CHEMNITZ UNIVERSITY OF TECHNOLOGY

Department of Computer Science

Computer Architecture Group

Diploma Thesis

Evaluating and Improving the Performance of MPI-Allreduce on QLogic HTX/PCIe InfiniBand™ HCA

Nico Mittenzwey

Chemnitz, March 31, 2009

Supervisor: Professor Dr.-Ing. Wolfgang Rehm
Advisor: Dipl.-Inf. Frank Mietke

Task of the Thesis

The task of this diploma thesis is the evaluation and improvement of the collective all-reduce operation on the InfiniBand^{TM} QLogic InfiniPath QLE7140 Host Channel Adapter. Since a wide rang of parallel applications depend on this operation, an efficient implementation for the given hardware is of high importance. The QLogic InfiniPath QLE7140 Host Channel Adapter (HCA) will be analyzed and compared to the Mellanox InfiniHost III Lx HCA. Based on the results of the analysis, optimizations for all-reduce algorithms shall be proposed. Known all-reduce algorithms will be investigated theoretically, using well-known communication models and practically on clusters. Finally a new algorithm shall be proposed.

Theses

I A InfiniBand™ onload architecture can outperform an offload architecture on modern host hardware.

II The LogGP model can predict the behavior of all-reduce algorithms accurate enough for large messages on InfiniBand™ HCAs.

III There cannot exist a general non-adaptive all-reduce algorithm which is optimal in all possible scenarios.

IV OFED RDMA-CM offers an easy way to use multicast over InfiniBand™ and can be combined with PSM point to point messages.

V The Open MPI library's send and receive overhead for the PSM interface is very low.

VI Open MPI offers an easy way to implement new collective algorithms.

Contents

List of Figures iv

List of Tables vi

1 Introduction 1
- 1.1 Organization of this Document . . . 3
- 1.2 The MPI Standard . . . 3
  - 1.2.1 Point-To-Point Communication . . . 3
  - 1.2.2 Collective Operations . . . 4
  - 1.2.3 Existing MPI Implementations . . . 6
- 1.3 Open MPI . . . 6
  - 1.3.1 Architecture . . . 6
  - 1.3.2 Point-To-Point Communication Design . . . 8
  - 1.3.3 Collective Operations Framework . . . 9
  - 1.3.4 Properties and Behavior . . . 9
- 1.4 Infiniband . . . 10
  - 1.4.1 Topology and Components . . . 11
  - 1.4.2 Addressing of Endpoints . . . 11
  - 1.4.3 Point-to-Point Communication . . . 12
  - 1.4.4 Hardware Multicast . . . 15
  - 1.4.5 Verbs . . . 15
  - 1.4.6 Host Channel Adapter . . . 18
  - 1.4.7 PSM InfiniPath Messaging API . . . 20
- 1.5 Performance Models . . . 21
  - 1.5.1 Hockney Model . . . 21
  - 1.5.2 LogP . . . 21
  - 1.5.3 P-LogP . . . 22
  - 1.5.4 LogGP . . . 22
  - 1.5.5 LogfP . . . 24
  - 1.5.6 Lop . . . 24
  - 1.5.7 Conclusion . . . 25

2 Evaluation Benchmarks 26
- 2.1 Benchmark Environment . . . 26

2.1.1 Jack . . . 26
2.1.2 Darwin . . . 26
2.1.3 CHiC . . . 27
2.2 Uni-Directional Latency and Bandwidth Benchmarks . . . 27
2.2.1 Bandwidth . . . 27
2.2.2 Latency . . . 29
2.2.3 Conclusion . . . 29
2.3 Bi-directional Bandwidth . . . 29
2.4 Multiple Process Bandwidth . . . 30
2.5 Overlapping Communication and Computation . . . 34
2.5.1 Overlapping Impact on Bandwidth . . . 34
2.5.2 Overlapping Impact on Latency . . . 36
2.5.3 Summary . . . 36
2.6 Round Trip Time, Send and Receive Overhead . . . 37
2.6.1 P-LogP Model Benchmarks . . . 37
2.6.2 LogGP Model Benchmarks . . . 39
2.6.3 LoP Model Benchmarks . . . 40
2.7 Conclusion . . . 44

3 MPI_Allreduce() Algorithms 45
3.1 Linear . . . 46
3.2 Recursive Doubling . . . 47
3.2.1 Power-of-Two Case . . . 47
3.2.2 Non-Power-of-Two Case . . . 48
3.3 Rabenseifner . . . 49
3.3.1 Preparation . . . 49
3.3.2 Reduce-Scatter: Recursive Vector Halving and Distance Doubling 50
3.3.3 All-Gather: Recursive Vector Doubling and Distance Halving . 50
3.3.4 Finalization . . . 51
3.3.5 Summary . . . 51
3.3.6 Optimizations . . . 52
3.4 Binary Blocks . . . 53
3.4.1 Preparation . . . 53
3.4.2 Reduce-Scatter: Recursive Vector Halving and Distance Doubling 53
3.4.3 All-Gather: Recursive Vector Doubling and Distance Halving . 53
3.4.4 Summary . . . 53
3.5 Bandwidth Optimal Algorithm . . . 56
3.5.1 Finding a contention-free logical ring . . . 56
3.5.2 Reduce-Scatter . . . 57
3.5.3 All-Gather . . . 58
3.5.4 Summary . . . 58

3.6 Hierarchical and Heterogeneous Algorithms 59
3.7 Verification of the LogGP Prediction 59
3.8 Conclusion 62

4 Reduce Followed by a Practical Constant-Time Broadcast 65
4.1 Practical Constant-Time Broadcast 65
4.2 Prototype Implementation 67
4.2.1 Complications 68
4.3 Testing 68

