Overlapping Computation and Communication through Offloading in MPI over InfiniBand

by

Grigori Inozemtsev

A thesis submitted to the
Graduate Program in Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Applied Science

Queen's University
Kingston, Ontario, Canada
May 2014

Abstract

As the demands of computational science and engineering simulations increase, the size and capabilities of High Performance Computing (HPC) clusters are also expected to grow. Consequently, the software providing the application programming abstractions for the clusters must adapt to meet these demands. Specifically, the increased cost of interprocessor synchronization and communication in larger systems must be accommodated. Non-blocking operations that allow communication latency to be hidden by overlapping it with computation have been proposed to mitigate this problem.

In this work, we investigate offloading a portion of the communication processing to dedicated hardware in order to support communication/computation overlap efficiently. We work with the Message Passing Interface (MPI), the *de facto* standard for parallel programming in HPC environments. We investigate both point-to-point non-blocking communication and collective operations; our work with collectives focuses on the allgather operation. We develop designs for both flat and hierarchical cluster topologies and examine both eager and rendezvous communication protocols.

We also develop a generalized primitive operation with the aim of simplifying further research into non-blocking collectives. We propose a new algorithm for the non-blocking allgather collective and implement it using this primitive. The algorithm has constant resource usage even when executing multiple operations simultaneously.

We implemented these designs using CORE-Direct offloading support in Mellanox

InfiniBand adapters. We present an evaluation of the designs using microbenchmarks and an application kernel that shows that offloaded non-blocking communication operations can provide latency that is comparable to that of their blocking counterparts while allowing most of the duration of the communication to be overlapped with computation and remaining resilient to process arrival and scheduling variations.

Acknowledgments

First I would like to express my sincere appreciation for the guidance and support of my supervisor, Dr. Ahmad Afsahi, throughout this work. I also would like to thank the Natural Science and Engineering Research Council of Canada (NSERC) and Queen's University for providing me with financial support. I also thank the HPC Advisory Council and Mellanox Technologies for the technical resources used to conduct the experiments in this thesis.

Thanks for stimulating discussions go to my coworkers at the Parallel Processing Research Lab: Ryan Grant, Judicael Zounmevo, Reza Zamani, Mohammad Rashti, Iman Faraji, and Hessam Mirsadeghi.

Last but not least I say a heartfelt thank you to my wife Laura without whose support this would not have been possible.

Table of Contents

List of Tables

List of Figures

Glossary

allgather a *collective* operation in which each participating process sends a block of data and receives the concatenation of all the blocks in process *rank* order, p. 55.

cluster a computer system comprised of interconnected subsystems that can cooperatively solve a computational problem; each subsystem is also capable of independent operation, p. 10.

collective (communication) having well-defined semantics performed by a group consisting of one or more processes, p. 17.

communicator a logical grouping of *MPI* processes in a *point-to-point* or *collective* operation for the purpose of referring to the group, p. 15.

Completion Queue (CQ) an *InfiniBand* queue containing *CQEs* indicating completions of *WQEs* on send and/or receive queues associated with the CQ, p. 23.

Completion Queue Entry (CQE) an entry in a *Completion Queue* describing the completion of a *WQE*; provided information can be used to determine the success or failure of the WQE, as well as the originating *SQ* or *RQ*, p. 23.

compute node a unit in a *cluster* that performs computation, p. 10.

CORE-Direct an extension of the *InfiniBand* specification that allows sequences of data transfers to be offloadedto the *HCA* for processing, p. 24.

Direct Memory Access (DMA) ability of a device to access system memory without involving the CPU, p. 22.

eXtended Reliable Communication (XRC) an *InfiniBand* transport mode that enables an *SRQ* to be shared among multiple processes, p. 23.

Floating Point Operations Per Second (FLOPS) a measure of computer performance commonly used in scientific computing. Unlike the measurement of instructions per second, the FLOPS metric emphasizes the performance of the system in computations involving floating-point numbers, p. 9.

High Performance Computing (HPC) a collection of technologies focused on delivering fast solutions to computational problems; performance is usually derived from parallel computation, p. 10.

Host Channel Adapter (HCA) a device connecting a host in an *InfiniBand* network to the fabric. Usually has capability for *Direct Memory Access* to application memory buffers, p. 22.

InfiniBand (IB) a specification for a switched high-speed *interconnect* with applications in *High Performance Computing* and enterprise datacentres, p. 22.

interconnect a network connecting the subsystems of a *cluster* into a single system, p. 10.

Management Queue (MQ) a type of queue provided by *InfiniBand* devices with *CORE-Direct* support; provides features for synchronizing multiple *Queue Pairs*, p. 26.

Maximum Transmission Unit (MTU) maximum message size that can be transmitted by a network adapter as a single network packet, p. 101.

Memory Region (MR) a region of main system memory that has been set up for access by the *Host Channel Adapter*, p. 23.

Message Passing Interface (MPI) a specification defining a message passing system. MPI is the *de facto* message passing standard in *High Performance Computing*, p. 13.

microbenchmark a measurement of performance of a small and specific part of code, usually a single feature, p. 5.

Multiple-Program Multiple-Data (MPMD) a paradigm for dividing work among processors in which the data are partitioned among multiple processes running instances of several different programs that are written to cooperatively solve the computational problem, p. 15.

Non-Uniform Memory Access (NUMA) a type of multiprocessor system design in which the latency of access to a region of memory depends on the location of the memory relative to the requesting processor, p. 10.

Open Fabrics Enterprise Distribution (OFED) a suite of open-source software providing support for *InfiniBand*. Contains both OS kernel drivers and libraries implementing user application interfaces, p. 22.

OS noise large delays in a cluster system arising from the propagation of small process scheduling variations due to interference of the operating system services on the individual nodes, p. 19.

Partitioned Global Address Space (PGAS) a programming model in which an address space is shared by all processes, yet portions of the space are designated as local to certain processes, p. 14.

point-to-point (communication) between a pair of processes performed by matching a *send* operation on one side with a *receive* operation on the other, p. 16.

Queue Pair (QP) a communication endpoint abstraction in *InfiniBand*. Consists of a *Send Queue* and a *Receive Queue*, p. 22.

rank a unique number identifying a process in a *communicator*. Ranks in *MPI* are assigned sequentially starting at 0, p. 15.

Receive Queue (RQ) an *InfiniBand* queue containing `recv` *WQEs* to be processed in order, p. 22.

Receiver-Not-Ready (RNR) a condition arising when no `recv` WQE is available on an *RQ* at the receiver to match an incoming `send` WQE. May optionally trigger a retry sequence with a specified timeout, p. 33.

Remote Direct Memory Access (RDMA) a feature of the *Host Channel Adapter* where it performs *DMA* in response to a remote request, p. 23.

Remote Memory Access (RMA) ability to access the memory of a process executing on a different processor, p. 16.

Send Queue (SQ) an *InfiniBand* queue containing `send` *WQEs* (and variants thereof, such as RDMA read/write requests) to be processed in order, p. 22.

Shared Receive Queue (SRQ) a variant of an *RQ* in which multiple *RQ*s belonging to the same process share a set of memory buffers, p. 23.

Single-Program Multiple-Data (SPMD) a paradigm for dividing work among processors in which the data are partitioned among multiple processes running instances of the same program, p. 15.

supercomputer see *High Performance Computing*.

TOP500 a biannually updated list of the 500 fastest supercomputers in the world, as determined by their performance in the LINPACK benchmark, p. 2.

unexpected message queue a queue holding *MPI* messages that have been received but not yet handled by the application, p. 34.

verbs a set of commands defining the application interface to *InfiniBand*. Verbs differ from functions in that they specify only semantics, not the function signatures, p. 22.

Work Queue Element (WQE) an element in a *Send Queue* or a *Receive Queue* specifying a unit of work to be performed by the *Host Channel Adapter*, p. 22.

Work Request (WR) see *Work Queue Element*, p. 33.

Chapter 1

Introduction

Modern High Performance Computing cluster systems support most of the world's scientific computing [95]. These systems are responsible for computation that is crucial for research in fields as diverse as physics, chemistry, climate modelling, and materials science, to name just a few [69]. Demand for greater computing power from these fields continues to increase. Larger and faster HPC systems do not just allow research results to be obtained more quickly; performance improvements also enable larger and more detailed scientific simulations and therefore new discoveries. Efforts are currently underway to increase the performance of the fastest HPC systems a thousandfold as part of the Exascale project [1].

Scaling up the hardware capabilities of HPC is not enough. In order to reach the Exascale target, the software executing on these machines will have to optimally utilize their capabilities. Clusters are comprised of a number of interconnected processing modules. Unsurprisingly, efficient interprocessor communication is an issue of utmost importance in HPC cluster systems, and will only become more critical as clusters grow to exascale and the number of communicating processors increases [1].

This work is a contribution towards the goal of mitigating the effects of the increasing communication latencies and synchronization overhead in HPC clusters. We developed techniques for hiding communication latency by overlapping it with computational work

and techniques for reducing the interprocessor synchronization implicitly required by communication operations. Our designs make use of specialized hardware to offload certain communication tasks from the main processor. We showed through experimental evaluation that communication offloading can efficiently support the overlapping of computation and communication without incurring significant overhead.

1.1 Motivation

The largest High Performance Computing systems in the world are ranked semiannually by the TOP500 project [95]. From the project's statistics it is clear that the available computing capacity has been growing steadily, with the current combined performance of the systems on the TOP500 list reaching 250 PetaFLOPS, or 2.50×10^{15} floating point operations per second. Demands for computing power are unlikely to show a reversal, thus HPC system performance is also expected to grow. The current milestone for HPC performance is the exascale goal; the aim of the Exascale project is to build a machine capable of 10^{18} floating-point operations per second [1].

The most common platform for High-Performance Computing today is a cluster comprised of machines built using commodity hardware and connected by a network. Though the first clusters were small-scale installations, the cluster architecture has since permeated the HPC world. Clusters have largely replaced other architectures even among the most powerful supercomputers. Whether the computational power for a scientific project is supplied by one of the world's largest supercomputers or a smaller-scale machine, it is likely that this machine is a cluster. Among the TOP500 machines, 84.6% of the systems were clusters as of November 2013 [95].

The fact that an HPC cluster is comprised of a number of networked nodes means that solving a computational problem on a cluster involves both processing that is performed locally on the cluster nodes, and cooperation between the nodes of the cluster through

network communication. In order to make optimal use of the resources provided by a cluster, it is crucial for the software executing on the cluster to efficiently orchestrate local processing and network communication. This is not a trivial task; simply speeding up network communication is not enough. Consider the problems of process arrival pattern and scheduling noise, which we discuss in more detail in Chapter 2. If computation on a node depends on communication with another node, any delay at the sending node will be propagated to the receiver. Small delays propagating through the cluster in a cascading fashion can lead to dramatic slowdowns, especially in large clusters [72].

It is important for scientific applications to perform well on HPC clusters; however, burdening computational scientists responsible for implementing these applications with low-level details such as InfiniBand [39] networking or the interaction between local computation and network communication is impractical. Although they are intrinsic to the modern HPC cluster, these issues are outside the research goals of the users of the cluster hardware. Furthermore, these issues are common to all applications running on clusters, therefore a more practical approach would be to solve them in a fashion that could be shared among research projects and would obviate every project from having to duplicate the work. This can be accomplished by presenting scientific software developers with a higher-level programming model that provides abstractions to ease programming HPC systems. We briefly review the existing models for programming HPC clusters in Chapter 2. In this work, however, we focus on MPI (Message Passing Interface) [64], the most widely used programming model in HPC.

1.2 Research Objectives

As High Performance Computing clusters grow in scale to address ever-larger computational challenges, overlapping communication with computation is becoming an increasingly important tool for maximizing the performance of applications in the cluster environment [43,

20, 34].

Scientific application developers could reap the benefits of communication/computation overlap if they had usable abstractions that allowed them to describe the parts of their program that could be executed in this fashion. In MPI, this capability is provided by non-blocking operations, which we discuss in Chapter 2. Although the MPI specification provides a definition of these abstractions, the issue of ensuring that applications using these abstraction perform well in cluster environments remains open.

In this work, we examine the non-blocking communication abstractions in MPI that provide the capability of overlapping communication and computation. We investigate the issues involved in making these abstractions a reality on modern clusters using InfiniBand hardware. Having surveyed the problem, we present designs for non-blocking communication operations that provide consistently high performance for communication and for the computation that is overlapped with it, while scaling to run on large systems. We aim to answer the following questions:

- Can we ensure that overlapped communication, whether between pairs or among groups of processes, does not negatively affect the performance of the computation?
- How can we mitigate the effects of the process arrival pattern and scheduling noise?
- Can we efficiently support multiple concurrent communication operations and overlap them with computation?

1.3 Contributions

Overlapping communication and computation is an approach to improving application performance and scalability that has been successfully applied in the past, and continues to show promise in the face of the increasing performance requirements imposed by the

largest clusters [81, 6]. In Chapter 2, we review the background in the area of computation/communication overlap that lays the foundation for this thesis. The work immediately related to the contributions of the individual chapters of this thesis will be discussed as the contributions themselves are presented.

Building on this prior research, we turn our attention to the non-blocking communication operations in MPI. We investigate both messages exchanged between a pair of processes (point-to-point messages) and messages that are part of communication of a group of processes (collective operations). For the investigation of collective operations we focus on the allgather collective operation. This operation among participating processes communicates a concatenation of the contributions of all the processes to the entire group.

The overarching theme of this thesis is the design of communication operations that take advantage of the hardware of a modern cluster and support the Message Passing Interface that is familiar to most scientific software authors. For each MPI operation that we have investigated, we have developed a design to support effective non-blocking communication. To validate these designs, we have implemented them using the recently added capabilities of Mellanox [61] InfiniBand adapters that support offloading communication processing from the node's CPU.

Specifically, we make the following contributions:

1. We introduce a design of hardware offloaded progression of large messages. This design adapts MPI message matching semantics to the capabilities provided by the CORE-Direct hardware for exchanging messages between a pair of nodes in a cluster (point-to-point). An evaluation of this design is conducted using microbenchmarks.

2. We design and implement asynchronous flat and hierarchical network offloaded non-blocking allgather collective operations using standard exchange [10], Bruck [14], and ring [93] algorithms with single-port and multiport modelling. We also discuss the applicability of multileader and multigroup design techniques to the allgather algorithm

using offloading.

3. We redesign the radix sort application kernel to utilize the proposed non-blocking allgather collective to achieve speedup through communication/computation overlap. The radix sort application kernel was used along with microbenchmarks to evaluate the non-blocking allgather collective on a 128-core and a 16-core cluster.

4. We propose a pipelined allgather algorithm that preserves the best characteristics of previously explored algorithms, improves tolerance to process scheduling discrepancies, and reduces resource consumption. We also propose a new primitive, Receive-Store-Replicate, to aid with the construction of offloaded collectives. An evaluation of this design was conducted using microbenchmarks.

The ultimate aim of this work is to contribute towards a general approach to designing efficient non-blocking communication with hardware offloading in point-to-point and collective scenarios. The techniques and approaches described in this thesis should be adaptable to communication operations other than the ones covered. Moreover, all the non-blocking operations presented are fully compatible with the MPI-3 specification [64], and can be employed in compliant applications.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 provides the background information for the remainder of the work and introduces the Mellanox CORE-Direct offloading technology that supports the implementation of our designs.

Chapter 3 explains how CORE-Direct offloading can be applied to transferring single messages between a pair of nodes in a cluster while overlapping the transfer with background computation. In addition to being useful in its own right, this work forms the foundation of the subsequent chapters.

Chapter 4 deals with collective communication in flat and hierarchical environments. In a flat environment, no special consideration is given to the hierarchical structure of a typical cluster, which is built using multiple processors and processor cores per node. Consequently, the investigation of the flat environment focuses on the network communication. Taking advantage of the hierarchical structure for collective communication is also considered in Chapter 4.

In Chapter 5, an additional collective optimization is introduced and explored. Communication pipelining is shown to improve the tolerance of the collective to scheduling variability across the cluster. As part of our design of the pipelined collective, we propose the Receive-Store-Replicate primitive, which is a general solution that can be used as a building block for other non-blocking collectives, and show how it can be applied. We also propose an algorithm for the allgather collective operation based on the structure of a Cayley tree. We employ this algorithm in a collective design that reduces resource consumption. We then implement this algorithm using the new primitive, and evaluate its performance.

Chapter 6 concludes the work and discusses directions for future research.

Chapter 2

Background

Amdahl's 1967 paper famously asserted that the speedup obtained through parallelism is necessarily limited by the portion of the program that cannot be executed in parallel [2]. Almost every program will have some amount of sequential code, therefore Amdahl's Law predicts diminishing returns from increasing the number of processors that are executing the program in parallel. However, in practice the situation is often not as dire as Amdahl's Law predicts and much higher speedup is attainable with parallel processing. The difference lies in the definition of speedup. Gustafson pointed out that the fundamental assumption of Amdahl's Law – that the aim of parallel computing is to speed up a fixed amount of computation – often does not hold in practice [28]:

> When given a more powerful processor, the problem generally expands to make use of the increased facilities.

The time taken to execute the serial part of many scientific application programs stays relatively constant, while the work done in the parallelizable parts of their algorithms scales with the problem size. Such applications can see an improvement much larger than predicted by Amdahl's Law. Instead of considering the time necessary to complete a fixed amount of work, Gustafson considers the amount of work that can be done in a fixed amount of time.

The speedup predicted by this model is much more optimistic: it scales linearly with the number of processes.

While this definition of speedup is not the same as that of Amdahl's Law, Gustafson's Law nevertheless strengthens the argument for the usefulness of massively parallel computation for many problem domains. Problems that are well-served by parallel computing are common in a number of scientific fields, including chemistry, physics, biology, and geology. In order to model the phenomena they are studying in greater detail, increase the solvable problem sizes, and to make discoveries more quickly, researchers in these fields demand ever-increasing computing power. This power is supplied by High Performance Computing facilities.

The use of commodity systems for parallel computing was researched extensively in the early 1990s. The best-known projects from this era, the Berkeley Network of Workstations (NOW) [4] and the Beowulf project [88], used clusters of PC workstations for scientific computing tasks and found this approach to be cost-effective. Though these first clusters were small-scale installations by today's standards, the cluster architecture proved to be able to scale to the largest deployments. Clusters have largely displaced other architectures in high-performance computing. They made up 84.6% of the systems in the worldwide TOP500 supercomputer ranking [95] as of November 2013.

Progress in High Performance Computing is customarily measured in factors of 1000. The fastest machines at present are able to perform over 10^{15} floating-point operations per second (FLOPS), and are part of the petascale generation of systems [69]. The fastest petascale system at the time of writing, Tianhe-2 at the China National University of Defense Technology, can reach peak performance of 33.86 PFLOPS in the LINPACK benchmark [21] that is used to rank systems on the TOP500 list [95].

Exascale machines represent the next landmark on the HPC roadmap. These machines would be able to perform over 10^{18} operations per second, providing a 1000-fold increase in available performance for scientific applications. Although no exascale designs currently exist, and there is no consensus on how these systems should be designed, there is widespread

agreement that reaching this milestone will be challenging [1]. An exascale machine using current processor, memory, and network technologies would consume more power than a medium-sized city [69]. Clearly, simply scaling up existing petascale designs is impractical; instead, a new approach must be sought that would achieve exascale computational performance without a dramatic increase in power consumption.

However, power is not the only obstacle on the road to exascale computing. It is almost certain that the number of processing modules in an exascale system will be increased compared to the machines of the present. Coordinating these processing resources effectively will require involvement from the software that is to run on the exascale machines. One issue that is already of utmost importance in the petascale era, and will only become more critical as HPC moves to exascale, is that of interprocessor communication and synchronization [1]. Specifically, as the number of processors increases, the cost of synchronizing them grows correspondingly. It is therefore beneficial to restrict synchronization to cases where it is absolutely necessary and to employ asynchronous processing elsewhere. This work is a contribution towards this goal.

2.1 Cluster Hardware

As we already mentioned, the most common High Performance Computing architecture today is a cluster of compute nodes connected through an interconnection network. The nodes making up an HPC cluster contain multiple processing units that share a memory address space. The processors in the nodes can typically access all of the node's main memory, though the access times of the various parts of the main memory may differ if the node is of a Non-Uniform Memory Access (NUMA) design [69]. Modern processors operate much faster than main memory and thus typically include one or more levels of cache memories that serve to speed up access to frequently used data. These caches may be shared or private to a single processor. Today, multiple processing units are commonly placed on a single

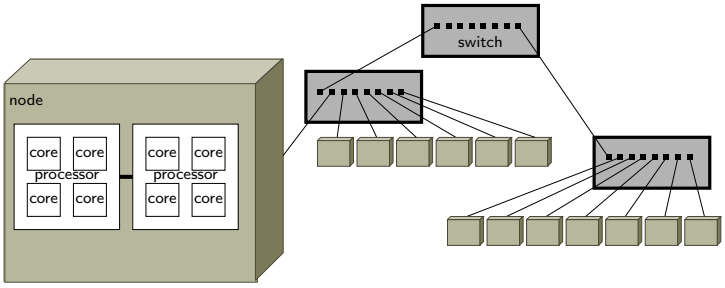


Figure 2.1: Architecture of a cluster using multi-core processors

silicon die. It is also common for multiple dies to be joined into a single package. Processing units in such a product are usually called processor cores. This packaging provides a natural hierarchy for cache memories to follow. For example, the Intel Nehalem and Ivy Bridge processors include two private levels of cache for each core, as well as a shared third cache level [41].

Because network communication is an integral part of cluster computing, its performance has a significant impact on the overall performance of the HPC cluster. Accordingly, cluster network architectures have diverged from those most commonly found in networks of servers and workstations. It is common for clusters to replace low-cost network technologies such as Gigabit Ethernet with a high-speed interconnect. The most common technology used to interconnect the computation nodes of HPC clusters is InfiniBand. InfiniBand is used in 41.4% of the clusters on the TOP500 list [95] and this number has been growing steadily in recent years. As we will see in Section 2.6, the InfiniBand technology aggressively optimizes network communication. Unlike many conventional network interfaces, where transferring data to the network hardware is the responsibility of the operating system (OS), InfiniBand bypasses the OS and handles networking at the application level. Figure 2.1 shows a visual representation of the architecture described in this section. It represents a cluster with multi-core processors and an interconnection network consisting of two levels of switches.

2.2 Programming Models

Making effective use of the hardware resources in an HPC cluster is a challenging task. It is important to ease the burden on computational scientists by providing a consistent and usable programming model that abstracts away the low-level details of the hardware. At the same time, the implementation supporting the model's abstractions must be optimized to extract the most performance from the hardware while letting scientific software developers focus on their work.

A programming model provides an interface between the application and its execution on the cluster hardware while hiding the implementation details from the application programmer. The programming models used in scientific software can be broadly classified as belonging to one of two groups: *message passing* models that present the application developer with a representation of independently executing processes that communicate by passing messages, and *shared memory* models that instead present an abstraction of a large shared memory that all executing processes share. Clearly, the programming model's choice of abstraction will favour or discourage certain application designs. The programmers using the model in their application are much more likely to favour the abstractions that are a natural fit to their application and the model.

Libraries and programming languages used in practice will often combine certain aspects of both models. For example, the Message Passing Interface (MPI) specification, while considered a quintessential example of a message-passing system, introduced shared-memory programming features in version 2.2 [63].

Although the implementation details of a programming model's abstractions are hidden, they are crucial to the performance of the application on the physical hardware of a cluster. In order to be useful in practice, the abstractions provided by a programming model must be designed to match real usage scenarios and be implemented efficiently. An abstraction that is awkward to use or provides poor or inconsistent performance is unlikely to find

widespread adoption.

2.2.1 Message Passing

The message-passing model is based upon explicitly specified communication between participating processes. Information is communicated in this model by having one process construct a message and send it through the interconnection network to another process, which in turn receives the message. In the message-passing model, each processing unit has a private memory space that it can access, with no memory being shared between the sender and the receiver. All communication between processes is accomplished through messages.

The Message Passing Interface (MPI) [64] is one example of this type of system. Although many other message-passing systems exist [69], MPI is the *de facto* standard for message passing in scientific applications. We will discuss MPI in more detail in Section 2.3.

2.2.2 Shared Memory

In contrast to the message-passing model, processes in the shared-memory model coordinate work by sharing parts of their memory address spaces. Communication between processes in the shared-memory model is done implicitly, through reads and writes to data structures residing in shared memory regions [69].

The shared-memory model is frequently said to be easier for programmers to use and reason about than the message-passing model [30]. Of course, even though no additional work is required to communicate updates to shared data among processes, coordination of work among the participating processes is still required in order to ensure that the results are correct. Coordination can be achieved through mutual exclusion algorithms [26] or lockfree data structures [19].

Shared-memory programming is the most popular programming model for standalone multiprocessor machines. Two examples of programming interfaces for this model are

Pthreads [94] and OpenMP [68]. Even though MPI includes some shared-memory programming features, as we previously mentioned, it is typically not classified as an example of the shared-memory programming model.

2.2.3 Distributed Shared Memory

As we mentioned in Section 2.2.2, many programmers consider using the shared-memory model easier than dealing with explicit message passing [30]. For this reason, a shared-memory programming model can be provided even when multiple machines are involved that do not physically share memory. In this case, shared-memory semantics are emulated using message passing to transfer data between the disparate memory address spaces.

Though the abstraction of a large, uniform, globally accessible address space may be convenient, it can be challenging for the programmer to reason about the performance of an application written using such a programming model. The time taken by a given processor to access different parts of the distributed shared memory may vary depending on the physical location of the memory providing the data in relation to the processor. In a distributed shared-memory system powered by a cluster of interconnected compute nodes, memory access times may be highly non-uniform due to the large variation in latency between local and remote accesses. Because hiding the details of the distributed memory layout from the programmer can make it difficult to write well-performing software, the Partitioned Global Address Space (PGAS) [69] model has been developed. Programming languages and libraries that implement the PGAS model offer the programmer control over the allocation of data in the distributed memory by specifying affinity between the data and the processing units that need to access them frequently. Some examples of PGAS languages are UPC [98], Chapel [17] and X10 [18]. The Global Arrays [66] library also implements PGAS concepts.

2.3 Message Passing Interface

Let us return to the message-passing paradigm, having briefly looked at the shared memory programming model. The Message Passing Interface is the dominant message-passing standard in the area of High Performance Computing. MPI is supported on almost all platforms used in HPC, and its implementations have as a rule been extensively optimized for performance on their respective hardware. MPI therefore enables message-passing programs to be ported between hardware platforms while maintaining a high level of performance without additional optimization work in the common cases. MPI provides a distributed memory abstraction, though implementations of MPI work in shared- and distributed-memory environments, as well as hybrid environments such as clusters.

An MPI program consists of multiple running processes, each with its own private address space. The processes are started by a *launcher*, which provides enough information to the MPI library to establish communication between the processes, largely transparently to the user. MPI processes are commonly instances of the same program, though it is possible to create an MPI job with different programs. The former is the Single-Program Multiple-Data (SPMD) model, while the latter is the Multiple-Program Multiple-Data (MPMD) model. Once launched, the processes execute asynchronously. When running on a single compute node or a cluster, the MPI processes sharing the node can use the shared memory to implement message passing.

For identification, processes are grouped together in communicators. In each communicator, processes are assigned a rank by numbering them sequentially starting at 0. Because a rank in a communicator uniquely identifies a process, in the rest of this work we will sometimes use the term rank to refer to an individual process. Note, however, that a process may belong to multiple communicators, and have different ranks in each.

Once processes know how to address each other by the combination of the communicator and rank, they can cooperatively solve a computational problem by communicating with one

another. Because communication operations involve participation of multiple ranks, they introduce dependencies, and potentially synchronization, into the execution of processes. MPI communication operations belong to one of three categories: point-to-point, collective, and remote memory access (RMA) operations. The latter category introduces a distributed shared-memory programming facility to MPI, as we previously mentioned. We will not discuss RMA operations further in this thesis, choosing to focus instead on point-to-point and collective operations.

2.3.1 Point-to-point Communication

Point-to-point communication involves a pair of processes; one executes a send operation, and the other executes a receive operation. The MPI implementation is responsible for passing the message that the programmer describes by transferring data from the source buffer in the sender's address space to the destination in the receiver's address space.

The send and receive point-to-point operations have several variants that differ in their completion timelines and buffering strategies. *Blocking* operations return control to the application only after the data transfer is completed from the view of the MPI rank, and the data have been either sent or received and placed in the correct buffer in the participating process. In contrast, *non-blocking* operations are split-phase, with one call beginning the communication and a separate call confirming its completion. Between these calls, the application is free to perform other operations that will overlap the execution of the background data transfer. There also exist several buffering modes that can be specified by the programmer to optimize the communication performance. All variants of the send and receive calls are interoperable. For example, a non-blocking receive call can receive data from a blocking send.

The processes execute asynchronously, meaning it is possible that either the send or the receive operation is issued first. Furthermore, multiple outstanding operations are possible due to the non-blocking communication feature. To deal with these possibilities, MPI has a

well-defined set of message-matching semantics that ensure that communication between processes follows a predictable pattern. Messages are exchanged within a context of a communicator. They are further distinguished by the message tag and the sender rank. Messages in MPI are non-overtaking, meaning that messages satisfying the same matching rules are received in the same order as they were sent.

