explore the coverage of a potential implementation as well as to verify that it works as intended. For our evaluations, we compare our implementation against OSHMPI [24], the most commonly used and accessible OpenSHMEM implementation. OSHMPI can be backed by either OpenMPI or MPICH, and we compare against both, using both TCP/IP and Infiniband as our underlying communication protocols. In the cases of Infiniband, GET operations achieve 6x times lower latency. We also achieve 1.25x times better latency in atomic memory operations for remote operations.

The rest of the thesis is structured as follows: Section 1.1 contains an overview of related work to this thesis. Section 2 provides in-depth information regarding our work.Finally, in Section 3 we present our work's evaluation results.

1.1 Related Work

Cray OpenSHMEMX [22] is a proprietary OpenSHMEM implementation by Cray Inc. Cray was the first to offer an implementation of the SHMEM model, which later evolved in the OpenSHMEM. It has also played a major role in the shaping and evolution of the OpenSHMEM standard. Cray's internal SHMEM API is specifically designed around DMAPP [4], a customized communication library used for their in-house developed Gemini and Aries architectures. OpenSHMEMX aims to bring their SHMEM API up to code with the OpenSHMEM standard. The latest versions offers full compliance with the 1.4 version of the standard. OpenSHMEMX is designed in a modular fashion that allows fine-grained tuning based on the underlying hardware, networking protocols and applications needs.

Nvidia has its own OpenSHMEM implementation namely NVSHMEM [12]. NVSHMEM is tailored specifically for CUDA enabled Graphics Processor Units (GPU). Traditionally, GPUs use CUDA and MPI. The CPU utilizes MPI for communications while the GPU is handling the actual computation. This introduces overheads caused by CPU usage and latency due to CPU to GPU communication. NVSHMEM's PGAS spans across GPUs on the cluster enabling direct communication from one GPU to another with close to zero CPU involvement. By using special GPU-Kernels (not to be confused with the Kernel in the operating system) a GPU has interleaving communication and computation. This enables GPUs to handle both communication and computation without the involvement of the CPU. Also, the asynchronous one-sided communications implemented per the OpenSHMEM specification allow for optimal communication between the GPU threads, significantly boosting throughput and lowering latency, while also reducing code complexity.

OSHMPI [11] implements the OpenSHMEM standard based on the MPI-3

specification. The MPI-3 specification, introduces improved one-sided communication compared to its predecessors. It defines a more strict memory model, with better ordering guarantees as well as remote synchronization allow for more optimized communications between processes. OSHMPI can be backed by both OpenMPI and MPICH, two of the most popular opensource MPI implementations. Both variations can run on top of TCP/IP. However in the case of OpenMPI, the OpenSHMEM implementation can only be used if the Unified Communication X (UCX) framework is present. The MPICH variation also requires the UCX framework but only when the user runs on top of Infiniband. While the end user experience is the same, the architectures of OpenMPI and UCX are very complex making it extremely difficult to improve or implement additional features in the implementation. Moreover, both OpenMPI and UCX accept their own set of parameters, which most often than not result in collisions that can negatively affect the performance, making fine tuning a very daunting task.

GASPI [3, 10] is an open source PGAS API, implemented in C, C++ and Fortran, offering the same benefits of OpenSHMEM without being an OpenSHMEM compliant implementation. It is centered around scalability, flexibility and fault tolerance. GASPI's one-sided data transfers produce a notification on the remote side upon completion, allowing users to develop a highly customized code of computation and communication. It also fully supports RDMA features. It also abolishes conventional symmetric memory, and opts for a more fine tuning model where the user can choose where and how the segments of the Global address space will be spawned. Timeouts of remote operations allow for the discovery of potential failures while its dynamic modification of the active node sets, allow for the recovery from failures.

Chapter 2

Architecture

In this thesis, we introduce Gmem, an implementation of the OpenSHMEM standard. Our implementation is built on top of the Global Address Space(GSAS) shared memory framework. GSAS follows a Partitioned Global Address Space (PGAS) shared memory model. A PGAS model provides the participating processes with a virtual global address memory space, partitioned evenly between different nodes. GSAS provides an API that the participating processes use to interact with the global address space memory, as well as with each other. This API provides functionality to allocate, read, and write memory in any of the nodes, allowing processes to store data in such a way that memory locality can be fully utilized. GSAS provides a slim software stack in contrast to other state-of-the-art alternatives such as OpenMPI and MPICH, which include numerous software dependencies. Additionally, the OpenSHMEM implementation part of this thesis introduces Infiniband networking support in GSAS

The architecture section is organized as follows:

1. Section 2.1 presents an overview of GSAS.
2. Section 2.2 explains how we introduce Infiniband communications in GSAS.
3. Section 2.3 explains how we implement the OpenSHMEM Standard in GSAS.

2.1 GSAS Architecture

GSAS is a centralized framework, meaning that every node that is part of it, needs to have the Atomic Service (AS) running. The Atomic Service is responsible for managing the memory segment of GSAS in a node, such as the allocation and de-allocation of memory, as well as forking processes running a GSAS application. Figure 2.1 shows an overview of GSAS. The instances of Atomic Service are able to communicate with each other via the underlying network. In addition to this they can also receive requests directly from a GSAS process. This allows a process to request data from a memory partition of the global address space residing in a

remote node. Moreover, when an instance of Atomic Service allocates a new block of memory, that block is also made available to all of its local processes through shared memory. This allows processes to directly retrieve data residing in their local node without having to receive the data through the network stack. In this case the data are transmitted to the requesting process by a simple memory copy. The operations provided by the GSAS API are running in user-level, thus avoiding any kernel involvement, allowing for low latency and fast communication. The performance bottleneck of these operations usually lies on the underlying network used for the communication. The addition of Infiniband (IB) support in GSAS networking protocols, in contrast to the currently supported TCP/IP stack via Reliable Datagram Sockets (RDS) or User Datagram Protocol (UDP), gives the ability to utilize a modern and fast network. Infiniband is an open standard interconnect protocol aiming to provide low latency and high throughput. The main difference between standard TCP/IP models and IB is that IB is through its Remote Direct Memory Access (RDMA) semantics. RDMA allows a remote process to transmit data across a network by talking directly to the network adapter, without having to utilize operating system, thus limiting kernel-level calls to a minimum. This reduces the network latency in GSAS boosting its performance.

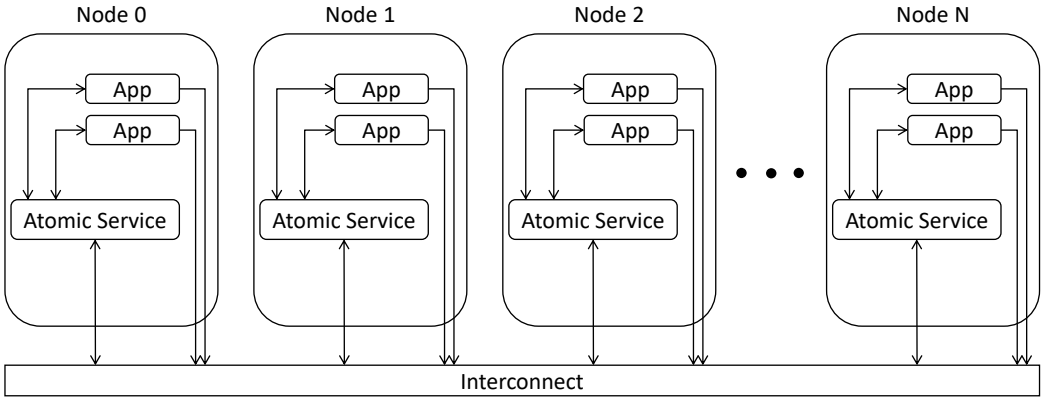


Figure 2.1: Overview of GSAS architecture.
Each application presented utilizes GSAS. Applications running in any node, are able to communicate with both the remote Atomic Services and all GSAS application processes.

2.1.1 GSAS Extended Functionality

GSAS original design provides support for the atomic primitives of 64b words. In addition, the supported atomic primitives are: Compare and Swap (CAS), Fetch and Add(FAD) and atomic SWAP. Moreover, in the CAS and SWAP operations, GSAS functionality only provides support for returning a boolean value for the status of the operation and not the actual value. The OpenSHMEM implementation supports all atomic primitives, which include fetching/non-fetching bitwise operations, and based on the API requirements, CAS and SWAP need to return

the value to the caller. For this reason we have extended GSAS functionality with all the required primitives. In GSAS, when an atomic primitive is called, the caller process prepares a packet containing the type of operation, the address of the variable to be changed and the desired value, and transmits this packet to the desired atomic service. When the atomic service receives this packet, it deciphers the address and proceed to perform the necessary operation. For the atomic primitives, the gcc atomic builtins are used. To that end, we have introduced new type headers for the atomic packets, and enriched the atomic service in order to properly handle requests.

2.2 Infiniband

Prior to any IB communication, a process needs to setup an IB context containing all the required resources for the RDMA operations. As depicted in Figure 2.2, once a process calls the initialization functions, the network adapter will proceed to create and return a collection of resources back to the process. This collection consists of three queues. The first two namely, Send Queue (SQ) and Receive Queue(RQ) are paired together in a construct named Queue Pair (QP). These two are paired together due to their very similar operation. The third queue is named Completion Queue Entry (CQE). Once this initialization is complete, the process can then start posting Work Queue Elements on the adapter. WQEs contain a buffer allocated on the process memory space, that will be used in order to either receive data from a remote node or send its data to a remote node. Upon receiving these elements, the adapter will put them in the proper queue of the QP. In the case of a receive WQE, when the adapter receives a packet addressed at a specific process, it will place the received data on the buffer pointed by the first available WQE in the RQ and consume it. In a send WQE, the adapter will fetch the buffer location containing the data to be transmitted from the first available WQE of the SQ and then transmit them. The WQEs are served in a first in first out (FIFO) order. When a WQE is consumed, a new element is generated and placed on the CQE. Once a process has posted a WQE, it can only track its progress though polling the CQE. For security reasons, a process has to initialize a Memory Region (MR) that is locked with a cryptographic key pair. These MRs need to be tied to a specific QP. Any memory that the process will be posting as WQEs in a QP need to be part of the MR tied to that QP.

Similar to traditional networking models such as the TCP/IP protocol stack, the Infiniband architecture also has its own notion of the transport layer. In the TCP/IP protocol, an end user can choose whether to make use of connection or connectionless communication by either TCP or UDP transport modes respectively. Infiniband also supports the concept of a connection based communication like TCP as well as simple datagram transmission like UDP, which is incorporated in QPs. Infiniband provides different types of QP configurations, each with its own properties called Service Types. When a QP is setup, it must be configured

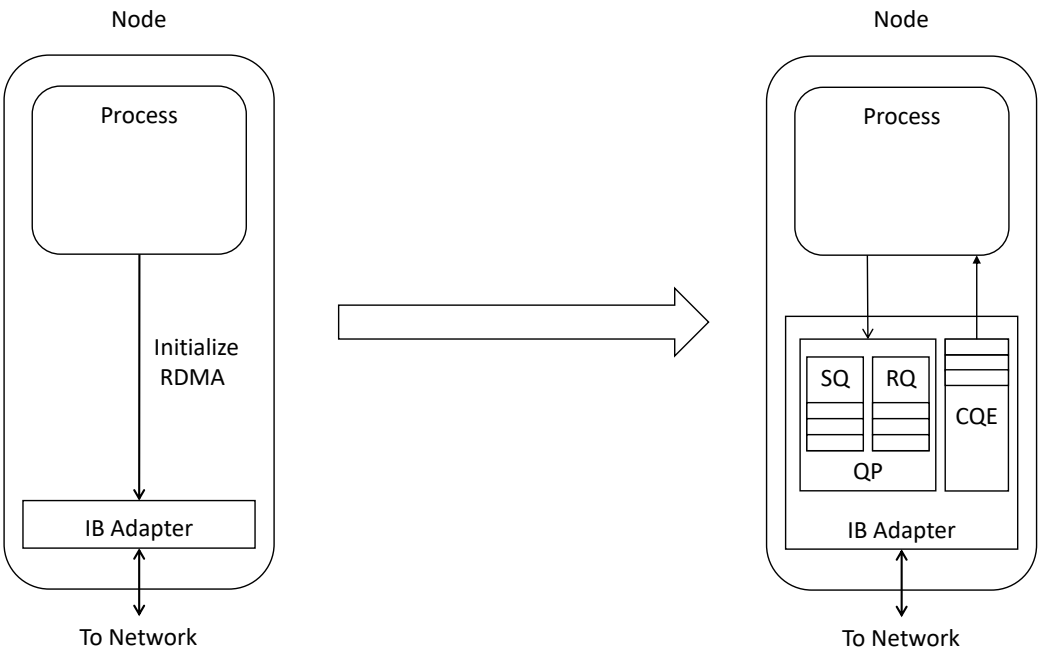


Figure 2.2: Infiniband Initialization

about what type of connection it will support. Only one type is allowed per QP. These types are organized based on two fundamental attributes, connectivity and reliability. An end user can choose from any permutation of the two such as: Reliable Connection (RC), Unreliable Connection(UC) and Unreliable Datagram(UD). Table 2.1 presents an overview of the properties for each service Type. Based on GSAS requirements, in our implementation we make use of the RC type.

Addressing in IB is a bit more complex than the traditional TCP/IP stack. Instead of a port and an IP address, IB uses a bundle of several information to route packets. Each network adapter has a unique identifier named Global Identifier (GID), and each port of the adapter has a unique identifier named Local Identifier (LID). Moreover, each QP has its own unique Queue Pair Number (QPN). The address for a send consists of all three aforementioned fields as well as the public key for the MR of the QP. In addition to this, before any transmission takes place both sides need to choose a Packet Sequence Number (PSN). The PSN works similar to TCP's sequence numbers and is used for capturing potential loss of packets and re-transmitting said packets. Due to the fact that the QPN, the MR keys and the PSN are available at runtime before a connection is set up, these data need to be exchanged between two hosts prior to any data transmission and can not be setup in advance. There are two possible ways of exchanging this information. The first one is through the subnet manager provided by IB router hardware. The second one is using an out of band communication, from a secondary network such as TCP/IP. Since IB hardware is expensive and might not be always available, we chose to use the later, and implement a mechanism that works through the usage

Properties	Services		
	Reliable Connection	Unreliable Connection	Unreliable Datagram
Errors Detected	YES		
Packet Delivery Guaranteed	YES	NO	NO
Packet Retransmission	YES	NO	NO
Packet Ordering	YES	NO	NO
Error Recovery	YES	NO	NO
Is Alive Check	YES	NO	NO
Connection Setup Required	YES	NO	NO

Table 2.1: Infiniband Service Types

of TCP/IP's RDS sockets in order to exchange this information and enable IB communication.