5 Conclusion and Future Work 69

Acronyms 70

Bibliography 73

A Benchmarks 79
A.1 Implementation Notes for the Benchmarks 79

List of Figures

1.1 Top500 System Architectures over Time (image taken from `www.top500.org`) . . . 1
1.2 Top500 System Interconnect Family over Time (image taken from `www.top500.org`) . . . 2
1.3 MPI_Allreduce() Data Distribution . . . 6
1.4 Open MPI Modular Component Architecture . . . 7
1.5 Open MPI Layered Point-To-Point Communication Architecture . . . 8
1.6 InfiniBand™ Fabric Subnet (taken from [1]) . . . 11
1.7 InfiniBand™ Architecture Layers (taken from [1]) . . . 12
1.8 InfiniBand™ Queue Pairs and Completion Queue (taken from [1]) . . . 13
1.9 InfiniBand™ CA with Two Applications and Three QPs (taken from [1]) . . . 13
1.10 QLogic InfiniPath QLE7140 HCA (taken from [2]) . . . 18
1.11 Mellanox InfiniHost III Lx HCA (taken from [3]) . . . 19

2.1 OSU 3.1 MPI Bandwidth Benchmark Comparison between QLogic InfiniPath QLE7140 and Mellanox InfiniHost III Lx HCAs . . . 28
2.2 OSU 3.1 MPI Latency Benchmark Comparison between QLogic InfiniPath QLE7140 and Mellanox InfiniHost III Lx HCAs . . . 30
2.3 OSU 3.1 MPI Bi- and Uni-Directional Bandwidth Comparison . . . 31
2.4 Effective Bandwidth Loss in Bi-Directional Case for One QLogic InfiniPath QLE7140 HCA . . . 31
2.5 OSU 3.1 MPI Multiple Process Bandwidth Benchmark of the QLogic InfiniPath QLE7140 HCA . . . 32
2.6 OSU 3.1 MPI Multiple Process Bandwidth Benchmark of the Mellanox InfiniHost III Lx HCA . . . 33
2.7 QLogic InfiniPath QLE7140 : OSU 3.1 MPI Overlapping 1 Communication Thread and 3 Computation Threads Bandwidth Benchmark . 34
2.8 Mellanox InfiniHost III Lx : OSU 3.1 MPI Overlapping 1 Communication Thread and 3 Computation Threads Bandwidth Benchmark . . . 35
2.9 OSU 3.1 MPI Overlapping 1 Communication Thread and 4 Computation Threads Bandwidth Benchmark . . . 36

2.10 OSU 3.1 MPI Overlapping 1 Communication Thread and 4 Computation Threads Latency Benchmark 37
2.11 Comparison of OSU Latency Benchmark and P-LogP Benchmarks LogGP Prediction for QLogic InfiniPath QLE7140 HCAs 38
2.12 Comparison of OSU Latency Benchmark and P-LogP Benchmarks LogGP Prediction for Voltaire HCA 410Ex HCAs 38
2.13 Comparison of OSU Latency Benchmark and LogGP Predictions for QLogic InfiniPath QLE7140 HCAs 40
2.14 Comparison of OSU Latency Benchmark and LogGP Prediction for Voltaire HCA 410Ex HCAs 41
2.15 Average Send Post Overhead for a Message of 1 byte 42
2.16 Average Receive Post Overhead for a Message of 1 byte 42
2.17 Average Send Post Overhead for 10 Chained Posts 43
2.18 Average Receive Post Overhead for 10 Chained Posts 43
2.19 Average RTT per Process for QLogic InfiniPath QLE7140 HCAs . . . 44

3.1 Recursive Doubling (image taken from [4]) 48
3.2 Example for Rabenseifners Algorithm (image taken from [5]) 49
3.3 Example for Binary Blocks Algorithm (image taken from [4]) 54
3.4 Time needed for the Binary Blocks Example in 3.3 55
3.5 Contention-free Ring of Switches (image taken from [6]) 57
3.6 Example for a Reduce-Scatter Operation Implemented as Ring 57
3.7 Recursive Doubling LogGP Prediction and IMB Benchmark for 8 Processes on 8 Nodes using QLogic InfiniPath QLE7140 HCAs 60
3.8 Recursive Doubling LogGP Prediction and IMB Benchmark for 8 Processes on 8 Nodes using Voltaire HCA 410Ex HCAs 61
3.9 Recursive Doubling LogGP Prediction and IMB Benchmark for a Message Size of 4096 bytes using Voltaire HCAs 62
3.10 LogGP Prediction for Message Sizes between 1 byte and 1KiB (a) and 1KiB and 1MiB (b) 63

4.1 Steps of the Broadcast: 1. Multicast 2. Fragmented Chain Broadcast . 66

List of Tables

1.1 InfiniBand™ Types of Service 15

2.1 Overview over N/2 for Different Numbers of Sending Processes 33
2.2 P-LogP Parameter Results for the P-LogP Model Benchmark over Open MPI for a Message Size of 1 byte 39
2.3 LogGP Parameter Results for the LogGP Model Netgauge Benchmark over Open MPI 40