Tags allow the programmer to enforce the correct order of message arrival without concern for issuing outgoing messages in the same order. This capability enables a degree of non-determinism in the MPI message-matching. For example, messages may originate from distinct application threads, or be received using a combination of a non-blocking receive and the `MPI_Waitany` function that completes a single request out of a set. As a further source of non-determinism, MPI includes a feature that allows the receiver to not specify the source of the message, and receive a message from any sender (`MPI_ANY_SOURCE`). A similar feature (`MPI_ANY_TAG`) exists for tags. Even with these sources of non-determinism, as long as the programmer supplies correct message matching information, the correctness of the program remains intact.

2.3.2 Collective Communication

Whereas point-to-point operations involve only a pair of processes, collective operations carry out communication within a communicator in a specific pattern. For example, the `MPI_Bcast` operation broadcasts the contents of a buffer belonging to a single process to all ranks in the specified communicator. Many collective operations are part of MPI, each with its own set of semantics.

An MPI library is free to substitute any implementation of a collective operation as long as the operation's semantics are preserved. For example, the broadcast operation may be implemented using a tree-based communication pattern rather than with a naive algorithm in which the process at the root of the broadcast iteratively sends a message to all receivers. Designing efficient collective communication operations has been an active area of research

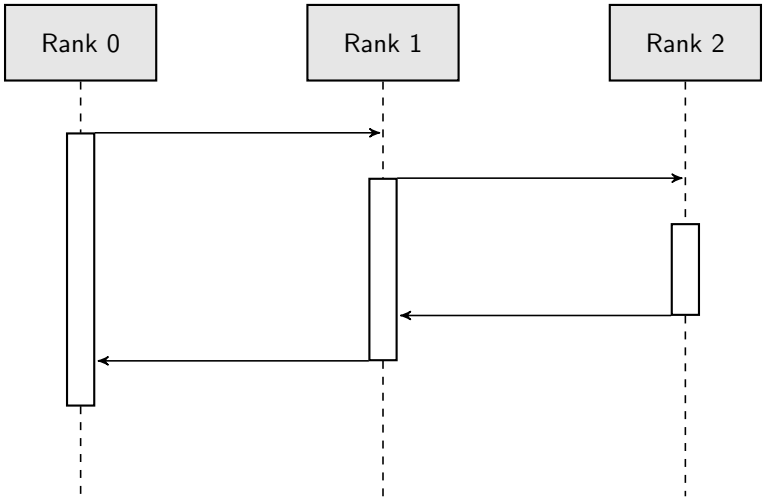


Figure 2.2: Process skew in a blocking collective operation. Processing of messages that arrive when a process is inactive (indicated by dashed lines) is delayed.

because the performance of collectives significantly affects the execution time and scalability of MPI applications [27].

Optimizing collectives for large-scale HPC systems presents a number of challenges. One scalability issue that existed in previous versions of the MPI specification was that all collectives were defined to be blocking operations. Therefore, no rank in a communicator could perform any other tasks while in a collective call. Ranks would have to wait until all operations implementing the collective semantics had been performed, at least from the point of view of that rank. This usually meant that collective operations were *synchronizing*: all ranks would have to wait for the entire communicator to complete the collective call.

The blocking collective operations specified by the MPI-2.2 standard do not allow for overlap between computation and communication; however, non-blocking collective operations [35] have recently been introduced in the new MPI-3 standard [64]. Non-blocking collective operations are similar in concept to non-blocking point-to-point operations discussed in Section 2.3.1, with one call commencing the operation, and another call completing it. Unlike point-to-point operations, blocking collective calls do not interoperate with non-blocking ones [64].

Like their point-to-point counterparts, non-blocking collectives reduce requirements for synchronization among processes, and allow for overlap between computation and communication. The benefits of non-blocking semantics are potentially more significant for collectives than for point-to-point operations: as illustrated by Figure 2.2, there is greater opportunity for delays due to synchronization as the number of processes increases. In the sequence diagram, rank 0 begins a blocking collective operation that depends on a response from rank 1, which in turn must communicate with rank 2. Any delay in the activation of the collective call at ranks 1 and 2 results in significantly increased delay at rank 0. The situation becomes worse when the collective communication consists of multiple steps, as is usually the case with collective operations in practice.

2.4 Effects of Process Arrival Pattern and Operating System Noise

Any communication operation, whether point-to-point or collective, necessitates actions from more than one process. Clearly, an operation cannot complete without the completion of work assigned to each of the processes. This implies that a delay in starting the work by any process is likely to contribute to the overall latency of the operation.

Ideally, all processes would begin the collective operation simultaneously, and would spend no time waiting for each other. This ideal is unachievable in practice due to a number of factors, such as variance in partitioning of work across ranks, inconsistencies in network latency, and the interference of operating system services, or OS noise, to name a few. Scheduling variations can significantly prolong the execution time of communication operations [72, 42, 38]. Because collectives involve many ranks, they are especially susceptible to these effects. Reducing the effects of the process arrival pattern and scheduling noise is crucial in optimizing the performance of collective operations. We will describe these issues in more detail and present a design aimed at addressing them in Chapter 5.

2.5 Non-blocking Communication Progression

In Section 2.3.2, we briefly discussed MPI's collective operations, which offer programmers a set of useful primitives for communicating information among a group of processes. We saw that data dependencies often arise in collective communication, and therefore implicit synchronization is often involved between the participating processes. It is advantageous to structure scientific code so that it can perform useful work while waiting for network communication to complete. Efficiently implementing support for the overlap between computation and communication has been a long-standing goal in high performance computing.

One of the most important factors in achieving a high level of overlap is the ability of the MPI communication subsystem to make timely progress on the outstanding background communication operations. Furthermore, the overhead of message processing in non-blocking operations must be kept low to ensure maximum availability of the host processor for computation.

Non-blocking communication progression methods can be broadly categorized into approaches that make use of the host processor for progression and those that offload this task to a dedicated hardware device. Although we focus on offloading in this thesis, we will review the host-based techniques to provide context.

2.5.1 Host-based Progression

When using the host processor to progress communication while performing computation, the MPI *progress engine* must be periodically invoked to attend to the network and make progress on pending operations. The function of the progress engine is described in more detail in Chapter 3.

One way to accomplish this is to manually restructure the application code to periodically call the `MPI_Test` function [36]. Although this approach has low overhead, it leads to complicated code and cannot be easily applied when using libraries to perform computation.

Additional complications arise from the need to keep the workload balanced across the processes involved in the communication. For optimal performance, calls to `MPI_Test` on all processors should occur at similar intervals, which need to be tuned for the specific workload.

A second option is to use a helper thread to handle message progression [36]. This approach works well with libraries and results in cleaner application code. Unfortunately, the context switching overhead and competition for the processor between the computation and communication threads lead to an increase in latency and a corresponding drop in application performance. Thread-based solutions are also susceptible to the negative effects of process arrival pattern variations and OS noise [72, 38].

Making communication progression the responsibility of the OS kernel is a third host-based approach. It generally results in lower overhead compared to designs that employ userspace threads, and allows for better communication/computation overlap for small messages [67, 83]. However, installation of custom kernel modules may not be permitted in a shared environment of a supercomputing facility. Additionally, the host processing resources must still be shared between computation and communication in this approach.

Reserving a processor core for handling communication can ameliorate the issues associated with host-based progression at the cost of reducing the processing power available for computation. Alternatively, communication can be offloaded to special-purpose networking hardware. This is the approach that we will describe next.

2.5.2 Offloaded Progression

Offloading communication progression to dedicated hardware has been shown to reduce the effects of OS noise and to allow for effective communication/computation overlap by a number of studies. Previous investigations of hardware offloading of collective operations have been performed by Buntinas et al. using Myrinet network interfaces [16], Yu et al. on Quadrics Elan [103], and Hemmert et al. using the proprietary Cray interconnect [29].

We have investigated the effects of using a different offloading technology, namely Mellanox

CORE-Direct on the InfiniBand interconnect. We will describe the CORE-Direct technology in Section 2.6.1 using an example of its use for offloading communication progression.

2.6 InfiniBand

The InfiniBand Architecture (IBA) [39] is an open standard for a high-speed switched system-area network. Hosts are connected to the network by Host Channel Adapters (HCAs). One defining feature of InfiniBand is its operating system bypass capability. Using direct memory access (DMA), HCAs can directly access application memory buffers without relying on a software network stack in the operating system kernel. Operating system bypass is an important feature for High Performance Computing because the overhead of calling into the OS kernel from the application can be significant [53].

Software support for InfiniBand is provided by the open-source Open Fabrics Enterprise Distribution (OFED) [39], which includes both OS kernel drivers and userspace application libraries that implement an interface to InfiniBand. The main abstraction used to program InfiniBand is IB *verbs*. The InfiniBand standard defines the semantics of the verbs, but not the exact programming interface. The functions in the library supporting the verbs specification implement the abstraction.

An application communicates over InfiniBand by invoking verbs that manage Work Queue Elements (WQEs) on the HCA. IB WQEs include `send`, `recv`, RDMA `write` and `read`, and atomic operations. WQEs are posted to a Send Queue (SQ) or a Receive Queue (RQ), which together form a Queue Pair (QP) – a communication endpoint in the IB network.

QPs may use one of several transport types that provide different ordering and reliability characteristics. For the designs presented in this thesis, we use the Reliable Connection (RC) transport type that guarantees reliable in-order delivery of messages, as well as extensions to the RC transport. There are three other transport types: Reliable Datagram (RD), which removes the ordering requirement, Unreliable Connection (UC), which has ordering but not

reliability guarantees, and Unreliable Datagram (UD), which does not guarantee message ordering or successful delivery.

IB Send and Receive Queues are associated with Completion Queues (CQs). This mapping is not necessarily one-to-one: multiple queues can share a CQ. The application fetches Completion Queue Entries (CQEs) from a CQ to determine the completion of WQEs.

InfiniBand adapters access memory by directly using physical addressing, bypassing the operating system's paging mechanism. Because RDMA operations target application buffers, the expected buffer must always be found at a known physical address. This requirement is met through Memory Region (MR) registration: the user buffer is pinned in physical memory so that it cannot be swapped out, and the address translation is communicated to the HCA.

The InfiniBand `send` verb and the corresponding `recv` verb invoked at the destination together provide the information about the memory locations on the sender and receiver systems that is sufficient to perform the data transfer. Alternatively, the Remote Direct Memory Access (RDMA) verbs `rdma_write` and `rdma_read` permit one-sided communication that can be performed as long as all the requisite address information is provided.

The Shared Receive Queue feature provided by most modern InfiniBand adapters allows a number of RQs to share a single pool of memory buffers. eXtended Reliable Communication (XRC) is an extension of the SRQ feature that enables receive queue sharing across processes on the same node.

In this work, we make use of Mellanox CORE-Direct extensions to the InfiniBand Architecture that enable sequences of WQEs to be managed by the HCA. We will describe these extensions in Section 2.6.1.

2.6.1 CORE-Direct Extensions to InfiniBand

As discussed in Section 2.6, a communication endpoint in InfiniBand is represented by a Queue Pair (QP), which consists of a Send Queue and a Receive Queue. Tasks posted

Table 2.1: CORE-Direct QP initialization options

Flag	Description
`IBV_M_QP_EXT_CLASS_1`	Enable CORE-Direct functionality
`IBV_M_QP_EXT_CLASS_2`	Hold SQ WQEs until explicitly enabled
`IBV_M_QP_EXT_CLASS_3`	Hold RQ WQEs until explicitly enabled

to these queues describe the data transfer operations to be carried out by the InfiniBand adapter. Mellanox CORE-Direct extends this model by introducing three new types of tasks, or Work Queue Elements (WQEs). Like the `send` and `recv` WQEs described in Section 2.6, these WQEs become the responsibility of the InfiniBand HCA once they are posted to the QP. The execution of the WQEs is thus decoupled from the code executed by the host processor. The completion of a WQE is still reported as a Completion Queue Entry (CQE) on a CQ.

We describe the functionality of the CORE-Direct tasks individually at first, followed by an illustration of how these WQEs can be combined to achieve offloading of communication progression.

Enable WQE

Send and receive tasks usually begin executing as soon as they are posted to a Queue Pair. Therefore, if the aforementioned tasks make up a sequence of communications that requires progression based on incoming data, a host-based progression implementation must wait before posting the tasks to the HCA.

When a QP is configured for use with CORE-Direct, it can optionally hold send and/or receive tasks in the corresponding queue instead of immediately processing them, in accordance with the flags listed in Table 2.1. The job of `send_enable` and `receive_enable` WQEs is to enable a task being held in a queue. These WQEs receive the number of tasks to enable and the relevant QP as parameters.

Wait WQE

As we previously mentioned, the completion of a task is signaled by a Completion Queue Entry arriving on a Completion Queue. The `wait` WQE provides a way for the HCA to react to the arrival of a CQE on a given CQ. A `wait` WQE receives a CQ reference and the number of entries to wait for as parameters. The execution of a `wait` WQE completes once these conditions are met.

Calc WQE

In addition to handling communication progression, the processor on an HCA with CORE-Direct support is capable of performing arithmetic and logic operations on data elements. This feature can be used to implement operations that combine data movement with computation, such as the `MPI_Reduce` and `MPI_Allreduce` collectives [46]. Even though the computational performance of the HCA is lower than that of the host CPU, offloading computation from the host is beneficial for certain applications. However, we do not deal with computation offloading in this work.

Management Queues

In the current implementation of CORE-Direct, `enable`, `wait`, and `calc` tasks are posted to Send Queues, where they can be interleaved with regular `send` tasks. Revisiting the discussion on QP initialization for CORE-Direct gives rise to the following question: how can we progress communication if a QP is set up to hold tasks in the Send Queue until an enable task activates them, and enable tasks are themselves treated as sends? The answer is that the enable tasks act across QPs.

We can envision two possible scenarios for implementing cross-QP synchronization. One possibility is to set up QPs to *not* hold their `send` tasks and instead use only `wait` tasks for synchronization. In the second approach, a separate QP can be set up to manage other

QPs using enable WQEs. This Management Queue (MQ) does not hold `send` tasks, but the remaining Data QPs do. The two approaches can also be combined for greater flexibility in representing communication patterns.

2.6.2 Barrier Example

Non-blocking collectives using CORE-Direct were first studied by Graham et al. [24, 25] targeting the barrier collective operation. We use a simplified version of the barrier operation to illustrate the use of the CORE-Direct concepts we described in the previous section.

The barrier collective as defined in MPI has the following semantics: a barrier is considered completed by any process only once all processes have started the collective call. Unlike the communication operations we will discuss in subsequent chapters, a barrier does not exchange any user data, and is used purely to synchronize processes. Even though a barrier can be considered the simplest collective operation, it is nevertheless an important one, and is a useful starting point for discussing the implementation of offloaded non-blocking collectives.

One of the possible algorithms to implement the barrier operation is recursive doubling [93]. In this algorithm, $\log_2 n$ steps are taken to complete a barrier among n processes, where n is a power of 2. In each step i, process p communicates with another process q separated by 2^i ranks from p to confirm that process q has entered the barrier. As can be seen from Figure 2.3, carrying out communication in this pattern allows the processes to infer the arrival at the barrier of the processes they do not directly communicate with. For example, by the time rank 2 communicates with rank 0 in step 1, rank 2 has previously synchronized with rank 3. Therefore, rank 0 can infer the arrival of rank 3 when it receives a message from rank 2.

We could envision a naive barrier algorithm in which every process communicates with every other process. The benefit of the recursive doubling barrier is that the number of messages sent by a process is reduced to $\log_2 n$. However, unlike the naive algorithm,

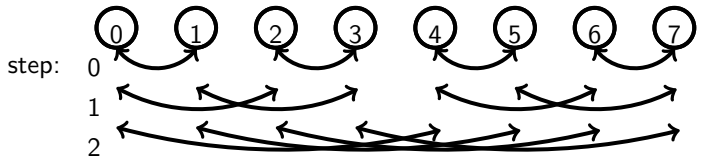


Figure 2.3: Recursive doubling communication pattern for 8 ranks

recursive doubling relies on the order of message arrival. A process cannot simply send all its messages at once: it must do so in response to incoming messages. Returning to the problem of non-blocking collective implementation, we see that this requirement necessitates the use of collective progression, either through software intervention or hardware offloading.

To see how the barrier collective can be implemented using CORE-Direct, let us consider the collective communication from the point of view of a single participating process. As can be seen from Figure 2.3, the process with rank 0 communicates with ranks 1, 2, and 4. During initialization, Queue Pairs are set up and connected to those ranks. Receive Queues are configured to process their tasks immediately, while the Send Queues hold their tasks until explicitly enabled. We use a single Management Queue for this example.

When rank 0 initiates the barrier call, the tasks shown in Table 2.2 are posted to their respective queues. The `send` to rank 1 is enabled immediately after being posted by the first `send_enable` task in the MQ. The next `send` (to rank 2) is only enabled once a message arrives from rank 1. This ensures that rank 2 can infer the arrival of both ranks 0 and 1 once it receives the message. The message to rank 4 is processed similarly. Finally, an additional `wait` task is used to detect the arrival of a message from rank 4 and signal the completion of the collective operation by rank 0.

In this example we saw how CORE-Direct features can be used to express dependencies among messages comprising the collective. We note that the tasks are handled entirely by the InfiniBand HCA once they are posted in the `MPI_Ibarrier` call that initiates the non-blocking barrier. The only interaction required from the host is the detection of collective completion in an `MPI_Wait` call. While the barrier communication is in progress, the user application is free to perform useful work.

Table 2.2: CORE-Direct tasks for recursive doubling barrier. The first column lists the tasks posted to the MQ on rank 0. The Send and Receive Queues are connected to ranks 1, 2, and 4

MQ	RQ (1)	SQ (1)	RQ (2)	SQ (2)	RQ(4)	SQ (4)
send_enable (1) wait (1) send_enable (2) wait (2) send_enable (4) wait (4, signal)	recv	send	recv	send	recv	send

2.7 Summary

In this chapter, we briefly described the programming models for High Performance Computing and set the stage for the remainder of the work. We described MPI, the *de facto* standard for supporting the message-passing paradigm in clusters. We discussed the host-based techniques that can be used for progression of non-blocking operations, as well as offloaded progression.

We also introduced the InfiniBand interconnect and its programming abstractions. We then provided an introduction to the offloading technology used in the rest of this work: Mellanox CORE-Direct. In the following chapters, we will discuss our designs for offloading MPI point-to-point and collective operations, along with our optimization decisions and performance evaluations.

Chapter 3

Offloaded Point-to-point Rendezvous Progression

In the previous chapter, we described the most common programming models for high-performance computing and noted that we will focus our attention on MPI, which is an example of a model using the message-passing paradigm. Recall that MPI provides two kinds of communication primitives: point-to-point and collective. MPI communication functions exist in a blocking form, in which the function must complete before control is returned to the calling program, and a non-blocking form which executes communication in parallel with other work done by the caller. This thesis focuses on non-blocking communication.

In Chapter 2, we introduced the hardware technology that we use to support the designs developed in this work: the InfiniBand architecture (IBA) and the Mellanox CORE-Direct communication offloading extensions to the IBA. We illustrated the primitives of CORE-Direct with an example design and implementation of the `MPI_Ibarrier` collective operation that could synchronize MPI processes in the background while the processes continued their computation. Although the barrier operation served as a convenient demonstration of the use of offloading to deal with communication dependencies, it did not address an important

aspect of communication in MPI: the transfer of user data. Communicating data among processes is a key feature of MPI: it must be provided in all communication operations except the barrier. Therefore, it behooves us to discuss data transfers in the context of offloaded non-blocking MPI operations and to make sure that our designs can support data transfer in an efficient manner. In this chapter, we will deal with non-blocking point-to-point operations. Since point-to-point operations do not need to address communication dependencies between more than two processes, we will be able to narrow our focus exclusively to the issue of data transfer.

Prior to the appearance of non-blocking collectives, the `MPI_Isend` and `MPI_Irecv` family of routines specified in MPI-2.2 [63] were the only way to obtain overlap between communication and computation. Because these operations are well-established, many MPI applications rely on them. In addition to being useful in their own right, point-to-point communication operations can serve as a basis for collective communication designs. In subsequent chapters, we will build upon our point-to-point design to support non-blocking collectives that are specified as part of the MPI-3 standard [64].

In this chapter, we will discuss point-to-point message transfers. We will begin by providing the background on this area of MPI, as well as discussing related work, in Section 3.1. This section will also specify the semantics of MPI message matching that our design will have to respect. Section 3.2 will describe the design of our offloaded eager and rendezvous protocols, as well as their implementation using Mellanox CORE-Direct technology. In Section 3.3, we will present the experimental evaluation of our design and discuss the implications of the results we obtained. Section 3.4 concludes the chapter.

3.1 Background and Related Work

We first introduced MPI point-to-point operations in Section 2.3.1. Recall that a point-to-point communication operation involves a pair of processes: a sender and a receiver. Before

point-to-point communication can occur, the sender and the receiver must have a way of addressing their counterpart and setting up a communication channel. We talked briefly about connection setup and addressing in MPI in Chapter 2. Recall that processes in MPI belong to one or more communicators, and that a process in a communicator is identified by a rank. The MPI library handles the translation of communicators and ranks to the addressing information specific to the communication channel. Connection setup is largely outside the scope of this work: we will describe the structure of the channel necessary for our designs, but will assume that setting up a communication channel is a solved problem, because MPI implementations typically include connection management [11, 104]. We will, however, discuss the addressing of the processes communicating over these channels, and the mapping between the channel addressing scheme and that of MPI.

3.1.1 Message Matching in MPI

When non-blocking communication is employed, the application is free to start a communication operation, and then start another one before the first operation completes. Thus, in the general case, there may be multiple messages in flight between any pair of MPI processes at any given time. Therefore, a method is needed for making message delivery non-ambiguous. As illustrated by the prototypes of the non-blocking point-to-point calls in Figure 3.1, during the invocation of a point-to-point operation, the programmer must indicate both the rank and the communicator of the process to communicate with. The communicator is necessary to fully specify the destination because an MPI process may belong to multiple

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

Figure 3.1: Non-blocking MPI send and receive C function prototypes

communicators. Messages between the same processes are further disambiguated by the tag field. It is also worth restating that messages in MPI are non-overtaking, meaning that messages satisfying a set of matching rules are received in the same order as they were sent. Even when multiple messages are in flight simultaneously, the non-overtaking property of MPI messages must be preserved.

MPI message matching by the receive operation includes wildcards for both the rank and the message tag. Specifying the value `MPI_ANY_SOURCE` for the sender rank allows a receive operation to match a message from any process in the communicator. The `MPI_ANY_TAG` value allows the same semantics for tags. This feature offers additional flexibility to the programmer, because in some scientific applications the source of data to be received may be initially unknown. However, wildcards complicate message matching semantics because they introduce a degree of non-determinism into the communication. When wildcards are used, it is not possible for the receiver to unambiguously determine ahead of time which incoming message will match the receive. The programmer must take additional care to ensure the correctness of message passing in an MPI application using wildcards.

3.1.2 InfiniBand Semantics and Message Passing

In point-to-point communication, the sender specifies a memory buffer in its address space that supplies the data to be transferred to the receiver. The receiver specifies a receiving buffer of the same or larger size into which incoming data will be placed. In order for data transfer to occur, the send operation must be matched with its corresponding receive operation and a message buffer must be ready on both sides of the exchange. However, because MPI processes execute asynchronously, it is possible that either the send or the receive operation is issued first. This means that it is possible for a receiver to find that it does not yet have available data to receive, or for a sender to find that no message buffer is available at the destination.

Generally speaking, similar problems often arise in networking. They can be dealt with

either by buffering or by synchronization. In MPI, this corresponds to two broadly defined categories: *eager* and *rendezvous* protocols. In an eager protocol, a receiver that is late to start its receive operation will buffer any unmatched incoming messages and copy data to the final destination once the destination becomes known. When a rendezvous protocol is used, the sender and the receiver will instead be synchronized, ensuring that the message can be placed into the final location without an intermediate copy operation. Choosing between an eager protocol and a rendezvous protocol involves the tradeoff between the cost of synchronization and the cost of copying the message. Intuitively, short messages are quick to copy, so it would follow that we should prefer the eager protocol for short messages. This assumption is supported by empirical results [23].

Eager Protocols

Let us consider the issues in supporting eager and rendezvous protocols using InfiniBand hardware. We will begin with a discussion of an eager protocol using InfiniBand send/receive semantics. Recall from Chapter 2 that IB adapters directly access application memory buffers and do not have built-in memory for buffering incoming messages. Instead, the InfiniBand `send` WR expects a corresponding `recv` WR to have made a memory buffer available at the destination. The target HCA then places the incoming data into the receiving buffer. If no receive verb has been invoked at the destination by the time the incoming `send` arrives, the target HCA sends a control packet back to the sender signaling the Receiver-Not-Ready (RNR) condition. The sender can then attempt the data transfer again after a timeout. The choice of timeout value involves a tradeoff between increased network traffic due to control packets and increased transfer latency due to the timeout. In general, for optimal performance the RNR state should be avoided whenever possible.

Consequently, MPI implementations of eager protocols will usually manage a set of memory buffers that are ready to receive incoming messages. The MPI library will *eagerly* post `recv` WRs for these buffers to the HCA before the actual MPI-level receive operations

are issued. The eager buffers form the unexpected message queue that the `MPI_Recv` function has to check for any incoming messages that have arrived prior to the start of the call, and continue checking until the matching message is received. Once the message arrives, it can be copied to its final destination.

From the sender's perspective, all of the receiver's eager buffers are equivalent as long as they are large enough to receive the sender's message. Recall from the introduction of Section 3.1 that an MPI process sets up channels to other processes in MPI communicators. In InfiniBand, these channels are implemented using Queue Pairs. As we just discussed, the receiver has to post a `recv` WR to the HCA in anticipation of incoming messages. However, the MPI library generally cannot predict when a given pair of processes will communicate, or indeed whether they will communicate at all. An MPI rank therefore can expect a message from any other rank. That said, associating a set of eager buffers with every connection is wasteful: we may be tying up buffers in channels where they are not needed. Two extensions to the InfiniBand Queue Pairs address this problem. The first is the Shared Receive Queue (SRQ). This feature allows multiple QPs to share a pool of buffers [84, 91]. A further improvement, eXtended Reliable Communication (XRC) [85], extends this sharing to multiple MPI processes running on the same node for additional memory savings.

Rendezvous Protocols

Rendezvous protocols avoid the memory copies employed by eager protocols and place data directly into user application buffers. Although this communication mode is well-supported by InfiniBand hardware, there are certain issues involved in designing IB-based MPI rendezvous protocols.

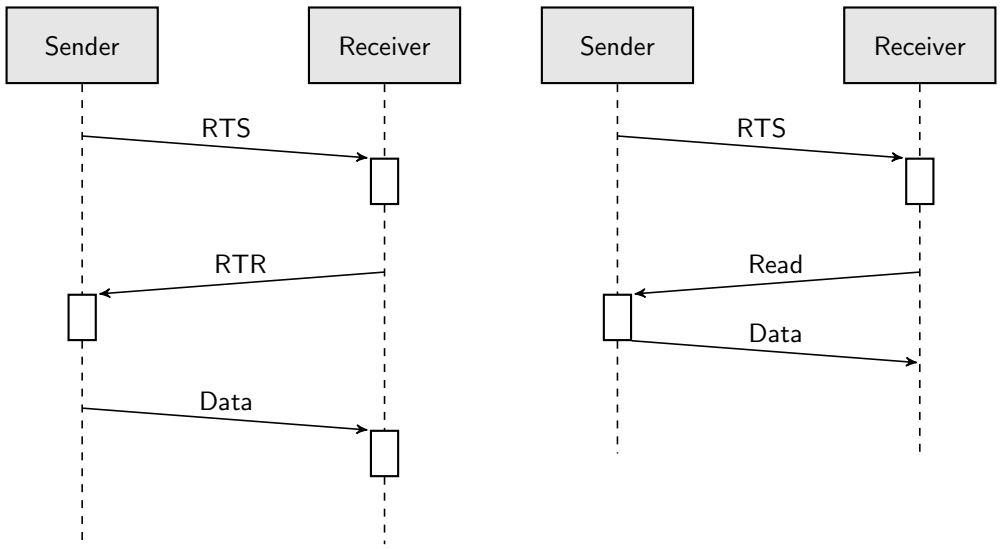
Recall that IB adapters work directly with physical memory. The MPI library does not have knowledge of physical memory addresses, therefore a mapping must be established between the virtual address space and the physical addresses visible to the IB adapter.

This is done by tasking the InfiniBand kernel driver with creating a registered Memory Region (MR). In addition to creating the memory mapping and communicating it to the IB device, memory pages making up the MR must remain pinned in physical memory because it would not be safe for these pages to be swapped out. The call into the kernel, device communication, and page table manipulation for pinning the pages make MR registration an expensive operation compared to most other IB operations, which can be carried out entirely in userspace [62].

The requirement to associate a memory buffer with an IB memory region before it can be used for communication is not a significant detriment to eager protocols because the registration overhead for eager buffers would apply only at application startup. However, for rendezvous protocols the registration requirement presents a problem. Because the address of the buffer supplied for a point-to-point MPI operation is not known prior to its invocation, MR registration overhead becomes a part of the message transfer latency. To reduce this overhead, most MPI implementations using InfiniBand implement a registration cache: they leave user-supplied memory buffers registered in the hope that they will be used in an MPI operation more than once, amortizing the cost of registration [23].