In the RDS implementation of GSAS, each node has a specific IP and different ports starting from a common initial number and increment upwards is used for addressing. For example, if Node 0 has the IP xxx.xxx.xxx.1, then the atomic service will listen on that IP and port 9000, while the first GSAS app will listen on the same IP and port 9001 etc. In the IB specification we use this IP and Port combination in as a side channel communication mechanism. The side-channel exchange of IB routing information is implemented as a thread called MBOX Manager (MBM) in every GSAS application including the Atomic Service. This thread starts running before any communication takes place to and from any process. The Mbox Manager is responsible for setting up connections with remote processes as well as bookkeeping information on every connection established. This information is stored in a hashtable. This hashtable receives as key the combination of IP and Port of the remote process and as data a struct containing information for each connection. Similar to the TCP/IP handshake, we implement a three step protocol for the initial setup of the IB communications. In our protocol we define and use 3 types of packets, signaling each step. The three steps are namely INIT CONNECTION, START CONNECTION and ACK START.

An example of MBM's operation is as presented in Figure 2.3. In this example we have two Nodes 0 and 1 with IPs A and B respectively, each belonging on the same subnet. An instance of the Atomic Service is running in each of the nodes and listens on port 9000. Also some GSAS is running (let it be M), and it listens on Port 9001 in node 1. In this scenario process M wants to communicate with the atomic service in node 0. M enters the INIT CONNECTION step, initializing its IB resources. It's MBM then sends a CREATE CONNECTION with all the required IB information to the MBM of atomic service listening on IP A and port

9000. Upon receiving this message, the MBM of the atomic service in node 0 stores it in a hashtable using as key the source IP and Port (in this case B + 9001) and initializes its own IB resources. Once this step is complete it replies to the sender an ACK START packet. This packet contains its own IB information. The initiator then finalizes its IB initialization with the missing information and stores the information for IP A and port 9000 to its own hashtable. Once this is complete, M is now able to transmit IB messages to the AS0. This procedure needs to occur every time a process tries to communicate with a remote process for the first time. If prior communication has taken place between two processes, they can retrieve the required information from their personal hashtable. If the information on the hashtable is outdated for a specific IP and Port key, then the sender will try to transmit, but will receive an unsucessful CQE. This signals that the process that was previously listening on that specific IP and Port key is no longer reachable. In this case it will clear its information from the hashtable and re-run the MBM protocol to retrieve the new info.

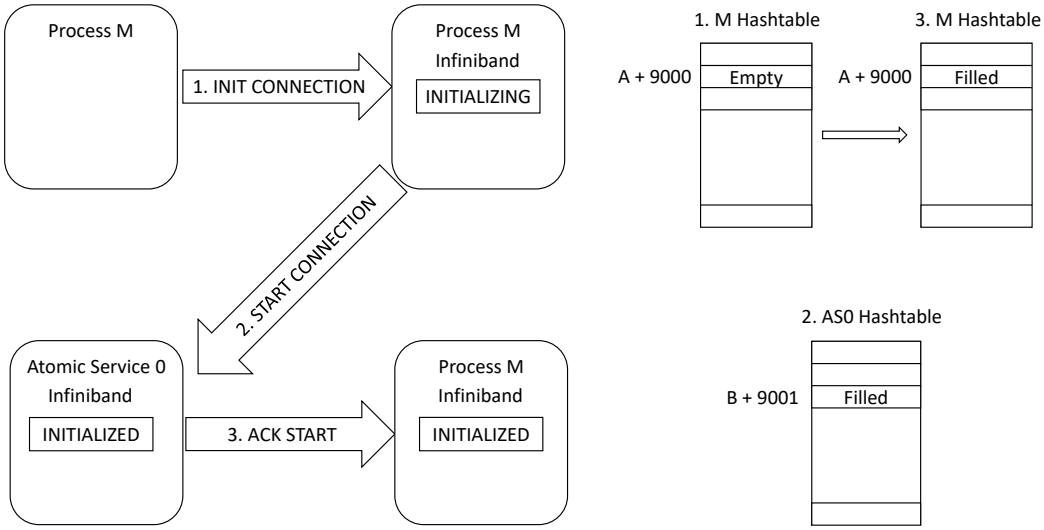


Figure 2.3: MBox Manager Protocol

2.2.1 GSAS Communication

In order to access remote segments of the PGAS memory space, GSAS processes need to communicate with remote nodes. This communication includes special control packets that are responsible for utility operations such as the allocation and de-allocation of memory, as well as forking a new GSAS process in a remote node. In addition to that, GSAS divides accesses to remote memory in three distinct categories, writes, reads and atomics. The GSAS API provides the WRITE and READ functions for writing and reading to and from remote areas. Any GSAS process that wants to modify a remote memory segment has to communicate with

the atomic service responsible for that segment. The atomic service operates similarly to a server, always looking to serve an incoming request. Due to legacy reasons the request packet that it expects has a fixed size of 32 Bytes. These 32 Bytes are split in 4 unsigned long integers. The structure of the control packets is depicted in Figure 2.4. The first integer contains the packet's header. The header includes information about the source of the packet and its TYPE. The TYPE indicates what kind of operation this packet signals. The rest 3 integers are used for the payload. While this is the structure of packets that the atomic service is waiting for, GSAS allows bigger size packets to be transmitted. However, these large packets have the limitation that they can only be transmitted unidirectionally, from an atomic service to an application. This introduces the problem that, while an atomic service can transmit packets as long as the network allows, an application can only transmit packets of 32B to the atomic service. For example, when an application performs a READ operation, it sends a control message to the appropriate instance of atomic service with the TYPE: READ and with payload the address and size that it wants to read. The atomic service then proceeds to send back the data in large packets, minimizing the number of packet that will be sent. However, in the case of WRITE operations an application has to transmit the entirety of the data in packets with only 24 bytes as the payload. This introduces heavy performance overheads.

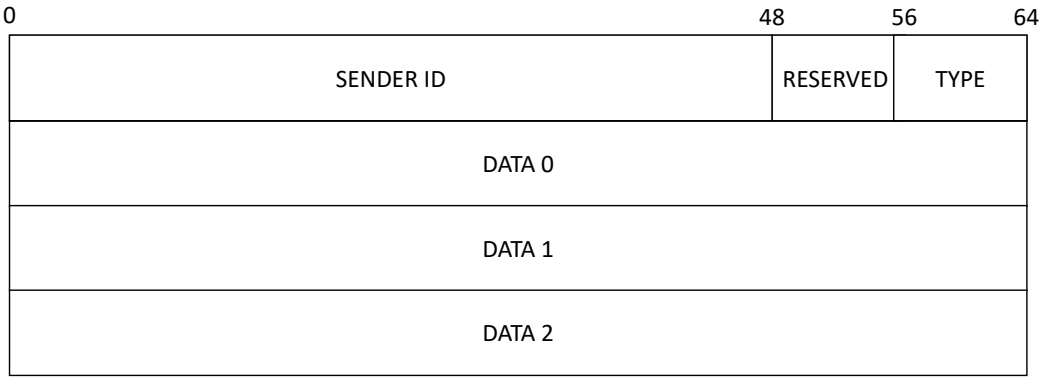


Figure 2.4: GSAS Control Packet Structure

In contrast to instances of atomic service, processes of a GSAS application can receive packets of either 32B or 64B size. In both cases the packet is split in two segments, the header and the payload. The header has a constant size of 8B, leaving the rest available for the *payload*. Transmitted packets with size greater than 32B are only allowed from an atomic service to an application. When an application wants to conduct a READ, it sends a control packet to the appropriate atomic service. This packet contains the type READ as well as the address and *length* of the data it requests. Upon receiving such a packet, the atomic service proceeds to send back the requested data, fragmented in packets of at most 64B size, with a payload capability of 56B. The total number of packets required by

a READ is calculated by the following equation: $1 + \lceil length \ / \ payload \rceil$, where *length* equals the data size and *payload* indicates the payload available per packet. One packet is used for the control message and the remaining are used for the payload data. In a scenario where a process wants to read 1024B of data, the number of required packets would be 20 packets. Out of these, the first packet is the control that signals the initiation of the transmission and the remaining 19 are the packets required for the data transmission. In our IB implementation we are able to optimize GSAS's READ operations. To do so, we implemented a new GSAS mechanism called BULKREAD. In addition to the 32B and 64B that remain for legacy reasons, we also introduce packets that can have a size as big as the Maximum Transmission Unit (MTU) that the network's link supports. We call this packets, bulk packets. This significantly improves our *payload* amount as IB's MTU is 4096B. When an atomic service receives a control message with the BULK READ type, then that means that the requester is now expecting neither 32B nor 64B, but bulk packets. Following the same previous scenario the required number of packets now drops from 20 to 2. By lowering the amount of packets, we are able to better utilize our links capability. In addition to this, both the time an atomic service takes to serve a request and the time a process spends blocked waiting for a packet to arrive, are reduced significantly. This effectively means that an atomic service can serve a higher number of requests in the same amount of time. It also saves CPU cycles on an application that would previously be lost blocked on network I/O, that can be now utilized in a more useful manner.

2.3 Gmem Implementation

Any implementation of the OpenSHMEM[6] standard can be divided in the following four areas of interest:

1. Establishing the underlying communication between remote nodes.
2. Introduce an easy to use tool to quickly start an OpenSHMEM program from any node.
3. Setup the environment.
4. Implement various operations of OpenSHMEM.

Iterating over these briefly, first an underlying communication protocol between the processes participating in an OpenSHMEM program execution must be established. In our case we are building on top of GSAS, which enables communications between nodes through its own API. The next area is to be able to launch an OpenSHMEM application from any participating node. This is important because during this step we set the groundwork for the OpenSHMEM internals initialization, which are required upon the beginning of the OpenSHMEM code execution. Lastly, the two final components are the actual implementation of the OpenSHMEM API. We split these in two different parts, first the environment, which

consists of the library initialization like naming and addressing of each process as well as the memory mirroring features of OpenSHMEM called Symmetric Memory. Lastly, we implement the various API methods for Data transmission, synchronization and Atomic Memory Operations (AMO). Out of the 12 functionality categories specified in OpenSHMEM, our implementation does not provide: Thread Support, Communication Management Routines and Cache Management. Thread support is a limitation originating from GSAS. GSAS does not currently support multi-threaded transmissions between nodes. Cache management is officially deprecated in the OpenSHMEM specification and was thus not implemented.

The core behind any OpenSHMEM implementation lies within proper library initialization and the memory model setup. In accordance to the OpenSHMEM specification, each participating process is named Processing Element (PE). During initialization, each PE is assigned a distinct identifier called Rank. Ranks begin from 0 and increment linearly. The memory of each PE is split in two segments the Private Data, and the Remotely Accesible Symmetric Data (Symmetric Memory). The Private Data encapsulate all the local variables that are only accessible by each PE. These variables follow the traditional C memory model and consist of local function variables and any space allocated by the any of the allocation functions such as malloc, calloc and realloc. The Symmetric Memory on the other hand includes all of the global and static variables and any space allocated (Shared Allocated) by the OpenSHMEM defined allocation functions such as shmem_malloc. The difference between the two is that the symmetric memory is accesible by any other PE. This means that at any given moment, a global, static or Shared Allocated space is available on each PE and have the same name, space and type. Shared allocations reside in a memory region called Symmetric Heap. All shared communications in OpenSHMEM are one-sided in nature meaning that a Symmetric Memory variable might change at any time without the owner knowing. Remote modifications to a Symmetric Memory variable can only be executed through the proper OpenSHMEM library call. Prior to any API function call, the shmem_init function must called. Shmem_init is responsible for assigning Ranks to PE's and also set up the Symmetric Memory.

2.3.1 Library Setup

For proper execution and initialization of an OpenSHMEM program we implement the GSHRUN tool. This tool takes as arguments the number of PE's that participate and the desired executable to run with its arguments. GSHRUN uses the GSAS API for forking processes on the remote nodes. In GSAS, fork can also take as an argument a struct, namely shm_args that contains information available to the forked process. We allocate this struct in GSAS shared memory allowing any GSAS process to read and write to it. An overview of the shm_args struct is presented on Table 2.2. The id field is used by forked processes to acquire their Rank. The PE_slot array has as many elements as running PEs. This array is used in initialization and in collective operations when PE's need to exchange remote

addresses in GSAS memory space. The sync_barrier is used for synchronization between PE's. GSHRUN forks processes evenly between nodes, and assures that the Ranks assigned on a node's PEs are continuous. In a scenario with two nodes and 4 processes, GSHRUN will spawn 2 processes on each node, and node's 0 processes will get ranks 0-1 while node's 1 will get 2-3. In a scenario with 2 nodes and 5 processes, node's 0 will receive ranks 0-2 and node 1 ranks 3-4. When a process gets forked, it calls the shmem_init function. During this, each process will call a GSAS Fetch and Increment on the ID variable to receive their Rank. After receiving a Rank, each PE blocks on the sync_barrier. GSHRUN busy-waits on the ID variable knowing when to move to the next node. Once all PE's are forked the sync_barrier is released.