3.1 LogGP Time of All-reduce Algorithms (* power-of-two case) 64

1 Introduction

In recent years, clusters of workstations using commodity networks as interconnect have become a preferred platform for high performance computing (HPC). In combination with the open source operating system Linux and the Message Passing Interface (MPI) standard clusters of workstations have emerged as a powerful solution for solving scientific and business problems. Not only the lower costs but also the familiarity to a wide range of computer scientists and technicians, robustness, the accessibility of hard- and software designs and open standards make clusters based on commodity hardware the preferred choice for many scientific and business computing solutions. This trend is reflected by the number of cluster systems in the Top 500[1] computing list (see Figure 1.1). Beginning with a small number in 1998, cluster systems now dominate the list of the 500 fastest public known supercomputers of the world. The downside is that one workstation can only host a limited

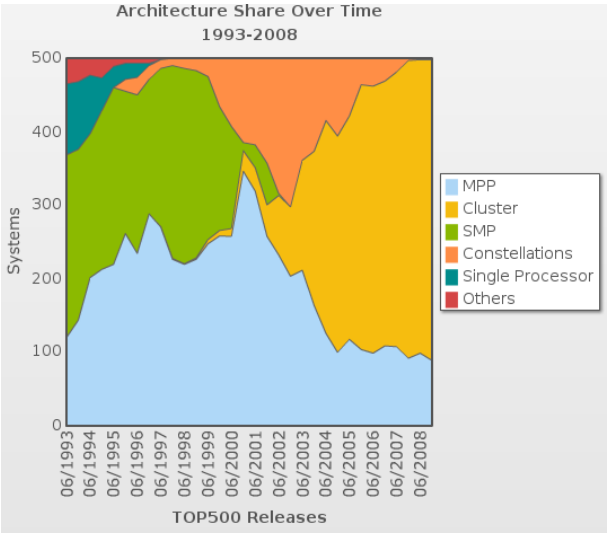

Figure 1.1: Top500 System Architectures over Time (image taken from `www.top500.org`)

number of CPUs and memory. This is compensated by connecting hundreds or even thousands of them through an interconnect. Since most HPC applications need to

[1] `http://top500.org/`

communicate between their processes to synchronize and share data, the network is a critical component in terms of application speed, scalability and cluster efficiency. Thus, the interconnect should provide a high bandwidth and low latency to match the interprocess communication needs of the applications. In the beginning clusters

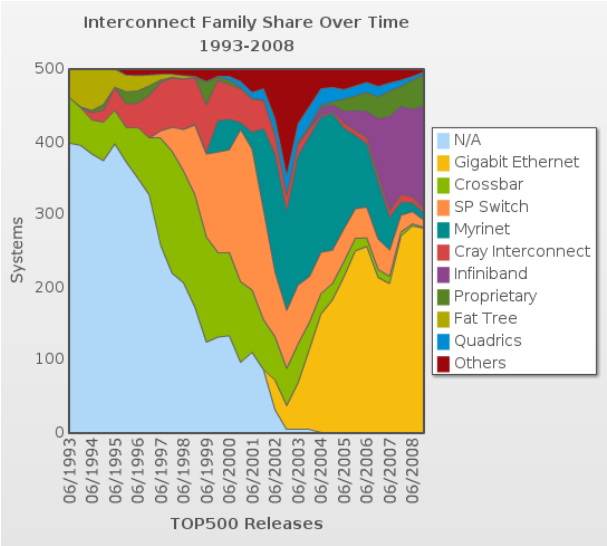

Figure 1.2: Top500 System Interconnect Family over Time (image taken from `www.top500.org`)

were most likely connected through a standard 100Mbit Ethernet network which was eventually replaced by 1Gbit and will certainly be replaced by 10Gbit in the near future. Being the most used and therefore cheapest network technology, Ethernet was the first choice for cost efficient clusters. In recent years InfiniBand™ gained a raising market share (as of November 2008 28% of the Top500 supercomputer use InfiniBand™ cp. Figure 1.2) by providing a much higher performance in terms of latency and bandwidth in comparison to Ethernet while simultaneously usually being cheaper than any other high performance interconnect technology like Myrinet or Quadrics.

Today, interprocess communication of an application is most likely handled by an MPI library which provides an Application Programming Interface (API) to hide the complexity of communicating over a network and to make the application itself more hardware independent and therefore portable. One of those functions is the all-reduce operation. It is used to calculate a result from input values of a number of processes so that after the operation has finished all processes know the result. For example all processes of an application could search for the minimum of a function for different input values. After each process computed the result for its input values the

minimum is determined by suppling these results to the all-reduce operation which returns the minimum of all results. A five year survey of a productive supercomputer system at the University of Stuttgart revealed that over 40% of the execution time of MPI routines was spent in the functions MPI_Allreduce and MPI_Reduce [7]. Thus, the MPI_Allreduce() function can have a heavy impact on the performance of an application and should therefore be as fast as possible under the given environment. The goal of this thesis is to find an optimal all-reduce algorithm for the QLogic InfiniPath QLE7140 InfiniBand™ host channel adapter .

1.1 Organization of this Document

The first part of this thesis gives an introduction and theoretical background to the MPI standard, the all-reduce operation, InfiniBand™ and performance models. In the second part, the QLogic InfiniPath QLE7140 InfiniBand™ adapter and its properties are evaluated through a series of microbenchmarks. Based on these benchmarks and the theoretical background from Chapter 1, several all-reduce algorithms are then presented end evaluated in Chapter 3. In Chapter 4 a new all-reduce algorithm is proposed and in Chapter 5 the results of this work are summarized.