As previously discussed, rendezvous protocols eliminate the overhead of copying messages from the unexpected message queue to their final destinations. However, a tradeoff is involved: in order to ensure that a buffer is available to receive the message, the sender and receiver must first synchronize by following a handshake protocol. Several handshake protocols for rendezvous are well-known in host-progressed InfiniBand-based MPI implementations. We will discuss each of them in turn in order to provide context for the offloading-compatible protocol that we will present later in the chapter.

The 3-message handshake (Figure 3.2a) can be used to provide a rendezvous protocol that does not need hardware support beyond the basic IB `send` and `recv` verbs [3]. The solid boxes in the figure represent process activation. In the first step of the protocol, the sender sends a Request-to-Send (RTS) message to the receiver over the eager message channel. This



(a) 3-message (send or RDMA write) (b) RDMA read

Figure 3.2: Conventional rendezvous protocols. Solid boxes in the timeline represent process activation.

message contains the details necessary to match the message with an `MPI_Recv` operation. The receiver then sets up the user buffer for receiving the message using an IB `recv` verb on a separate large message Queue Pair and sends a Ready-To-Receive (RTR) message to the sender. At this point the sender has ensured the availability of a destination buffer for the message, therefore it can perform the message transfer using an IB `send` verb on the large message QP while avoiding the Receiver-Not-Ready condition.

An optimization to this protocol is possible if the InfiniBand adapter supports one-sided Remote Direct Memory Access `write` functionality [3]. In this variant of the protocol, the final data transfer is carried out using an RDMA `write` from the sender. The sender must have information about the receiver's memory buffer in order to perform the write. The RTR message from the receiver carries this information in addition to performing its role in the handshake sequence. This optimization of the 3-message protocol is so common that many recent papers refer to it as the RDMA write rendezvous protocol and omit the send/receive based variant of the protocol [87]. The sequence of operations for the RDMA write variant

is identical to that shown in Figure 3.2a.

If the InfiniBand adapter supports the RDMA `read` feature, the number of handshake messages can be reduced [92]. In the RDMA `read`-based protocol (Figure 3.2b), the sender supplies enough details to the receiver so that the latter is able to directly read the data from the sender's buffer. An acknowledgement message is necessary to notify the sender that the data transfer is complete. As an optimization, acknowledgements can piggyback on other messages or be batched.

Another possibility is to have the receiver initiate the rendezvous message transfer by sending a Ready-to-Receive message to the sender. This protocol is illustrated in Figure 3.3. Although receiver-initiated rendezvous would allow the number of control messages to be reduced even when using adapters without RDMA `read` support, such a rendezvous protocol cannot support the entirety of the MPI message-passing semantics. Specifically, the receiver has to know the sender of the message, which is an impossible requirement in the presence of the `MPI_ANY_SOURCE` flag. Nevertheless, receiver-initiated rendezvous protocols have received some attention [70, 79]. Most implementations of receiver-initiated rendezvous make the assumption that the application does not make use of `MPI_ANY_SOURCE` and fall back to a different rendezvous protocol should an exception to this rule occur.

3.1.3 Rendezvous Message Progression and Communication/Computation Overlap

The data transfer of a non-blocking MPI point-to-point operation proceeds asynchronously in parallel with the execution of the calling program. As we discussed in Chapter 2, computation/communication overlap can hide communication latency in clusters leading to improved application performance. In order to realize this potential, it is important to ensure that progress can be made on the communication without impeding computation. Ideally, the progression of computation and communication would be completely independent [13].

When no specialized hardware support for overlapping computation and communication

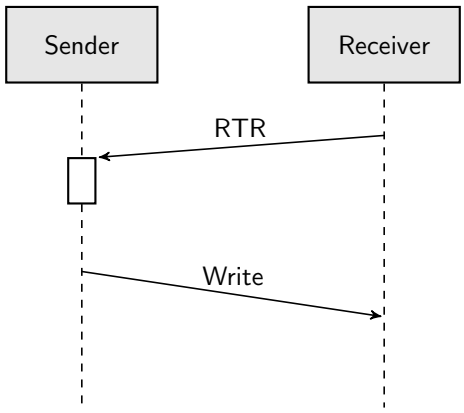


Figure 3.3: Receiver-initiated rendezvous protocol

is available, non-blocking operations can be implemented by having the host processor variously attend to the computation and the communication. Point-to-point communication operations employing a rendezvous protocol must exchange control messages prior to the transfer of user data. Their performance depends on the timely handling of these messages, and consequently on the process arrival pattern at the point-to-point MPI call. Relatively small variations in the process arrival pattern can propagate through the cluster and become amplified when other operations depend on the point-to-point communication [22, 74].

Let us illustrate the problem with an example of a message transfer using a rendezvous protocol with RDMA `read` and host-based progression. As we saw in Section 3.1.2, RDMA `read`-based protocols minimize the number of control messages required. However, even these designs have inefficiencies. In the sequence diagram in Figure 3.2b, the sender executes an `MPI_Send` call before the receiver executes the matching `MPI_Irecv`, meaning that the RTS message will wait in the unexpected message queue at the receiver. It will have to be retrieved by the receiver, who will then execute an RDMA `read` operation and complete the data transfer. In this example, the delay is due solely to the receiver's arrival. Consider, however, what would happen had the processes arrived at their respective calls in a different order (Figure 3.4). Specifically, if receiver executed its `MPI_Irecv` call before the RTS message arrived, it would have to periodically check its unexpected message queue while

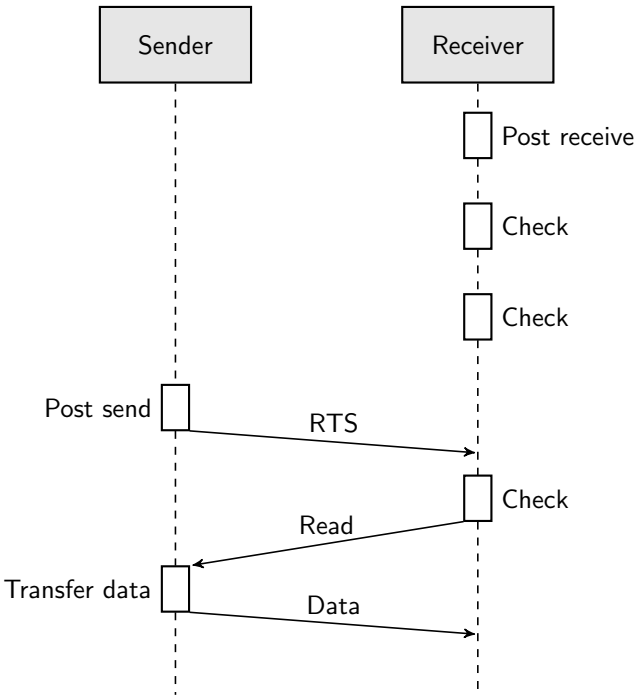


Figure 3.4: Inefficiency in RDMA read rendezvous protocol: the receiver must repeatedly probe for the arrival of an RTS message from a late sender

waiting for the RTS message, switching away from any overlapped computation.

We discussed host-based communication progression approaches in Section 2.5.1. Generally speaking, host-based progression approaches reduce the availability of the processor for computation and add varying amounts of overhead, whether they employ a progression thread, interrupts, or kernel-based progression. This is counter to our goal of making communication and computation independent. Alternatively, communication can be offloaded to special-purpose networking hardware. In this thesis, we improve the overlap between computation and communication by designing communication operations to take advantage of communication offloading. We will now discuss the current work in the field, and then present our design of a rendezvous protocol suitable for hardware offloading. With this design, we aim to achieve greater processor availability for computation by eliminating context switching associated with software-based progression.

3.1.4 Related Work

Eager and rendezvous protocols are in use in most MPI implementations using InfiniBand and its precursor technology, the Virtual Interface Architecture (VIA) [9, 12, 23]. Both eager and rendezvous protocols have seen a number of refinements since their inception. For the eager protocol, the improvements have largely focused on reducing the memory footprint of the pinned memory buffers. This has been achieved through the use of Shared Receive Queues (SRQs) [91] that allow a buffer pool to be shared between multiple connections. It is also possible to optimize eager buffer pool utilization by stratifying messages by priority and splitting the pool accordingly, as is done in the B-SRQ protocol in OpenMPI [84]. The memory footprint and scalability of MPI applications has recently been reduced even further through the use of the eXtended Reliable Communication (XRC) transport that permits buffers to be shared among several MPI processes on the same node [85]. A different approach to optimizing eager MPI messaging using a polling-based RDMA design has been shown by Liu, Wu, and Panda [57]. We do not focus on memory footprint reduction in this chapter, but we will revisit this topic in Chapter 5.

Rashti and Afsahi examined the conditions for switching between eager and rendezvous protocols in their work and found another source of performance improvement [80]. They concluded that it is possible to improve application performance by using the rendezvous protocol for frequently reused message buffers even for short messages that would normally be processed by the eager protocol.

Handshaking protocols in rendezvous point-to-point MPI messaging have received significant attention from researchers. RDMA `read`-based rendezvous is discussed in a paper by Sur et al. [92]. Pakin investigated receiver-initiated rendezvous protocols and discussed their advantages and limitations in MPI compatibility [70]. Our approach has similar compatibility limitations, but offloads the communication to CORE-Direct hardware in order to improve the overlap between communication and computation. Rashti and Afsahi [78, 79] as well as

Small and Yuan [87, 86] pursued various rendezvous protocol designs that can be viewed as hybrid designs where sender- and receiver-initiated rendezvous are used simultaneously. Although these approaches are interesting, we reserve investigations of offloaded hybrid rendezvous protocols for future work.

Optimization of host-based progression of rendezvous protocols is the subject of papers by Kumar et al. [50] and Zounmevo and Afsahi [106]. Unlike these designs, our progression is based on hardware offloading. Our protocol is most similar to that used by Venkata et al. for broadcast collective communication [100]. However, MPI collective communication operations do not need to support source and tag matching, unlike MPI point-to-point operations. We therefore provide this additional feature in our design. In the message envelope matching area our work bears some similarity to the TupleQ design by Koop, Sridhar, and Panda [49] in that a CQ is used for message envelope identification, though we do not make use of the XRC InfiniBand transfer protocol. The TupleQ design does not use communication offloading, unlike the design we present in this chapter.

Brightwell and Underwood investigated the performance of non-blocking operations and have found that for optimal performance, the communication and computation must be overlapped while making progress independently [13]. Brightwell and Underwood suggested offloading as a promising approach to meet these goals, though they did not empirically evaluate communication offloading in their work. In this work we show that offloading can indeed be used to improve non-blocking communication in practice.

3.2 Design for Offloading Non-blocking Point-to-point Communication

Taking communication out of the critical path of the program execution allows communication latency to be hidden while additionally conferring to the application the benefit of tolerance to scheduling jitter. However, an implementation of communication/computation

overlap might require some overhead for managing the background communication. Ideally, rendezvous message progression would be done independently without having to interrupt the computation being performed by the application. We aim to achieve this goal by transferring the responsibility for message progression to dedicated hardware. In this section, we discuss our design for offloading both eager and rendezvous protocol progression using CORE-Direct.

3.2.1 Eager Protocol

Let us first discuss the eager protocol in the context of CORE-Direct. Eager non-blocking MPI transfers can be carried out using offloading in a similar fashion to those employing host-based progression. In the eager protocol, whether in the offloaded or host-progressed case, the receiver maintains a number of pre-posted buffers for incoming messages. The eager protocol uses these pre-posted buffers to receive messages regardless of whether the receiver has already indicated before the message arrives that it is expecting the message through an MPI call in the `MPI_Recv` family. Once the receiver has ensured that a buffer is available to receive the message, non-blocking progression can be achieved simply by delaying the check of the unexpected message queue until non-blocking receive completion is explicitly requested by the application through a call to `MPI_Wait`. For its part, the sender can handle blocking and non-blocking sends identically. It can send the message across the InfiniBand network when convenient, because it can assume that a buffer is available at the destination to receive the message.

Only a single `send` or `recv` InfiniBand verb invocation is needed for handling an eager message, therefore progression of eager messages is in effect already fully offloaded to the HCA even when CORE-Direct extensions are not in use. The purpose of the CORE-Direct extensions is to deal with chains of dependent communication operations. Because no control messages are involved in the eager protocol, the inefficiencies of using the processor to check the unexpected message queue do not apply to eager transfers. Using CORE-Direct for offloading a single eager transfer therefore does not offer an advantage in

computation/communication overlap. Because of this, we will not presently discuss offloading eager communication further. However, we note that offloaded eager transfers find a use as building blocks of collective communication operations. Because collective communications are comprised of multiple communication operations, CORE-Direct extensions are beneficial in this application, as we shall see in Chapter 4.

3.2.2 Rendezvous Protocol

In contrast to the eager protocol, the rendezvous protocol presents a challenge for host-based progression, as we discussed in Section 3.1.3. The difficulties stem from the requirement to exchange control messages in order to complete the operation and facilitate interruptions of the application by the message progression handler. Offloading the rendezvous protocol to CORE-Direct hardware can result in better overlap between communication and computation, because the sharing of the host processor between message progression and computation becomes unnecessary.

Any rendezvous protocol we design must correctly handle message matching. As we saw earlier in Section 2.3.1, messages in MPI are matched based on their source, communicator, and tag. When host-based progression is used, this information can be included in the Request-to-Send (RTS) message from the sender to the receiver. Once this message arrives over the eager message channel, the receiver can correctly match it to the corresponding `MPI_Recv` or similar call. The receiver can then progress the rendezvous protocol further.

In contrast, the CORE-Direct specification provides no way of examining the contents of the message. We are therefore unable to examine the RTS message as part of the message matching process. The only kind of information about message arrival that is available to an offloaded protocol using CORE-Direct is in the form of Completion Queue Entries (CQEs) appearing on Completion Queues (CQs). Consequently, in our design we infer the matching information for incoming messages from the completion event arriving on a specific CQ.

There is a second issue in designing a rendezvous protocol for CORE-Direct. In order

not to interrupt the application's computation, all InfiniBand Work Requests making up the rendezvous transfer must be posted to the HCA at the beginning of the point-to-point operation. This means that we can only use information that is known at the start of the transfer when creating these WRs, and must design our protocol to deal with this limitation. Unfortunately, the limitation means that our protocol is unable to support the MPI wildcard matching features. Let us discuss how this implication comes about.

Consider a design in which the data transfer is initiated by the sender. The sender knows which receiver to contact at the time when it invokes the `MPI_Isend` call. However, it does not know whether the receiver has arrived at its corresponding receive call. Because the sender cannot make an assumption about the availability of a buffer to receive its message, and because we choose to avoid triggering the RNR condition, the sender should first establish buffer availability by sending an RTS signal to the receiver. However, if the receiver is using either the `MPI_ANY_SOURCE` or the `MPI_ANY_TAG` feature when it invokes `MPI_Irecv` (or similar function), it cannot determine which CQ will receive the RTS message, and as a result cannot post all the CORE-Direct tasks at the start of the operation.

In Section 3.1.4, we discussed receiver-initiated point-to-point rendezvous transfers. To handle rendezvous transfers using CORE-Direct, we adopt a similar approach. In our design, a separate QP is created for each $(communicator, source, tag)$ triple. Each of these QPs serves as a simplex channel with the MPI process at one end acting as a sender and the other as a receiver. If a duplex connection is desired between the two MPI ranks, a second offloaded message channel can be created for the other data transfer direction.

The receiver uses the Send Queue (SQ) of the QP to transfer a Clear-to-Send (CTS) signal to the sender. The sender's Receive Queue (QP) always has a number of pre-posted InfiniBand `recv` WRs. This ensures that the QP does not enter the Receiver-not-Ready state and generate retransmissions, making the delivery of CTS messages similar to the control messages sent over the eager message channel in host-based rendezvous progression. The contents of the CTS message serve no purpose, therefore the `recv` verbs for CTS signals

are invoked with zero length and do not receive any data. On the receiver side, we use the RDMA `write with immediate data` verb for the CTS signal. The immediate data feature ensures that a `recv` WR will be consumed at the destination and the CTS condition will be signalled to the sender.

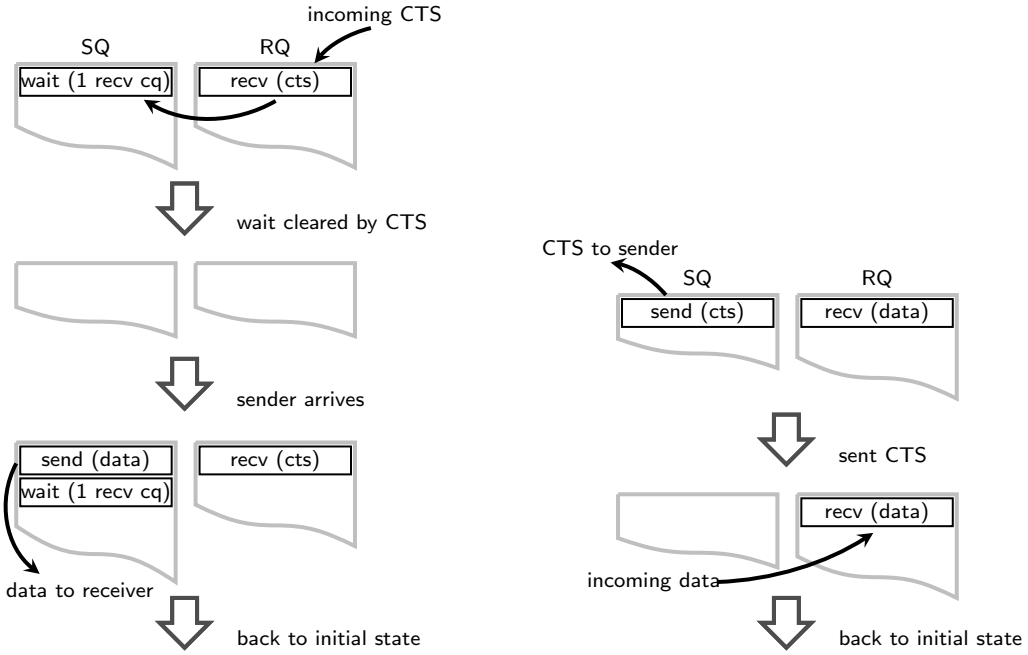
The first CORE-Direct task posted by the sender to its SQ is a `wait` task, as illustrated in Figure 3.5. This task waits for a CQE to appear on the RQ's CQ in response to an incoming CTS signal. The completion of this task implies that the receiver has invoked the matching receive call and has a buffer waiting for the sender's message. The task following the `wait` is the actual large message `send`. This task will begin once the receive buffer availability is established by the Clear-to-Send (CTS) signal.

This design works both for the case where the receiver arrives at the point-to-point call before the sender, and vice versa. In the first case (shown in Figure 3.6), the receiver executes its operation before the sender, and because its buffers are ready to receive data, it sends a CTS control message. After being received by the sender, this message clears a `recv` WQE in its RQ and generates a CQE on the corresponding CQ. The sender then arrives at the send call and posts a `wait` WQE to detect the arrival of the CTS message. Because a CQE is already present, this `wait` WQE will be immediately completed. Upon the completion of the CTS `wait`, the `send` will perform the actual data transfer to the receiver. In the second possible scenario, the sender arrives before the receiver and all WQEs are posted by the sender before the CTS message arrives. In a sequence of events similar to the early receiver case, the `wait` WQE is cleared by the incoming CTS message, allowing the data transfer to proceed.

Lastly, we note that the requirement of a dedicated QP for every $(communicator, source, tag)$ triple may lead to a significant increase in resource consumption by applications utilizing a large number of such triples. To handle this possibility, as well as to allow for compatibility with the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` features, we implement a fallback mode which disables offloaded rendezvous progression when necessary.

(a) Sender (b) Receiver

Figure 3.5: Offload endpoint queues at the start of communication



(a) Sender (b) Receiver

Figure 3.6: Offloaded rendezvous progression for early receiver case

3.3 Experimental Results and Analysis

In this section, we examine the latency of our non-blocking rendezvous algorithm as well as its communication/computation overlap potential. In our evaluation, we compare our CORE-Direct-based implementation of the offloaded rendezvous protocol for point-to-point messages with that of the MVAPICH2 1.9 [65] Nemesis InfiniBand channel [58]. The threshold for the switch between eager and rendezvous protocols in MVAPICH2 was set to the default value of 12KB. Although the use of CORE-Direct hardware for eager messages is valuable when designing collective operations, as we will do in Chapter 4, it offers no benefit for single

eager message transfers, as we discussed in Section 3.2.1. Since the focus of this chapter is on offloading the rendezvous protocol, we used it for all message sizes when testing our the offloaded implementation.

For consistency, we used the NBCBench benchmark [37] to measure latency and overlap potential in this as well as subsequent chapters. The benchmark was run for 1000 iterations for each data size. To trigger host-based progression, the `MPI_Test` function was called whenever a 2048-byte block was transferred. NBCBench synchronizes the starting points of operations under test.

3.3.1 System Configuration

The testbed for the rendezvous benchmark was made up of two Dell PowerEdge 2850 servers, each containing two 2.80GHz dual-core Intel Xeon Paxville processors, 4 GB DDR2 memory, and a Mellanox ConnectX-2 MT25418 HCA. The nodes use Mellanox OFED 1.5.3-1 and 64-bit CentOS 5.5 with kernel 2.6.18-194.26.1.el5. On both nodes, a single HCA port was connected to a Mellanox Infiniscale-IV switch.

3.3.2 Latency

Because we are only interested in testing the rendezvous protocol, the smallest message size used was 16KB, which is above the default rendezvous threshold in MVAPICH2. Figure 3.7 shows the latency of the non-blocking send and receive operations of MVAPICH2 and of our offloaded design. When both implementations utilize a rendezvous protocol, the latency scales identically: it is linearly related to the message size. Though the offloaded implementation has a slight performance penalty in this test, its latency is consistently within a constant factor of the host-based implementation. However, the offloaded implementation offers the additional benefit of computation/communication overlap, which we will examine in the next section.

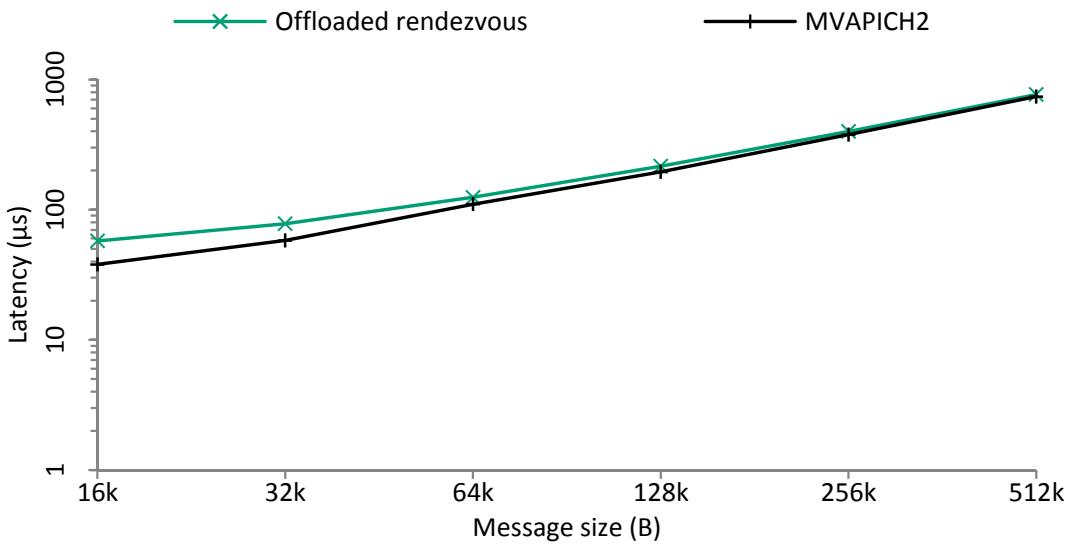


Figure 3.7: Offloaded vs. host-progressed rendezvous message latency

3.3.3 Overlap Potential

Overlap potential is a measure of the overhead required to manage the background communication. If we could come up with a design that would not require any attention from the host processor then the entirety of the communication time could be spent on computation. A certain amount of processor overhead is unavoidable, even when using offloaded communication. In this section, we empirically obtain a measure of the percentage of the communication time that can be overlapped with computation.

Because the transfer time of short messages is insignificant compared to the overhead of offloading, using offloading to overlap these transfers is not worthwhile. As the message size increases so does the opportunity for overlap. Therefore we continue investigating message sizes of 16KB and above.

The NBCBench benchmark measures overlap according to the formula

$$overlap = 1 - \frac{t_{comp+comm} - t_{comp}}{t_{comm}}$$

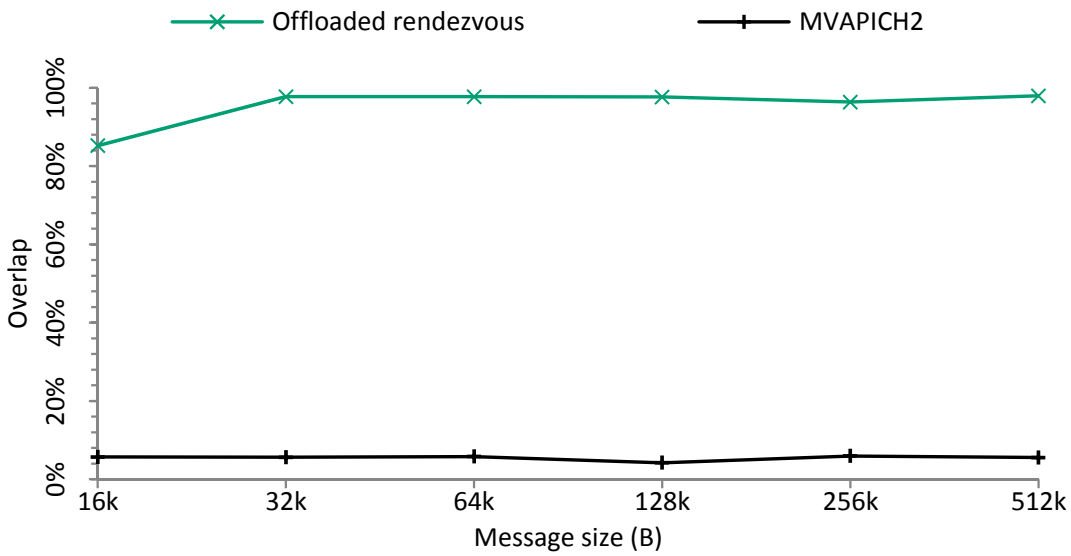


Figure 3.8: Offloaded vs. host-progressed rendezvous overlap capability

where $t_{comp+comm}$ is obtained by starting the nonblocking communication first, then performing a predefined amount of synthetic computation, and finally waiting for the communication to complete. t_{comm} refers to blocking communication time. t_{comp} is chosen to be close to t_{comm} in an effort to minimize measurement error.

The overlap potential results are presented in Figure 3.8. In our comparison testing the host-progressed rendezvous protocol exhibits consistently lower overlap potential than the offloaded design. The offloaded rendezvous overlap potential was as high as 98% for 512KB messages, while MVAPICH2 did not achieve good overlap performance in our testing.

We position our design for offloaded rendezvous point-to-point communication using larger messages, as is traditional for rendezvous protocols. However, as we mentioned in Section 3.1.1, short incoming messages may successfully match large receives. This implies that our design must be able to handle all message sizes. The impact of this requirement on application performance may vary. As long as most messages can benefit from computation/communication overlap, we can tolerate using the offloaded rendezvous

Table 3.1: Percentage of injected noise delay propagated from receiver to sender

Message Size	MVAPICH2	Offloaded Rendezvous
1B – 12KB	4%	2%
16KB - 512KB	99%	2%

channel for short messages without incurring a significant penalty. Should the application use mostly short messages, however, it might be beneficial to fall back to software progression, at least temporarily. This tradeoff likely merits additional investigation.

3.3.4 Noise Tolerance

We noted in Section 3.1.3 that offloading non-blocking communication can alleviate the problem of scheduling variability. To test this in practice, we modified the NBCBench benchmark to insert a 1 second pause between the issuing of the `MPI_Irecv` call and its corresponding `MPI_Wait`. We then measured the message latency at the sender to obtain the percentage of the delay that is propagated from the receiver to its communication peer. The results are summarized in Table 3.1. We performed our testing for all message sizes in this section, including those smaller than 12KB.

As expected, the eager protocol was essentially immune to the inserted delay because no control messages are involved. However, once the message size passed 12KB and both implementations started using the rendezvous protocol, we saw that the host-progressed solution propagated most of the receiver's delay to the sender. In contrast, the offloaded protocol was largely unaffected: the sender's latency stayed essentially the same.

Even though the large delay at the receiver is an exaggeration, this test showcases the resilience to OS noise of the offloaded rendezvous protocol. We can see that this protocol can maintain its latency and overlap characteristics even in the face of a large discrepancy in scheduling between the communicating processes. Programmers using host-progressed point-to-point communication in MPI currently take care to match the arrival times of

the sender and the receiver, however, an offloaded point-to-point design imposes no such restrictions, and promises to free programmers from this burden.

3.4 Summary

In this chapter, we explored the point-to-point messaging features of MPI. We described the general principles of eager protocols that are most useful for short messages, as well as rendezvous protocols that optimize for large message transfers by avoiding additional memory copying. We talked about the issues that MPI designers must address in order to implement these operations efficiently using the InfiniBand interconnect. We discussed the inefficiencies in using the host for progression of rendezvous protocols and showed how offloading can be used to alleviate these inefficiencies.