At this stage of the shm_init function, each process calculates the size it requires for its symmetric variables. All global and static variables in a C program are placed in the initialized and uninitialized data segments. In our implementation we leverage two symbols exposed by the linker, __data_start and _end. The addresses of these two, point to the beginning of the initialized data memory segment of a process and the end of the uninitialized data(bss) respectively. With the data size known, each PE allocates the required amount of memory from GSAS's Atomic Service. It then proceeds to mmap this space over the pages containing the global and static variables. This enables an Atomic Service to write directly to a variable without having to go through the PE, thus guaranteeing the one-sided communication property of the OpenSHMEM standard. Once this is complete every PE writes the address of the memory returned by GSAS to their respective PE_slot cell of the PE_slot array that resides on the shm_args struct, thus exchanging their new global segment address with the rest in an all-to-all fashion. Each PE stores this information in a hashtable using as key the rank for each PE and as data the address of said PE's global address space. Once all PE's have gathered the information of their peers, a barrier is used again for synchronization. After this barrier is released shmem_init finishes execution. Shmem_init is a costly operation, but this cost can be considered negligible as this function is run only once during a programs lifetime.

Struct shm_args	
Variable	Type
id	int
PE_slot	uint64_t
sync_barrier	GSAS_Barrier

Table 2.2: Overview of shm_args structure

2.3.2 Memory Operations

In order to utilize the Symmetric Memory space of OpenSHMEM, its API provides functions mirroring the malloc family of functions with the shmem_ prefix. In this

section we present an overview of the shmem_malloc and shmem_free functions. Shmem_calloc and shmem_realloc are build on top of malloc with minor additions.

As stated in 2.3, any space allocated on the symmetric heap needs to be available to all processes by the end of the shmem_malloc function. In the shmem_malloc implementation, the allocation is comprised of three steps. Synchronization between all PE's is required to transition between each step and for this reason the sync_barrier GSAS barrier is reused. In the first step all processes that call shmem_malloc execute barrier wait as a first instruction. The second step is the actual allocation of the memory, where all processes allocate memory from their node's Atomic Service. To support remote memory accesses by remote processes, we allocate memory through the Atomic Service. Once the memory is allocated, each process proceeds to write the pointer to this newly allocated memory on their respective PE_slot cell. The PE_slot array is part of the shm_args structure presented on 2.3. Finally, in the third step each PE stores its peers allocated memory. To do so, every PE reads the PE_slot array, and stores the information to a private hashtable.

2.3.3 Remote Memory Accesses

OpenSHMEM defines two different categories for remote memory access operations namely Remote Memory Access Routines and Non-Blocking Remote Memory Access Routines. The difference between the two is that the former requires a synchronization call to ensure the data are transmitted to the remote destination. In our implementation all remote access operations are made through GSAS READ and WRITE operations. GSAS follows a strict memory model in which, by the end of a remote memory operation the data are ensured to be available on the remote destination. This effectively means that in Gmem all the provided operations are blocking. The functions available for remote operations in OpenSHMEM are presented in Table 2.3. All of these functions receive the following arguments:

1. The Pe rank. This is the rank of the remote PE from which the data will be retrieved or transmitted to.

2. The nelems number of elements to be exchanged.

3. The dest address. In a case of Put this is the remote memory address the data will be send to. For a GET, this is the remote data to be retrieved.

4. Source address. In a GET operation, this is the local address where the data will be stored. In a PUT operation, this is the local data to be transmitted.

The nelems argument is not available in the single element PUT and GET operations. In Gmem whenever a remote memory operation is called, the caller searches the proper hashtable for the remote address using pe as key. Afterwards it calls the appropriate GSAS API function for the actual transmission.

Function	Description
shmem_put	Put a continuous block to remote
shmem_p	Put a single element to remote
shmem_iput	Put a strided block to remote
shmem_get	Get a continuous block from remote
shmem_g	Get a single element from remote
shmem_iget	Get a strided block from remote
shmem_put_nbi	Put a continuous block of memory without blocking
shmem_get_nbi	Get a continuous of block memory without blocking

Table 2.3: OpenSHMEM Remote Memory Operations

2.3.4 Atomic Memory Operations

In addition to regular remote memory operations, OpenSHMEM also allows for Atomic Memory Operations. The operations available to OpenSHMEM are presented on Table 2.4. Each of these operations combine a read and modification operation in a single step. Each of these functions perform a specific operation on a variable. This variable can be either local or remote. In the functions containing the fetch keyword, the value of the variable prior to the operation is also transmitted back to the caller. An atomic operation guarantees that the reading and modifying a variable will be done in a single step. This effectively means that an atomic operation will either be fully executed or no executed at all. After an atomic operation takes place and while the result is being transmitted back to the caller, a new operation may take place that alters the same variable. This is acceptable by the atomicity guarantees designated on the OpenSHMEM specification. Similar to the Remote Memory Operations, the atomic operations receive arguments that designate the target address to read/modify, the destination PE and a value or condition depending on the operation. The target PE and target address are used to retrieve the proper remote address from the caller's hashtable. Afterwards the caller calls the underlying GSAS API for the actual operation to take place. OpenSHMEM defines more atomic operations than the GSAS original design allowed. For this reason we have implemented the missing functionality in GSAS to support all of the OpenSHMEM operations. During a GSAS remote memory atomic operation, the caller sends a request to the Atomic Service responsible for the memory that will receive the modification. Upon receiving the request, the Atomic Service calls the appropriate GCC built-in atomic memory operation to conduct the operation on the designated memory and transmits the results back to the caller.

Atomic Memory Operations	
shmem_atomic_fetch	shmem_atomic_set
shmem_atomic_compare_swap	shmem_atomic_swap
shmem_atomic_fetch_inc	shmem_atomic_inc
shmem_atomic_fetch_add	shmem_atomic_add
shmem_atomic_fetch_and	shmem_atomic_and
shmem_atomic_fetch_or	shmem_atomic_or
shmem_atomic_fetch_xor	shmem_atomic_xor

Table 2.4: OpenSHMEM Atomic Memory Operations

2.3.5 Collective Operations

OpenSHMEM defines a category of operations called collective operations. This category contains functions that are used for synchronization of a specific set of PEs. This set is passed as argument to every operation. A collective operation cannot progress until all PEs in the set call it. OpenSHMEM's collective operations are presented on Table 2.5. In the following sections we examine each operation in depth.

Collective Operations	
shmem_barrier_all	shmem_barrier
shmem_sync_all	shmem_sync
shmem_broadcast	shmem_collect
shmem_fcollect	shmem_alltoall
shmem_alltoalls	shmem_and_to_all
shmem_min_to_all	shmem_max_to_all
shmem_sum_to_all	shmem_prod_to_all
shmem_xor_to_all	shmem_or_to_all

Table 2.5: OpenSHMEM Atomic Memory Operations

2.3.5.1 Barrier & Sync

Both shmem_barrier and shmem_sync have the same functionality of registering a PE at a barrier and awaiting until all PE's in the specified set call the operation. The difference between the two is that barrier ensures completion of previously issued remote memory operations while sync does not. GSAS implements a stricter memory model, by which all remote memory operations are completed and visible to the remote side when a remote call is made, thus in our implementation there is no distinction between the two. These operations are the core behind the rest collectives, since they are used for the synchronization of the PE's before a collective can run. Figure 2.5 demonstrates our shmem_barrier implementation. During library initialization each PE allocates a shared variable called barrier_slot residing

on its local node's Atomic Service and shares it with the rest of the PEs. Initially every barrier_slot is initialized to 0. For the barrier we implement an algorithm by which the first PE of the set is elected the leader of the barrier. The rest of the PE's atomically increment the leaders slot and spin on their own slot. This grants the benefit that the spinning on one's slot is done via shared memory and not through network communication, which would add significant overhead. The leader monitors his slot and when all participating PE's have arrived at it proceeds to unblock each PE by writing to their respective slots. The unblocking of PEs on the same node is done via shared memory, but for remote PEs network communication is required. With a high number of PE's in the set and given GSAS centralized nature where communication to a remote node is only available through the Atomic service this introduces heavy overheads. For this reason we further expanded the algorithm to also have sub-leaders. A sub-leader is a PE responsible for unblocking the PEs on its node, while the leader is responsible for unblocking the sub-leaders. The leader can also be a sub-leader. This makes the unblocking in a hierarchical fashion, with minimal remote communications. Shmem_barrier_all and shmem_sync_all functions operate in the exact same manner with the sole difference that instead of barrier_slot they use their own global_barrier_slot. This new variable is required to avoid overlapping modifications when a barrier call is followed by a barrier_all.

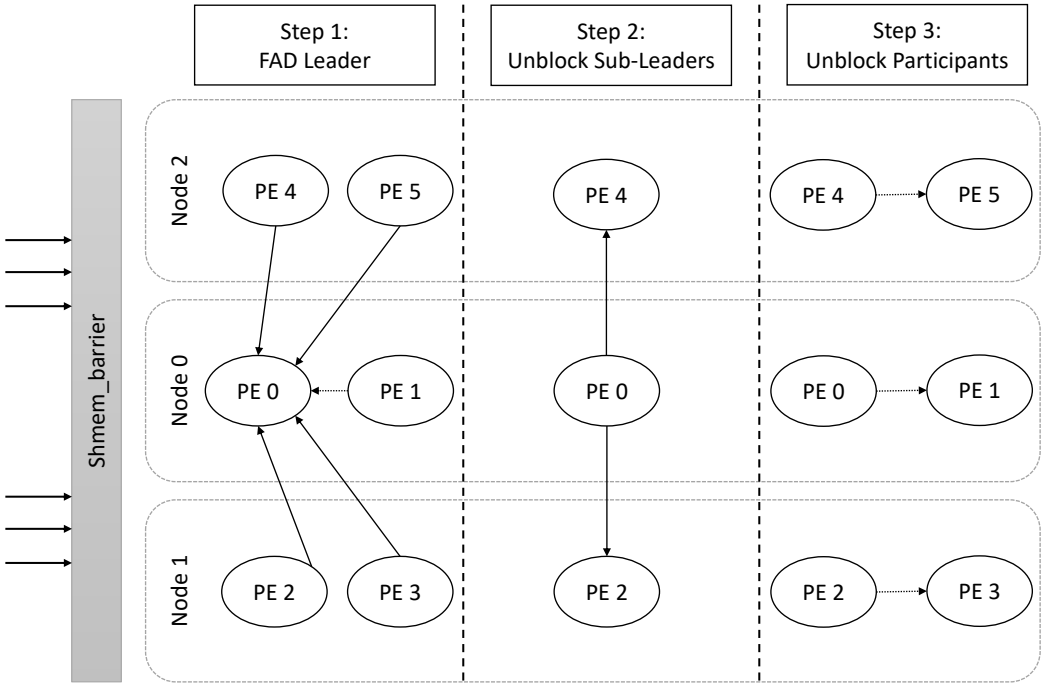


Figure 2.5: Gmem Barrier Operation
Shmem_barrier call with an active set of 6 PEs ranging from Rank 0 to 5. Straight arrows represent remote memory communication, while dashed arrows shared memory communication.

2.3.5.2 Broadcast

The shmem_broadcast method allows the transmission of data from a PE to all PEs in a designated set. Since this is a one-to-many transmission we make use of our the hierarchical approach used in shmem_barrier explained on 2.3.5.1 in order to avoid excess network communication. In a broadcast, we designate the PE that will be broadcasting as the leader PE and the PE with the lowest rank in each node as a sub-leader. The leader can also be a sub-leader. Each PE has an shared variable, allocated through GSAS called barrier_slot initialized to 0. These slots are accesible from PEs of the same node through shared memory accesses, and to remote PEs through network operations. During a broadcast each PE performs an Atomic Increment operation on the barrier_slot of the leader and proceed to spin on their own barrier_slot to unblock. The leader monitors this variable and when all PEs are present, it first broadcasts the data to the sub-leader of each node and then releases them. In turn, the sub-leaders propagate the data to the PEs in their node via shared memory and unblock them. This hierarchical approach minimizes network traffic and reduces load on the Atomic Services.

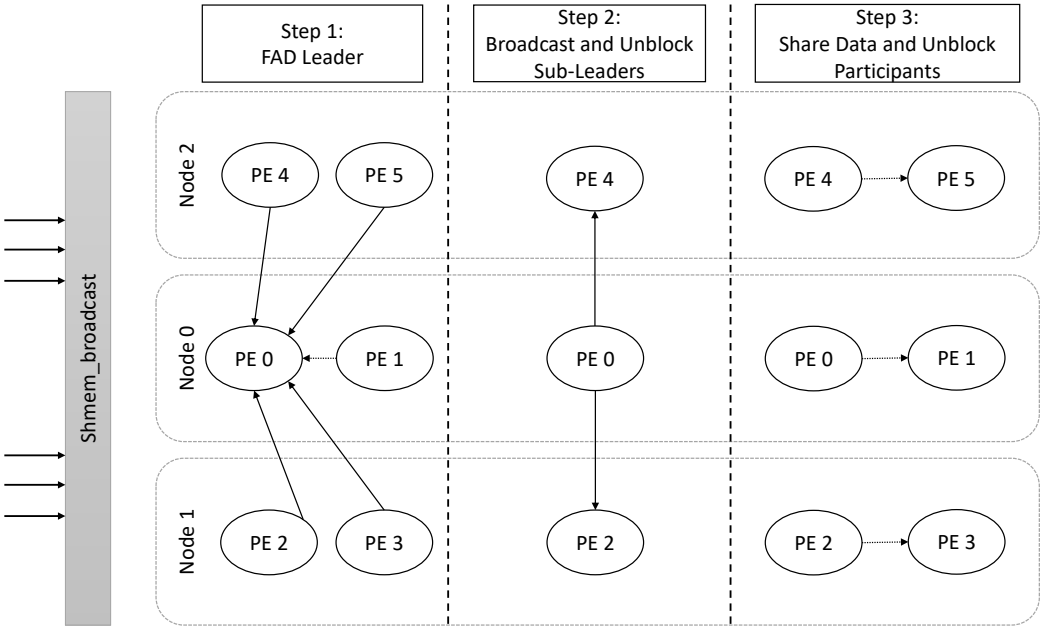


Figure 2.6: Gmem Broadcast Operation
Shmem_broadcast call with an active set of 6 PEs ranging from Rank 0 to 5. Straight arrows represent remote memory communication, while dashed arrows shared memory communication.