1.2 The MPI Standard

MPI stands for Message Passing Interface. It specifies a portable standard for the communication between distinct processes by messages. In 1994 the MPI-Forum released the first version: MPI-1.0 [8], defining the syntax and semantics of all elemental message passing functions like point-to-point (P2P) communication or collective operations. Besides clarifying and correcting errors in MPI-1 resulting in MPI-1.3 in 2008 the MPI Forum released MPI-2.0 [9] in 1997. MPI-2.0 extended the standard by defining process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. MPI-2.1 was released in September 2008 containing clarifications and corrections to MPI-2.0. At the moment, the MPI-Forum is discussing additions[2] to the MPI-2.2 standards which will eventually lead to the new MPI-3.0 standard.

1.2.1 Point-To-Point Communication

The basic P2P communication operations are send and receive. The processes are identified by a unique ID called *rank* and different messages are distinguished by a *tag*. The MPI standard defines blocking and non-blocking P2P operations. The blocking MPI_Send and MPI_Recv functions return, when the data transfer finished

[2] Up to date information can be found on `http://meetings.mpi-forum.org/`.

successfully. Thus, the application calling either of these functions has to wait while the data is transferred over the network. This may take a considerable amount of time which could have been used for computation by the application. For this reason the MPI standard also defines the non-blocking MPI_Isend and MPI_Irecv operations. These functions return immediately after initiating the send or receive operation. This allows the application to compute while waiting for the transfer operation to finish. Before the send buffer may be used again and the receive buffer is read the application must ensure that the operations are finished. This is accomplished by querying the MPI library for the status of the operation with special calls like MPI_Wait.

1.2.2 Collective Operations

Message passing operations in which all processes of a group participate are called collective operations. They must be called by all processes of one group with matching arguments. The group of processes is defined by a unique data type called *communicator* which also provides a context for the operation. Generally, collective operations can be in one of the following classes:

All-To-All All processes contribute to and receive the result of the operation. Examples: MPI_Alltoall, MPI_Allgather, MPI_Allreduce

All-To-One All processes contribute to the result of the operation but only one receives it. Examples: MPI_Gather, MPI_Reduce

One-To-All One process sends data to all other processes. Examples: MPI_Bcast, MPI_Scatter

Others Operations that do not fit into one of the above. Example: MPI_Barrier

At the moment, the MPI standard specifies only blocking collection operations. In recent years Höfler et al. proposed non-blocking collective operations to rise application level performance [10].

The MPI_Allreduce() Function

The MPI_Allreduce() function is one of the most used collective operation in scientific computing [11, 7]. A common task in parallel programming is the accumulation of values across all members of a defined group of processes. Therefore, every participating process provides data which is combined by an accumulation operation. The result is either returned to one or all processes. The MPI standard defines three reduction operations: MPI_Reduce(), MPI_Allreduce() and MPI_Reduce_scatter(). The MPI_Reduce() call returns the result to the buffer of a specified root process.

MPI_Reduce_scatter() combines the values and scatters the result over the participating processes. The MPI_Allreduce() operation is specified as follows:

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

sendbuf	IN	starting address of send buffer (choice)
recvbuf	OUT	starting address of receive buffer (choice)
count	IN	number of elements in send buffer (non-negative integer)
datatype	IN	data type of elements of send buffer (handle)
op	IN	operation (handle)
comm	IN	communicator (handle)

int MPI_Allreduce(void sendbuf, void* recvbuf, int count,*
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

It accumulates the elements of the data in *sendbuf* of each process in the communicator *comm* by applying the operation in *op* and returns the result in the *recvbuf* of each process. The parameters *comm, op, datatype* and *count* must be identical for each participating process. The *sendbuf* can be equal to the constant *MPI_IN_PLACE*. If this is the case the original data of each process is taken from *recvbuf* instead of *sendbuf* to minimize the amount of memory. Figure 1.3 shows the content of the send (IN) and receive (OUT) buffers before and after the MPI_Allreduce() call for three processes (P0 to P2). The accumulation operation *op* can be either user defined or one of a predefined list of commutative and associative operations:

MPI_MAX	-	maximum
MPI_MIN	-	minimum
MPI_SUM	-	sum
MPI_PROD	-	product
MPI_LAND	-	logical and
MPI_BAND	-	bit-wise and
MPI_LOR	-	logical or
MPI_BOR	-	bit-wise or
MPI_LXOR	-	logical xor
MPI_BXOR	-	bit-wise xor
MPI_MAXLOC	-	maximum value and location
MPI_MINLOC	-	minimum value and location

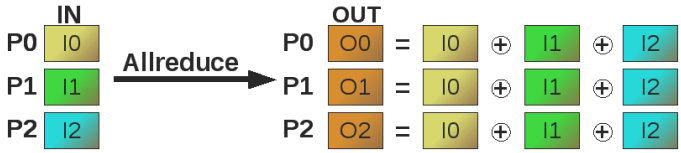

Figure 1.3: MPI_Allreduce() Data Distribution

1.2.3 Existing MPI Implementations

Besides MVAPICH2[3] the most commonly used open source MPI-2 implementation with support for InfiniBand™ is Open MPI[4]. MVAPICH2 is based on MPICH2[5] and developed mostly at the Ohio State University. Open MPI was intruduced in 2004 [12] as an all new MPI-2 implementation by a consortium of academic, research, and industry partners. It is influenced by code bases of LAM/MPI [13], LA-MPI [14] and FT-MPI [15]. The component-based design of Open MPI provides an easy way for developers and researchers to extend and modify the library. Thus, in this thesis, Open MPI is used to evaluate, develop and test collective MPI algorithms.