We evaluated the latency and the potential for overlapping communication with computation that is afforded by offloaded rendezvous point-to-point operations and compared these characteristics to those of a host-progressed implementation in MVAPICH2. We found that offloaded operations provide competitive latency and greater potential for computation/communication overlap, in addition to having better tolerance to scheduling noise than host-based operations.

Additionally, we touched on the applicability of offloading to eager protocols. Although offloading individual eager message transfers ultimately offers no benefit, the methods for handling eager and rendezvous transfers developed in this chapter form a foundation for the work on collective communication that we will discuss next.

Chapter 4

Flat and Hierarchical Non-blocking Offloaded Collectives

In the previous chapter we discussed the process of transferring a single message between a pair of MPI processes. We will now build upon this foundation and investigate higher-order MPI operations – collectives – which we introduced in Section 2.3.2. As we have seen, a collective operation involves communication among a group of processes belonging to a communicator. The MPI specification defines the semantics of collective operations in terms of their results but does not stipulate specific implementations. This affords MPI designers plenty of opportunities for optimizing collective operations for execution in specific systems.

It is desirable for a process participating in a collective operation to perform useful work while waiting for other processes to begin their part of the exchange, for data transfers to complete, and for message dependencies to be satisfied. We previously examined overlapping computation with point-to-point communication operations in Chapter 3. Recall that we concluded that it is not trivial to overlap communication with computation in a manner that compromises neither the latency of the communication nor the availability of processor time for computation. Offloading communication progression emerged as the solution to

these issues for point-to-point communication.

In this chapter, we investigate how offloading can help in obtaining efficient overlap between computation and collective communication. Section 4.1 starts the chapter with a review of related work. In Sections 4.2, 4.3, and 4.4, we develop non-blocking allgather collectives that meet the computation/communication overlap demands using CORE-Direct offloading in flat and hierarchical communicators. We evaluate the performance of our designs using microbenchmarks and an application kernel in Section 4.5. Section 4.6 concludes the chapter.

4.1 Related Work

In Chapter 3, we briefly discussed host-based progression of non-blocking point-to-point operations. This approach can also be used to implement non-blocking collectives. For example, thread-based progression has been used by Hoefler et al. in their work on the libNBC library [33]. We use the libNBC library as a baseline against which to compare our design of offloaded non-blocking collectives. We continue using the large message rendezvous transfer mechanism described in Chapter 3 in the context of collective communication.

Kandalla et al. [48] implemented the `MPI_Alltoall` collective operation using CORE-Direct and investigated the speedup of parallel 3D FFT due to overlapping communication and computation. This chapter focuses on the `MPI_Allgather` collective operation and evaluates its performance using a Radix Sort application kernel. Other works that discuss CORE-Direct offloading in the context of various MPI collectives include work on the barrier collective by Graham et al. [24, 25], its extension to hierarchical collectives by Rabinovitz et al. [77], and work on the `MPI_Allreduce` collective by Kandalla et al. [46]. The allreduce work is distinguished by its leverage of the calculation capabilities of the CORE-Direct hardware.

The material in this chapter is based on the investigation of offloaded non-blocking

allgather collectives by Inozemtsev and Afsahi [40]. We should point out two relevant works that have been published since the completion of [40]. The first is the investigation of the non-blocking all-to-all collective by Venkata et al. [101]. The main contribution of this work is the optimization of the communication using RDMA and scatter-gather features of the HCA. Although we do not examine these optimizations in this chapter, we note that they are applicable to the `MPI_Allgather` operation as well. The authors of [101] noted that the use of the SGE feature limits the scalability of their algorithm. In a follow-up work focusing on the `MPI_Allgather` operation, Ladd et al. [52] present a k-nomial tree algorithm suitable for CORE-Direct offloading that avoids the SGE feature.

A number of authors have explored the use of shared on-node memory for optimizing collective communication without the use of offloading. The works most pertinent to this thesis focus on the application of hierarchical communicators to allgather and the related all-to-all collective operation. These works include the studies of hierarchical allgather optimizations by Mamidala et al. [60, 59] and Kandalla et al. [47], as well as the adaptive allgather algorithm proposed by Träff [96]. Ladd et al. published a detailed investigation of the effects of the cache memory hierarchy in modern clusters on collective operations [51]. Li et al. investigated optimization of shared-memory communication in NUMA systems that make up many modern clusters [54]. Buntinas et al. compared different techniques of using shared memory for intranode communication in MPI [15]. We chose an approach to maintaining a shared memory region that does not rely on special support from the operating system.

Several groups have sought to improve the blocking `MPI_Allgather` operation by spreading the workload of network message processing among multiple processor cores. The multileader allgather design by Kandalla et al. [47] and the multigroup algorithms by Qian et al. [76] have been shown to decrease communication latency. We investigated the applicability of these approaches to offloaded collectives. Additionally, we investigated the use of multiple network interface ports for offloaded collective communication. This technique has previously been shown to improve the latency of blocking collective operations [75, 73].

4.2 Non-blocking Collective Design

In the allgather collective, each participating process contributes a block of data and receives the concatenation of all the blocks in the collective in process rank order [64]. A naive algorithm implementing the `MPI_Allgather` collective operation could simply have each rank send its block of data to every other rank. In practice, better algorithms are employed for `MPI_Allgather`. We used three well-known algorithms: standard exchange [10], Bruck [14], and ring [93].

4.2.1 Standard Exchange

The recursive doubling algorithm that we explored in the context of the barrier collective operation in Chapter 2 is also applicable to the allgather collective [93]. In the discussion below, we assume that communication can be carried out simultaneously on k ports. The *standard exchange* algorithm that we discuss in this section is a generalization of recursive doubling for k-port modelling [10, 75].

Standard exchange uses $\log_{k+1} N$ rounds when the number of processes N is a power of $k+1$. In each round j of the algorithm, process p communicates with processes $p+(k+1)^j$, $p+2(k+1)^j$, ..., $p+k(k+1)^j$ modulo N. $(k+1)^j$ blocks of data are sent and received in each round and are placed at the required offset in the destination buffer. Figure 4.1 shows an example of the execution of the standard exchange algorithm with 8 ranks and the number of ports $k=1$. Each square block represents the data item contributed by the corresponding process. We see that in round 0 the data blocks are communicated between processes 1 step away with the distance and the message size doubling in each round. The communication completes in $\log_2 8 = 3$ rounds with the contributions of all ranks known to the entire communicator. If the number of ranks is not a power of $k+1$, an additional round of communication is required to propagate data to the excess ranks.

The example in Figure 4.2 shows the effect using two ports for communication. This

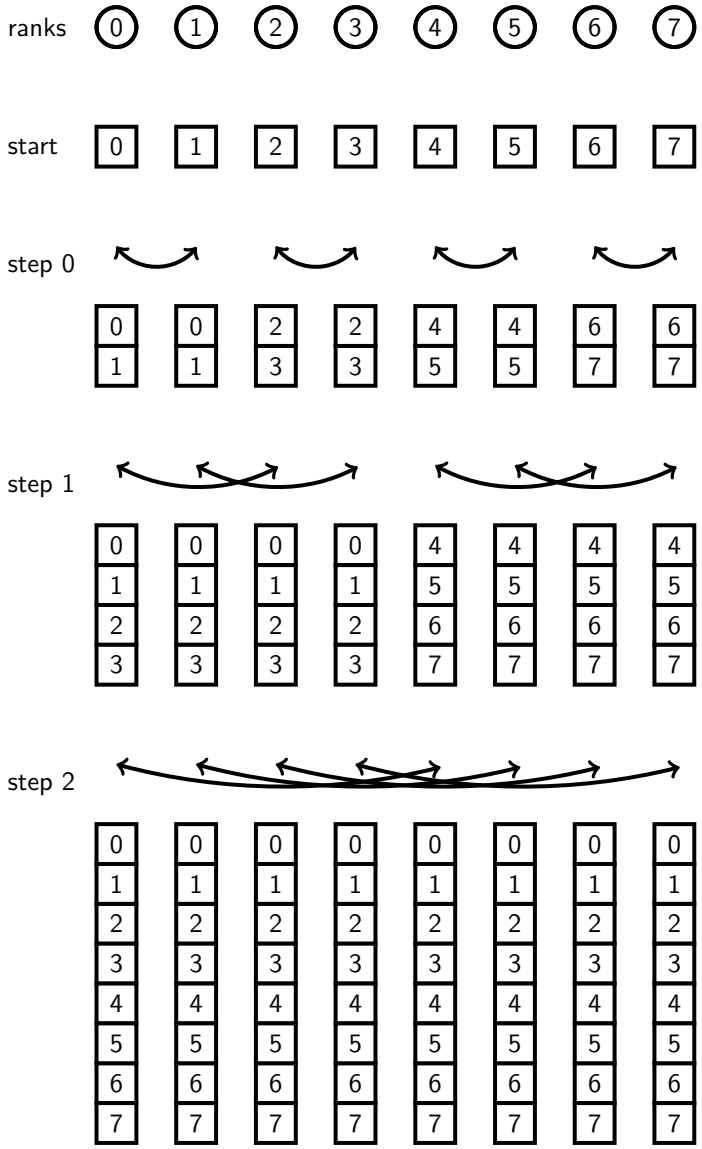


Figure 4.1: Execution of the standard exchange allgather algorithm with 8 ranks and single-port modelling

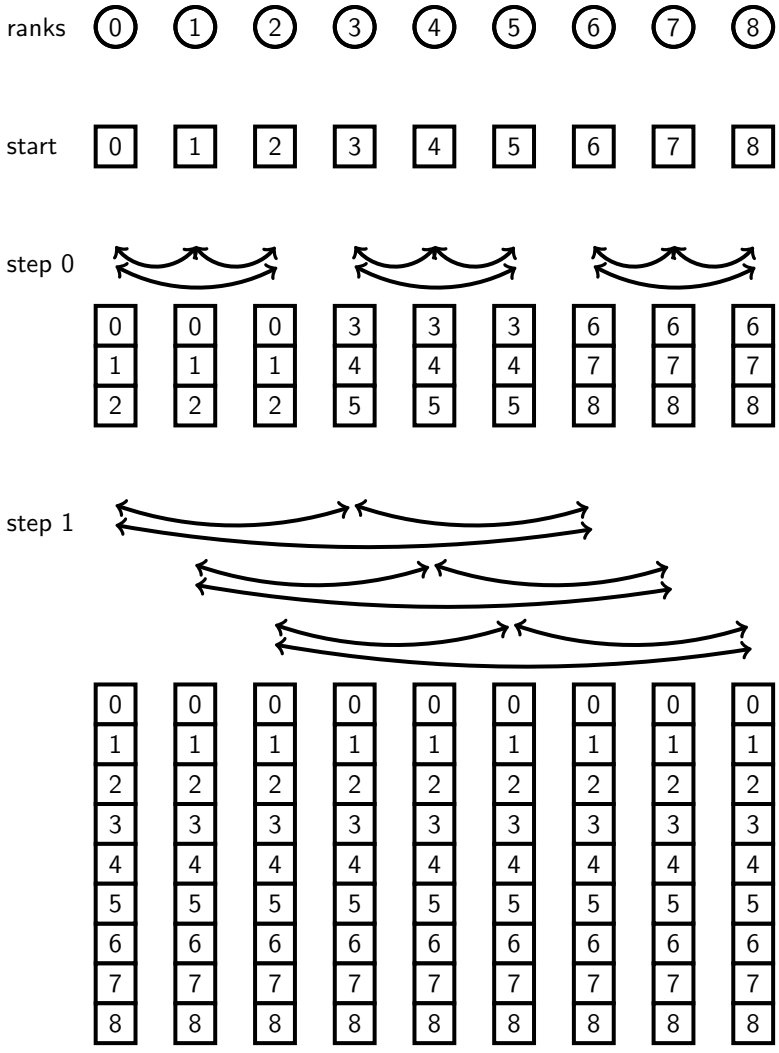


Figure 4.2: Execution of the standard exchange allgather algorithm with 9 ranks and 2-port modelling

time the ranks communicate in groups of $k+1=3$ in each round. The distance and message length triple in each step. The algorithm proceeds in the same fashion as in the $k=1$ case and completes in $\log_3 9 = 2$ steps.

The ports described by the model may be implemented either as separate physical interface ports, or be multiplexed over a smaller number of physical ports. An increase in the number of ports in the communication model means that more messages are transferred in each round. Although this property of multiport algorithms may reduce the latency of the collective, especially if multiple hardware ports are available, it also potentially increases network contention, which may have an adverse effect.

4.2.2 Bruck

The Bruck algorithm [14] has two phases: network exchange and local shift. In the discussion below, we again assume that communication can be carried out on k ports. The first phase consists of $\lceil \log_{k+1} N \rceil$ steps, where N is the number of processes in the communicator. If we number the rounds of the network phase starting at 0, in round j process p sends its data to processes $p+(k+1)^j, p+2(k+1)^j, ..., p+k(k+1)^j$ modulo N and receives messages from $p-(k+1)^j, p-2(k+1)^j, ..., p-k(k+1)^j$ modulo N. The incoming data are appended to the data that have been previously received. The result is sent out in the next round of communication.

Figure 4.3 shows an example of execution of the Bruck algorithm with 5 ranks and the number of ports $k=1$. We see data being sent to ranks at a distance 1 away in step 0. The message sizes and distances increase by a factor of $k+1=2$ in each round.

Unlike standard exchange, the Bruck algorithm gracefully handles the case where N is not a power of $k+1$. In this case, only the data blocks that were missed by the previous rounds are communicated in the last round. An example of this can be seen in Figure 4.3 in Step 2: because $N=5$, this extra round is required to communicate the single block that each process is missing.

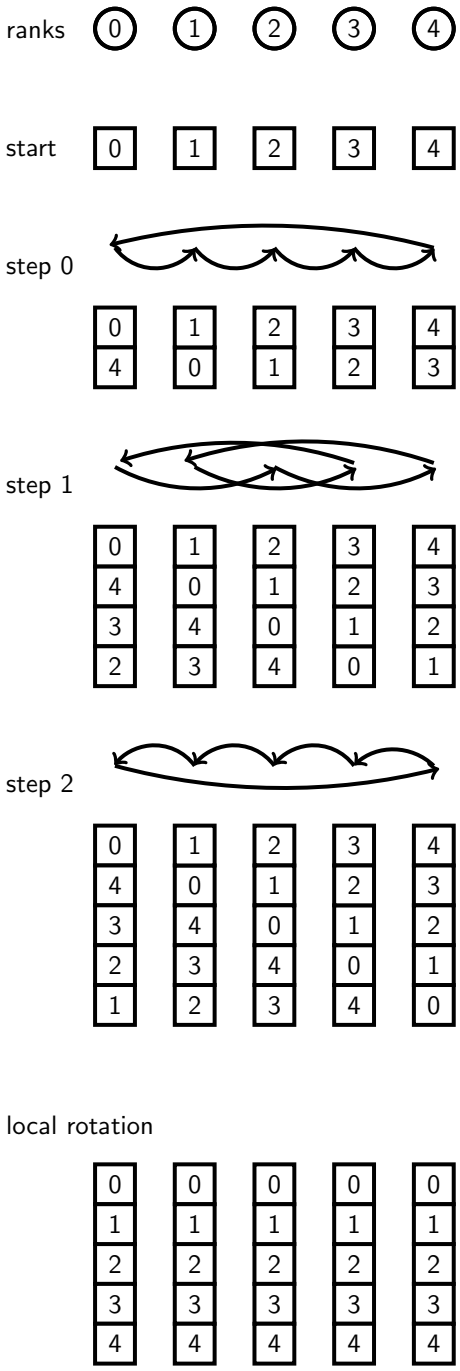


Figure 4.3: Execution of the Bruck allgather algorithm with 5 ranks and single-port modelling

Though the property of efficiently handling any number of processes is a desirable one, the Bruck algorithm has one additional requirement that may be a drawback. The second phase of the Bruck algorithm performs a local circular rotation by p blocks on process p to place the data in the correct order. This additional local memory operation is not necessary in standard exchange or the ring algorithm that we will discuss next. We should note, however, that in a hierarchical collective design, all the mentioned algorithms will require a memory copy operation to complete, though Bruck remains at a slight disadvantage because its memory copy is non-contiguous. Lastly, standard exchange has been shown to have a performance advantage over the Bruck algorithm in networks where a pairwise communication pattern is beneficial [8].

4.2.3 Ring

The algorithms we have seen so far increase the size of the messages used in each round in order to reduce the number of rounds required to complete the communication. In contrast, the ring algorithm keeps the message size constant in each round. The ring algorithm requires $N - 1$ rounds for N processes [93]. Process p sends a single block of data to $p + 1$ and receives a block from $p - 1$ in every round. In Figure 4.4 the execution of the ring algorithm is illustrated for a communicator with $p = 4$ ranks. We can see that each rank has only two connections in the ring algorithm, and the execution takes $p - 1 = 3$ steps.

Even though the linear $O(N)$ number of steps appears inefficient in comparison to the Bruck and standard exchange algorithms, the ring algorithm is often the best choice for large messages in practice. This is due to its nearest-neighbour communication pattern which translates into reduced network contention and a small number of connections that need to be set up.

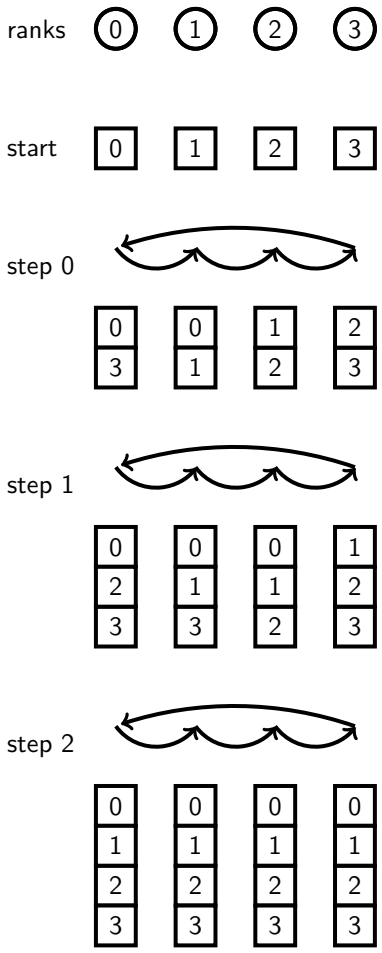


Figure 4.4: Execution of the ring allgather algorithm with 4 ranks

4.2.4 Algorithm Comparison

We have presented two algorithms where the number of communication rounds scales logarithmically, and one linearly scaling algorithm to implement the `MPI_Allgather` collective operation. The choice of algorithm in practice depends on the size of the messages that are being exchanged. To examine this tradeoff, we employ Hockney's model [31]

$$t = \alpha + \beta c$$

where α refers to the communication startup time of the individual messages, β is the reciprocal of the network bandwidth, and c is the size of the message. Ignoring the effects of network contention, the lower bound on the cost of the collective operation can be inferred from the cost of the messages sent by the individual processes. Treating c as the size of individual data blocks in the allgather operation, and summing the latency terms over all rounds, we obtain

$$t \in O(\log N)\alpha + O(N \log N)\beta c$$

for the standard exchange and Bruck algorithms, and

$$t \in O(N)\alpha + O(\log N)\beta c$$

for the ring algorithm.

If the messages are small, startup costs expressed by the α term dominate the data transfer latency, and the algorithms where the startup term scales logarithmically are preferable. In the case of large messages, however, the linear ring algorithm is preferable, because it minimizes the bandwidth-dependent β term.

Comparing the logarithmic algorithms further, we see that in the standard exchange algorithm no local rotation step is required. The pairwise communication pattern of standard exchange and the lack of local rotation can provide an advantage over the Bruck algorithm

[8]. However, when the number of processes is not a power of $(k+1)$, Bruck usually handles the excess ranks more efficiently.

Because no algorithm is a clear best choice for all scenarios, we choose the algorithm at runtime based on the size of the communicator and the size of the messages.

4.3 Flat Collective Design

Once the choice of algorithm has been made, we can set up the required connections and generate a task template that describes the communication to be performed. The template contains all the information about the required CORE-Direct tasks except for the pointers to data buffers. At each rank, the local portion of the communication is expressed using `send` and `recv` tasks, with rounds delimited by `wait` tasks. These are followed by `send_enable` tasks that start the next round. The last `wait` task generates a CQE that signals the completion of the offloaded operation. The task template is preserved between `MPI_Iallgather` calls. Upon an invocation, a copy of the template is filled in with the addresses and sizes of the data blocks involved in each step of the transfer.

Because the processes participating in a collective cannot be expected to invoke the collective call simultaneously, a collective implementation must efficiently handle situations where one of the processes starts communication before its peer is ready. Two approaches to addressing this issue exist in the design space, namely, the *eager* and *rendezvous* protocols we introduced in Chapter 3. We will now see how these protocols can be applied in the context of collective communication.

4.3.1 Eager Protocol

As discussed in Chapter 3, in the eager protocol messages are transferred between buffers that are internal to the MPI library. To avoid triggering the Receiver-Not-Ready condition, we would like to pre-post the `recv` tasks involving these buffers to their QPs before the collective

Table 4.1: Eager protocol task list for rank 0 in single-port standard exchange among 8 ranks

Step 0	recv from 1 send to 1 wait (1 recv CQ) from 1
Step 1	recv from 2 recv from 2 send to 2 send to 2 wait (2 recv CQ) from 2
Step 2	recv from 4 recv from 4 recv from 4 recv from 4 send to 4 send to 4 send to 4 send to 4 wait (4 recv CQ) from 4

communication begins. However, this choice poses a problem: because the communication has not yet begun, there is no way to find out the size of the messages involved, and by extension – the offset at which the contribution of each process should be placed.

To deal with this issue, we place the data blocks contributed by processes into individual eager buffers. We use multiple `send` and `recv` tasks to emulate sending a longer message. An example of this can be seen in the list of tasks implementing standard exchange that is shown in Table 4.1. In each step, the number of tasks and the expected completion entries doubles.

Once communication completes, the result is copied out from the eager buffers into the destination user buffer. Because the eager buffers are pre-registered with the HCA, this scheme avoids the overhead of registering user buffers. However, the overhead of copying the data to and from these buffers is too high for large messages. In this such cases, we employ the rendezvous protocol that we will discuss next.

4.3.2 Rendezvous Protocol

We employed the rendezvous protocol design discussed in Chapter 3 for large message support in our allgather collective. The only difference is in message identification: because the `MPI_Iallgather` operation does not use tags for its messages, and blocking collective calls cannot match non-blocking ones, only a single set of queue pairs is required in our design. We continue to use a registration cache to reduce the cost of user buffer registrations.

Unlike the immediate message transfer of the eager protocol, the handshaking process of the rendezvous protocol establishes buffer availability prior to the data transfer. Thus, no `recv` tasks need to be pre-posted to avoid the RNR state. This means that when the rendezvous protocol is in use, the `send` and `recv` tasks can use the correct message size, and the use of multiple tasks to emulate longer messages is not required.

Additionally, the overhead of copying messages to and from the eager buffers is avoided. However, the latency of each transfer is increased due to the round trip of the CTS message. Therefore, the rendezvous protocol works best when the collective execution time is dominated by the transfer time of the messages, not communication startup. Theoretical modeling and empirical results previously obtained with other collective designs suggest that this is the case when the messages are large. In Section 4.5 we will examine the performance of the three algorithms described above in combination with eager and rendezvous transfers to verify that this trend holds for offloaded collectives.

4.4 Hierarchical Collective Design

One issue with the flat offloaded collective design is the number of QPs that need to be set up. Because IB HCAs perform best when QP information fits in the adapter's network context cache [25], QP count should be kept low. For each connection, our non-blocking allgather collective design requires separate QPs for eager messages, for handling CTS messages in the rendezvous protocol, and for large message transfers, along with the associated CQs.

These requirements lead to pressure on the context cache and a consequent increase in communication latency. In order to reduce the total number of queues, as well as the traffic across the InfiniBand interconnect, we decrease the number of processes that communicate using CORE-Direct by adopting a hierarchical collective scheme.

In our design, a leader rank is selected on each node. It sets up a shared memory buffer that is registered with the HCA, and also sets up the QPs to handle data transfers. The allgather operation then executes in three phases:

1. **Intranode gather** In the first phase, processes sharing a node copy their data into the shared memory region, creating a single larger message. Per-process arrival flags are used to signal the completion of this stage to the leader.

2. **Internode allgather** In the second phase, each leader engages in collective communication by posting the CORE-Direct tasks to the HCA and periodically checking for completion of the internode phase.

3. **Intranode broadcast** Upon internode phase completion, the leader signals the leaf ranks through their arrival flags. Each rank then copies the result of the collective from the shared memory region to the destination.

Because only the leader processes make use of CORE-Direct, this approach reduces the pressure on the HCA context cache by creating fewer QPs. Additionally, using shared memory for intranode data transfer decreases latency. However, unlike in a flat collective, communication/computation overlap can only occur in phase 2 described above, because the host is busy performing memory copies in phases 1 and 3. Thus we expect to see a reduction in latency of the collective communication due to faster data transfers through shared memory and a reduction in the use of HCA resources. On the other hand, due to the use of the host to copy data to and from the shared buffers, we expect the overlap potential of a hierarchical non-blocking collective to be lower than that of a flat collective design such

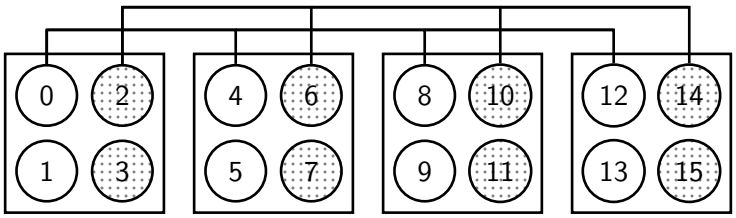


Figure 4.5: Two-group allgather algorithm with 16 ranks (adapted from [76])

as that presented in Section 4.3. In Section 4.5, we present experimental results confirming this assertion.

4.4.1 Multileader and Multigroup Collectives

In Section 4.1 we mentioned multileader [47] and multigroup [76] collective algorithms; we will describe them further here. These approaches have been shown to achieve a reduction in blocking collective latency by spreading the work done by the leader ranks in a hierarchical collective across multiple processor cores.

Multileader algorithms can be readily derived from single-leader hierarchical algorithms. If we were to select n leaders per cluster node, we could partition the remaining ranks and assign each of them to one of the leaders. We could then treat the resulting groupings as residing on separate cluster nodes: communication between all the leaders would be carried out in the internode communicator. As a result, we would trade off a potential increase in latency due to not maximally utilizing shared memory for a quicker response to network events resulting from an increase in the number of processors handling network traffic.

Multigroup algorithms achieve the same goal as multileader algorithms, albeit through a different method. In a multileader algorithm, once the leader ranks are chosen, the algorithm typically places all of them in the internode communicator and treats them identically, without taking into account the fact that some leader ranks share nodes. In contrast, the internode phase of a multigroup algorithm partitions the leader ranks into multiple independent communicators, and collects the results in a final shared memory phase.

Figure 4.5 illustrates the execution of the multigroup algorithm by Qian et al. [76] with 16 ranks and two groups. In this algorithm, ranks are partitioned into groups, with each group having a leader on every node. In Figure 4.5, ranks 0, 1, 4, 5, 8, 9, 12 and 13 belong to group 0, and the rest of the ranks belong to group 1. The algorithm executes in three phases:

1. **Per-group intranode gather** The leaders collect the data from the leaf ranks on their respective nodes.

2. **Per-group internode allgather** An allgather operation is performed among the group's leaders. At the end of this phase the leaders will have all the data to complete the original allgather operation.

3. **Per-group intranode broadcast** The leaders broadcast the results to the leaves.

Although the algorithm bears a similarity to the multileader approaches, the leaders in this algorithm do not communicate across groups. This feature of the design reduces network traffic while keeping the advantages of spreading the handling of network communication across multiple processor cores.

In contrast to blocking collective operations, their non-blocking versions aim to keep the processor cores from being occupied by network tasks, so it would seem that the benefits of multileader and multigroup algorithms would be limited in this application. However, to investigate whether some of the benefits of multileader and multigroup designs are due to the changes to the communication pattern rather than load distribution, and are thus applicable to non-blocking collectives, we designed, implemented, and tested a CORE-Direct-based version of the multigroup algorithm by Qian et al. [76]. We also evaluated the multileader approach using our existing hierarchical collective design by varying the number of node leaders in our testing.

4.5 Experimental Evaluation and Analysis

In this section, we examine the latency of various non-blocking allgather algorithms, as well as their communication/computation overlap potential under single-port and k-port modelling. In Section 4.5.3 we evaluate the impact of our non-blocking algorithms on the radix sort application kernel.