2.3.5.3 Collect & fCollect

The shmem_collect and shmem_fcollect functions concatenate block of data from all PEs of an active set to an array in each PE. The difference between the two is that fcollect requires the data to have the same size in all PEs while in collect each PE can have different data sizes. Since this is an all-to-all operation we can no longer benefit from our hierarchical approach. Both functions operate in three steps. Figure 2.7 presents an overview of the collect function. Fcollect operates in a similar manner. In the first step, when a collect or fcollect call is made, all PEs block on a barrier. This is required so that all participating PEs start running on the same time. In the second step, the barrier is released and all PEs start transmitting the data to the destination buffer on the proper offset on each PE. Lastly on the third step a second barrier call is required to ensure that all transmissions have concluded before finishing the function call.

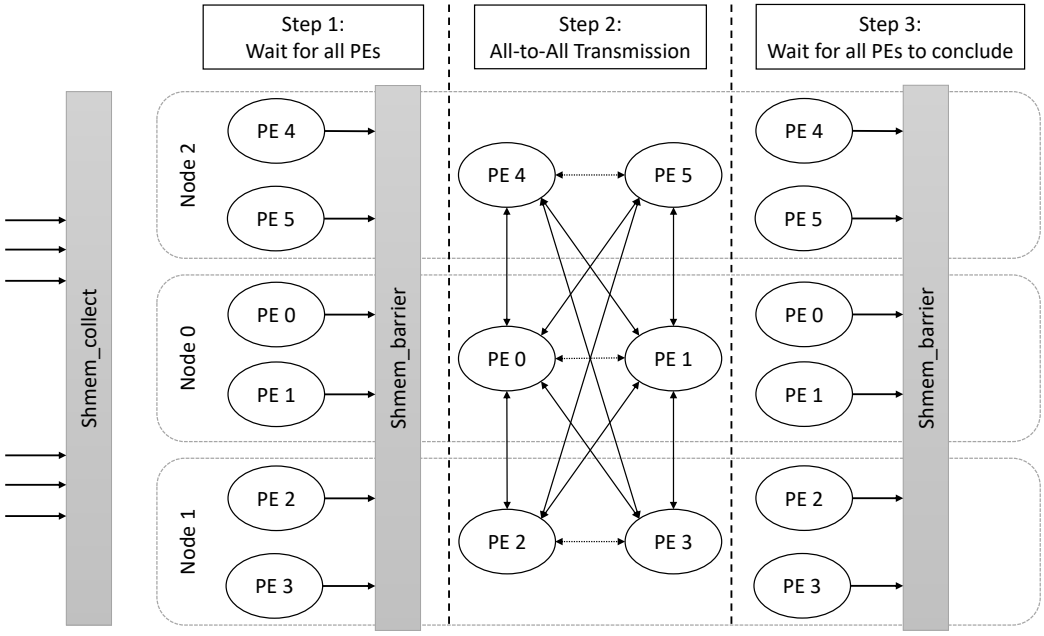


Figure 2.7: Gmem Collect Operation
Shmem_collect call with an active set of 6 PEs ranging from Rank 0 to 5.
Straight arrows represent remote memory communication, while dashed arrows shared memory communication.

2.3.5.4 Reduce

OpenSHMEM defines a set of collective operations used in data reductions across PEs in an active set. All of the reduction operations are identical in our implementation with the sole difference of the actual operation commenced. The operations available are presented in Table 2.6. The functions receive the following arguments: the dest and source arrays, the nelems and the active set. The source array provides the address from which it will retrieve the data to be reduced. The dest array contains the address where the reduced results will be stored. Nelems is used to indicate the number of elements that will be reduced. Figure 2.8 displays an overview the reduce operations. On entry, reduce operations synchronize all PEs using a barrier call. When all PEs sync at the barrier, everyone retrieves the data from the rest PEs and performs the reduction storing the result in their local dest array. In order to ensure that all PEs have finished the reduction we use a second barrier before the function call is finished.

Collective Ruduction Operations	
shmem_and_to_all	
shmem_min_to_all	shmem_max_to_all
shmem_sum_to_all	shmem_prod_to_all
shmem_xor_to_all	shmem_or_to_all

Table 2.6: OpenSHMEM Atomic Memory Operations

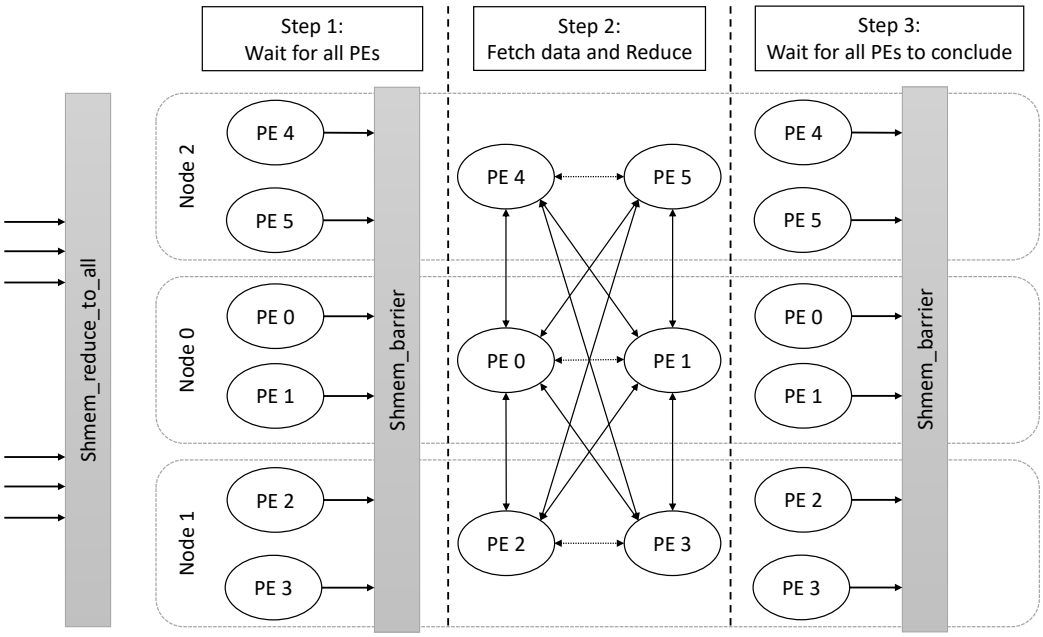


Figure 2.8: Gmem Reduce Operations
Shmem_collect call with an active set of 6 PEs ranging from Rank 0 to 5. Straight arrows represent remote memory communication, while dashed arrows shared memory communication.

2.3.5.5 AlltoAll & AlltoAlls

The shmem_alltoall and shmem_alltoall allow PEs to exchange a fixed amount of data with the rest PEs of a specific active set. The two functions operate in a similar manner with the difference that while alltoall exchanges a continuous block of data, alltoalls exchanges a strided block. The two functions operate in an all-to-all fashion. Figure 2.9 displays how alltoall operations work. When either function is called, all PEs need to be synchronized using a barrier call. This ensures that all PEs have their data ready for the exchange. Once all PEs have arrived at the barrier, they proceed to retrieve the data from every PE in the set. The retrieval of the data can be done in two ways. A PE can either write or read the data to/from the participating PEs. In our implementation we choose to read the data. This is because the architecture of GSAS does not provide optimized PUT operations. Finally, when all PEs have exchanged the data, a second barrier is required to ensure that all PEs have finished before finishing the function call.

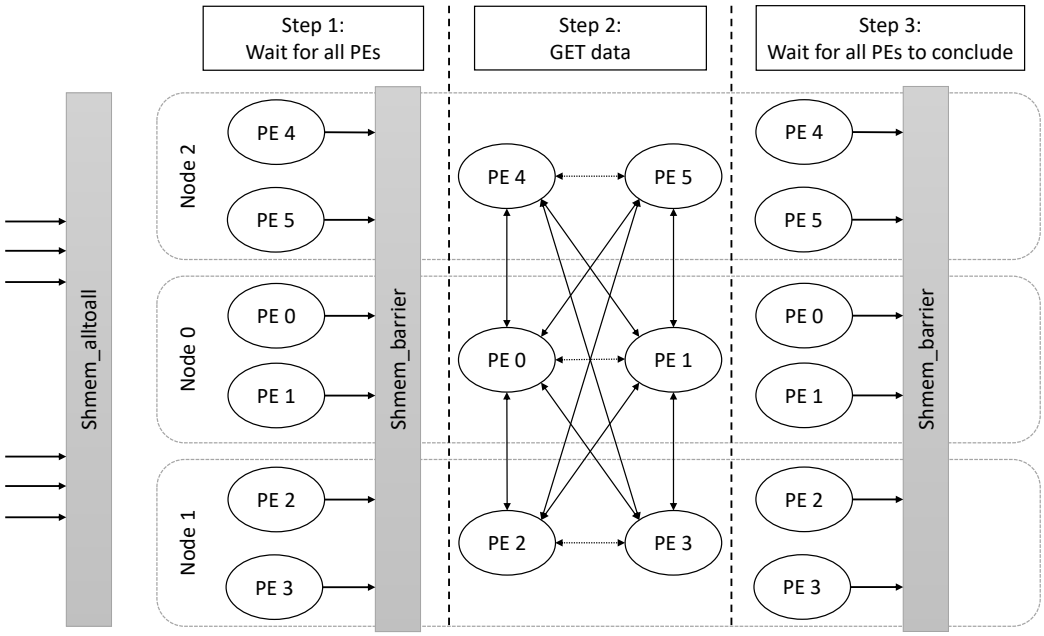


Figure 2.9: Gmem AlltoAll Operations
Shmem_alltoall call with an active set of 6 PEs ranging from Rank 0 to 5. Straight arrows represent remote memory communication, while dashed arrows shared memory communication.

2.3.5.6 Point-To-Point Synchronization

Given OpenSHMEM's one-sided communication nature, a PE is oblivious on the status of its symmetric variables. At any point in time a remote PE might modify a shared variable on any PE. OpenSHMEM defines two functions that allow PEs

to check whether or not such an event has occured. These two functions are namely: shmem_wait_until and shmem_test. Both functions receive as arguments the address of the variable to be compared, a compare operator, and a comparison value. Both functions operate similarly, with the difference that Shmem_wait_until blocks until the requested variable has changed according to the compare operation and value specified, while shmem_test does not. Due to GSAS strict memory model, changes on remote hosts will be available immediately on the remote side. This allows the implementation to operate in the same manner as it would in traditional memory.

2.3.5.7 Memory Ordering Routines

OpenSHMEM provides two functions, namely shmem_quiet and shmem_fence that affect the ordering of remote operations. Fence is repsonsible for ensuring the ordering of remote operations. For example a Put operation is guaranteed to finish before a future one, if a fence call is made between the two. Quiet on the other hand waits until all previously issued remote operations are finished. In our implementation we make use of GSAS for all remote operations. GSAS follows a strict memory model, by which most memory operations are guaranteed to be performed in-order. Due to this, in our OpenSHMEM implementation both shmem_fence and shmem_quiet are not required and thus exist as no-op functions.

2.3.5.8 Distributed Locking Routines

Due to the distributed nature of OpenSHMEM, a locking mechanism is defined to enable mutual exclusion. A shared lock is defined that PEs can acquire in order to enter critical regions. The specification dictates that the lock can be any symmetric variable of type long. Additionally, PEs trying to acquire the lock will be served in a first-come, first-served manner (FIFO) in order to prevent potential starvation of PEs. The lock API is presented in Table 2.7. All functions receive as argument the address of the lock variable of type long. Shmem_clear_lock releases a previously acquired lock. Shmem_set_lock blocks until it acquires a lock. Shmem_test_lock tries to acquire a lock without blocking. In order to ensure the FIFO property of the lock, we choose to use the ticket lock algorithm. In our implementation we split the lock variable in two halves. The first one is used as the ticket, and the second half is used as the now-serving. This is required so that changes in both variables can be made in a single atomic instruction. In the set_lock function, a PE first calls a Fetch and Add on the ticket to receive its ticket. It then proceeds to busy-wait until the now serving is equal to the ticket it owns to enter the critical region. In the clear_lock function the PE that owns the lock, calls a Fetch and Add on the now-serving to increment the now-serving and exit the critical region. The test_lock function can not be completed in a single atomic instruction. When a PE calls it, it first reads the lock variable. Afterwards, a Compare and Swap operation is called on the lock, to check if the next ticket to be served is still the

one previously read. If the Compare and Swap succeeds, the PE has acquired the lock. If not, the function returns.

Locking Routines
shmem_clear_lock
shmem_set_lock
shmem_test_lock

Table 2.7: OpenSHMEM Locking Routines

Chapter 3

Experimental Evaluation

For our evaluation, we benchmark the OpenSHMEM implementation of GSAS for all of its supported network connectivity. We are comparing against OpenMPI and MPICH that both support OpenSHMEM. For commodity Ethernet, MPICH provides a native implementation of OpenSHMEM, whilst OpenMPI requires the Unified Communication X (UCX) framework in order to work. For Infiniband (IB) they both require UCX. Due to them using the same underlying network stack, we only compare our IB implementation against OpenMPI and not both. Both MPI implementations are configured with the default configuration as cloned from their repositories. UCX is configured via runtime environmental variables. In order for the testing environments to be as close as possible to Gmem, we disable multi-railing on UCX, which automatically splits and transmits packets evenly across all available network devices. In Ethernet connectivity, both MPI implementations make use of TCP sockets, while GSAS supports RDS and UDP sockets. In IB, all implementation use the same transport protocol, namely Reliable Connection Queue Pairs. Furthermore, we conduct our experiments with cpu binding. We chose to bind our processes to physical CPU cores and not hyperthreads to fully utilize the CPU core and avoid oversubscribing that can lead to unreliable experiments. We also bind our processes to CPUs in NUMAs closer to our network adapters to achieve the best possible results.

Our testing environment consists of two servers connected with 1 Gbit Ethernet via a switch and 50 Gbit IB directly to each other. Both servers are identical, equipped with 2 Intel CPU Xeon E5-2620 with 12 physical cores and 24 hyperthreads, running at the 2.6 GHz frequency and 128GB of DDR4 DRAM. The operating system for both machines is Centos 7 and the kernel is 4.14 with RDS sockets enabled.