1.3 Open MPI

In 2003 developers of LAM/MPI, LA-MPI and FT-MPI decided to create a new flexible open source MPI-2 implementation from scratch to combine different strong points of the existing implementations and new concepts in one project. One of the main goals is to prevent "the forking problem" by allowing uncomplicated community and 3rd party involvement in the development process and providing a production quality research platform. Other goals are high user-friendliness, portable efficiency through a component architecture and the ability to safely act on environment variations like hardware defects or resource changes [12]. This section provides an overview of the MPI implementation Open MPI.

1.3.1 Architecture

Open MPI is split into three main parts, the Open MPI layer (OMPI), the Open Run-Time Environment (ORTE) and Open Portability Access Layer (OPAL). OMPI includes all MPI semantics like data types, functions and communicators and is heavily optimized to provide high performance message passing. ORTE acts as an interface to the back-end run-time system and provides services to OMPI as the identification

[3] http://mvapich.cse.ohio-state.edu/
[4] http://www.open-mpi.org/
[5] http://www-unix.mcs.anl.gov/mpi/mpich1/index.htm

and allocation of resources, process to node mappings, launch of processes, data conversion, error management, simple message passing and user notification. A more detailed description of the ORTE architecture and functions is provided in [16, 17]. OPAL provides portable helper functions like memory management, IO operations, processor and memory affinity, high resolution timers and different C macros and utility classes. For more details see [17]. Both OMPI and ORTE use one of the key features of Open MPI to be able to utilize different hardware: the Modular Component Architecture (MCA).

Modular Component Architecture

Open MPI uses a component architecture to allow the user to change the behavior and extend the MPI library at run time. This architecture also allows Open MPI to choose the fastest communication interface available at a given host without the need of recompiling or relinking. The MCA manages several independent subsystems called MCA frameworks for each major functionality of Open MPI.

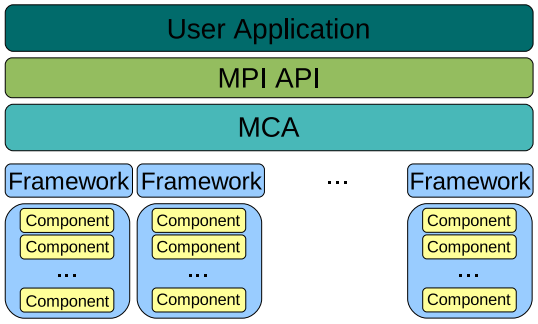

Figure 1.4: Open MPI Modular Component Architecture

Each framework has a targeted set of functionality such as P2P data transfer, collective operations or resource management for which it provides a well defined API. The implementation of a framework is called a component. Each framework can host several components to access different hardware or run different algorithms (see Figure 1.4). The framework is responsible for selecting, loading, using and unloading the components. Components are selected at run time based on parameters supplied by the user or automatically determined attributes like the fastest available interconnect or the number of processes in a communicator. Each selected component returns a module with pointers to framework specific functions and module specific data.

1.3.2 Point-To-Point Communication Design

The P2P communication in Open MPI is implemented in layers [18] utilizing the component architecture as shown in Figure 1.5. In 2005 Barret et al. showed that this approach introduces only a very small measurable overhead [19]. The topmost Point-to-point Messaging Layer (PML) provides all MPI P2P semantics like synchronous and asynchronous message passing needed by higher level MPI functions and applications. This message passing includes message buffering, matching and scheduling of transfers. Depending on the available hardware the PML uses either the OB1 or CM[6] component.

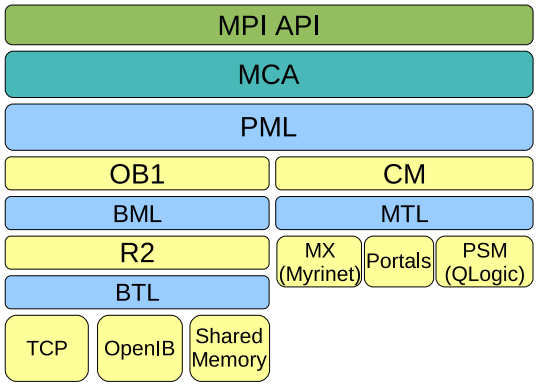

Figure 1.5: Open MPI Layered Point-To-Point Communication Architecture

OB1 Component

The OB1 component implements message matching and scheduling within the MPI layer and uses the Byte Transfer Layer (BTL) framework for message transfers. The Byte Management Layer (BML) is primarily used in initialization and BTL component selection and later bypassed for higher performances. The BTL framework includes components for several interconnects like InfiniBand™ (OpenIB[7]), Myrinet (GM), Ethernet (TCP) and shared memory (SM).

CM Component

The CM component is designed for interconnect APIs which are capable of supporting MPI communication protocols. CM only handles memory management for requests and buffer management for buffered sends. Anything else needed for MPI's P2P semantics is implemented in the components of the Matched Transport Layer (MTL)

[6] PML names do not have any official meaning
[7] using the OFED API see Section 1.4.5

framework which acts as interface between the CM component and the actual hardware. At the moment, the MTL framework supports Myricom MX, Cray Portals and Qlogics PSM library. In order to provide a high message passing performance, the MPI message matching logic and the transfer is left to the hardware libraries. This allows for completely asynchronous transfers without the need for progress threads in the MPI layer which control the data transfer.

1.3.3 Collective Operations Framework

The framework responsible for all collective operations in Open MPI is called COLL. This framework allows the implementation of new collective algorithms in components with little knowledge of the Open MPI internals. Collective routines can use MPI P2P functions, other collective components and direct hardware access. The component is selected on communicator creation and returns a module containing function pointers. These functions are called by top-level MPI collective functions like MPI_Allreduce() after validating the parameters given to the function. The communicator is always bound to the selected and the BASIC component if the selected component does not implement all collective operations.