In our evaluation, we compare the latency of our CORE-Direct-based offloaded collective design with the blocking allgather provided by MVAPICH2 1.7 Nemesis [65] and with the non-blocking allgather provided by libNBC 1.0.1 [55]. At the time of writing, libNBC was the only public implementation of the non-blocking allgather collective for InfiniBand clusters. Our implementation is integrated with the MVAPICH2 1.7 Nemesis InfiniBand module [58]. Note that MVAPICH2, with the default settings on our platform, uses the recursive doubling algorithm for all message sizes.

libNBC may either use an MPI library [33] or call IB verbs directly [32] to perform data transfers; we tested both transport modes. libNBC can make progress on non-blocking communication either in `NBC_Test` calls or with the aid of a progress thread. Unfortunately, due to compatibility issues we were unable to get the progress thread option to work reliably. Because the performance of test-based progression is dependent on the frequency of `NBC_Test` calls, we present the best results, which were obtained when calling `NBC_Test` at intervals of 1024 bytes. We used the NBCBench benchmark [37] previously described in Section 3.3 to measure latency and overlap potential.

4.5.1 System Configurations

Cluster A is a cluster of 16 Dell PowerEdge M610 blade servers with 24 GB DDR3 memory, dual 2.93 GHz hexa-core Intel Xeon Westmere X5670 processors, and a Mellanox ConnectX-2 mezzanine HCA. The nodes run 64-bit Red Hat Enterprise Linux 5.5 with kernel version 2.6.18-194.el5 and Mellanox OFED 1.5.2-1. All nodes are connected to a single switch. 128

MPI ranks were used, resulting in 8 ranks per node.

Cluster B is made up of four Dell PowerEdge 2850 servers, each containing two 2.80GHz dual-core Intel Xeon Paxville processors, 4 GB of DDR2 memory, and a Mellanox ConnectX-2 MT25418 HCA. The nodes use Mellanox OFED 1.5.3-1 and 64-bit CentOS 5.5 with kernel 2.6.18-194.26.1.el5. Both ports of all HCAs are connected to a single switch. The tests on this system used 16 MPI ranks with 4 ranks per node.

4.5.2 Microbenchmark Results

Latency of Single-port Flat Allgather

We first evaluated the single-port allgather collective on *Cluster A*. In order to make a comparison with the MVAPICH2 recursive doubling algorithm, the number of MPI ranks was chosen to be a power of 2. We used 128 ranks on our *Cluster A* test system, resulting in 8 ranks per node. Latency was measured by averaging the latency obtained on every process over a large number of runs of the NBCBench benchmark.

The results in Figure 4.6 demonstrate that the CORE-Direct-based algorithms outperformed the libNBC library by a large margin for small and medium messages. Among the offloaded algorithms, standard exchange and Bruck have the best latency for small and medium messages, as expected from the theoretical analysis performed in Section 4.2.4.

However, for messages larger than 16 KB the performance of all offloaded algorithms becomes similar as our implementation switches to the rendezvous protocol, making the handshaking round-trip the deciding factor in latency. Messages larger than 16 KB result in a speedup of up to 20% using standard exchange and Ring algorithms.

For very small messages, offloaded collectives have a latency penalty compared to the blocking MVAPICH2 `MPI_Allgather`. Besides the overhead of setting up a CORE-Direct task list, an additional source of overhead for small messages stems from the inability to use the InfiniBand *inline send* feature which copies the data for a small message to the HCA

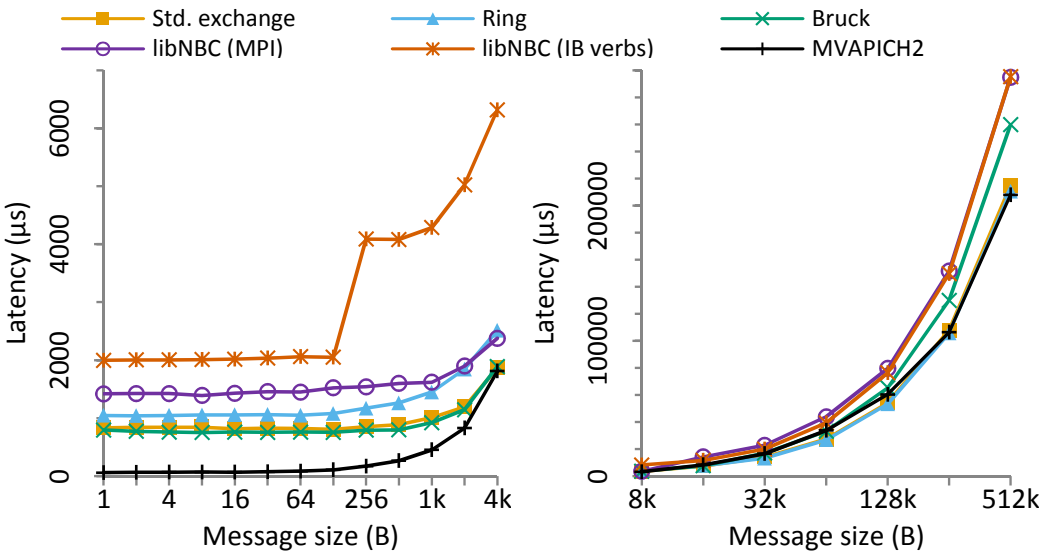


Figure 4.6: Single-port flat allgather latency on Cluster A (128 processes)

together with a work request. Using inline sends is impossible because the data involved have yet to arrive at the time the offloaded `send` WR is posted.

Latency of Single-port Hierarchical Allgather

Comparing the results presented in Figure 4.7 with the ones in Figure 4.6 we see that in the hierarchical collective latency is reduced by the use of shared memory, as expected. Offloaded collectives beat MVAPICH2 by up to 68% for medium, and 46% for large messages. libNBC did not perform well in our hierarchical test using MVAPICH2 as its transport; we excluded its results from Figure 4.7 as they could not be displayed at a reasonable scale. The IB verbs variant of libNBC is also excluded because the library cannot make use of shared memory in this mode.

Interestingly, the ring algorithm has slightly better performance than standard exchange and Bruck algorithms. At the smaller internode communicator size, the QP footprint of all three algorithms is similar and the $O(n)$ steps taken by the ring algorithm do not translate into a large penalty. On 16 nodes, the ring algorithm needs 16 steps, whereas standard

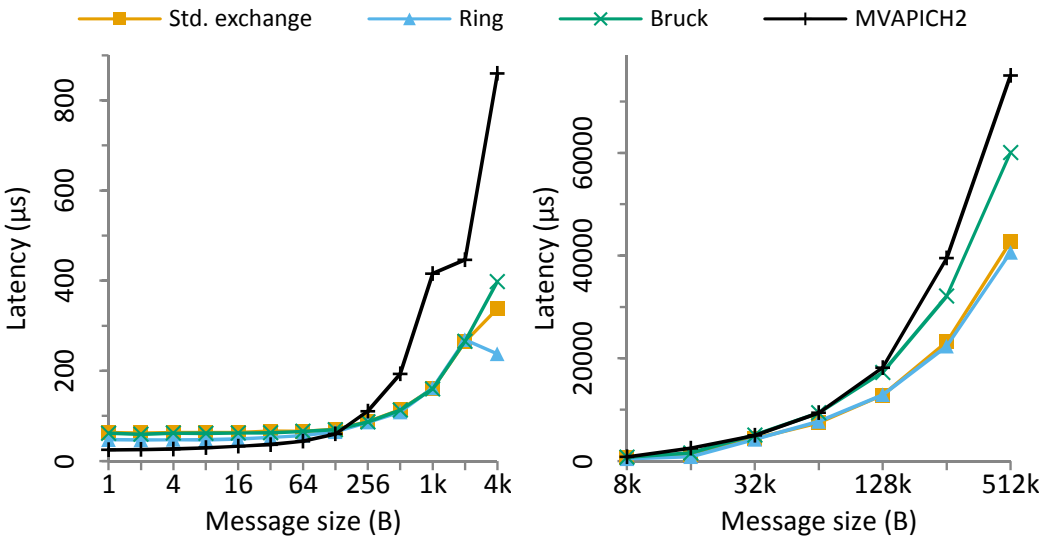


Figure 4.7: Single-port hierarchical allgather latency on Cluster A (128 processes)

exchange completes in 4 steps. In the flat communicator, the ring algorithm was slower for small message sizes, because it required 128 rounds, whereas standard exchange only required 7 rounds.

Overlap Potential of Single-port Flat Allgather

We measured the overlap capabilities of the non-blocking allgather collective by using the NBCBench benchmark [37] to estimate the amount of computation that can be overlapped with communication, as described in Chapter 3. As shown in Figure 4.8, when using single-port algorithms, almost the entire communication time can be overlapped with computation if the message sizes in the offloaded collective range between small and medium. The overlap performance decreased slightly as the message size grew. At 16 KB, our implementation switches to the rendezvous algorithm, which again allows for near 100% overlap for large message sizes. It is likely that better performance could be obtained by tuning the rendezvous threshold for a particular system; however, even with no additional tuning the non-blocking collectives can achieve good overlap performance.

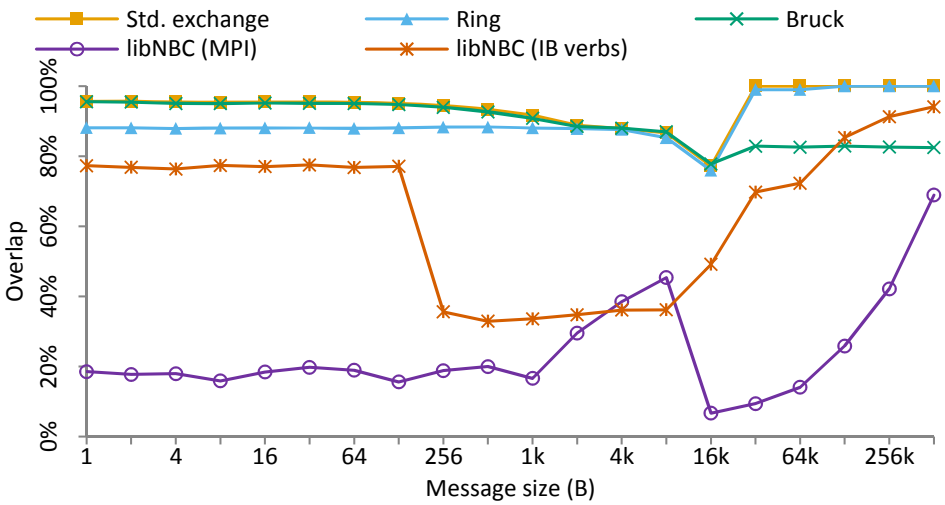


Figure 4.8: Single-port flat allgather overlap on Cluster A (128 processes)

We observe that the overlap for the flat Bruck algorithm was lower for large message sizes because of the memory copy requirement in the local rotation step. This reduced the availability of the processor for computation.

Overlap Potential of Single-port Hierarchical Allgather

The capability of the hierarchical non-blocking collectives to overlap computation with communication was measured following the same methodology as in the evaluation of the flat collectives. Compared to the flat collectives discussed in that section, the hierarchical collectives introduce data copying into the operation, which reduces the availability of the processor for computation. The effect can be seen by comparing Figure 4.8 with the hierarchical collective results in Figure 4.9.

For small messages, we observed a small reduction in overlap (5–10%). However, an increasing amount of time is spent copying data from the shared memory region into destination buffers at larger message sizes, reducing the availability of the processor for computation to approximately 50%.

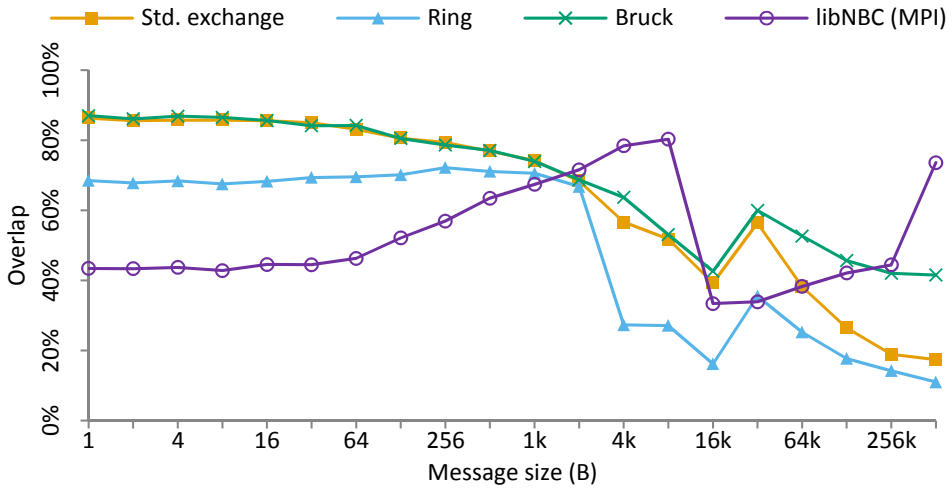


Figure 4.9: Single-port hierarchical allgather overlap on Cluster A (128 processes)

libNBC's overlap performance varied in our testing, ranging between 40 and 80%. Note that we do not present the overlap results for libNBC using its IB verbs transport in a hierarchical communicator because this transport mode is unable to take advantage of shared memory.

Resource Consumption of Flat Single-port Allgather

The Bruck algorithm requires more connections, and thus more QPs per process ($2 \cdot 3 \log_2 128 = 42$) than standard exchange ($3 \log_2 128 = 21$), while ring has the lowest QP footprint ($2 \cdot 3 = 6$ QPs). Because each connection is associated with a set of preposted buffers, there is a corresponding effect on the memory footprint of the MPI library. As seen in Table 4.2, we measured the resident set size of NBCBench to be 105 MB for standard exchange, 60 MB for ring, and 160 MB for Bruck. Shared memory usage was insignificant.

Table 4.2: Flat communicator per-process memory usage by allgather algorithm (MB)

Algorithm	Resident	Shared
Standard exchange	105	1
Ring	60	1
Bruck	160	1
libNBC MPI	42	1.5
libNBC ibverbs	591	1.5

Table 4.3: Hierarchical communicator per-process memory usage by allgather algorithm (MB)

Algorithm	Resident	Shared
Standard exchange	201	161
Ring	201	161
Bruck	201	161
libNBC MPI	42	2

Resource Consumption of Single-port Hierarchical Allgather

Compared to the resource consumption of the flat offloaded non-blocking collectives, in a hierarchical communicator all algorithms have reduced QP usage. Standard exchange required $3 \log_2 16 = 12$ QPs per node in our environment, the Bruck algorithm needed 24, and the ring algorithm used 6. More importantly, however, the memory footprint in the hierarchical case was dominated by the shared region. Thus, the differences between the algorithms are insignificant, as can be seen in Table 4.3.

Multiport Modelling

As described in Section 4.2, the Bruck and standard exchange algorithms support a communication model in which messages can be sent simultaneously on multiple ports. We have investigated k-port allgather performance on *Cluster B*, because this platform has dual physical interface ports. To enable comparison with standard exchange, we used a 3-port communication model to cover communicators of size 16 and 4. The 3 virtual ports were

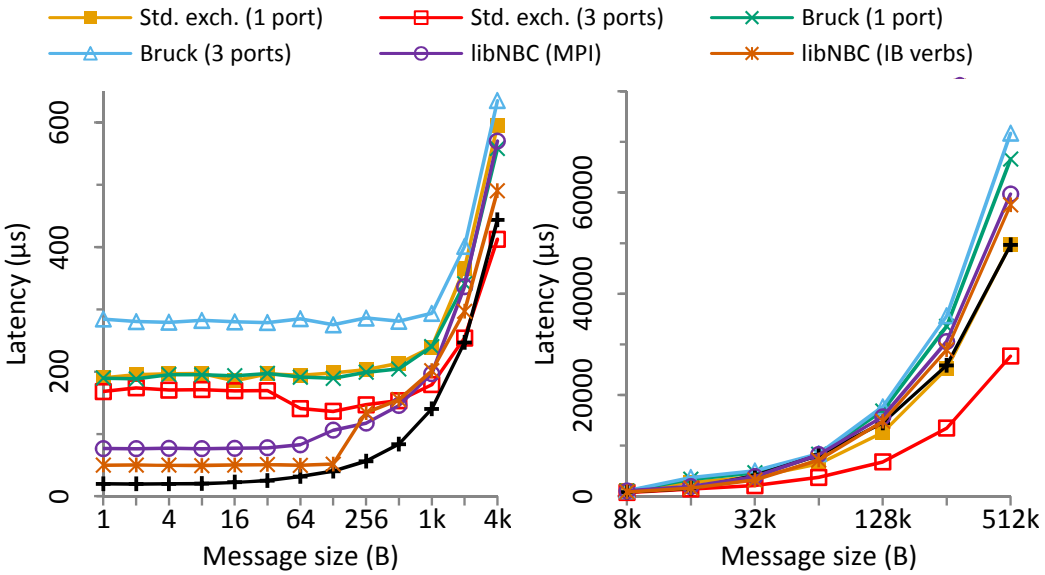


Figure 4.10: k-port flat allgather latency for Bruck and standard exchange algorithms on Cluster B (16 processes)

allocated over 2 physical HCA ports using a round-robin pattern.

In a flat communicator (Figure 4.10), 3-port standard exchange has a performance advantage over the single-port variant. For messages larger than 16 KB, the 3-port allgather achieves a speedup of up to 49% compared to single-port, and up to 54% compared to MVAPICH2. The Bruck 3-port algorithm has a slight penalty compared to the single-port latency. The overlap performance of the single-port and 3-port algorithms was similar, as illustrated by Figure 4.11. As with the tests on *Cluster A*, the Bruck algorithm has reduced overlap performance with large messages due to the memory copy in the rotation step.

Unlike in our tests on *Cluster A*, libNBC's allgather algorithm provided competitive latency in the smaller *Cluster B*. However, the overlap performance of libNBC was significantly lower than that of the offloaded collectives.

We also tested multiport versions of the hierarchical collectives. As shown in Figure 4.12, the 3-port standard exchange and Bruck algorithms have reduced latency in a hierarchical communicator compared to the single-port versions, because the internode exchange can

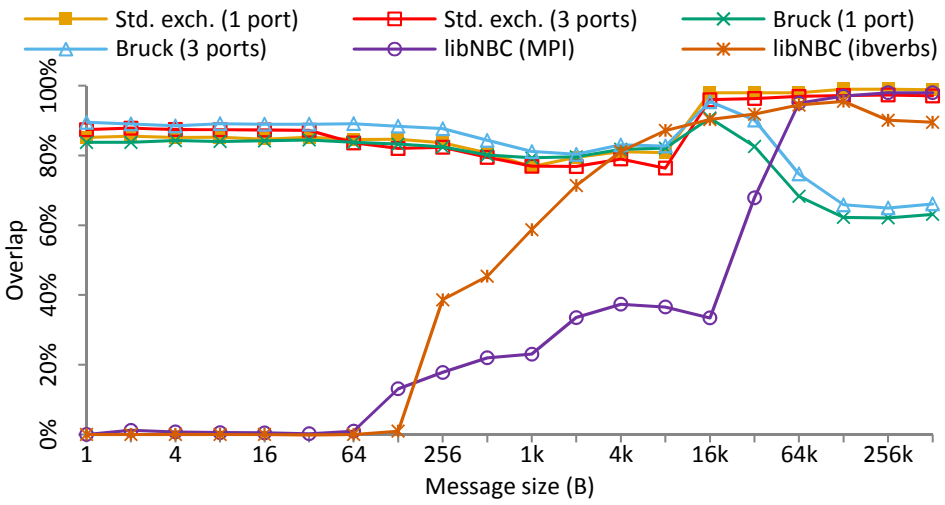


Figure 4.11: k-port flat allgather overlap for Bruck and standard exchange algorithms on Cluster B (16 processes)

be completed in a single step. The largest improvement of 16% was observed for small messages. As the message size increases, the latency becomes dominated by the memory copy, eliminating the speedup. The overlap performance remains largely similar in the single-port and 3-port collectives, even though there is a slight reduction, as illustrated by Figure 4.13.

Referring back to the results for the flat collective, we recall that in a flat communicator multiport standard exchange also had a performance advantage, whereas the multiport Bruck algorithm had a slight penalty compared to the single-port latency. Overall, the effects of switching to a multiport collective are similar in the flat and hierarchical environments: in both cases latency is reduced, while overlap capability remains similar. However, as the message size increases, the latency of the hierarchical collectives becomes dominated by the shared memory operations, negating the performance gains due to the use of multiple ports.

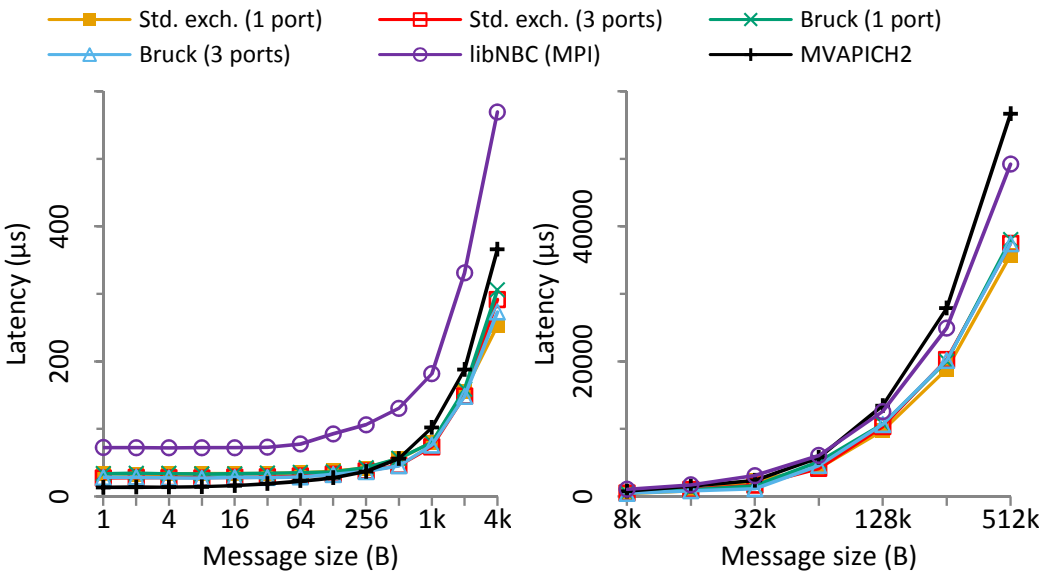


Figure 4.12: k-port hierarchical allgather latency for Bruck and Standard Exchange algorithms on Cluster B (16 processes)

Multileader and Multigroup Allgather Algorithms

In addition to the multiport algorithm variants, we looked at the effectiveness of multileader and multigroup modifications to the hierarchical collectives. These algorithms have been previously shown to reduce latency of blocking allgather collectives by spreading the work of collective progression to multiple processor cores [47, 76]. We investigated whether similar benefits can be obtained for non-blocking offloaded collectives. We described the multileader and multigroup collective algorithms in Section 4.4.1.

In our testing, a two-group `MPI_Iallgather` took more time to complete than the single-leader hierarchical version, with the four-group version being slower still. The overlap potential was largely unaffected. The same outcome was seen with multiple leaders. The change in communication pattern does not appear to help. Because all communication is being handled by the HCA instead of being distributed among multiple cores, the multigroup and multileader approaches are not beneficial when using CORE-Direct. In order to obtain speedup comparable to that seen in host-progressed multileader and multigroup collectives,

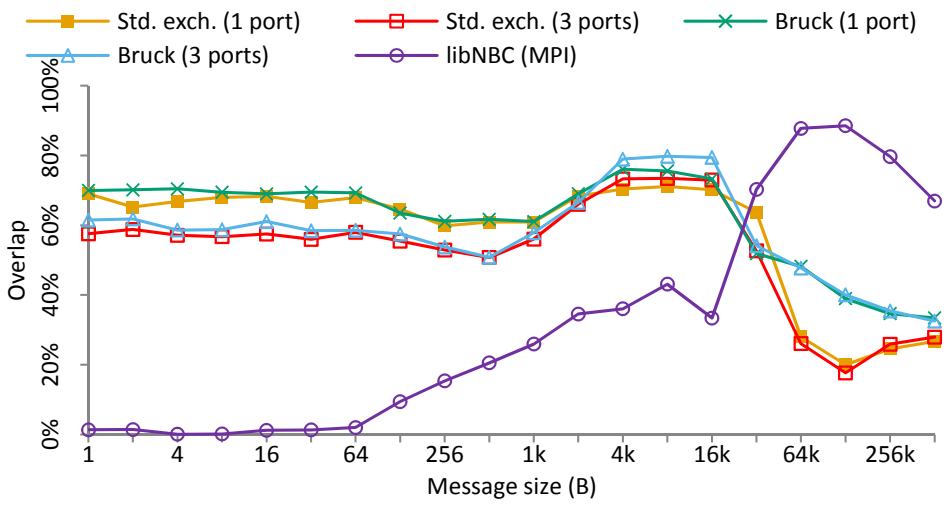


Figure 4.13: k-port hierarchical allgather overlap for Bruck and standard exchange algorithms on Cluster B (16 processes)

multiple HCAs would be required.

In summary, our tests on *Cluster A* confirmed that keeping the number of QPs low is essential to achieving low latency, and that the improvements due to multileader and multigroup algorithms that were seen in previous works have been due to spreading the workload over multiple processor cores rather than a change in communication pattern.

4.5.3 Application Results: Radix Sort Kernel

Overview of Radix Sort

To evaluate the performance of our non-blocking allgather collective on code that is more realistic than a microbenchmark, we created a redesigned variant of the *radix sort* routine. Our radix sort kernel is based on the commonly used algorithm by Zagha and Blelloch [105] that has been modified to take advantage of computation/communication overlap.

Radix sort is a non-comparative sorting algorithm that is amenable to parallelization. It is an efficient and practical method for sorting numeric keys, and finds applications in

computer graphics, database systems, and sparse matrix multiplication, among other areas.

Parallel radix sort using radix r begins by splitting keys into digits with a value in the range $[0..(r-1)]$. The keys are then sorted on each of these digits in individual rounds, one round per digit. We assume that at the beginning of the algorithm's execution keys are distributed among multiple processes. Sorting proceeds from least to most significant digit. Each round consists of six steps:

1. **Bucket count** Each process computes a count of keys that would go in each bucket based on the current digit being sorted.

2. **Local prefix sum** The prefix sum of the bucket histogram obtained in step 1 is computed. The result is a set of offsets at which keys belonging to each bucket must be placed locally.

3. **Local redistribution** Keys at each process are placed at offsets computed in step 2.

4. **Histogram allgather** The bucket counts from step 1 are communicated among all processes to create a global histogram.

5. **Global prefix sum** The global offsets of the buckets are calculated using the histogram obtained in step 4.

6. **Global redistribution** Keys are moved between processes, being placed at the correct global offset.

Adding Overlap Capability to Radix Sort

Our version of parallel radix sort works similarly to the algorithm described at the beginning of this section, except that it takes advantage of the non-blocking allgather operation. In our radix sort, histogram communication in step 4 is overlapped with local prefix sum computation and key movement in steps 2 and 3.

Table 4.4: Radix kernel run time percentage improvement over MVAPICH2 using flat single-port standard exchange on Cluster A (128 processes)

Radix	2^{12} keys	2^{16} keys
2	-49.76	-42.25
4	-41.16	-33.42
8	0.92	0.053
10	-2.37	-0.07
12	5.33	-2.64
14	6.43	4.06
16	-0.95	7.02

In the first step of the algorithm, every process calculates the number of keys that need to be placed into each of the r buckets. The bucket counts need to be exchanged between the processes in order to rearrange the keys globally. However, while the allgather communication is in progress, it is possible to prepare for the key data transfer step by rearranging keys locally. To obtain the offset for the keys in each bucket, we calculate the prefix sum of the bucket counts. The keys are then moved to the correct offset at each rank. Once the network communication completes, we compute the global offsets using a prefix sum of the global histogram, and move the keys using one-sided MPI operations. We present the observed improvement due to overlapping local prefix sum computation and key movement with allgather collective execution in flat and hierarchical communicators.

Flat Collective Results

We first evaluate the improvement in radix sort run time due to communication/computation overlap using single-port modelling. The results were obtained on *Cluster A* and are presented in Table 4.4. The baseline for the comparison is the radix sort kernel without overlap, using the blocking recursive doubling `MPI_Allgather` provided by MVAPICH2 1.7 Nemesis.

Because the size of the messages involved was small to medium, we used the standard exchange algorithm for offloaded allgather. The performance of radix sort is dependent on its

Table 4.5: Radix kernel run time percentage improvement of 1-port and 3-port flat standard exchange over MVAPICH2 on Cluster B (16 processes)

(a) 2^{12} keys

Radix	1-port	3-port
2	-9.50	0.30
4	-26.04	0.03
6	-14.96	2.93
8	-3.83	8.18
10	-5.44	4.29

(b) 2^{16} keys

Radix	1-port	3-port
2	6.93	-7.03
4	-13.23	-0.74
6	-5.35	5.18
8	-0.95	4.18
10	1.95	3.97

chosen parameters. In our testing the performance of the non-blocking allgather was poor for small radices (and thus more rounds of communication). The communication/computation overlap could not make up for the increase in latency over the blocking collective that we saw in Section 4.5.2.

We also measured the improvement due to multiport modelling, again using the performance with blocking MVAPICH2 allgather as a baseline. The results presented in Table 4.5 were obtained on *Cluster B*. The blocking allgather performed similarly to the non-blocking 3-port collective, whereas the single-port non-blocking collective could not match the performance of the blocking allgather due to the reduced amount of communication in the smaller *Cluster B*. 3-port allgather reduced run time by up to 20% compared to the single-port collective.

As we can see, a non-blocking allgather design that does not take advantage of the memory hierarchy in a cluster exhibits poor latency characteristics with small messages. Therefore, we see no improvement in the radix sort kernel, because the amount of overlapped computation is not enough to make up for the increase in latency.