Our tests revolve around:

- Latency and throughput of PUT and GET operations for a variety of sizes
- Latency of atomic operations of 8 Byte WORDs
- Latency of collective operations

- Performance on Exanet

3.1 PUT and GET

In our first experiments, we study the performance of Remote Memory Access (RMA) operations performance. We measure the throughput average latency of PUT and GET operations, i.e. GET and PUT. More specifically, we measure the throughput and latency for PUT and GET operations. We perform 10^6 GET (or PUT) operations for variable sizes of transmitted data. This amount of operations and data sizes is large enough to saturate the network, allowing us to measure the maximum possible performance. We calculate latency as the time taken for the operations to finish divided by the number of operations. Also, we calculate throughput as the amount of data transmitted divided by the time taken. Each operation is followed by a synchronization opearation. We examine the cases of using shmem_barrier and shmem_fence as synchronization operations. The shmem_barrier ensures that all processes are synchronized, meaning that in order for the barrier to be released, all the previously remote memory updates should be completed. Shmem_fence forces a process to complete all the outstanding remote memory operations. The reason behind the usage of these operations lies within the OpenSHMEM specification. More specificaly, the OpenSHMEM specification defines a weak memory model regarding data transmission allowing an OpenSHMEM implementation to follow a weak or a stronger memory model. By following a weak/lazy memory model a process is allowed to complete an RMA operation without completing the transmission of the data. We have observed that without using a synchronization call the MPICH and OpenMPI implementations, data are not actually transmitted after a PUT or GET finishes. This is a result of following a lazy memory model, which leads to their remote memory operations being buffered and not actually processed until a synchronization call takes place to flush them. Gmem on the other hand, implements a stronger remote memory model in which the data are ensured to be transmitted by the end of an RMA operation.

In Figure 3.1, the latency performance of GET operations synchronized with Fence operations is measured. The vertical axis of the Figure shows the latency measured in microseconds (usecs) while the horizontal axis displays the size of data transfers performed. In the commodity case of Ethernet, we compare GSHMEMs RDS and UDP implementations with the TCP implementation of OpenMPI and MPICH. In the IB cases, we compare Gmem against OpenMPI. Delving more in to the Figure 3.1, we can see that the IB implementation of Gmem is not only competitive, starting at 1.55 times behind the MPI which uses UCX and progressively reducing the gap, but for sizes of 64KB and more, we manage to be almost 6 times better, maintaining a greater and a more consistent latency. This is a result of OpenMPI's complex architecture. OpenMPI splits data in packets called fragments. The maximum size of a fragment is 64KB. When data sizes

are greater than 64KB, OpenMPI switches to a more complicated protocol that introduces overheads causing the performance to drop. Oppositely, Gmem is using a simpler approach of only having two types of messages. For sizes up to 128B, we are transmitting inline IB messages. The inline feature is an implementation extension that is not strictly defined in any specification and its support varies based on the network adapter's manufacturers. We chose 128B as our maximum inline message size, since it is a low enough size that all manufacturers support. For sizes greater than this, we switch to standard IB messages. For the commodity Ethernet cases, our implementation falls behind and is only able to stay close to the MPI implementations for sizes up to 32B. This is due to the limitations of GSAS design that is being reflected direcly on Gmem. In the Ethernet cases, GSAS was designed to provide very low latency for small messages up to 64B, with half of it being our headers and the rest consisting of the actual payload. This explains the linear increase for sizes of 64B and more, since more than one packet need to be transmitted. Although we provide a fast path mechanism for sizes of 4KB and more, explaining the drop at the 4KB mark, it is still unable to keep up with both MPI implementations. In addition to this, our testing environment does not support native RDS sockets, which explains RDS being worse than UDP and TCP. RDS natively is implemented to work over Infiniband. In our implementation we are forcing it over Ethernet, which leads it to be simulated over TCP sockets, adding some more overhead on top of it.

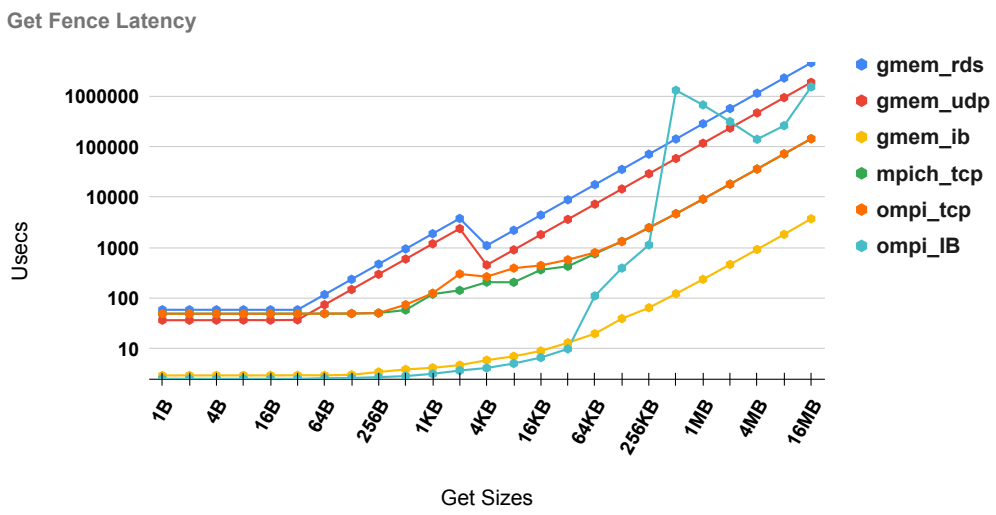


Figure 3.1: Get Latency synchronized with a fence operation

In Figure 3.2 we examine the throughput performance of GET operations synchronized with Fence. The vertical axis of the Figure shows the transmission rate measured in Megabits per second (Mbps) and the vertical axis contains the size of transmitted data. In the IB cases, the initial throughput is relatively small,

something expected since the sizes of transfers is not large enough for the link to be saturated and hit its cap. We can see that GSHMEMs throughput is very close to OpenMPI until the 64KB mark, where it surpasses it. The reason behind the OpenMPI behavior after 64KB is that it switchs its standard transmission protocol, to a more complex one for large packets introducing heavy overhead. Gmem utilizes 72% of the links capability, peaking at a throughput of 36000 Mbps. We are not able to fully utilize the link's capability, since Gmem utilizes a single threaded atomic service communicating with a single client. In the Ethernet cases, OpenMPI has better performance utilizing 100% of the links capability. As explained in 3.1, the Ethernet implementation of GSAS is only optimized for small data transfers and it's performance degrades in larger data sizes.

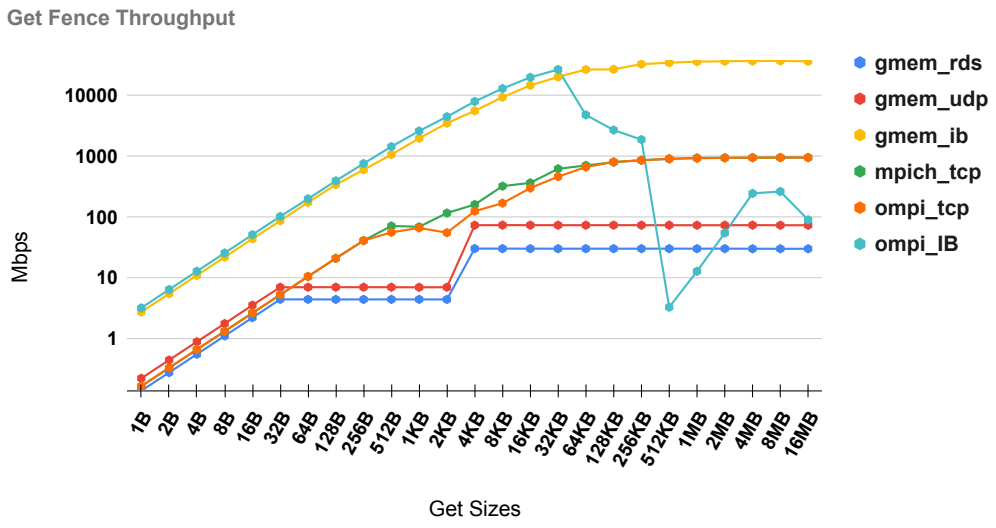


Figure 3.2: Get Throughput synchronized with a fence operation

In Figure 3.3 we examine the performance of latency and in Figure 3.4 the throughput of GET, but in these cases a barrier is used as synchronization operation. We conduct identical measurements consisting of 10^6 GET operations for variable sizes. Both figures display similar behavior to the experiments with fence synchronized operations i.e. Figures 3.1 and 3.2. For all implementations and different networking protocols, the initial performance is slightly worse compared to experiments performed with the fence counterparts. However, their peak performance remains the same. The explanation to this lies in the fundamental difference between the barrier and fence operations. A fence operation takes place in one node, and it behaves similarly to a memory fence and does not require any external communication with the other nodes. A barrier on the other hand, requires an all-to-all communication between all processes taking part in it, to signal their progress before it can be released. This introduces extra network traffic and synchronization which affects the performance of the actual remote memory operations. For smaller data transmissions this introduces a considerable overhead. As the data size increases, this cost keeps getting amortized, ultimately becoming negligible for amounts greater that 32KB, leading to the same results as the fence experiments.

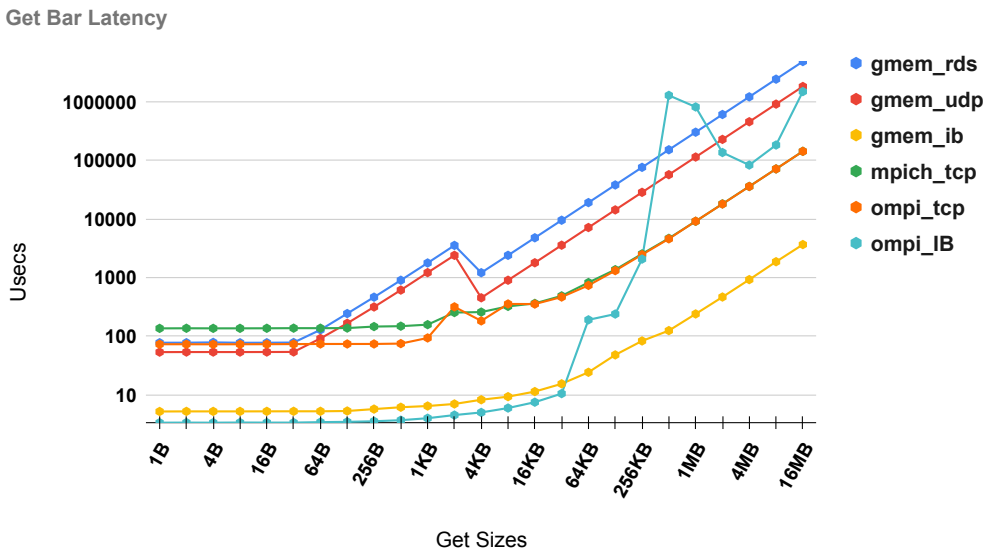


Figure 3.3: Get Latency synchronized with a barrier operation

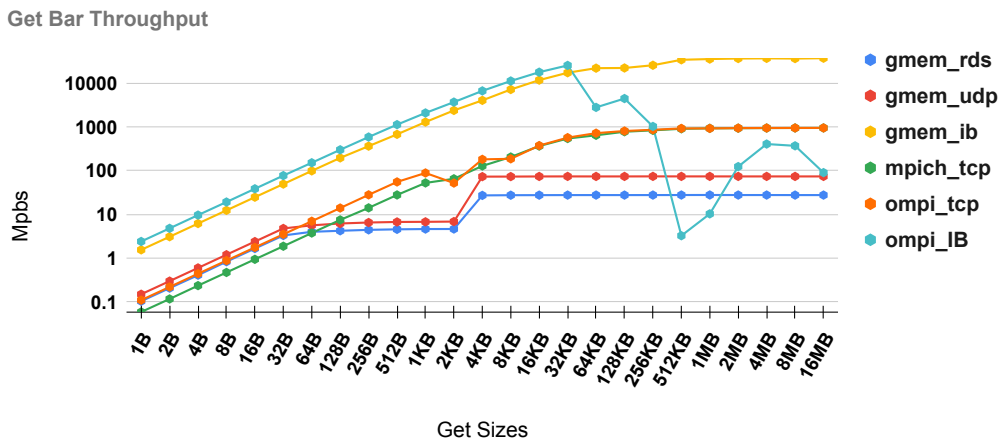


Figure 3.4: Get Throughput synchronized with a barrier operation

In the next experiments we measure the performance of PUT operations. Starting with Figure 3.5, we examine the latency of PUT synchronized with a Fence operation. Similarly to 3.1, the vertical axis displays the latency in microseconds and the horizontal axis the size of data transfers. As depicted on the Figure, Gmem falls behind the MPI implemenations, in both IB and Ethernet. The reason behind this is due to its design. More specifically, Gmem is built upon GSAS which was originally designed to work on top of customized hardware. That lead to a constraint of the atomic service being able to only receive packets of 32B. Out of this 32B, 8B are used as our header, leaving 16B as the payload. This is the reason behind the latency increase from 16B and more, since more than 1 packets need to be transmitted. Since the Atomic Service is a server handling requests from clients, it can reply to a request in any number of packets and packet sizes, but it only listens for requests of a specific format and size. In the case of GET operations, a client requests from a nodes Atomic Service to receive data. The Atomic Service upon receiving this request is able to split the requested data in MTU size packets and transmit them in a burst. In the case of a PUT operation, a client can not use the MTU size to transmit to the server and instead is limited to transmitting packets of 32B size. This leads in an increased number of packets required that affect negatively both the latency and the throughput. Given that this is a design constrain, all supported networking protocols of Gmem are affected, leading to the static latency of Gmem in IB, UDP and RDS as depicted on the Figure.