At the creation of a new communicator (e.g. at start-up MPI_COMM_WORLD or by calling MPI_Comm_create) each components *query* function is called. This function must return a priority value (from 0 to 100) based on the properties of the processes in the communicator (e.g. the interconnection type). The component may allow the user to override the priority through a command line parameter or a special configuration file. The COLL component with the highest priority is then selected by the framework and its *initialization* method is called. This method prepares the components module for its operation (e.g. allocate memory, setting up network connections or benchmark and pre-compute communication patterns). When the initialization is finished, the module including pointers to the implemented functions and data types is returned. These functions are called whenever a collective operation is executed on the communicator. On destruction of the communicator (e.g. by calling MPI_Comm_free) the *finalization* method of the module is called. This function frees all occupied resources of the module.

1.3.4 Properties and Behavior

For interconnects which are not capable of supporting MPI communication protocols, Open MPI handles the message transfer through the BTL framework. Even though the components for different interconnects use different methods for the message transfer some communication patterns are the same. One of this pattern is the distinction in eager and rendezvous protocol.

Eager and Rendezvous Protocol

In order to transfer small messages as fast as possible, Open MPI uses the eager protocol. This protocol allows for the sending of messages to the peer without informing it in advance. All processes have a buffer for this kind of messages where incoming data is stored temporarily and later copied to its destination defined by its associated tag. For big messages, these buffers may be too small, or operating system and/or network limitations may not allow this kind of transfer. In this case, Open MPI switches to the rendezvous protocol at a certain message size. In this protocol, the sender needs to request and wait for an acknowledgment from the receiver before transferring any data. This allows the receiver to allocate appropriate resources prior to receiving any message. For the OpenIB component, the rendezvous protocol is used for messages bigger than 12288 bytes. This size can be changed by the user through the MCA command line parameter *btl_openib_eager_limit*. A detailed description and evaluation of both protocols can be found in [20].

RDMA Transfers

The OpenIB component uses RDMA to transfer small messages eagerly. However, RDMA connections are valuable resources and thus Open MPI only establishes a RDMA connection to peers which have sent more than 16 short messages. This number can also be changed by the user through the parameter *btl_openib_eager_rdma_threshold*.

Both parameters must be considered when benchmarking the performance of different HCAs since they may significantly influence the measurements.

1.4 Infiniband

The InfiniBand™ Architecture (IBA) was originally designed by the Infiniband Trade Association[8] to provide a high speed I/O interface for directly connecting CPUs and thus eliminating buses like PCI and networks like Ethernet. The aim of this industry standard is to provide low-latency and high bandwidth communication by using RDMA and user level communication which avoids the costs for entering kernel functions. The specification was first released in September 2000 as version 1.0. Over the time errors were corrected and new features were included resulting in the last official release of version 1.2.1 in January 2008 [1]. The specification is freely available on the Infiniband Trade Association homepage.

[8] `http://www.infinibandta.org/`

Today, InfiniBand™ is mostly used as interconnect network in high performance computing and storage area networks.

1.4.1 Topology and Components

The IBA defines a network for interconnecting endpoints being either complex processing nodes called Host Channel Adapters (HCA) or I/O nodes called Target Channel Adapters (TCA). The CAs are connected through a switched fabric consisting of switches and routers as shown in Figure 1.6. The routers connect different subnets which are managed by a subnet manager. CAs can belong to one or more subnets. Switches and CAs can be connected to one or more switches. Thus, if one connection fails or is saturated the subnet manager can assign new routes in the subnet for point-to-point connections.

In this thesis the term HCA is used for the InfiniBand™ connection device in a standard computer.

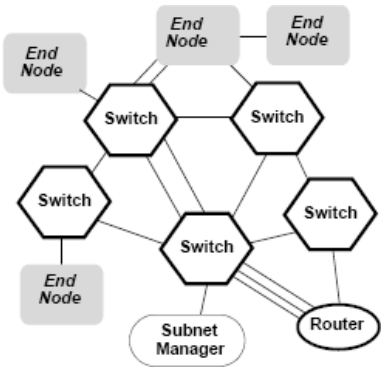

Figure 1.6: InfiniBand™ Fabric Subnet (taken from [1])

1.4.2 Addressing of Endpoints

An endpoint in the InfiniBand™ fabric can have several CAs which may have more than one physical port. Thus, on the startup of the CA each port gets a Local ID (LID) assigned by the subnet manager, which is unique in the local subnet and used by the switches to dispatch packets. Besides the Local ID each port also has a Global ID (GID) which can be used in the optional Global Route Header to address CAs in other subnets. This GID is only processed by routers.

1.4.3 Point-to-Point Communication

The InfiniBand™ communication stack consists of five layers as can be seen in Figure 1.7. In the fifth layer the management protocols and any number of other upper level protocols are placed. A queue based model is used in the interface between upper level protocols and the transport layer.