Table 4.6: Radix kernel run time percentage improvement over MVAPICH2 using single-port hierarchical standard exchange algorithm on Cluster A (128 processes)

Radix	2^{12} keys	2^{16} keys
2	40.19	11.09
4	19.83	13.47
8	20.09	9.51
10	-1.05	-0.12
12	5.07	1.74
14	3.18	4.72
16	6.42	7.07

Table 4.7: Radix kernel run time percentage improvement of 1-port and 3-port hierarchical standard exchange over MVAPICH2 on Cluster B (16 processes)

(a) 2^{12} keys

Radix	1-port	3-port
2	13.20	12.50
4	7.15	8.33
6	4.39	4.03
8	3.32	3.42
10	3.10	3.99

(b) 2^{16} keys

Radix	1-port	3-port
2	11.90	12.02
4	7.26	6.48
6	4.50	3.70
8	3.66	4.07
10	4.49	3.76

Hierarchical Collective Results

We repeated the radix sort kernel benchmark with a hierarchical collective. On Cluster A (Table 4.6) good speedup (up to 40%) was obtained when the radix was small, and thus many rounds of communication were needed. Increasing the number of keys makes the global redistribution phase dominate the run time, diminishing the speedup.

To test the effects of multiport modelling, the radix sort benchmark was also performed on Cluster B (Table 4.7). Because in a hierarchical communicator the latency of non-blocking collectives for small and medium size messages is closer to that of MVAPICH2 (see Figure 4.12), we observed a speedup of up to 13% in the hierarchical communicator compared to the blocking collective, with the 1-port and 3-port versions having similar

performance. We expect the 3-port allgather to have a performance advantage in a larger cluster, as the difference in the number of steps required to complete the communication will increase.

Overall, the performance of radix sort is sensitive to the latency of the collective using small messages. Thus, we see a speedup due to computation/communication overlap in the hierarchical collective case, which does not have the latency penalty that the flat communicator does.

4.6 Summary

Building upon the foundation of offloaded point-to-point transfers introduced in Chapter 3, in this chapter we developed a family of offloaded non-blocking algorithms for the allgather collective. We talked about the design issues involved, and evaluated the performance of these collectives with latency and overlap microbenchmarks, as well as in the radix sort application kernel. We saw that although the performance of the flat non-blocking allgather collectives was good, there was room for improvement. We therefore also developed and tested a design that takes advantage of the shared on-node memory in a typical modern HPC cluster. We found that while the use of shared memory significantly reduces latency, the communication/computation overlap potential of large messages suffers, because the processor time has to be shared between performing computation and memory copies.

With the radix sort benchmark in a hierarchical collective, we showed that non-blocking offloaded collectives can achieve good performance. Additionally, as we have demonstrated in the previous chapter, radix sort is very sensitive to small message latency; we were unable to achieve a speedup in a flat communicator. Therefore, the benefit from non-blocking collectives will depend on the computation and communication characteristics of a particular application.

In the next chapter, we will continue optimizing offloaded non-blocking collectives. We

will specifically focus on minimizing the effects of the issues of OS noise and process skew that were first presented in Chapter 2.

Chapter 5

Process Arrival Pattern Tolerant Pipelined Hierarchical Offloaded Collectives

In Chapter 3, we introduced offloaded operations to transfer a single message across an InfiniBand network. In Chapter 4, we built on this work by developing offloaded collective operations. In our investigations in these chapters, we saw that non-blocking operations typically have latency competitive with their blocking counterparts and allow the programmer to overlap communication with computation. We also saw that offloaded operations provide better overlap capabilities and better latency than host-based non-blocking operations.

However, these offloaded operations still have a weakness. When the latency of a collective operation is measured alone, the experimenter usually takes care to synchronize the invocations of the collective calls across the participating processes. In a real MPI application, however, the collective starting points may be significantly out of sync due to imbalances in the computational load. As we will see in Section 5.2, previous work has explored ways to make blocking collectives better handle variance in process arrival patterns.

Because they allow communication to be overlapped with computation, non-blocking collectives are likely to be a better fit than blocking collectives for situations where the computational load is difficult to carefully balance across the processes [44]. In such a situation, there is likely to be increased variability in the process arrival pattern. Additionally, the variability would be amplified when multiple non-blocking collective operations are used in parallel. Therefore, tolerance to process arrival pattern variations is arguably even more important in the context of non-blocking collectives.

In this chapter, we propose an approach to designing offloaded non-blocking collectives that uses pipelining to make the collective tolerant of arrival pattern variation. Section 5.1 presents an example that motivates the need for such a design. Section 5.2 presents related work and explains the novelty of this chapter. Section 5.3 explains the pipelined approach in detail, and discusses the Cayley allgather algorithm, as well as the Receive-Store-Replicate primitive designed to aid in implementing this and other offloaded collectives. Section 5.4 describes our implementation of the approach. Section 5.5 describes our evaluation of the implementation using microbenchmarks, and an evaluation of concurrent collective calls. We found that our design is better at handling variations in the process arrival pattern and has improved resource consumption scalability for parallel operations. Section 5.6 summarizes the chapter.

5.1 Motivating Example

We will begin the discussion of process arrival pattern tolerant offloaded collective design with a motivating example. In Chapter 4, we saw that offloading communication to dedicated hardware allows the host processor to perform computation without affecting the communication or having to manage its progress. However, this holds true only after the offloaded communication operation has started. During the time window when the ranks of the communicator are commencing collective communication, there still exists a source of

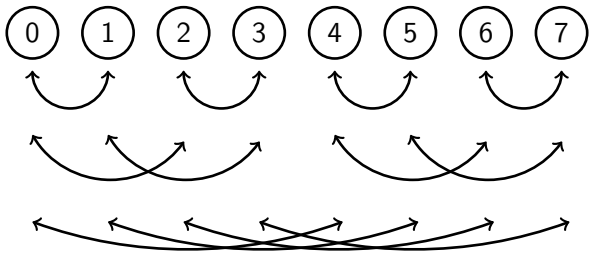


Figure 5.1: Recursive doubling allgather communication pattern

scheduling noise, and this noise can propagate through the cluster.

To illustrate the issue with an example, let us consider an allgather collective with eight participating processes that employs a standard exchange communication pattern with single-port modelling, also known as recursive doubling. Although we use a single-port communication pattern for clarity, every feature of the design described in this chapter applies to the general multi-port algorithm as well. As we saw in Section 4.2.1, the communication proceeds with the processes first communicating with their immediate neighbour and then doubling the distance to their peer in each subsequent round of the exchange. This pattern is shown in Figure 5.1.

Suppose rank 1 arrives at the start of the allgather collective much later than the other ranks. In this situation, even if rank 0 sends its data on time, its message cannot make further progress through the collective because it must be forwarded by rank 1 in rounds 2 and 3, and rank 1 has not yet arrived. In addition, this effect propagates through the later rounds of communication, because, for instance, rank 3 will not be able to communicate with rank 7 until it has received rank 1's message, and rank 0 will not begin rounds 2 and 3 until it receives rank 1's data.

One could argue that, by definition, a collective operation cannot complete without transferring all of the data, and thus handling delays due to process arrival is not necessary. However, if the other ranks could have carried out communication that would have otherwise depended on rank 1's arrival, the total time taken by the collective operation would be reduced.

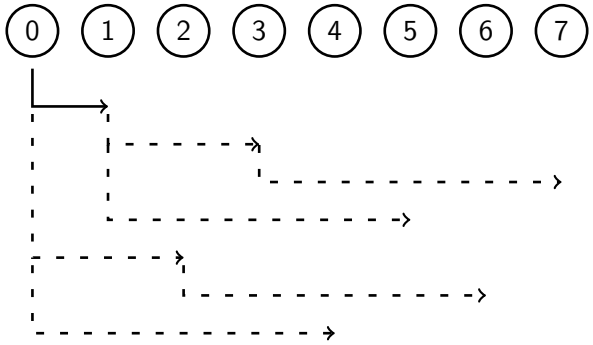


Figure 5.2: Propagation of the contribution of rank 0 in recursive doubling allgather, considered in isolation. Dashed lines represent the data transfers blocked by the late arrival of rank 1

When the dependencies are present, any delays propagate through the communicator and increase the execution time of the collective.

To understand this problem in more detail, consider Figure 5.2, which shows the path that the data block contributed by rank 0 takes through the communicator. Note that the figure considers *only* rank 0's data block and ignores all other data blocks. We can see that rank 0's data are propagated in a tree pattern, and that most of the work in the propagation tree is delayed by the late arrival of rank 1. Specifically, every communication indicated by a dashed line will be performed late as a result of rank 1's late arrival. Even a small initial delay in the arrival of rank 1 thus becomes amplified while propagating through the collective.

In the next section, we will see how previous work has addressed this problem by making blocking collectives aware of variation in the process arrival pattern. Following that, in Section 5.3, we will describe our method of solving this issue for a non-blocking collective.

5.2 Related Work

The problem described in our motivating example stems from variability in the arrival of processes at the site of the collective call. The importance of this problem has been noted by

many researchers. Faraj et al. performed a study of process arrival patterns and found that they can have a significant effect on the performance of collective operations [22]. Patarasuk and Yuan studied the issue of process arrival variability and the performance of various algorithms for the `MPI_Bcast` operation in imbalanced conditions, finding that adverse effects were significant. They also presented two host-progressed algorithms that took the process arrival pattern into account [71].

Qian and Afsahi presented a host-progressed arrival pattern aware allgather algorithm [74]. They found that variations in process arrival are common in MPI applications, as exemplified by the NAS Parallel Benchmark suite [5]. Qian and Afsahi's investigation dealt with the process arrival pattern in both the intranode and internode phases of a hierarchical blocking collective. Although we use similar techniques in the initial intranode gather phase of the collective, our focus is primarily on offloading of internode communication.

There are several differences between our design and the ones described above. First, we consider non-blocking rather than blocking collectives. Blocking collectives can have exclusive use of the host processor during their runtime; thus, the designs above focus on making the collective aware of the arrival pattern so that the processor can be used to actively manage the communication. In contrast, an offloaded collective should keep the host processor available for computation. We therefore take an alternate approach and try to make the communication independent of the arrival pattern variations. We call this an *arrival pattern tolerant* collective design to distinguish it from existing *arrival pattern aware* approaches.

Message fragmentation and pipelining of the transfers of the resulting fragments has been previously investigated by Bao et al. [7]. Their work focused on point-to-point communication over Ethernet interconnect, whereas we apply pipelining techniques to offloaded non-blocking collective communication over InfiniBand.

Träff et al. presented a pipelined algorithm for the blocking allgather collective that is based on the ring algorithm [97]. The algorithm's use of pipelining improved the performance

of the irregular allgather collective in situations where the contributions of the participating processes were highly imbalanced; that is, some ranks contributed significantly more data than others. We investigate the regular allgather collective in this chapter and aim to use pipelining to work around irregular arrival patterns.

Generalized primitives for building collective operations have been proposed by Hemmert et al. for the Portals framework [29]. Subramoni et al. also proposed a set of generalized primitives for CORE-Direct [89]. We will design a different primitive in Section 5.4.1 that will allow us to construct the collectives we would like to investigate.

The primitive that we develop in this chapter will allow us to simplify the allgather collective by effectively decomposing it into a set of broadcast operations. Optimizations of the broadcast collective have been extensively studied, and a number of algorithmic and implementation improvements have been proposed. Designs for the broadcast operation over InfiniBand have been developed by Sur et al. using RDMA operations [90] and Liu et al. using native InfiniBand multicast [56]. We do not make use of these hardware features in our design, but would like to investigate them in future work. Kandalla et al. designed and implemented a CORE-Direct offloaded MPI broadcast [45]. Their design incorporated an option to use the HCA for all steps in the collective, including the intranode broadcast. Although this had a detrimental effect on latency, the overlap potential of the collective was improved. We use the processor for the shared memory phase of our collective instead to realize the latency advantage. Venkata et al. have also investigated the broadcast operation in the context of CORE-Direct offloading [100]. Their design employed a rendezvous protocol similar to the one we used in Chapters 3 and 4. Our allgather collective in this chapter uses a protocol that is more similar to an eager protocol instead.

The work by Hemmert et al. [29] was recently expanded by Schneider et al. [82]. This work contains an in-depth discussion of the issues of offloaded collective design, such as tag matching and adapting eager and rendezvous protocols for offloading. The most recent work by Schneider et al. is close in spirit to the design proposed in this chapter. However, our

RSR communication primitive described in Section 5.4.1 allows fully offloaded asynchronous progression without explicit message pre-matching. Instead, matching is done in a post-processing step as part of the shared memory phase of the collective operation.

5.3 Design of the Pipelined Collective

A collective design must meet two criteria to be tolerant to process arrival patterns. First, there should be few or no dependencies between messages originating at different ranks. Second, the collective's communicator must retain connectivity among the arrived ranks even in the case of a late arrival of one or more other ranks. In other words, communication between ranks that have arrived must not be prevented by the late arrival of another rank. We will address each of these criteria in turn.

In our motivating example, the delay in the arrival of rank 1 delayed a large fraction of the collective communication. As can be seen in Figure 5.1, rank 0 is blocked in the first round of communication. Ranks 2 and 3 can complete a single round, and ranks 4-7 can complete two rounds of the recursive doubling algorithm before they encounter a data dependency that will cause a stall. Data dependencies cannot be completely avoided: by definition, the allgather operation cannot complete without rank 1's contribution. However, we can prevent stalls by modifying the collective algorithm such that ranks do not block when another rank is late.

In the standard exchange and Bruck algorithms, data dependencies arise because messages are concatenated before they are sent. Recall from Section 4.2.4 that concatenation is done to reduce message startup overhead, and that this optimization is effective for small and medium message sizes. Consequently, although removing concatenation prevents data dependencies, the overhead will increase. In Section 5.3.2, we investigate this modified version of the algorithm and evaluate the tradeoff.

For our second requirement, we point out that even if the messages in our running

example had been completely independent, our collective would still be affected by rank 1's delay. Until rank 1 is able to participate in the collective operation, parts of the collective are not able to communicate. For example, there is no way for rank 0 to send its data to rank 3 without rank 1's participation. Investigating solutions to this problem is a major theme of this chapter.

As we can see, a successful collective design must meet both requirements outlined above in order to deal with arrival pattern variations. Next, we will discuss how existing work and our set of solutions address these problems.

5.3.1 Direct Algorithm

Though their paper does not specify the same requirements that we proposed above, the allgather design by Qian and Afsahi [74] does meet them. The authors use the direct algorithm for the allgather collective where every rank is connected to every other rank. The first requirement is met because every rank is responsible for delivering its own data to the rest of the collective; no data dependencies arise. The second requirement is met because the communicator is fully connected; therefore, an edge is always present between any two arrived ranks in the communicator graph.

Though the direct algorithm is a complete solution to the problem of arrival pattern variability, and has been shown to be effective in small clusters, we would like to avoid the communication pattern where every rank must communicate with every other rank, and the $O(n^2)$ connection requirement associated with this communication pattern. We would prefer instead that ranks communicate with a limited set of neighbours in the communicator. In the sections that follow we will develop a design that requires only $O(1)$ connections and has a corresponding nearest-neighbour communication pattern.

5.3.2 Pipelined Standard Exchange and Bruck Algorithms

If we can avoid concatenating messages and thus creating data dependencies, it would be possible to avoid the magnification of the latency in the arrival of a given process. We begin our exploration of the design space for arrival pattern tolerant collectives with a simple modification to the standard exchange and Bruck algorithms: we remove message concatenation and instead send each individual data block as it becomes available.

Communication in such a collective executes in a *pipelined* fashion, with some portions of messages being propagated further in the collective while waiting for other portions to arrive. Thus, although the same amount of data is sent in total, the overall latency can likely be reduced if we could eliminate the time spent waiting for process arrival. The amount of work potentially amenable to this optimization is significant. In Figure 5.2, all the work represented by dashed lines could theoretically have been performed if the collective operation could proceed without waiting for the arrival of rank 1. In fact, Figure 5.2 underestimates the amount of blocked work, because it treats the propagation of the contribution of rank 0 in isolation. Messages from other ranks that are concatenated with rank 1's data are also affected by the late arrival of rank 1.

Additionally, if an application executes multiple concurrent non-blocking collective operations, a rank that is late to one of them will likely be late to the following ones as well. Thus, this optimization is even more beneficial in the presence of concurrent collectives. We can further increase the amount of work amenable to pipelining by splitting messages into smaller fragments. With this change, we can pipeline the processing of messages from processes whose arrival is not delayed. These optimizations are discussed further in Section 5.4.1.

However, avoiding data dependencies meets only one of the prerequisites for reducing the influence of the process arrival pattern. The process arrival pattern aware designs in both [74] and [71] assume that a rank may only participate in the collective once it has arrived.

The converse seems counterintuitive: a rank would be able to participate in the collective operation before starting it. This approach, however, is made possible by a specialized eager message protocol. The data transfer protocol we use for the collective is similar to the one described in Chapter 3, but it includes enough information for the message to both be matched to a collective operation and to be forwarded downstream prior to the rank's arrival. We will describe the protocol in detail in Section 5.4.1. The implementation of this forwarding capability can be provided by a host-based progress engine or by the HCA, as we will discuss in Section 5.4.

With the introduction of augmented eager messages, ranks that have yet to arrive at the collective call are able to cache unexpected messages for use once the collective is started. Additionally, they act as relays that ensure that the communicator connectivity is unbroken regardless of the arrival pattern.

Both the Bruck and standard exchange algorithms may be implemented using this approach. The resulting design maintains the $O(\log_k p)$ scalability in the number of InfiniBand connections of the standard exchange and Bruck algorithms, where p is the number of participating processes and k is the number of ports. However, due to the removal of message concatenation, the number of messages increases to $O(p)$. The total amount of user data transferred is unchanged, because in the allgather collective each rank needs to receive every data block only once.

With the two requirements met, our design can reduce the adverse effects of variations in the process arrival pattern. If we remove message concatenation from the standard exchange algorithm in our motivating example, and rank 1 is again delayed in starting this new collective operation, the rest of the ranks can proceed without it, while rank 1 acts as a relay for any unexpected messages it receives. For example, rank 0 is free to start the second round of communication earlier by sending out its own data block without waiting to receive rank 1's message in the first round. As a result, the late arrival of rank 1 does not affect the propagation of the remaining data through the communicator, and the progression of the

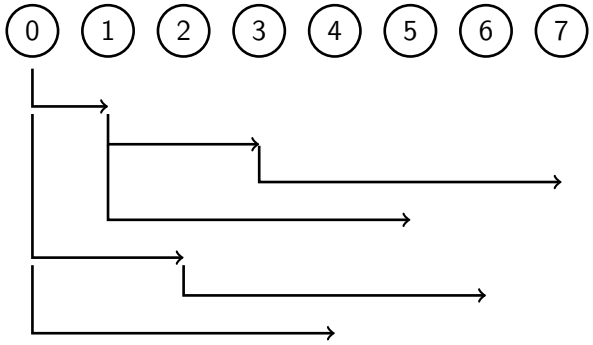


Figure 5.3: Propagation of the contribution of rank 0 considered in isolation. The algorithm used is pipelined single-port standard exchange allgather. Late arrival of rank 1 has no effect.

collective operation is improved. The dissemination tree shown in Figure 5.3 for rank 0's data block (considered in isolation) remains unchanged from that in Figure 5.2 illustrating recursive doubling with message concatenation, with the exception of the removal of data dependencies. This holds true for all blocks of data in the collective.

We will describe the implementation of the design in Section 5.4 after we present another arrival pattern tolerant algorithm with $O(1)$ resource consumption in the next section.

5.3.3 Resource-conserving Cayley Algorithm

Our removal of message concatenation from the Bruck and standard exchange algorithms in Section 5.3.2 increased the number of messages that need to be exchanged to $O(n)$. Although the change reduces the effect of the arrival pattern, it may adversely affect the latency of the algorithm for small messages due to the increased overhead. We made this tradeoff consciously, and will evaluate its effect in Section 5.5.

First, however, we will present one other solution for the arrival pattern aware non-blocking allgather problem. Recall that in Chapter 4 we discussed the ring algorithm for allgather. It meets our first requirement without modification because it does not concatenate messages. The ring algorithm instead immediately forwards incoming messages downstream.

The use of the ring algorithm alone does not meet our second requirement, and thus

does not fully address the arrival pattern problem. A rank arriving late introduces a break in the connectivity of the ring. Clearly, simply eliminating data dependencies is insufficient for an arrival pattern tolerant collective. However, we can apply the same solution of having ranks relay unexpected messages.

The ring algorithm is $O(1)$ in the number of required connections. However, there remains one disadvantage compared to standard exchange and Bruck: the ring algorithm cannot use multi-port communication. We solve this issue by introducing an algorithm for allgather based on the structure of the Cayley tree.

A Cayley tree is an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, and all vertices $v \in V$ have degree 1 or c, where c is called the coordination number of the Cayley tree [102]. In cases where this constraint cannot be satisfied due to the number of ranks participating in the collective operation, we use an incomplete Cayley tree with one of the vertices having degree between 1 and c. To construct a Cayley tree for an allgather collective, we start with a vertex representing rank 0. We then build the tree incrementally by selecting a leaf vertex w with the lowest rank and adding neighbours to this vertex in rank order until the degree of w reaches c or no unconnected ranks remain in the communicator. Once degree c is reached, a new leaf node is selected. Figure 5.4 illustrates the structure of a Cayley communicator with 9 ranks and a coordination number $c = 4$.

Because the resulting graph is a tree, it is by definition connected and cycle free. As we will see in Section 5.4.3, the lack of cycles will allow us to efficiently implement offloaded broadcast operations in the Cayley tree. By extension, this will lead to a design supporting the allgather operation.

We use the Cayley interconnection graph to construct broadcast trees for each rank's data item. From the point of view of a given rank p, the broadcast tree is a directed graph G_p rooted at p that has the same general structure as the originally constructed Cayley tree. The broadcast trees can be used to express the allgather collective among n processes as the joint execution of the n broadcasts. Each node in a broadcast tree is responsible for

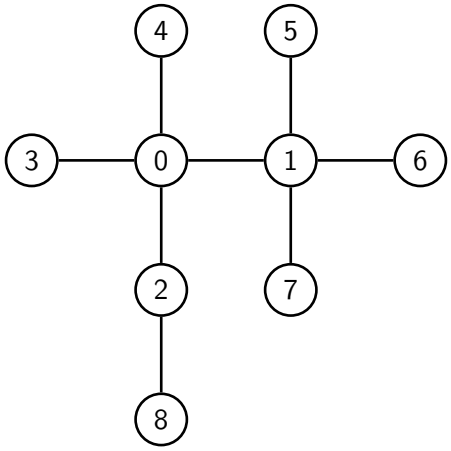


Figure 5.4: Incomplete 4-Cayley tree for a communicator of size 9

receiving a message from its parent and propagating the message to its k children. Each process belongs to all the broadcast trees representing the allgather operation.

Even though the depths of the vertices representing the process in the trees differ, the set of its neighbouring vertices remains the same, because we do not change the interconnection structure of the communicator. From the point of view of an individual rank, the neighbour acting as the parent will be different in every tree, with one tree having the rank itself at the root. The constant set of neighbours is the reason for choosing the Cayley tree, because it leads to constant resource usage in the resulting algorithm. Unlike the algorithms presented in Section 5.3.2, where the number of connections required for each rank grew with the number of processes participating in the collective, the number of connections needed by a rank in the Cayley algorithm is always at most the coordination number c of the Cayley tree. The Cayley allgather algorithm can be viewed as a generalization of the ring algorithm for multiport modeling where the coordination number $c = k + 1$, and k is the number of outbound ports on non-leaf ranks.

To see that the time complexity of this algorithm is $O(\log_c n)$, consider that the height – the length of the path from the root to the furthest leaf – of a Cayley tree with coordination number c is $O(\log_c n)$. The furthest that a data item has to travel is from one leaf node

Table 5.1: Properties of allgather algorithms

	Std. Exch. / Bruck	Ring	Pipelined Std. Exch. / Bruck	Cayley
Rounds	$O(\log_{k+1} n)$	$O(n)$	$O(\log_{k+1} n)$	$O(\log_{k+1} n)$
Messages	$O(\log_{k+1} n)$	$O(n)$	$O(n)$	$O(n)$
Connections	$O(\log_{k+1} n)$	$O(1)$	$O(\log_{k+1} n)$	$O(1)$

in the interconnection graph to another. Therefore, the data contributed to the allgather collective by any rank will take at most $2\lceil \log_{k+1} n \rceil$ steps to propagate to all other ranks.

5.3.4 Algorithm Comparison

The characteristics of the algorithms we have discussed so far are summarized in Table 5.1. As we have discussed, in our pipelined variants of the standard exchange and Bruck collectives, we opted for an increased number of messages in comparison to the logarithmically scaling original versions. We believe that improved arrival pattern resistance will offset any associated loss in performance. The number of communication rounds and connections remains the same.

Because the ring algorithm inherently has no data dependencies among messages, its characteristics are the same in the original and the pipelined versions. The Cayley algorithm offers a compelling combination of the constant resource usage of the ring algorithm with the reduced number of communication rounds of the standard exchange and Bruck algorithms. Since it has constant resource usage and is resistant to process arrival pattern variation, it should be an improvement over pipelined standard exchange and Bruck. We present the results of our experimental evaluation in Section 5.5.

5.4 Implementing the Pipelined Collectives

Although all the variants of the pipelined collective design that we have discussed in this chapter are possible to implement either in a system that supports only host-based progression

or using hardware offloading, we designed the collectives with the view of using the CORE-Direct technology for the implementation. We will therefore focus on the offloaded design in this section, and will discuss the host-based option briefly in Section 5.4.4.

Collective operation designs for HPC clusters using CORE-Direct offloading, while varied, also have much in common. In particular, we noticed that in many designs communication is done in rounds during which a rank receives data from a set of peer ranks, and sends data to another set. Additionally, a hierarchical structure is commonly present in which a leader rank handles internode communication and separately communicates with ranks sharing its node through shared memory. To ease our own implementation burden, and potentially save future researchers from this repetitive work, we designed a building block that provides these features for algorithms using CORE-Direct offloading: the *Receive-Store-Replicate* (RSR) primitive. The design of the primitive has deliberately been kept generic in order to broaden its applicability.

Due to the nature of our collective design, messages may arrive out of order and require reassembly before the collective is complete. Since the benefits of hierarchical collectives are well established, we chose to perform the reassembly during the final intranode phase of a hierarchical collective. We do not evaluate our design in flat communicators, and do not expect it to perform well in this scenario.

In the next section, we will discuss the RSR primitive that we used to implement the collective algorithms presented in this chapter. In Sections 5.4.2 and 5.4.3, we will discuss how the primitive can be used to provide the features and implement the algorithms we proposed previously.

5.4.1 Receive-Store-Replicate Primitive for Non-blocking Collectives

We define a Receive-Store-Replicate primitive as an operation that receives a message on one port from its set of inbound ports, stores the message for future use, and sends it out on all of its outbound ports. The primitive was designed to match the capabilities provided

by the CORE-Direct hardware, specifically, its inability to use the contents of the message to control the execution of the offloaded operation. In Chapter 3, we mentioned that the only way to distinguish between different kinds of messages in this system is by the CQ on which the completion entry for that message was received. Recall from our discussion of the design for message matching from Section 3.2.2 that we employ a unique Completion Queue for each message envelope, and infer the message identity from the arrival of a corresponding CQE. Unlike the rendezvous protocol in Chapter 3, the RSR model does not need to identify individual messages. Rather, we wish to distinguish messages that are forwarded to a different set of downstream ports. Therefore, there is a one-to-one mapping between identifying Completion Queues and RSR instances.

In our implementation, we employ the Mellanox eXtended Reliable Communication extension of the InfiniBand specification in a somewhat non-traditional manner. Instead of using an XRC SRQ to share buffers between processes on the same node, we use it for its completion semantics. Unlike a conventional Shared Receive Queue, an XRC SRQ has an associated Receive Queue. This RQ is used to report message receipt instead of the per-QP RQs. XRC allows our design to use a single RQ per RSR instance.