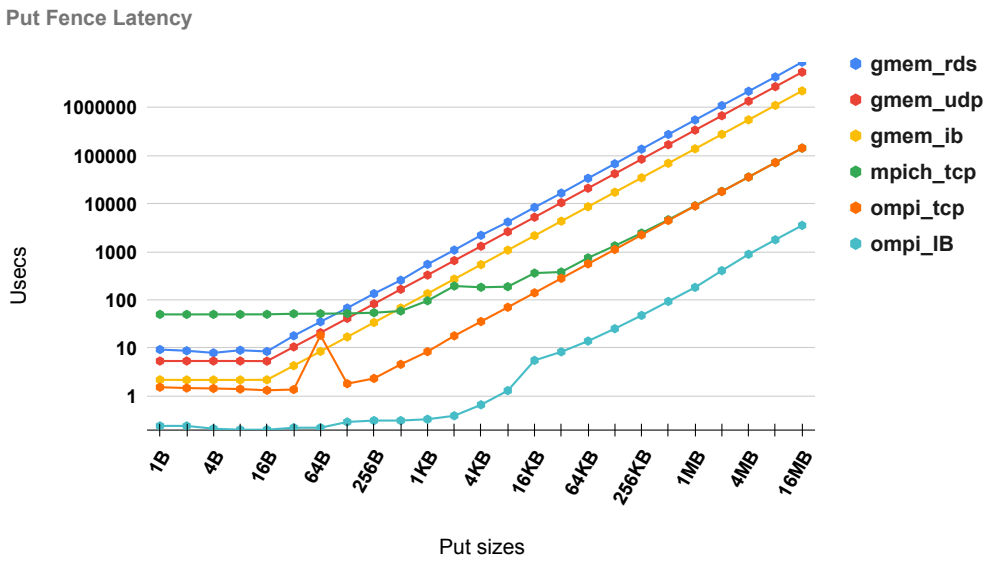


Figure 3.5: Put Latency synchronized with a fence operation

Figure 3.6 displays the throughput of PUT operations synchronized with fence operations. Similarly to Figure 3.5, Gmem 's performance cannot contest against the OpenMPI implementations due to the its design limitations discussed on 3.5. Gmem's IB implementation is only able to stay on par with OpenMPI up to sizes of 16B and degrades linearly afterwards. OpenMPI's latency on PUT operations is lower than the network's capability. This phenomenon is due to their optimized PUT operations, in which they use both multi threaded transmissions as well as multiple connections per pair of nodes. This behavior to the best of our knowledge is not configurable in either compile or run time and thus we are not able turn it off for our evaluations. For the Ethernet cases, Gmem's design limitation persist on both UDP and RDS transmissions displaying the same behavior as IB. MPICH follows a similar fixed sized packet transmission for packets with size less that 512B, and although it starts with a worse performance than Gmem, it manages to outperform it for packet sizes greater than 128B. OpenMPI's TCP manages to outperform both Gmem and MPICH for all packet sizes.

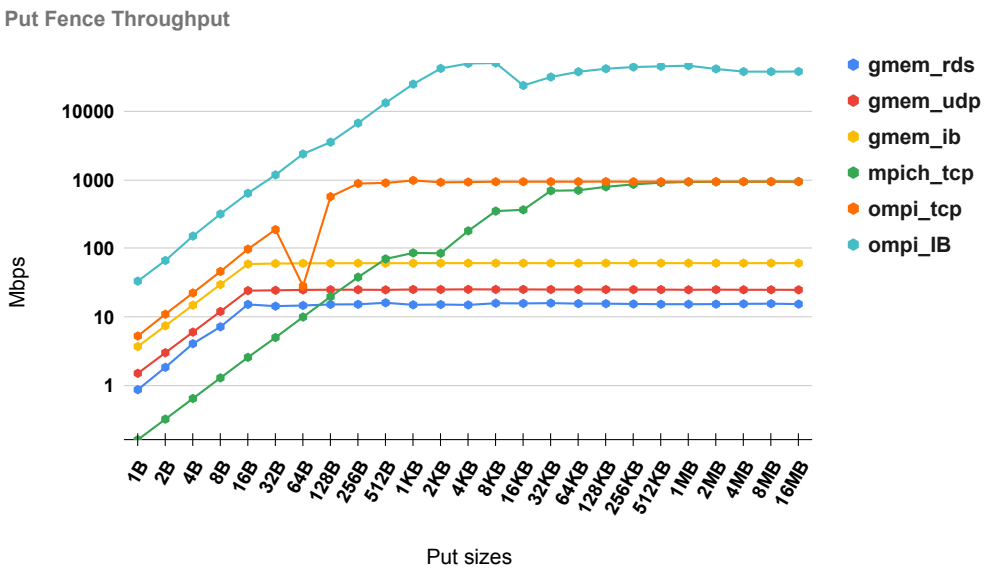


Figure 3.6: Put Throughput synchronized with a fence operation

Figures 3.7 and 3.8 depict the latency and throughput of PUT operations using barrier as their synchronization operation. In the TCP cases, we are able to perform better than both OpenMPI and MPICH for sizes less than 128B. In Infinibnnd OpenMPI is constantly ahead of Gmem. However, we observe that in our implementation, the synchronization method has minor effect on performance while both MPI implementations are considerably affected. This is because the OSHMPI relies on the MPI_windows for its symmetric memory, which are part of the MPI-3 specification and are used to introduce one-sided communications in MPI. MPI_windows work similar to GSAS memory allocation, in terms that memory space is allocated and then memory mapped in each process space. A remote pointer to this memory is exchanged between all processes and is used for one-sided operations. However, in contrast to GSAS, MPI_windows require some sort of internal synchronization between operations, which introduces overheads.

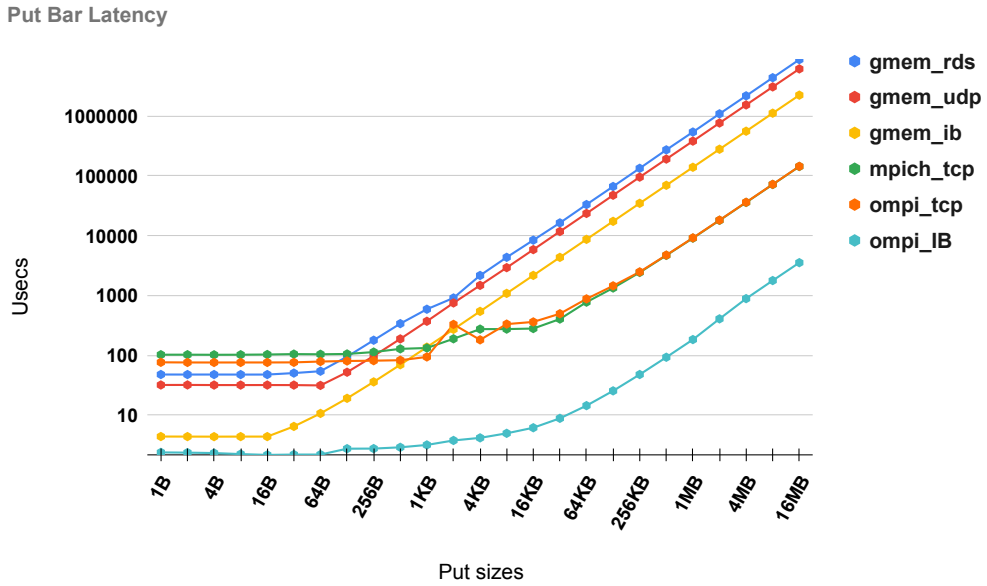


Figure 3.7: Put Latency synchronized with a barrier operation

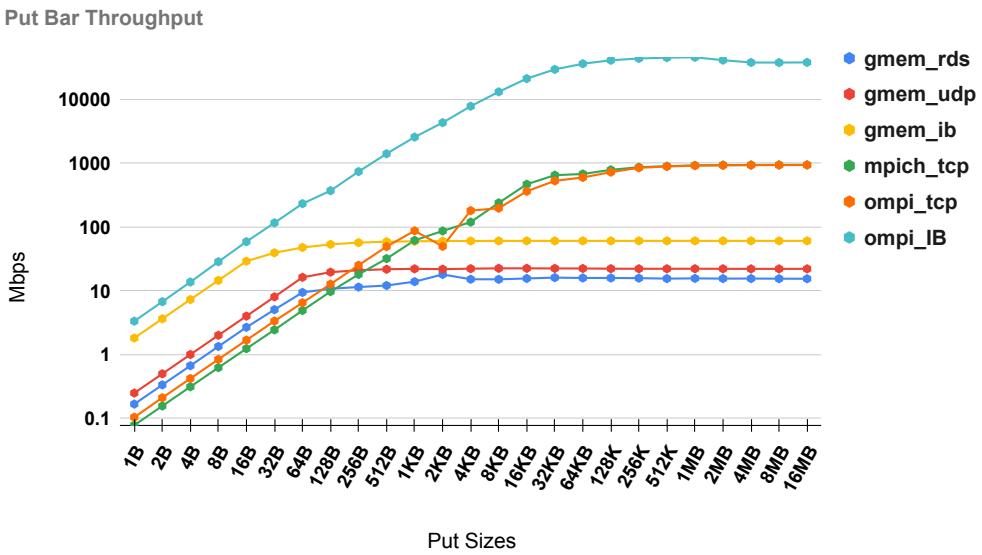


Figure 3.8: Put Throughput synchronized with a barrier operation

3.2 Atomic Memory Operations

Moving on to our next set of experiments, we examine the effectiveness of Atomic Memory Operations (AMO) defined in the OpenSHMEM standard for the GHSMEM, OpenMPI and MPICH implementations. We evaluate the Compare and Swap (CAS), Fetch and Add (FAD) and Atomic Add (AADD) operations. For each of these operations, we conduct 10^6 operations both for local and remote nodes, testing the average latency of each operation. We calculate the latency as the time taken for all operations to complete, divided by the time taken. Our testing environment is as follows: for the local experiments, we have two processes running on the same node with one performing the AMO's to the other. For our remote experiments, we have two processes running on two different nodes with one performing AMO's to the other. We chose 10^6 as our number of operation, so that our experiments could run for a sufficient amount of time to provide reliable results. In contrast to PUT and GET operations, no synchronization is needed after these operations.

Figure 3.9 displays the results of Compare and Swap AMO. The vertical axis displays the average latency for each case, measured in microseconds. Our horizontal axis is split in two segments, one containing the results for local and one for remote operations. In the case where the operation is performed locally, Gmem manages to achieve extremely low latency regardless of the underlying networking protocol used. This is due to the fact that in such cases Gmem has a fast path mechanism that allows it to call the native built-in atomic operations. This is possible because in GHSMEM, processes that are spawned on the same node are able to communicate using shared memory semantics. On the other hand, both MPI implementations seem to lack this feature and their transmission, even in local mode, is done through sockets transmitting in loopback. This is why it's performance varies based on the underlying networking protocol, due to the set up overhead of the sockets. On the remote cases, Gmem's IB implementation remains ahead, being 33% better than OpenMPI's performance. In the commodity Ethernet cases, our UDP manages to outperform both OpenMPI and MPICH, while RDS is slighty better than MPICH and worse than OpenMPI. This is due to the limitation of our testing environment's RDS usage over Ethernet instead of its native IB.

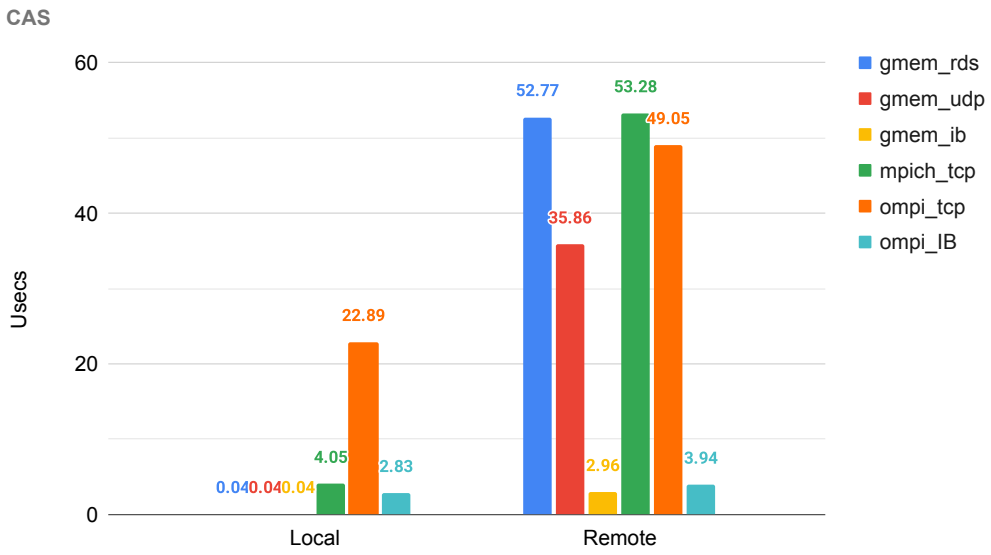


Figure 3.9: Atomic Compare and Swap

In Figure 3.10 we measure the performance of the Fetch and Add AMO. The vertical axis shows the latency measured in usecs. The horizontal axis displays the various test cases we examine. For our local operations, similarly to 3.9, Gmem operates better than the MPI implementations, regardless of the underlying networking protocol used. Both of the MPI implementations are using sockets for interprocess communication and are thus suffering from the overheads of initial setup as well as loopback transmissions. On the remote AMO's, Gmem's IB implementation outperforms OpenMPI, achieving 35% better latency. In Ethernet, Gmem'S UDP implementation achieves lower latency than both MPI implementations, while RDS has equal performance to MPICH and slightly worse performance than OpenMPI.