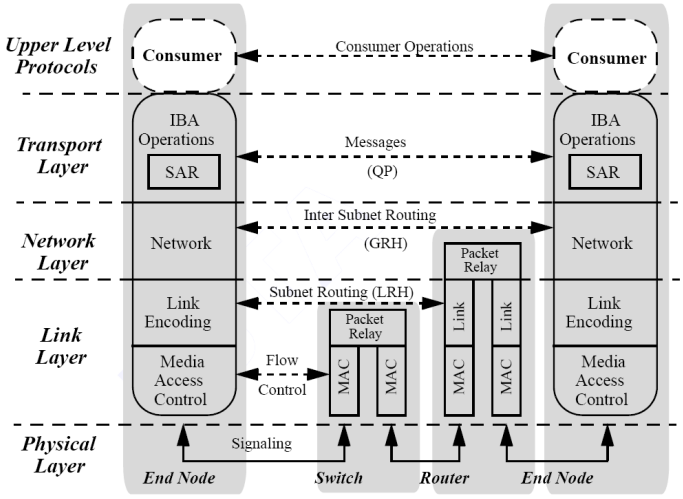

Figure 1.7: InfiniBand™ Architecture Layers (taken from [1])

Queues

The foundation of communication over InfiniBand™ for consumers (e.g. an application) are the work queues (see Figure 1.8). A consumer can post so called Work Requests (WR) to that queues, which will be executed by the hardware. The queue for receive operations and the one for send operations are always created together and thus are referred to as Queue Pair (QP). Submitting a WR to one of the queues creates a Work Queue Element (WQE) in that queue. After the Channel Adapter processed the WQR a Completion Queue Element (CQE) is posted to the Completion Queue (CQ). Consumers can create several independent QPs which can be associated to one or more created CQs. In order to identify Queue Pairs, each Queue Pair has a Queue Pair Number (QPN). As the example in Figure 1.9 shows, more than one consumer per node and HCA is allowed and one consumer can use multiple distinct QPs.

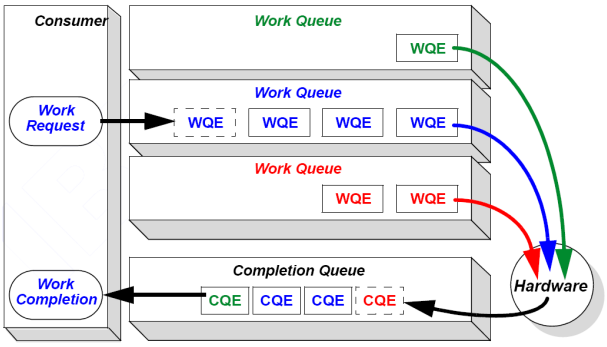

Figure 1.8: InfiniBand™ Queue Pairs and Completion Queue (taken from [1])

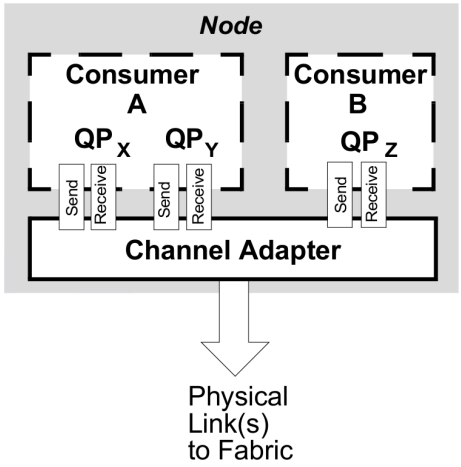

Figure 1.9: InfiniBand™ CA with Two Applications and Three QPs (taken from [1])

Operations

The receive queue supports only the *RECEIVE* operation which requires a local address as parameter. This address specifies the location of the receive buffer in the local memory were the incoming data from remote SEND operations is stored. After the data is successfully stored in the receive buffer a CQE is posted to the CQ.

The send queue supports three classes of operations: *SEND*, *Remote Direct Memory Access (RDMA)* and *Memory Binding*. The WR for a *SEND* operation must include a memory address where the data to be sent is stored. The data is then sent to the receiver which places it accordingly to its queued RECEIVE WQE. For *RDMA* operations the sender must have received a remote memory address and the associated R-key from the receiver. The R-key acts as security token and is issued by the remote CA. When the remote CA receives an RDMA request with an address the associated R-key is verified before issuing the operation. Thus, an RDMA operation does not need a pre-posted receive work request on the receiver side. InfiniBand™ supports three types of RDMA operations:

RDMA-WRITE with a local address, remote address and R-key as parameters writes the data of the local buffer to the remote address.

RMDA-READ with a local address, remote address and R-key as parameters reads data from the remote memory address and stores it in the local buffer

ATOMIC reads and updates the data at the remote address atomically and stores the read data in a local buffer

The *Memory Binding* operation registers a specified range of memory for RDMA access and returns the corresponding R-key. The R-key can then be transferred to remote nodes to allow them to access this memory region.

Types of Service

Each Queue Pair can be configured for a certain type of service which defines how the local and remote QPs interact. Both QPs must be configured for the same type of service. Table 1.1 gives an overview. The type of service is based on three attributes.

Connection Oriented QPs are connected with exactly one other QP. WRs posted to that QP are being send to the established destination QP. QPs that are not Connection Oriented are *datagram* QPs which can send and receive to/from any other datagram QP on any node.

QPs in the *Acknowledged* type of service return responses when they receive messages and therefore the data delivery is reliable. The response messages can be positive (ACK), negative (NAK) or contain response data. The Unacknowledged type does only guarantee that the delivered data is not corrupted.

Service Type	Connection Oriented	Acknowledged	Transport Type
Reliable Connection (RC)	yes	yes	IBA
Unreliable Connection (UC)	yes	no	IBA
Reliable Datagram (RD)	no	yes	IBA
Unreliable Datagram (UD)	no	no	IBA
RAW Datagram (RAW)	no	no	RAW

Table 1.1: InfiniBand™ Types of Service

The *RAW* transport mode allows the consumer to use the Channel Adapter as a data link engine for raw packets between nodes to support legacy protocols and networks.