Figure 5.5 illustrates the handling of a message by an RSR instance. An incoming message received by an RSR is placed in a buffer posted to the XRC SRQ and allocated in a shared memory region that is pre-registered with the HCA. The use of shared memory allows the message to be used for intranode communication as well as outbound forwarding over the network. A set of buffers is pre-posted to each RSR instance during its creation, making it similar to an unexpected message queue. As in the eager protocol, intermediate memory buffers of fixed size are used; messages are split into fragments to fit in these buffers. The Maximum Transmission Unit (MTU) of the IB adapter provides an appropriate upper limit for this chunk size. Because fragmentation already occurs at this message size at the InfiniBand layer, we can avoid additional fragmentation by choosing our chunk size to match the MTU. However, to negotiate buffer availability, the RSR implementation uses a

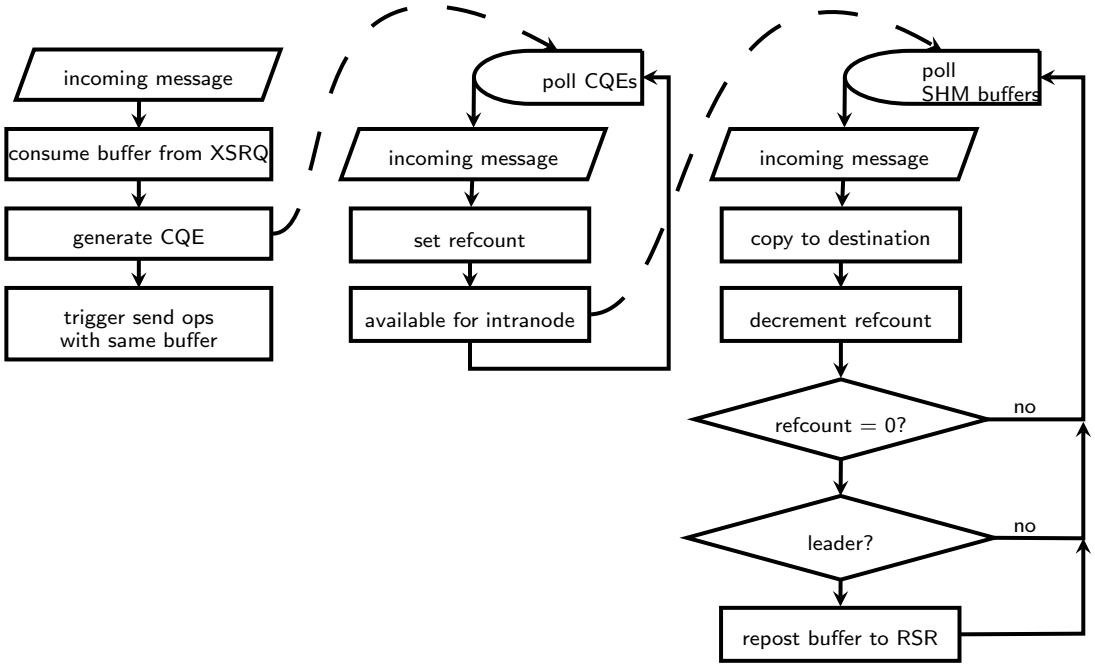


Figure 5.5: Receive-Store-Replicate control flow

handshaking protocol similar to that presented in Section 3.2.2. Therefore, the RSR design has characteristics of both the offloaded eager and rendezvous protocols.

Three asynchronous processes are at work in the RSR primitive. The data propagation pathway shown in the left section of Figure 5.5 is executed by the offloading engine and is independent of the host processor availability. It will forward any incoming message to a predefined set of destinations.

Completion events generated by incoming messages may be processed at the application's convenience. Because the downstream forwarding of the message is handled by the CORE-Direct offloading engine, the only work that must be performed on message receipt by the host progress engine is marking the shared buffer available for the intranode portion of the collective communication. This process is shown in the middle part of Figure 5.5. In the

case where the unexpected message belongs to a collective that a leader has not yet started, the message is placed on an in-memory unexpected queue to be re-examined later. Note that the forwarding of the message by the offloaded progress engine proceeds independently of this operation.

Intranode completion (shown on the right in Figure 5.5) is controlled by buffer reference counting. This portion of the algorithm is executed by both the node leaders and the leaf ranks in the hierarchical collective. When a message buffer is taken from the shared memory pool and pre-posted to an RSR, its reference count is initialized to reflect the number of intranode ranks participating in the collective served by that RSR. Once the CQE handler for the RSR signals that the message is available in shared memory, the ranks sharing the node can copy the data to their destination buffers at their convenience, as their timing does not affect the timing of the internode or intranode communication. Once the buffer is no longer in use, it is resubmitted to an RSR instance to be used again. Any outbound IB `send` operations hold additional references on the buffer to make sure it is not reused too early.

The preposted set of RSR buffers is refilled periodically by the progress engine. In addition, we make use of the asynchronous event feature of the SRQ to provide a backup mechanism for refilling the RSR from a helper thread. This is only necessary to keep the pipeline from stalling during times when the user application does not invoke the MPI progress engine. However, unlike the case of host-based non-blocking collective progression, the timing of the invocation of our refill mechanisms does not affect the latency of the collective, as long as the RSR does not run out of pre-posted buffers.

Figure 5.6 illustrates the structure of an RSR instance in terms of its constituent components. It shows a set of QPs implementing the inbound ports. Messages received on their RQs target an XRC SRQ and each consume a shared memory buffer. The outgoing sends are served from this buffer and are triggered by the completion of the receive. The shared memory broadcast runs asynchronously and propagates the data to the intranode ranks.

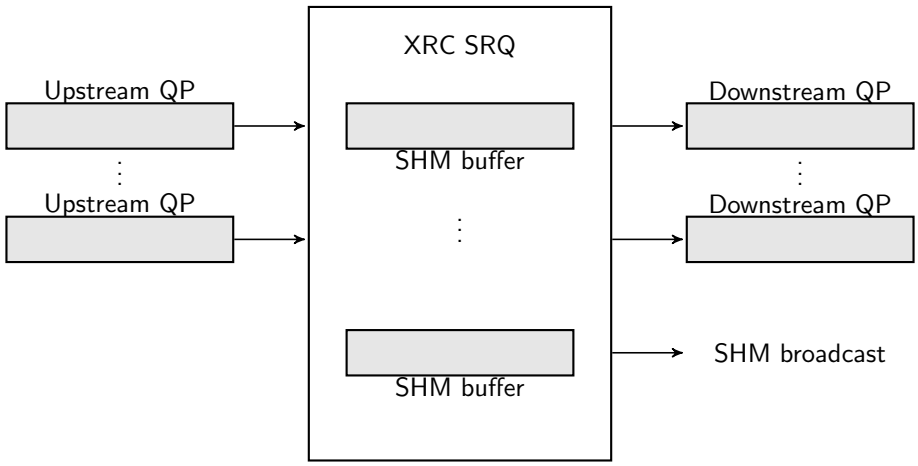


Figure 5.6: Receive-Store-Replicate primitive structure

Defining the RSR operation allows us to raise the level of abstraction in our discussion of the implementation. We no longer have to consider the delivery of individual messages, and can focus on the collective algorithm design. Now that we have presented the building block of our collectives, let us show how the primitive can be used to implement them.

5.4.2 Implementing Logarithmic Pipelined Algorithms

For the purposes of our work, the standard exchange and Bruck algorithms are almost identical. Because all messages are split into uniformly sized fragments and processed without concatenation, the standard exchange algorithm loses its advantage of omitting the local rotation step of the Bruck algorithm. The only difference that remains is in the communication pattern, specifically, the numbering of the ranks that communicate in each round of the collective. We will talk about implementing the Bruck algorithm using the RSR primitive in this section. Nonetheless, the techniques that we demonstrate apply equally to standard exchange.

Figure 5.7 shows the propagation of the contribution of rank 0 through the internode communicator using the Bruck algorithm with two ports. Recall from Chapter 4 that in round i of the Bruck algorithm using k ports, rank r sends data to ranks $r + (k+1)^i, r +$

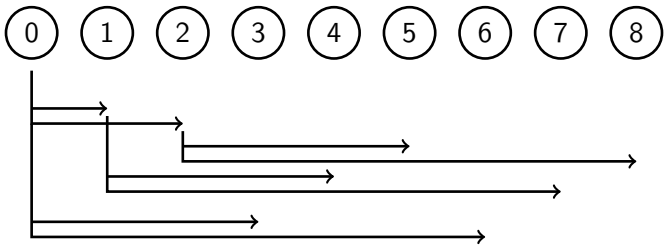


Figure 5.7: Propagation of the contribution of rank 0 in 2-port pipelined Bruck allgather

$2(k+1)^i, ..., r + k(k+1)^i$. In the first step, rank 0 sends its data item to ranks 1 and 2. In the second step, the ranks that have knowledge of the data originating at rank 0 will send them to ranks at offsets 3 and 6.

Consider the role that rank 2 plays in propagating the data item contributed by rank 0 in the internode communicator in Figure 5.7. If we examine in isolation the work that rank 2 does in the second step of internode communication, we can see that in addition to its own data it will send downstream the data that it received in the first step (specifically, the contribution of rank 0). Let us treat this communication step as a unit of work consisting of a *receive* step in which data from rank 0 arrive at rank 2, a *store* step in which rank 2 makes this data available to the intranode communicator, and a *replicate* step in which rank 2 propagates the data to ranks 5 and 8.

We implement rank 0's portion of the collective by instantiating one RSR primitive per communication round. The RSR instances differ in their inbound and outbound connections to RSRs on other ranks. The set of either inbound or outbound ports of an RSR may be empty, as signified by the symbol $\emptyset$. This corresponds to the beginning and end of the collective communication pattern. The RSRs instantiated on rank 0 are shown in Table 5.2. The notation A_b represents an RSR on rank A handling round b.

Once the RSR instances are connected, communication can be handled in asynchronous fashion. Recall that an RSR never stops receiving and forwarding incoming messages, regardless of whether a collective call is currently executing from the point of view of an individual rank. A late rank will still fulfil its message forwarding obligations, as well as store

Table 5.2: Receive-Store-Replicate units at rank 0 of the pipelined Bruck allgather

RSR ID	Data Items	Sources	Destinations
0	0	$\emptyset$	$\{1_1, 2_1, 3_2, 6_2\}$
1	7, 8	$\{7_0, 8_0\}$	$\{3_2, 6_2\}$
2	1, 2, 3, 4, 5, 6	$\{3_1, 6_1\}$	$\emptyset$

any unexpected messages for its own future use. Once the late rank arrives, it will process the unexpected messages in the intranode broadcast stage, contribute its data, and the collective operation can then complete. Thus, the RSR primitive satisfies the connectivity requirement that we specified in Section 5.3. Our collective design can therefore implement a process arrival pattern tolerant allgather while using fewer connections than the direct algorithm.

However, the number of connections per rank in this example is not $O(\log_{k+1} n)$, as Table 5.1 would suggest, but rather $O(\log^2_{k+1} n)$. This difference arises from the need to maintain independent progress of the RSR primitives representing the rounds of the collective. Because we wish for messages to be propagated without data dependencies, each RSR primitive requires its own set of QPs. The connection pattern in Table 5.2 can be derived by writing out the communication pattern of non-pipelined Bruck or standard exchange, and computing partial set sums in each round, working from higher to lower numbered rounds. Because each step adds $2k$ destinations, the total number of connections is the sum of an arithmetic series, $\sum_{i=0}^{\lceil \log_{k+1} n \rceil} 2ik \in 2k\lceil O(\log_{k+1} n)\rceil O(\log_{k+1} n) \in O(\log^2_{k+1} n)$.

Sharing connections among RSR instances would require examining the message prior to posting IB tasks to the HCA, which is not a feature provided by the CORE-Direct offloading engine. However, as we discussed in Section 5.3.3, we can reduce resource consumption via algorithmic improvements. Let us further our discussion on this topic.

5.4.3 Implementing the Cayley Allgather Algorithm

Although the offloaded collective presented in Section 5.4.2 meets our prerequisites for process arrival pattern tolerance, we would like to decrease the resource consumption of the collective further by implementing the Cayley allgather algorithm using RSR primitives.

In Section 5.3.3, we described the Cayley algorithm as being comprised of multiple broadcast operations, each using a directed broadcast tree based on a re-rooted undirected Cayley interconnection graph. Because the set of communicating peers for a given rank remains the same in all these broadcasts, in our implementation they all share a single set of QPs. In our implementation, every rank broadcasts any incoming data block on all ports other than the port of origin. The rank's own data items are broadcast to all of its connections. Because the interconnection graph is acyclic, messages can never propagate back in the direction of their arrival. Therefore, the collective is guaranteed to terminate.

The simplicity of the Cayley allgather algorithm makes it a good match for hardware offloading using the RSR primitive. The Cayley allgather communication pattern can be supplied by $k + 1$ RSR primitives instantiated on each node. Each of the first k instances will have a single upstream port, which is chosen in a way to ensure that one RSR instance is responsible for each of the rank's peers. The downstream ports of every RSR instance include all the peers except for the one that is acting as the upstream in that instance. An additional RSR instance without upstream connections is responsible for sending the rank's own message. It is connected to every peer. The total number of connections per rank is therefore $k(k + 1) \in O(1)$, because k is a constant.

As in the case of the logarithmic algorithms in Section 5.3.2, we can share the communication structures among multiple executing instances of the collective operation. This allows the Cayley allgather collective design to support any number of concurrent executions with constant resource usage while maintaining independent progress. Although serialization of individual message fragments still occurs in this model, message fragments belonging to

different collectives can be interleaved. This feature, along with the lack of data dependencies, allows the collectives to retain the ability to make independent progress.

5.4.4 Host-progressed Pipelined Non-blocking Collectives

Though we designed our collectives and the RSR primitive used to implement them with offloading in mind, the design does not depend on the presence of specialized hardware, and can be implemented using host-based progression alone. Because this line of research is outside of the scope of our work on offloading, we did not implement and test this design, leaving it as a possible future extension.

If we augmented the conventional MPI unexpected message queue with functionality that would allow unexpected messages to be forwarded over the network, we could implement all the collective algorithms described in this chapter. As long as the MPI application invokes the progress engine often enough, or a helper thread is provided, a rank that has yet to arrive at the collective can provide connectivity for the already-arrived ranks without being aware of the collective's details. Because the algorithms we use do not introduce data dependencies, the two requirements that we specified for process arrival pattern tolerance in Section 5.3 are satisfied.

5.5 Performance Evaluation

For our performance evaluation, we once again make use of the NBCBench benchmark [37] previously described in Section 3.3 to measure the latency and overlap potential of the allgather collectives. We implemented and tested the pipelined non-blocking Bruck allgather collective operation with 1-port and 3-port modeling, as well as the Cayley algorithm using 1 and 3 ports. We used the non-blocking `MPI_Iallgather` host-progressed collective provided by MVAPICH2 1.9 and the hierarchical Bruck collective design from Chapter 4 as baselines against which to evaluate the pipelined collectives.

For reasons discussed in Section 5.4, we only compared the allgather collectives in a hierarchical communicator. We feel that this decision is justified, because the use of hierarchical communicators is necessary to reap the benefits of low intranode communication latency on modern clusters with multicore processors.

We tested our collectives on the same Cluster B as described in Chapter 3. The testbed for the benchmark was made up of four Dell PowerEdge 2850 servers with two 2.80 GHz dual-core Intel Xeon Paxville processors, 4 GB DDR2 memory, and a Mellanox ConnectX-2 MT25418 HCA. The software used was Mellanox OFED 1.5.3-1 and 64-bit CentOS 5.5 with Linux kernel 2.6.18-194.26.1.el5. Two ports of each HCA were connected to the switch. Four MPI ranks were running on each node, and the system was set up to make use of shared memory for intranode data transfers.

At the time of writing, we have experienced a number of difficulties in combining the use of XRC with CORE-Direct. Therefore, our testing was limited to the range where the InfiniBand software stack was stable in this configuration. Specifically, we tested messages of up to 64KB in size. The MTU supported by our InfiniBand adapters is 1024 B in this configuration.

5.5.1 Latency and Overlap Potential

In terms of latency trends, the pipelined variant of the Bruck allgather collective performs similarly to the offloaded version without pipelining, as illustrated by Figure 5.8. This is not surprising, because the algorithms with and without pipelining have similar overhead in communicating with the CORE-Direct hardware and copying data from a shared memory region to the final destination. However, the pipelined Bruck allgather had a slight advantage in latency across all message sizes. The pipelined Cayley collective improved the latency of the collective operation for all message sizes compared to the pipelined Bruck design. We attribute this improvement to the reduced HCA resource usage by the Cayley collective, because IB HCA performance is sensitive to the number of queues that the HCA has to

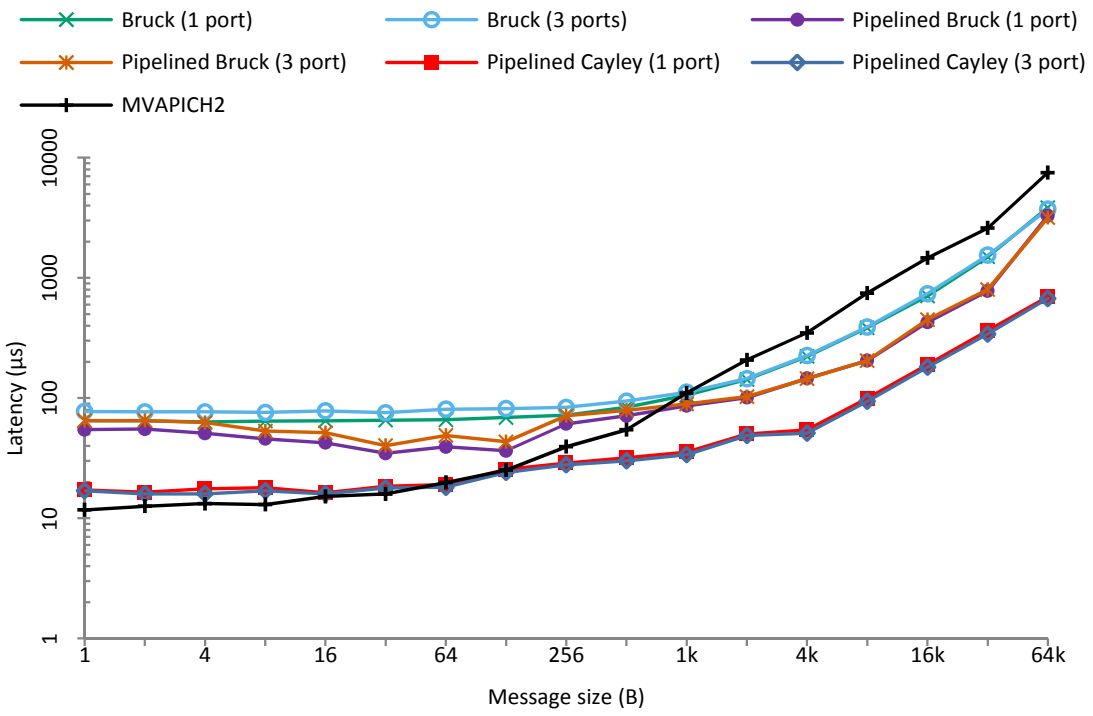


Figure 5.8: Non-blocking pipelined allgather collective message latency

manage, as we discussed in Section 4.4. Offloaded collectives are at a slight disadvantage compared to MVAPICH2 in messages smaller than 64 B and take the lead for larger messages.

For message sizes under 16 KB, the overlap potential of both pipelined collective designs is higher than that of the host-progressed collective, as demonstrated by Figure 5.9. The Cayley algorithm had a modest advantage in overlap over the pipelined Bruck design for small messages.

As the message size increases past 16 KB, the pipelined collectives lose their overlap advantage. We believe there are two reasons for this behaviour. First, as the message transfer time comes to dominate the overall collective latency, the influence of the process arrival pattern diminishes, because a greater portion of the work is done by the InfiniBand adapter even when host progression is used. Second, because our pipelined collective uses a fixed chunk size of 1024 B, as described in Section 5.4.1, message fragmentation begins to

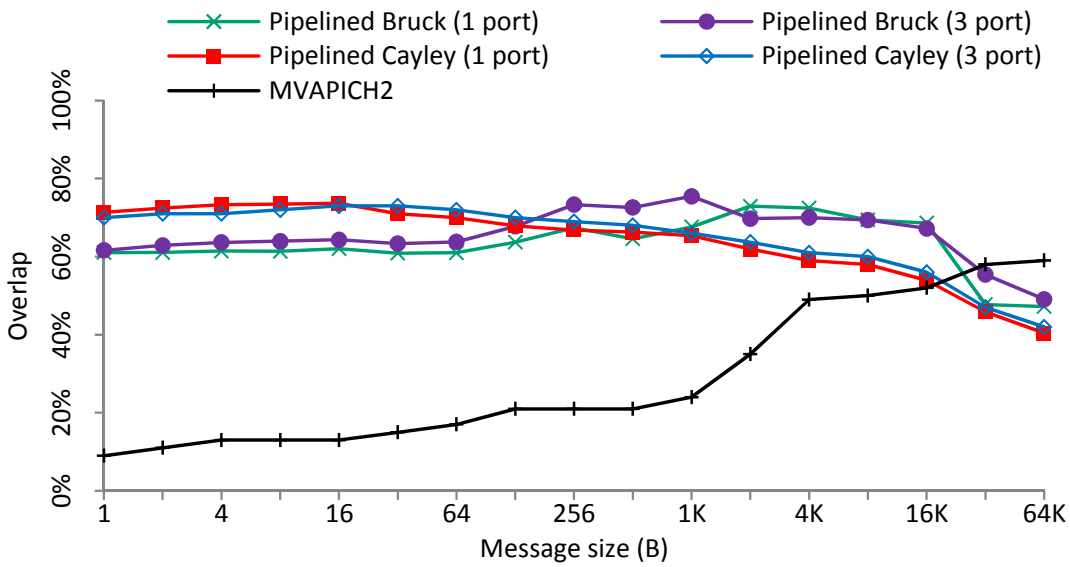


Figure 5.9: Non-blocking pipelined allgather collective overlap potential

outweigh the benefits of pipelining at large message sizes.

Interestingly, for small messages the overlap performance of the pipelined collectives was slightly below that of the variants introduced in Chapter 4. We believe that this is due to the implementation of the shared memory broadcast phase. In the non-pipelined collective, the data will be placed in the buffer contiguously and can be copied as a unit, reaping the benefits of prefetching. In contrast, the pipelined version must assemble the result of the communication from memory locations that are spaced further apart, defeating the prefetching features of the processor. At large message sizes, the non-blocking collectives act similar to one another. We also note that the use of multiple ports does not seem to provide a benefit on this testbed. Recall that multi-port offloaded allgather collectives had mixed performance results on this testbed in our investigation in Chapter 4. Flat collectives saw a small increase in latency, whereas hierarchical collectives enjoyed a performance improvement.

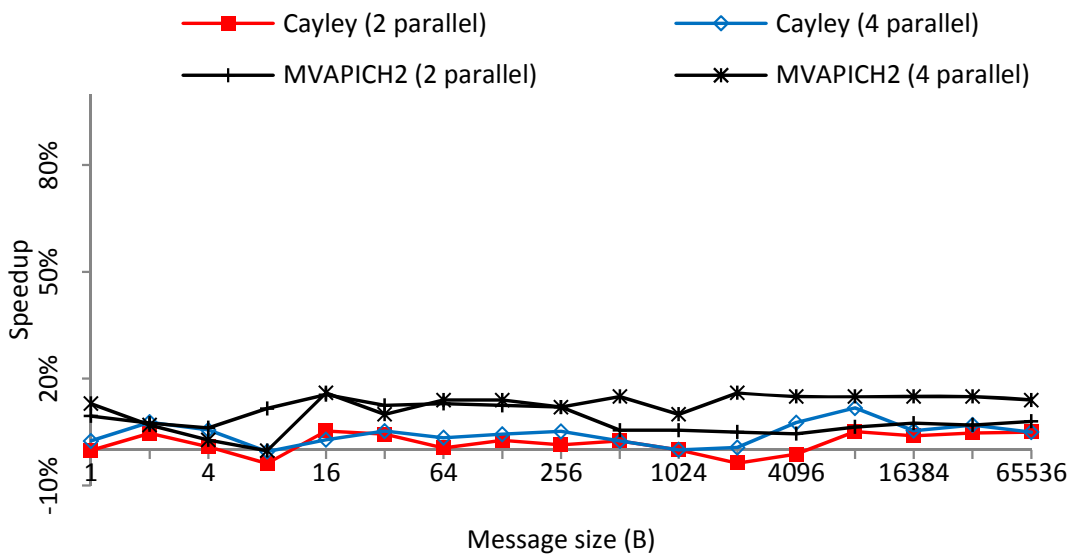


Figure 5.10: Improvement of latency in parallel instances of the allgather collective

5.5.2 Parallel Collective Operations

The Cayley allgather algorithm does not require additional resources for executing multiple instances of the collective concurrently. This property should translate into a scalability advantage in real-world applications, once non-blocking collectives become more popular with application developers. We used a modified version of the NBCBench benchmark to repeat the latency and overlap tests with multiple concurrent collective instances.

In our test, we started a batch of collective operations, performed synthetic computation, and then waited for all the collective instances to complete. The rest of the experimental setup remained unchanged. We measured the speedup that the parallel operations obtain in comparison to the sequential execution of the same operations. To simplify presentation, we only show the results for the 3-port Cayley collective. As in the previous section, the results for 1-port Cayley were similar.

As shown in Figure 5.10, the communication latency of the pipelined Cayley collective remains relatively unchanged, whether executing the collective calls in parallel or sequentially.

This behaviour can be explained by the good overlap capability of the non-blocking collective which enables the shared memory phase of a collective instance to be overlapped with the network communication of another. However, because the same amount of data must be copied within the node memory and transferred over the network as in the sequential case, the speedup is modest. Because the parallel operations share hardware resources, we do not expect a significant latency improvement. Our aim was to test the ability of the Cayley design to return consistent overlap performance in the presence of parallel operations.

In contrast, the host-progressed nonblocking collective supplied by MVAPICH2 sees a larger latency improvement. We believe that this effect is due to the ability of the progress engine to process multiple messages per invocation in the case of collective instances executing in parallel. However, this improvement comes at the cost of a significant deterioration in overlap capability, as illustrated by Figure 5.11. Because the progress engine must track a greater number of concurrent messages while performing network communication, its competition for processor time with the application's computation is increased, and a smaller fraction of processor time can be used for computation.

In contrast, the offloaded collective's overlap capability is essentially unaffected by the presence of parallel allgather calls. Because the message progression is performed by the HCA, and the amount of shared-memory data transfer that needs to be performed is unchanged, no additional burden is placed on the host processor.

5.5.3 Noise Tolerance

We sought to validate our hypothesis that pipelined collectives are better at dealing with OS noise. We performed preliminary testing of this hypothesis using articially generated noise. Noise was injected into the system by executing a noisemaker process on each processor core in the cluster. This process runs with real-time priority, preempting the other user processes. However, it spends most of its execution time sleeping. Each noisemaker process will wake up at random intervals and execute a busy wait pulse of 100 ms. The averaged duty cycle of

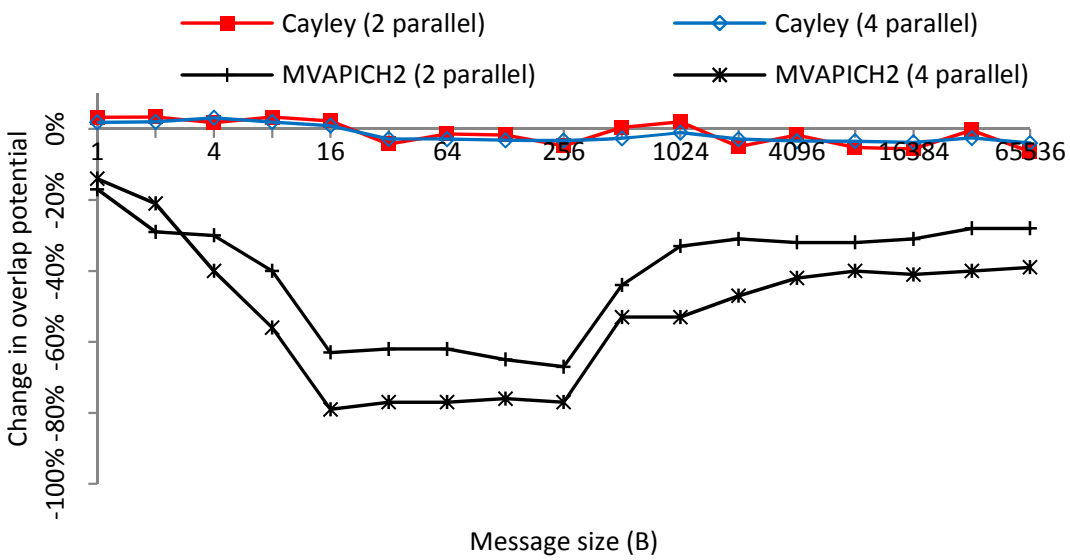


Figure 5.11: Degradation of overlap potential due to parallel instances of the allgather collective

the busy wait pulses is 20%.

The real-time priority causes the noisemaker processes to preempt the MPI processes, simulating interruptions from operating system services. Although the performance numbers varied with message size in this test, on average the pipelined collectives exhibited latency that was 16% lower than the collective without pipelining, and up to 40% lower than the MVAPICH2 CPU-progressed non-blocking allgather.

This test was used to expose the differences between the collective designs rather than act as a benchmark. While the amount of injected scheduling noise is high, it was necessary to simulate the effects of noise propagation in a large cluster using a much smaller system. In the future we would like to perform a study involving a larger cluster and realistic applications in the presence of scheduling noise.

5.6 Summary

In this chapter, we presented two designs for the allgather collective operation that are tolerant to variations in the process arrival pattern. One of the designs also offers potential for improved scalability, since it operates using a constant amount of resources regardless of the number of concurrent collective calls. As non-blocking collectives become more popular, we believe that the number of concurrent instances of non-blocking collectives will correspondingly increase. To meet this challenge, our collective design enables independent progress with constant resource requirements while maintaining latency and overlap characteristics. We conducted a microbenchmark-based investigation to validate these claims. In the future, we wish to test our design on a larger system with more applications.

The pipelined Bruck and Cayley designs are supported by the Receive-Store-Replicate primitive that we intend to serve as a generic building block for offloaded collective operations. The primitive maps well to the features provided by CORE-Direct HCAs. The RSR primitive has widely applicable semantics of receiving messages from a set of upstream ports and storing them before forwarding the messages downstream. We believe that it provides a useful abstraction of CORE-Direct hardware, and should be useful in designing collectives other than allgather.