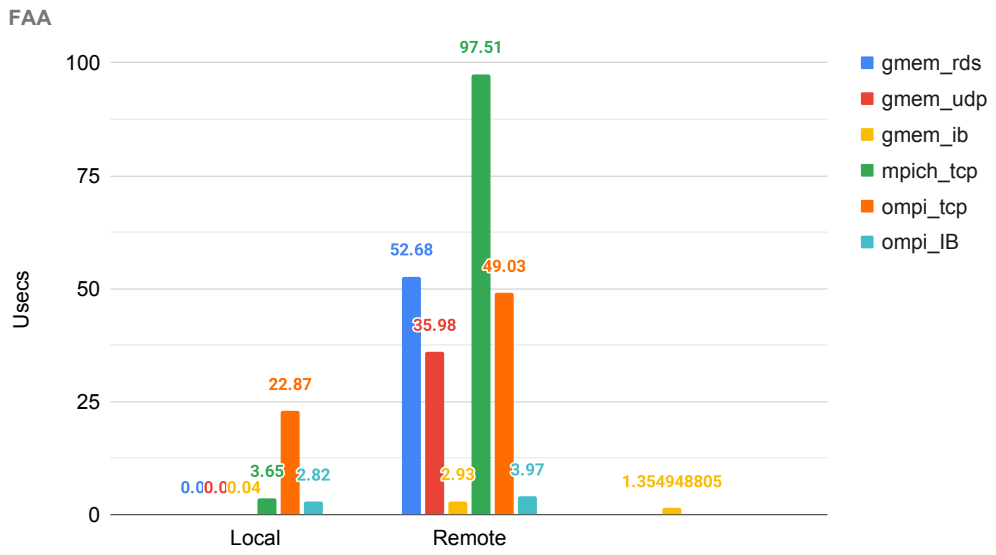


Figure 3.10: Atomic Fetch and ADD

Moving forward, Figure 3.10 displays the performance of the Atomic ADD operation. Following 3.9 and 3.10, the vertical axis shows the latency for each test measured in usecs. The horizontal axis contains the various implementations and network protocols we examine. In contrast to 3.9 and 3.10, we observe that in the local cases, OpenMPI achieves better results in both IB and TCP. This is due to the nature of AADD. AADD operates differently than CAS and FAA. CAS and FAA require communication to and forth from one node to another, introducing delays and interrupting optimizations. AADD on the other hand performs one way communication from one node to another. However, Gmem manages to outperform OpenMPI being 8 better in Ethernet transmissions and 6 times better in IB. On the remote cases, OpenMPI appears to perform better than Gmem's implementations, but the results produced are unrealiable. This is because, given that AADD operates similarly to a PUT, with a one way transmission of data, OpenMPI is able to perform optimizations that achieve better results than what the link is actually capable of. As presented on 3.6 OpenMPI uses multiple sockets and threads to transmit packets on a link, greatly reducing latency. These optimizations cannot be disabled in either compile time or runtime and thus our comparisons are uneven.

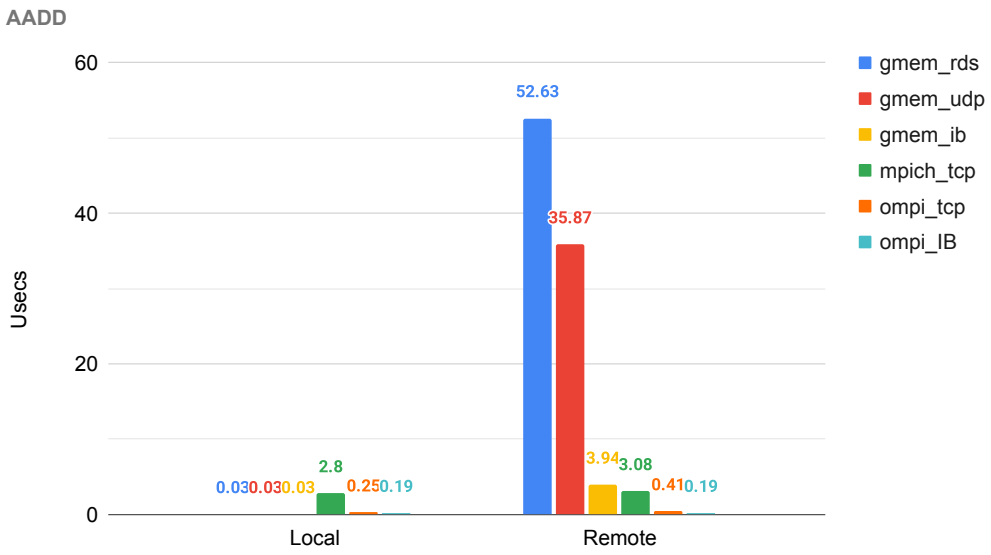


Figure 3.11: Atomic ADD

3.3 Collective Operations

Finally, we examine the performance of collective operations in OpenSHMEM. For the collective operations, we evaluate the latency of shmem_barrier, shmem_broadcast and shmem_collect for different number of participating processes. The maximum number of processes spawned is 22. We chose our maximum limit to be 22 in order to only spawn processes in the physical cores of our machines. In our benchmarks, all processes are initially spawned across our two computational nodes and each operation runs 10^6 times with an incrementing amount of processes participating every time.

Figure 3.12 presents the performance of the barrier operation of the OpenSHMEM specification. The vertical axis displays the latency of the operation in usecs. The horizontal axis contains the number of participating processes. In all implementations evaluated, processes with ranks 0-11 run on node 0, while ranks 12-22 run on node 1. When running intra-node, up to 11 processes, Gmem achieves the lowest latency in both TCP/IP and Infiniband cases. For more than 12 processes, we observe a steep increase in latency, in both our implementation and MPICH. This is due to the fact the interconnect (Infiniband or Ethernet) is utilized. OpenMPI's performance appears to be unaffected by remote operations. Gmem implementation utilizes a mechanism where only one process from each node synchronizes with the rest of the nodes, allowing for a hierarchical release of processes waiting at the barrier while limiting costly remote operations to a minimum. Overall our IB implementation performs 4.5x better when no remote communications are present, and 2x times better when remote operations are

present compared to OpenMPI over Infiniband. On TCP/IP our implementation also outperforms both OpenMPI and MPICH.

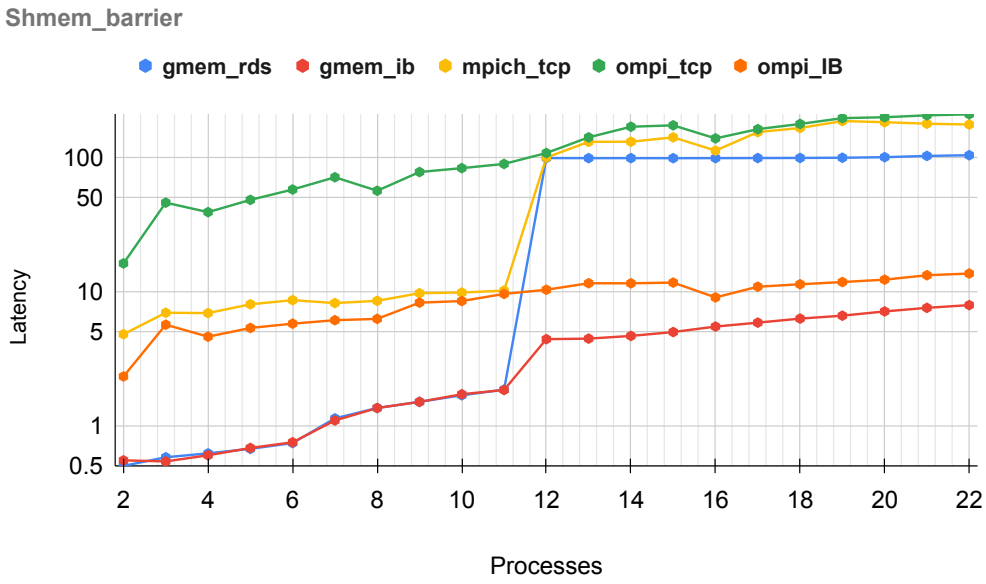


Figure 3.12: Shmem_barrier Performance

In Figure 3.13 we measure the performance of the broadcast operation. The vertical axis displays the latency of the operation while the horizontal indicates the number of processes participating in the operation. There are 22 total processes that are spawned evenly in both nodes. In a broadcast operation, one process transmits the same data to every other process of the active set. The number of data transmitted is 1 integer (4b WORD) per process. Our implementation utilizes an algorithm that combines the hierarchical barrier with the transmission of data. This allows us to perform the operation with minimal data transfers with only one barrier. We observe a steep increase in the latency of our implementation for 12 processes or more, regardless of the underlying networking protocol. The same is true for MPICH. This is due to the utilization of the interconnect. Our hierarchical broadcast implementation allows the leader process (i.e. the process that broadcasts) to transmit the data to sub-leader process of each node (i.e the first process of the active set in a specific node). When the sub-leaders receive the broadcasted data, they proceed to share it with the rest of local processes in their node via shared memory. This allows us to reduce the number of PUT operations from $N-1$ to M, where N equals the total number of participating process and M equals the total number of nodes containing participating processes, effectively lowering our latency. OpenMPI benefits from its very optimized PUT operations, achieving a 1.5x performance over our implementation when all the processes reside

on a single node, and a 10x performance when the participating processes span across 2 nodes.

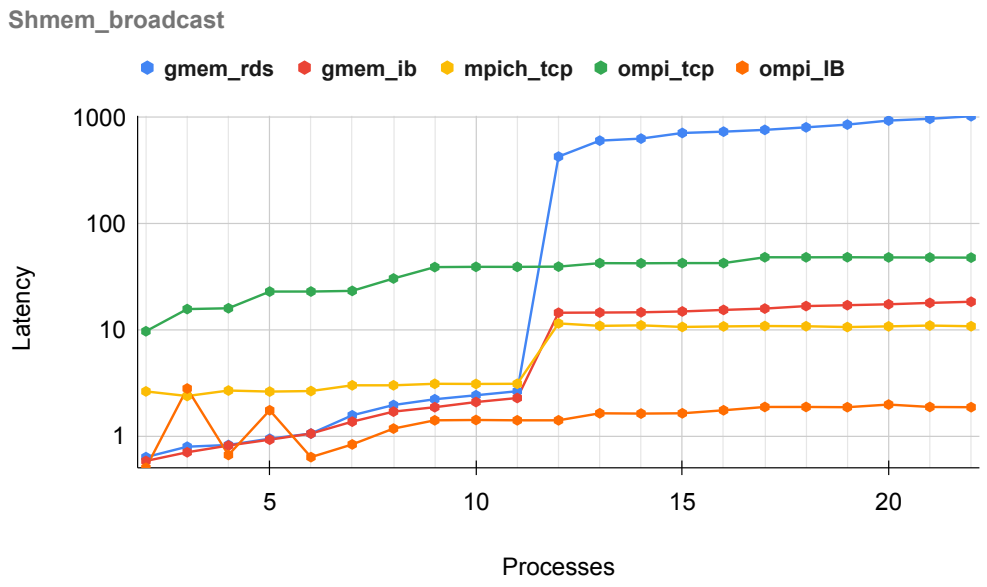


Figure 3.13: Shmem_broadcast performance

Figure 3.14 displays the performance of the collect operation. On the vertical axis the latency in usecs of the operation is displayed, while the horizontal axis displays the number of processes participating on the collective operation. The data transmitted equals to 1 integer per process. In a collect operation, all processes exchange data in an all-to-all fashion. Our implementation has the same performance in both underlying network protocols for up to 11 processes. This is due to the fact that the operations are performed though shared memory. For more than 11 processes, remote operations through network take place thus explaining the spike in the Figure at the point of 12 processes. MPICH operates in a similar manner. OpenMPI utilizes the network interface even for intranode communication, leading to an increased initial latency, but maintains a more gradual overall latency increase. Gmem limited performance for more than 11 processes regardless of the underlying network interconnect is due to its un-optimized Put operations, that is magnified due to the high number of remote operations required in an all-to-all operation.

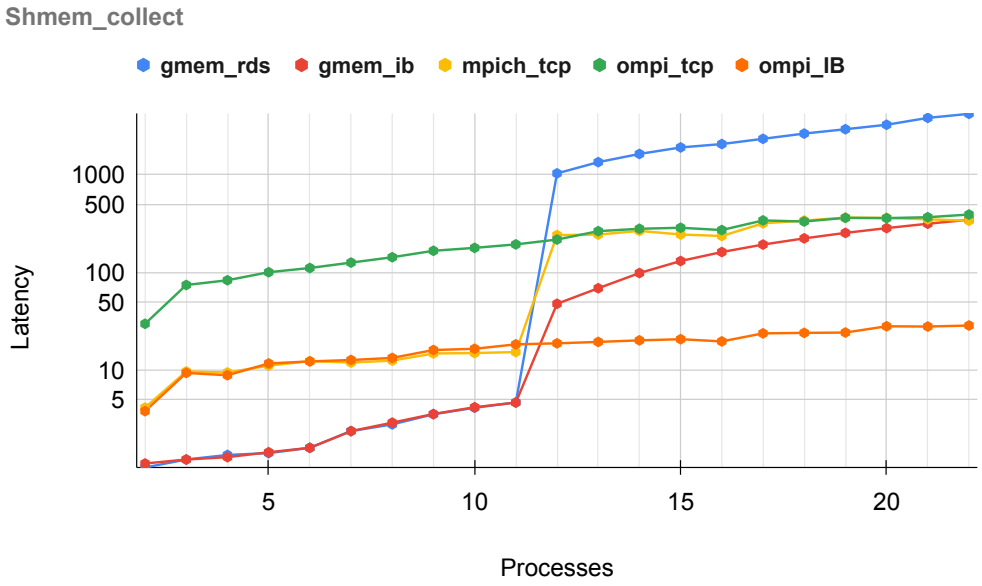


Figure 3.14: Shmem_collect performance

3.4 Exanet Performance

For the last part of our benchmarks, we evaluate the performance of Gmem on the Quad FPGA Daughter Boards (QFDB). As part of the ExaNeSt [8] project, FORTH's Carv laboratory has implemented a prototype cluster architecture consisting of several QFDB boards, connected through a custom interconnect called Exanet. GSAS was originally designed and implemented in order to deliver PGAS capabilities in the QFDB cluster. This architecture contains 32 QFDB boards, with each board having 4 FPGAs. This evaluation section serves as a proof-of-concept that Gmem is also supported in the arm-based QFDB boards with Exanet as our interconnect network. For our benchmarks, we utilize 1 QDFB board containing 4 FPGAs i.e. computation nodes. Each FPGA is equipped with a 4-core Cortex-A53 CPU running at 950Mhz and 16GB of RAM running at 1600Mhz. The FPGAs are connected through the Exanet interconnect with a link capacity of 17 Gbps.