In this chapter, we showed that in addition to being a valuable tool for reducing communication latency, pipelining can be used to break dependencies in the communication pattern of a collective operation. This approach has potential to improve the noise tolerance of the collective communication and improve the performance of the applications that use the collectives. Additionally, our design enables execution of multiple concurrent non-blocking collective operations in a scalable fashion. Although preliminary results have been mixed, we believe that the approach warrants further investigation.

Chapter 6

Conclusions and Future Work

6.1 Summary of Findings

The systems interconnected in HPC clusters continue to increase in number, leading to increased demands on the scalability of the cluster software infrastructure. In this work we investigated one approach to meeting these demands by employing specialized hardware to assist the software infrastructure in overlapping communication with computation. We examined the issues that MPI point-to-point and collective communication operations face in modern clusters, identified progression of message transfers as a specific area for improvement and proposed progression offloading as the solution.

In Chapter 3, we applied offloading to the progression of large single-message data transfers using the rendezvous protocol. We found that offloading the progression of the rendezvous protocol to Mellanox CORE-Direct hardware greatly improved the noise tolerance of the system and allowed for overlap between communication and computation without undue impact on latency.

In Chapter 4, we developed designs for flat and hierarchical offloaded non-blocking collective operations, focusing on the allgather collective. We found that while both flat and hierarchical variants of the offloaded collective outperformed collectives using software

progression in terms of overlap, communication latency was increased for small messages. The hierarchical design brought a reduction in latency. However, the improved latency of the hierarchical collective came at the expense of decreased overlap capability.

In Chapter 5, we improved the immunity of collective operations to the process arrival pattern and scheduling noise by reducing the amount of synchronization between the processes participating in the collective. We achieved this goal with a pipelined collective design that eschews message concatenation in order to reduce the dependencies between the ranks in the collective. Additionally, the ranks in this collective that are late to arrive can assist the other ranks with the collective communication. Next, we introduced a new algorithm for the allgather collective aimed at improving resource usage. The new algorithm, based on the structure of the Cayley tree, ensures that resource usage stays constant even with multiple active instances of the collective operation in progress.

In summary, in this work we developed designs for offloading the processing of non-blocking point-to-point and collective operations from the host processor. We showed that these designs have advantages over their host-progressed counterparts in their ability to overlap computation with communication, which is the *raison d'être* of non-blocking operations. We also showed that offloading can reduce implicit synchronization between processes, leading to better noise immunity and scalability of the non-blocking operations.

6.2 Future Work

We would like to continue the work in Chapter 3 by investigating the applicability of approaches that combine eager and rendezvous protocols to offloading. We would also like to try tuning the protocols for various message sizes.

The work in Chapter 4 could be extended by investigating how to maintain the overlap performance of the offloaded hierarchical collectives while preserving their latency advantage over flat collectives. A possible means of achieving this goal would be to free up the

CORE-Direct HCA by offloading message copying to hardware such as the Intel I/OAT [99] engine.

To continue the work in Chapter 5 we would like to investigate the performance of the pipelined allgather collective on a larger system to confirm the preliminary results discussed in the thesis. In addition, we would like to leverage the Receive-Store-Replicate primitive introduced in the chapter as a building block for other collectives. It would also be useful to investigate a purely host-based design for the primitive in order to improve compatibility with systems that do not offer specialized hardware for communication offloading.

The collective communication designs introduced in Chapter 5 demonstrate that it is possible to provide collectives that have scalable resource requirements and process arrival pattern immunity. This could ease MPI programming in the future by freeing the programmer from concern about process arrival patterns. We would like to extend the work in this chapter by investigating applications that would benefit from this treatment.

In addition, there are other ways that MPI applcation writing could be simplified. To improve the usability of non-blocking collectives, an interface to the collective could provide support for signaling partial completion. The MPI specification supplies the function `MPI_Get_count` that reports the amount of data transferred so far by a non-blocking point-to-point operation. In future work, we would like to provide a similar interface for non-blocking collectives.

Two final areas for future work are implementing offloaded collective operations with native InfiniBand hardware broadcast capabilities and integrating communication progression offloading with GPU acceleration technology, which is rapidly making inroads into HPC systems.

Bibliography

[1] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. *ExaScale Software Study: Software Challenges in Extreme Scale Systems*. Tech. rep. DARPA IPTO, Air Force Research Laboratory, Sept. 14, 2009.

[2] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proc. Spring Joint Computer Conference*. Atlantic City, New Jersey, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[3] G. Amerson and A. Apon. "Implementation and design analysis of a network messaging module using Virtual Interface Architecture". In: *Proc. IEEE Intl. Conf. Cluster Computing*. Sept. 2004, pp. 255–265. DOI: 10.1109/CLUSTR.2004.1392623.

[4] T. E. Anderson, D. E. Culler, and D. A. Patterson. "A Case for NOW (Networks of Workstations)". In: *IEEE Micro* 15.1 (Feb. 1995), pp. 54–64. DOI: 10.1109/40.342018.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. *The NAS Parallel Benchmarks*. Tech. rep. RNR-94-007. Department of Mathematics and Computer Science, Emory University, Mar. 1994.

[6] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. "MPI on Millions of Cores". In: *Parallel Processing Letters* 21.01 (2011), pp. 45–60. DOI: 10.1142/S0129626411000060.

[7] B. Bao, C. Ding, Y. Gao, and R. Archambault. "Delta Send-Recv for Dynamic Pipelining in MPI Programs". In: *Proc. IEEE Intl. Symp. Cluster Computing and the Grid*. Ottawa, ON, Canada, 2012, pp. 384–392. DOI: 10.1109/CCGrid.2012.113.

[8] G. Benson, C. Chu, Q. Huang, and S. Caglar. "A Comparison of MPICH Allgather Algorithms on Switched Networks". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by J. Dongarra, D. Laforenza, and S. Orlando. Vol. 2840. Lecture Notes in Computer Science. Springer, 2003, pp. 335–343. DOI: 10.1007/978-3-540-39924-7_47.

[9] M. Bertozzi, F. Boselli, G. Conte, and M. Reggiani. "An MPI Implementation on the Top of the Virtual Interface Architecture". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by J. Dongarra, E. Luque, and T. Margalef. Vol. 1697. Lecture Notes in Computer Science. Springer, 1999, pp. 199–206. DOI: 10.1007/3-540-48158-3_25.

[10] S. H. Bokhari. "Multiphase Complete Exchange on Paragon, SP2, and CS-2". In: *Parallel and Distributed Technology: Systems and Applications* 4.3 (Sept. 1996), pp. 45–59. DOI: 10.1109/88.532139.

[11] G. Bosilca, T. Herault, A. Rezmerita, and J. Dongarra. "On Scalability for MPI Runtime Systems". In: *Proc. IEEE Intl. Conf. Cluster Computing*. Austin, TX, Sept. 2011, pp. 187–195. DOI: 10.1109/CLUSTER.2011.29.

[12] P. Bozeman and B. Saphir. *A Modular High Performance Implementation of the Virtual Interface Architecture*. Tech. rep. LBNL-46455. National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, July 7, 2000.

[13] R. Brightwell and K. D. Underwood. "An Analysis of the Impact of MPI Overlap and Independent Progress". In: *Proc. International Conference on Supercomputing*. Malo, France, pp. 298–305. DOI: 10.1145/1006209.1006251.

[14] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. "Efficient Algorithms for All-to-All Communications in Multiport Message-passing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 8 (Nov. 1997), pp. 1143–1156. DOI: 10.1109/71.642949.

[15] D. Buntinas, G. Mercier, and W. Gropp. "Data Transfers between Processes in an SMP System: Performance Study and Application to MPI". In: *Proc. Intl. Conf. Parallel Processing*. 2006, pp. 487–496. DOI: 10.1109/ICPP.2006.31.

[16] D. Buntinas, D. K. Panda, and R. Brightwell. "Application-Bypass Broadcast in MPICH over GM". In: *Proc. IEEE Intl. Symp. Cluster Computing and the Grid*. Tokyo, Japan, 2003, pp. 404–411. DOI: 10.1109/CCGRID.2003.1199346.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language". In: *Intl. J. High Performance Computing Applications* 21.3 (Aug. 2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: an object-oriented approach to non-uniform cluster computing". In: *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*. San Diego, CA, 2005, pp. 519–538. DOI: 10.1145/1094811.1094852.

[19] G. Cong and D. Bader. "Lock-Free Parallel Algorithms: An Experimental Study". In: *High Performance Computing – HiPC 2004*. Ed. by L. Bougé and V. K. Prasanna. Vol. 3296. Lecture Notes in Computer Science. Springer, 2005, pp. 516–527. DOI: 10.1007/978-3-540-30474-6_54.

[20] A. Danalis, K. Kim, L. Pollock, and M. Swany. "Transformations to Parallel Codes for Communication-Computation Overlap". In: *Proc. ACM/IEEE Supercomputing Conference*. Seattle, WA, 2005, pp. 58–70.

[21] J. J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software*. Tech. rep. CS-89-85. University of Tennessee, 2013.

[22] A. Faraj, P. Patarasuk, and X. Yuan. "A Study of Process Arrival Patterns for MPI Collective Operations". In: *Intl. J. Parallel Programming* 36.6 (2008), pp. 543–570. DOI: 10.1007/10766-008-0070-9.

[23] A. Gavrilovska. *Attaining High Performance Communications: A Vertical Approach*. CRC Press, 2009. ISBN: 978-1-4200-9308-7.

[24] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. "ConnectX-2 InfiniBand Management Queues: First Investigation of the New Support for Network Offloaded Collective Operations". In: *Proc. IEEE Intl. Symp. Cluster Computing and the Grid*. 2010, pp. 53–62. DOI: 10.1109/CCGRID.2010.9.

[25] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. "Overlapping Computation and Communication: Barrier Algorithms and ConnectX-2 CORE-Direct Capabilities". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470854.

[26] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. 2nd ed. Addison Wesley, Jan. 2003. ISBN: 0201648652.

[27] W. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI – 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. 2nd ed. MIT Press, Nov. 1999. ISBN: 0262571323.

[28] J. L. Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31 (1988), pp. 532–533.

[29] K. Hemmert, B. Barrett, and K. Underwood. "Using Triggered Operations to Offload Collective Communication Operations". In: *Recent Advances in the Message Passing Interface*. Ed. by R. Keller, E. Gabriel, M. Resch, and J. Dongarra. Vol. 6305. Lecture Notes in Computer Science. Springer, 2010, pp. 249–256. DOI: 10.1007/978-3-642-15646-5_26.

[30] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers". In: *Proc. ACM/IEEE Supercomputing Conference*. 2005, pp. 35–44. DOI: 10.1109/SC.2005.53.

[31] R. Hockney. "Performance Parameters and Benchmarking of Supercomputers". In: *J. Parallel Computing* 17.10–11 (Dec. 1991), pp. 1111–1130. DOI: 10.1016/S0167-8191(05)80029-8.

[32] T. Hoefler and A. Lumsdaine. "Optimizing non-blocking collective operations for InfiniBand". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* Apr. 2008, pp. 1–8. DOI: 10.1109/ipdps.2008.4536138.

[33] T. Hoefler, A. Lumsdaine, and W. Rehm. "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI". In: *Proc. ACM/IEEE Supercomputing Conference*. Reno, NV, Nov. 2007, 52:1–52:10. DOI: 10.1145/1362622.1362692.

[34] T. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine. "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra. Vol. 4192. Lecture Notes in Computer Science. Springer, 2006, pp. 374–382. DOI: 10.1007/11846802_52.

[35] T. Hoefler, P. Kambadur, R. Graham, G. Shipman, and A. Lumsdaine. "A Case for Standard Non-Blocking Collective Operations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by F. Cappello, T. Herault, and J. Dongarra. Vol. 4757. Lecture Notes in Computer Science. Springer, 2007, pp. 125–134. DOI: 10.1007/978-3-540-75416-9_22.

[36] T. Hoefler and A. Lumsdaine. "Message Progression in Parallel Computing – to Thread or not to Thread?" In: *Proc. IEEE Intl. Conf. Cluster Computing*. Tsukuba, Japan, 2008, pp. 213–222. DOI: 10.1109/CLUSTR.2008.4663774.

[37] T. Hoefler, T. Schneider, and A. Lumsdaine. "Accurately Measuring Collective Operations at Massive Scale". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* Miami, FL, Apr. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536494.

[38] T. Hoefler, T. Schneider, and A. Lumsdaine. "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation". In: *Proc. ACM/IEEE Supercomputing Conference*. New Orleans, LA, 2010, pp. 1–11. DOI: 10.1109/SC.2010.12.

[39] *InfiniBand Trade Association*. URL: http://www.infinibandta.org/ (visited on 09/01/2013).

[40] G. Inozemtsev and A. Afsahi. "Designing an Offloaded Nonblocking MPI_Allgather Collective Using CORE-Direct". In: *Proc. IEEE Intl. Conf. Cluster Computing*. Beijing, China, Sept. 2012, pp. 477–485. DOI: 10.1109/CLUSTER.2012.75.

[41] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-026. Intel Corporation. Apr. 2012.

[42] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. "Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System". In: *Proc. ACM/IEEE*

Supercomputing Conference. Phoenix, AZ, 2003, pp. 1–20. DOI: 10.1145/1048935.1050161.

[43] L. Kale, S. Kumar, and K. Varadarajan. "A Framework for Collective Personalized Communication". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* 2003, pp. 69.1–69.8. DOI: 10.1109/IPDPS.2003.1213166.

[44] K. Kandalla, A. Buluc, H. Subramoni, K. Tomko, J. Vienne, L. Oliker, and D. Panda. "Can Network-Offload Based Non-blocking Neighborhood MPI Collectives Improve Communication Overheads of Irregular Graph Algorithms?" In: *Proc. IEEE Intl. Conf. Cluster Computing*. Sept. 2012, pp. 222–230. DOI: 10.1109/ClusterW.2012.40.

[45] K. Kandalla, H. Subramoni, J. Vienne, S. P. Raikar, K. Tomko, S. Sur, and D. K. Panda. "Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: A Case Study with HPL". In: *Proc. IEEE Symp. High Performance Interconnects*. Aug. 2011, pp. 27–34. DOI: 10.1109/HOTI.2011.14.

[46] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. R. de Supinski, and D. K. Panda. "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* May 2012, pp. 1156–1167. DOI: 10.1109/IPDPS.2012.106.

[47] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda. "Designing Multi-leader-based Allgather Algorithms for Multi-core Clusters". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160896.

[48] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda. "High-performance and Scalable Non-blocking All-to-All with Collective Offload on InfiniBand Clusters: a Study with Parallel 3D FFT". In: *Computer Science - Research and Development* 26.3 (June 2011), pp. 237–246. DOI: 10.1007/s00450-011-0170-4.

[49] M. J. Koop, J. K. Sridhar, and D. K. Panda. "TupleQ: Fully-asynchronous and Zero-copy MPI over InfiniBand". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* Rome, Italy, May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161056.

[50] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda. "Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by A. Lastovetsky, T. Kechadi, and J. Dongarra. Vol. 5205. Lecture Notes in Computer Science. Springer, 2008, pp. 185–193. DOI: 10.1007/978-3-540-87475-1_27.

[51] J. Ladd, M. G. Venkata, R. Graham, and P. Shamis. "Analyzing the Effects of Multicore Architectures and On-Host Communication Characteristics on Collective Communications". In: *Proc. Intl. Conf. Parallel Processing Workshops*. 2011, pp. 406–415. DOI: 10.1109/ICPPW.2011.15.

[52] J. S. Ladd, M. G. Venkata, R. Graham, and P. Shamis. "Assessing the Performance and Scalability of a Novel Multilevel K-Nomial Allgather on CORE-Direct Systems". In: *Proc. Euro-Par 2012 Parallel Processing*. Ed. by C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis. Vol. 7484. Lecture Notes in Computer Science. Springer, 2012, pp. 538–549. DOI: 10.1007/978-3-642-32820-6_53.

[53] C. Li, C. Ding, and K. Shen. "Quantifying the cost of context switch". In: *Proc. Workshop on Experimental Computer Science*. San Diego, CA, 2007, pp. 2.1–2.4. DOI: 10.1145/1281700.1281702.

[54] S. Li, T. Hoefler, and M. Snir. "NUMA-aware Shared-memory Collective Communication for MPI". In: *Proc. Intl. Symp. High-performance Parallel and Distributed Computing*. New York, New York, USA, 2013, pp. 85–96. DOI: 10.1145/2462902.2462903.

[55] *libNBC – Nonblocking MPI Collective Operations*. URL: http://www.unixer.de/research/nbcoll/libnbc/ (visited on 10/05/2013).

[56] J. Liu, A. Mamidala, and D. Panda. "Fast and Scalable MPI-level Broadcast Using InfiniBand's Hardware Multicast Support". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* 2004, pp. 1–10. DOI: 10.1109/IPDPS.2004.1302912.

[57] J. Liu, J. Wu, and D. K. Panda. "High Performance RDMA-based MPI Implementation over InfiniBand". In: *Int. J. Parallel Programming* 32.3 (June 2004), pp. 167–198. DOI: 10.1023/B:IJPP.0000029272.69895.c1.

[58] M. Luo, P. Lai, S. Potluri, E. P. Mancini, H. Subramoni, K. Kandalla, and D. K. Panda. "A Portable InfiniBand Module for MPICH2/Nemesis: Design and Evaluation". In: *Proc. 10th Workshop on Communication Architecture for Clusters, in conjunction with ICPP 2010*. San Diego, CA, 2010, pp. 1–10.

[59] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics". In: *Proc. IEEE Intl. Symp. Cluster Computing and the Grid*. May 2008, pp. 130–137. DOI: 10.1109/CCGRID.2008.87.

[60] A. R. Mamidala, A. Vishnu, and D. K. Panda. "Efficient Shared Memory and RDMA Based Design for MPI_Allgather over InfiniBand". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra. Vol. 4192. Lecture Notes in Computer Science. Springer, 2006, pp. 66–75. DOI: 10.1007/11846802_17.

[61] *Mellanox Technologies*. URL: http://www.mellanox.com/ (visited on 02/01/2014).

[62] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm. "Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack". In: *Proc. Euro-Par 2006 Parallel Processing*. Ed. by W. E. Nagel, W. V. Walter, and W. Lehner. Vol. 4128. Lecture Notes in Computer Science. Springer, 2006, pp. 124–33. DOI: 10.1007/11823285_13.

[63] *MPI: A Message-Passing Interface Standard, Version 2.2*. Tech. rep. Message Passing Interface Forum, Sept. 2009.

[64] *MPI: A Message-Passing Interface Standard, Version 3.0*. Tech. rep. Message Passing Interface Forum, Sept. 2012.

[65] *MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE*. URL: `http://mvapich.cse.ohio-state.edu/` (visited on 09/01/2013).

[66] J. Nieplocha and R. Harrison. "Shared Memory Programming in Metacomputing Environments: The Global Array Approach". In: *J. Supercomputing* 11 (2 1997), pp. 119–136. DOI: `10.1023/A:1007955822788`.

[67] A. Nomura and Y. Ishikawa. "Design of Kernel-Level Asynchronous Collective Communication". In: *Recent Advances in the Message Passing Interface*. Ed. by R. Keller, E. Gabriel, M. Resch, and J. Dongarra. Lecture Notes in Computer Science. Springer, 2010, pp. 92–101. DOI: `10.1007/978-3-642-15646-5_10`.

[68] *OpenMP Application Program Interface Version 3.1*. OpenMP Architecture Review Board. July 2011.

[69] D. Padua. *Encyclopedia of Parallel Computing*. Springer, 2011. ISBN: 978-0-3870-9765-7.

[70] S. Pakin. "Receiver-initiated Message Passing over RDMA Networks". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* 2008, pp. 1–12. DOI: `10.1109/IPDPS.2008.4536262`.

[71] P. Patarasuk and X. Yuan. "Efficient MPI_Bcast across different process arrival patterns". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* Apr. 2008, pp. 1–11. DOI: `10.1109/IPDPS.2008.4536308`.

[72] F. Petrini, D. J. Kerbyson, and S. Pakin. "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q". In: *Proc. ACM/IEEE Supercomputing Conference*. 2003, pp. 55–72. DOI: 10.1109/SC.2003.10010.

[73] Y. Qian and A. Afsahi. "Efficient Shared Memory and RDMA Based Collectives on Multi-rail QsNet^{II} SMP Clusters". In: *Cluster Computing* 11.4 (Oct. 16, 2008), pp. 341–354. DOI: 10.1007/s10586-008-0065-8.

[74] Y. Qian and A. Afsahi. "Process Arrival Pattern Aware Alltoall and Allgather on InfiniBand Clusters". In: *Intl. J. Parallel Programming* 39.4 (2011), pp. 473–493. DOI: 10.1007/s10766-010-0152-3.

[75] Y. Qian and A. Afsahi. "RDMA-based and SMP-aware Multi-port All-Gather on Multi-rail QsNet^{II} SMP Clusters". In: *Proc. Intl. Conf. Parallel Processing*. 2007, pp. 48–57. DOI: 10.1109/ICPP.2007.69.

[76] Y. Qian, M. J. Rashti, and A. Afsahi. "Multi-Connection and Multi-Core Aware All-Gather on InfiniBand Clusters". In: *Proc. Intl. Conf. Parallel and Distributed Computing and Systems*. Orlando, FL, 2008, pp. 1–7. ISBN: 978-0-88986-773-4.

[77] I. Rabinovitz, P. Shamis, R. Graham, N. Bloch, and G. Shainer. "Network Offloaded Hierarchical Collectives Using ConnectX-2's CORE-Direct Capabilities". In: *Recent Advances in the Message Passing Interface*. Ed. by R. Keller, E. Gabriel, M. Resch, and J. Dongarra. Lecture Notes in Computer Science. Springer, 2010, pp. 102–112. DOI: 10.1007/978-3-642-15646-5_11.

[78] M. J. Rashti and A. Afsahi. "Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects". In: *Intl. Symp. High Performance Computing Systems and Applications*. 2008, pp. 95–101. DOI: 10.1109/HPCS.2008.10.

[79] M. J. Rashti and A. Afsahi. "A speculative and adaptive MPI rendezvous protocol over RDMA-enabled interconnects". In: *Intl. J. Parallel Programming* 37.2 (Apr. 2009), pp. 223–246. DOI: 10.1007/s10766-009-0094-9.

[80] M. Rashti and A. Afsahi. "Exploiting Application Buffer Reuse to Improve MPI Small Message Transfer Protocols over RDMA-enabled Networks". In: *Cluster Computing* (June 3, 2011), pp. 1–12. DOI: 10.1007/s10586-011-0165-8.

[81] J. Sancho, K. Barker, D. Kerbyson, and K. Davis. "Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications". In: *Proc. ACM/IEEE Supercomputing Conference*. Nov. 2006, pp. 17–33. DOI: 10.1109/SC.2006.51.

[82] T. Schneider, T. Hoefler, R. Grant, B. Barrett, and R. Brightwell. "Protocols for Fully Offloaded Collective Operations on Accelerated Network Adapters". In: *Proc. Intl. Conf. Parallel Processing*. Lyon, France, Oct. 2013, pp. 593–602. DOI: 10.1109/ICPP.2013.73.

[83] T. Schneider, S. Eckelmann, T. Hoefler, and W. Rehm. "Kernel-Based Offload of Collective Operations – Implementation, Evaluation and Lessons Learned". In: *Proc. Intl. Conf. Parallel Processing*. 2011, pp. 264–275. DOI: 10.1007/978-3-642-23397-5_26.

[84] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. "InfiniBand Scalability in Open MPI". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* 2006, pp. 1–10. DOI: 10.1109/IPDPS.2006.1639335.

[85] G. M. Shipman, S. Poole, P. Shamis, and I. Rabinovitz. "X-SRQ - Improving Scalability and Performance of Multi-core InfiniBand Clusters". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by A. Lastovetsky, T. Kechadi, and J. Dongarra. Vol. 5205. Lecture Notes in Computer Science. Springer, 2008, pp. 33–42. DOI: 10.1007/978-3-540-87475-1_11.

[86] M. Small, Z. Gu, and X. Yuan. "Near-Optimal Rendezvous Protocols for RDMA-Enabled Clusters". In: San Diego, CA, Sept. 2010, pp. 644–652. DOI: 10.1109/ICPP.2010.72.

[87] M. Small and X. Yuan. "Maximizing MPI point-to-point communication performance on RDMA-enabled clusters with customized protocols". In: *Proc. Intl. Conf. Supercomputing*. Yorktown Heights, NY, 2009, pp. 306–315. DOI: 10.1145/1542275.1542320.

[88] T. Sterling, D. J. Becker, and D. F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, May 1999. ISBN: 026269218X.

[89] H. Subramoni, K. Kandalla, S. Sur, and D. K. Panda. "Design and Evaluation of Generalized Collective Communication Primitives with Overlap Using ConnectX-2 Offload Engine". In: *Proc. IEEE Symp. High Performance Interconnects*. 2010, pp. 40–49. DOI: 10.1109/HOTI.2010.22.

[90] S. Sur, U. K. R. Bondhugula, A. Mamidala, H. Jin, and D. K. Panda. "High Performance RDMA Based All-to-All Broadcast for InfiniBand Clusters". In: *High Performance Computing – HiPC 2005*. Ed. by D. A. Bader, M. Parashar, V. Sridhar, and V. K. Prasanna. Vol. 3769. Lecture Notes in Computer Science. Springer, 2005, pp. 148–157. DOI: 10.1007/11602569_19.

[91] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda. "Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* 2006, pp. 101–110. DOI: 10.1109/IPDPS.2006.1639336.

[92] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. "RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits". In: *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. New York, NY, 2006, pp. 32–39. DOI: 10.1145/1122971.1122978.

[93] R. Thakur, R. Rabenseifner, and W. Gropp. "Optimization of Collective Communication Operations in MPICH". In: *Intl. J. High Performance Computing Applications* 19.1 (Feb. 1, 2005), pp. 49–66. DOI: 10.1177/1094342005051521.

[94] *The Open Group Base Specifications, Issue 6, IEEE Std 1003.1*. IEEE and The Open Group. 2004.

[95] *TOP500 Supercomputing Sites*. URL: http://www.top500.org (visited on 12/02/2013).

[96] J. L. Träff. "Efficient Allgather for Regular SMP-Clusters". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra. Vol. 4192. Lecture Notes in Computer Science. Springer, 2006, pp. 58–65. DOI: 10.1007/11846802_16.

[97] J. L. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, and W. Gropp. "A Simple, Pipelined Algorithm for Large, Irregular All-gather Problems". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by A. Lastovetsky, T. Kechadi, and J. Dongarra. Vol. 5205. Lecture Notes in Computer Science. Springer, 2008, pp. 84–93. DOI: 10.1007/978-3-540-87475-1_16.

[98] UPC Consortium. *UPC Language Specifications, v1.2*. Tech. rep. LBNL-59208. Lawrence Berkeley National Lab, 2005.

[99] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. "Efficient asynchronous memory copy operations on multi-core systems and I/OAT". In: *Proc. IEEE Intl. Conf. Cluster Computing*. Sept. 2007, pp. 159–168. DOI: 10.1109/CLUSTR.2007.4629228.

[100] M. G. Venkata, R. L. Graham, J. S. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer. "ConnectX-2 CORE-Direct Enabled Asynchronous Broadcast Collective Communications". In: *Proc. IEEE Intl. Parallel & Distributed Processing Symp.* Anchorage, AK: IEEE, May 2011, pp. 781–787. ISBN: 978-1-61284-425-1. DOI: 10.1109/IPDPS.2011.221.

[101] M. G. Venkata, R. L. Graham, J. Ladd, and P. Shamis. "Exploring the All-to-All Collective Optimization Space with ConnectX CORE-Direct". In: *Proc. Intl. Conf. Parallel Processing*. Sept. 2012, pp. 289–298. DOI: 10.1109/ICPP.2012.28.

[102] E. W. Weisstein. *Cayley Tree. From MathWorld–A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/CayleyTree.html (visited on 11/13/2013).

[103] W. Yu, S. Sur, D. K. Panda, R. T. Aulwes, and R. L. Graham. "High Performance Broadcast Support in LA-MPI Over Quadrics". In: *Intl. J. High Performance Computing Applications* 19.4 (Nov. 1, 2005), pp. 453–463. DOI: 10.1177/1094342005056145.

[104] W. Yu, J. Wu, and D. K. Panda. "Fast and Scalable Startup of MPI Programs in InfiniBand Clusters". In: *High Performance Computing – HiPC 2004*. Ed. by L. Bougé and V. K. Prasanna. Vol. 3296. Lecture Notes in Computer Science. Springer, 2005, pp. 440–449. DOI: 10.1007/978-3-540-30474-6_47.

[105] M. Zagha and G. E. Blelloch. "Radix sort for vector multiprocessors". In: *Proc. ACM/IEEE Supercomputing Conference*. 1991, pp. 712–721. DOI: 10.1145/125826.126164.

[106] J. Zounmevo and A. Afsahi. "Investigating Scenario-Conscious Asynchronous Rendezvous over RDMA". In: *Proc. IEEE Intl. Conf. Cluster Computing*. 2011, pp. 542–546. DOI: 10.1109/CLUSTER.2011.65.