3.4.1 Get Operations

Figures 3.15 and 3.16 display the latency and throughput of Get operations on the Exanet interconnect. Each figure contains data on operations performed that are synchronized using shmem_fence and shmem_quiet functions. The vertical axis presents the latency and throughput of the figures respectively. The latency is calculated in usecs and the throughput is calculated in Mbps. The vertical axis

displays the size of the data in both figures. While fence synchronized operations have a better initial performance, both types of Get gradually converge and perform the same for sizes greater than 512B. Fence synchronized operations initially perfoms better than barrier due to the fact that in our implementation fence is a no-op function. The reason behind the convergence of the two operations is that, as the transmitted size grows, the performance impact of the barrier synchronization becomes negligible and is being amortized. Throughput is inversely proportional to latency, and for that reason the 3.16 displays a similar behavior, with fence synchronized operations initially performing slightly better that the barrier.

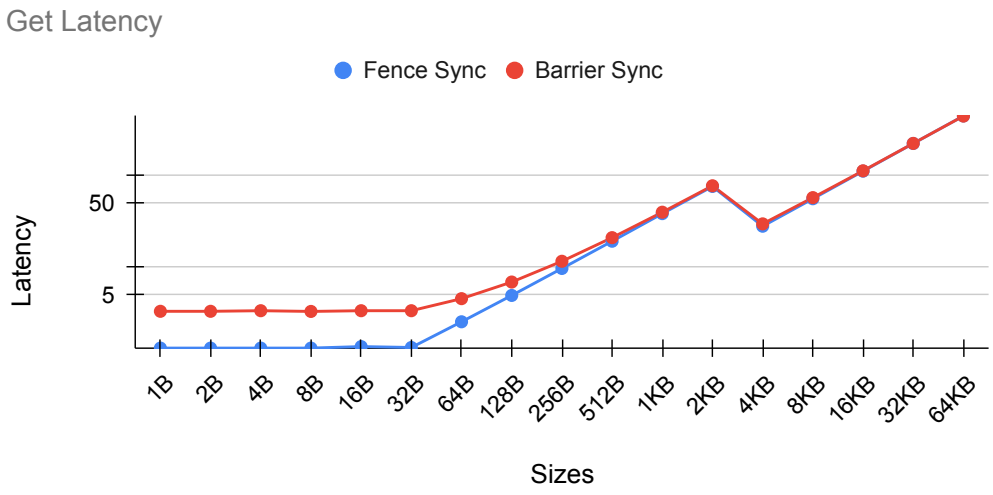


Figure 3.15: QFDB Get Latency performance

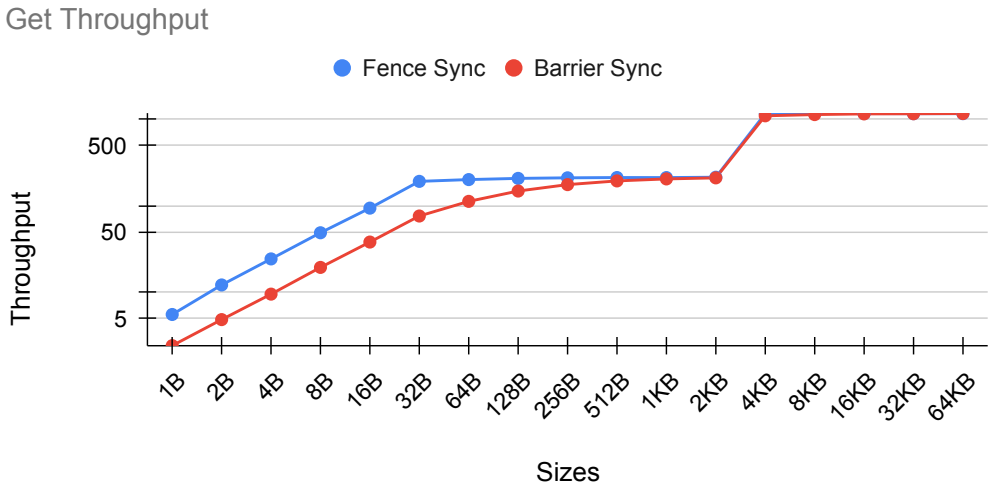


Figure 3.16: QFDB Get throughput performance

3.4.2 Put Operations

In Figures 3.17 and 3.18 we examine the latency and throughput of Put operations at Exanet interconnect. In these Figures we compare the performance of Put synchronized with fence against Put synchronized with a barrier operation. In Figure 3.17, the vertical axis displays the latency measured in usecs, while in Figure 3.18 the vertical axis displays the throughput measured in Mbps. The horizontal axis displays the size of the Put operations in both Figures. The latency of Put operations with fence performs slighty better than its barrier counterpart for sizes up to 128B. This is because in our implementation fence is a no-op operation. For sizes greater than 128B both test cases converge and display the same behavior. This is due to the reason that, as the sizes of Put increase, the cost of the barrier function becomes amortized by the costly network operations. In Figure 3.18 displaying the throughput we observe the same behavior. Fence initially performs better but for sizes greater than 128B fence and barrier synchronized Puts perform the same.

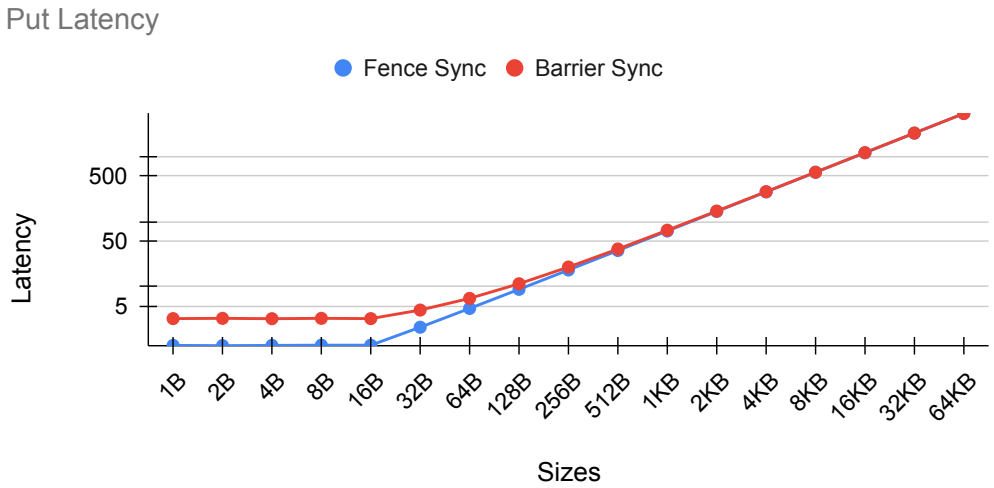


Figure 3.17: QFDB Put Latency performance

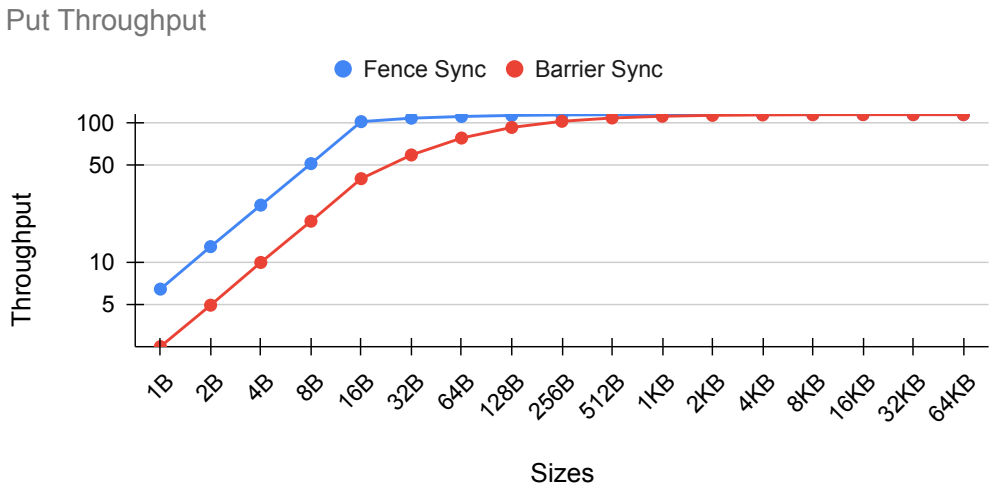


Figure 3.18: QFDB Put Throughput performance

3.4.3 Collective Operations

Figure 3.19 displays the performance of collective operations throught the Exanet interconnect. The vertical axis contains the latency of the operations measured in usecs, while the horizontal axis displays the number of participating processes in the operation. The collective operations performed are shmem_barrier, shmem_broacast and shmem_collect. For this benchmark, we make utilize all 4 FPGA's computation nodes and a total of 12 processes, 3 in each node. In the

broadcast operation, a process broadcasts a fixed amount of data with every other participating process, while in a collect operation all processes exchange a fix amount of data in an all-to-all fashion. We observe a considerable increase in the latency of all three operations when the number of processes is 4. This is because for more than 3 processes, the underlying interconnect is utilized.

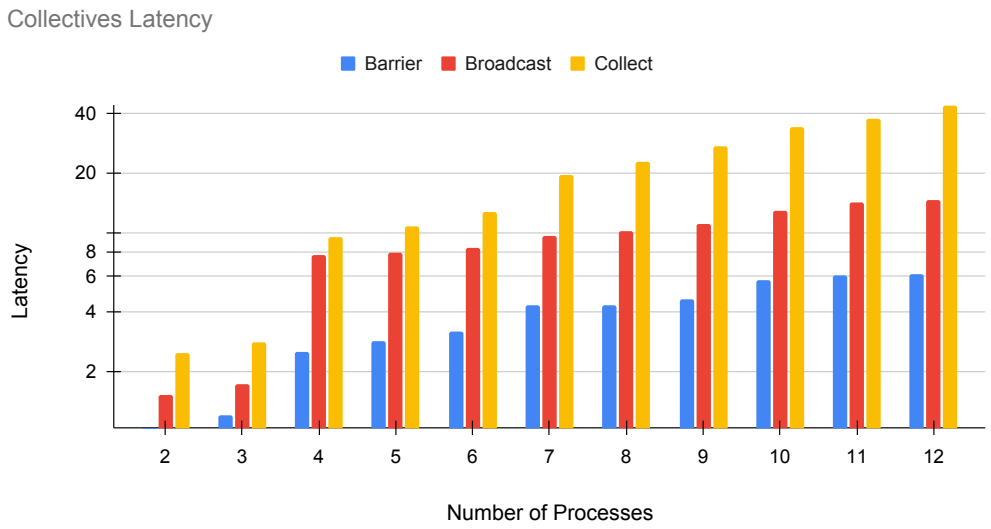


Figure 3.19: QFDB Collectives Latency performance

Bibliography

[1] Vikas Aggarwal, Alan D. George, Changil Yoon, Kishore Yalamanchili, and Herman Lam. Shmem+: A multilevel-pgas programming model for reconfigurable supercomputing. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3), aug 2011.

[2] Dan Bonachea and Paul H. Hargrove. Gasnet-ex: A high-performance, portable communication library for exascale.

[3] Jens Breitbart, Mareike Schmidtobreick, and Vincent Heuveline. Evaluation of the global address space programming interface (gaspi). In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 717–726, 2014.

[4] Monika Bruggencate, Cray Inc, and Duncan Roweth. Dmapp -an api for one-sided program models on baker systems. 02 2022.

[5] Georgel Calin, Egor Derevenetc, Rupak Majumdar, and Roland Meyer. A theory of partitioned global address spaces. *CoRR*, abs/1307.6590, 2013.

[6] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4), may 2015.

[8] European Union. ExaNeSt Portal. `https://exanest.eu/`. Accessed: March 11, 2022.

[9] ExaNode. Design of ExaNoDe Firmware. `https://exanode.eu/wp-content/uploads/2017/04/D3.6.pdf`, 2016. Accessed: March 11, 2022.

[10] Daniel Grünewald and Christian Simmendinger. The gaspi api specification and its implementation gpi 2.0. 2013.

[11] Jeff Hammond, Sayan Ghosh, and Barbara Chapman. Implementing openshmem using mpi-3 one-sided communication. 03 2014.

[12] Chung-Hsing Hsu, Neena Imam, Akhil Langer, Sreeram Potluri, and Chris J. Newburn. An initial assessment of nvshmem for high performance computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–10, 2020.

[13] Nikolaos D. Kallimanis. Synch: A framework for concurrent data-structures and benchmarks. *Journal of Open Source Software*, 6(64):3143, 2021.

[14] Nikolaos D Kallimanis, Nikolaos Chrysos, and Manolis Marazakis. A flexible & efficient shared memory abstraction with minimal hw assistance.

[15] Nikolaos D Kallimanis, Manolis Marazakis, and Nikolaos Chrysos. Gsas: A fast shared memory abstraction with minimal hardware support.

[16] Nikolaos D Kallimanis, Manolis Marazakis, and Manolis Skordalakis. Usecases for remote memory in the unimem architecture. In *ExascaleHPC: the ExaNoDe, ExaNeSt, EcoScale, and EuroEXA projects workshop at HiPEAC, Manchester*, 2018.

[17] Jarek Nieplocha and Bryan Carpenter. *ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems*, volume 1586, pages 533–546. 11 2006.

[18] OpenSHMEM Foundation. OpenSHMEM SOS Test Suite. `https://github.com/openshmem-org/openshmem-examples`. Accessed: March 11, 2022.

[19] K Parzyszek, J Nieplocha, and R A Kendall. A generalized portable shmem library for high performance computing.

[20] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. *OpenSHMEM - Toward a Unified RMA Model*, pages 1379–1391. Springer US, Boston, MA, 2011.

[21] Mirko Rahn. Gpi - global address space programming interface - experiences on scalability. In *PARCO*, 2011.

[22] Naveen Ravichandrasekaran, Bob Cernohous, Dan Pou, and Mark Pagel. Introducing cray openshmemx-a modular multi-communication layer openshmem implementation. 01 2019.

[23] Robert Ross, Rob Latham, William Gropp, Ewing Lusk, and Rajeev Thakur. Processing mpi datatypes outside mpi. volume 5759, pages 42–53, 09 2009.

[24] MIN SI, PAVAN BALAJI, KENNETH J RAFFENETTI, HUI ZHOU, SHINTARO IWASAKI, and DOD. Oshmpi: Open shmem implementation over mpi, 3 2021.