1.4.4 Hardware Multicast

The IBA also describes an unreliable hardware multicast in both IBA and RAW transport mode. This feature however is optional and thus may not be supported by all InfiniBand™ HCAs, switches and routers. Since the hardware multicast can be initiated by only one send operation, the overhead at the sender is very low in comparison to the overhead created by separately sending the same data to hundreds of nodes. Another benefit of using multicast is the reduction of bandwidth usage and contention. The multicast packets are duplicated by switches only when necessary, which eliminates multiple identical packets traveling through the same physical link. However, the unreliability is a big drawback and forces the consumer to verify the receipt of the data at the receiver if reliability is necessary.

1.4.5 Verbs

The IBA defines a set of semantics called Verbs for the consumer to interact with the host channel adapter. The Verbs are not an API but an abstract definition of the IBA functionality which allows operating system vendors to design their own API for user programs. This lead to the implementation of different verbs APIs by several HCA developers (e.g. the Mellanox verbs API) which reduced the portability of user programs greatly. Thus, the OpenIB Alliance was founded in June 2004 by several hardware developers to develop a complete standardized InfiniBand™ software stack including a common verbs API. In 2006 the name was changed to OpenFabrics Alliance[9] and the aim was shifted to develop a unified, cross-platform, transport-independent open source software stack for RDMA based interconnects called the OpenFabrics Enterprise Distribution (OFED). The following list shows all

[9] `http://www.openfabrics.org/`

(simplified) function needed in the appropriate order to establish a connection and send a message over InfiniBand™ with the OFED v1.3 API:

1. **ib_dev=ibv_get_device_list()** - get a list of HCAs
2. **context=ibv_open_device(ib_dev)** - create context for first HCA
3. **prot_domain = ibv_alloc_pd(context)** - create/pin down protection domain for buffer memory
4. **compl_q = ibv_create_cq(context)** - create the CQ
5. **ibv_reg_mr(prot_domain,address,buffer_size)** - register memory buffers for data to be send or received
6. **q_pair = ibv_create_qp(prot_domain,compl_q)** - create a QP and initialize to state Reset (RST)
7. **ibv_modify_qp(q_pair,RST->INIT)** - modifies QP state from RST to Init
8. **ibv_modify_qp(q_pair,rLID,rQPN,INIT->RTR)** - modifies QP state from Init to Ready to Receive (RTR) from remote LID and QPN
9. **ibv_modify_qp(q_pair,RTR->RTS)** - modifies QP state from RTR to Ready to Send (RTS)
10. **ibv_post_recv(q_pair,wr)** - post receive request with WR containing address of receive buffer
11. **ibv_post_send(q_pair,wr)** - post send request with WR containing address of send buffer
12. **ibv_poll_cq(compl_q)** - polls CQ for events
13. **ibv_dereg_mr(address)** - deregister buffers
14. **ibv_destroy_qp(q_pair)** - destroy QP
15. **ibv_destroy_cq(compl_q)** - destroy CQ
16. **ibv_dealloc_pd(prot_domain)** - destory PD
17. **ibv_close_device(ib_dev)** - close HCA

In order to ease the programming of RDMA capable devices, the OpenFabrics Alliance introduced the RDMA Connection Manager API (RDMA-CM). Similar to the programming of Ethernet interfaces, RDMA-CM supports sockets with servers and clients but uses QPs for the message transfer. To be able to use the RDMA-CM functions, the ipoib (allows to run traditional internet protocol over InfiniBand™) and rdma_cm module must be loaded into the Linux kernel. RDMA-CM simplifies the usage of multicast over the InfiniBand™ fabric as well (for a native InfiniBand™ implementation see [21]). The following list shows all needed functions to join a multicast group and send and receive multicast messages. Functions which are called by the server only are highlighted in red and those called only by the clients are highlighted in green.

1. **rdma_channel=rdma_create_event_channel()** - create a new rdma_channel
2. **rdma_create_id(rdma_channel,rdma_id,RDMA_PS_UDP** - create rdma_id for UDP transfers and associate it with rdma_channel
3. **rdma_bind_addr(rdma_id,src_addr)** - bind rdma_id to local RDMA device(s) specified by src_addr
4. **rdma_resolve_addr(rdma_id,src_addr,mcast_addr)** - map local RDMA device(s) to mcast_addr ("0.0.0.0" at the moment)
5. **rdma_get_cm_event(rdma_channel, rcm_event)** - receive resolve event
6. **rdma_ack_cm_event(rcm_event)** - acknowledge resolve event
7. **rdma_join_multicast(rdma_id,mcast_addr)** - join multicast group with mcast_addr being "0.0.0.0" to get a free multicast group IP[10]
8. **rdma_get_cm_event(rdma_channel, rcm_event)** - receive multicast join event
9. **MPI_Bcast(rdma_mcast_ip)** - Broadcast multicast group IP received from the event to all clients
10. **rdma_join_multicast(rdma_id,rdma_mcast_ip)** - join multicast group
11. **prot_domain = ibv_alloc_pd(rdma_id->verbs)** - create/pin down protection domain for buffer memory
12. **cq = ibv_create_cq(rdma_id->verbs)** - create the CQ
13. **rdma_create_qp(rdma_id,prot_domain)** - create a QP initialized to state Ready to Send (RTS)
14. **ibv_reg_mr(prot_domain,address,buffer_size)** - register memory buffers for data to be send or received
15. **ibv_create_ah(qp_ctx->pd, rcm_event->param.ud.ah_attr)** - create Address Handel (AH) for send operations
16. **rdma_ack_cm_event(rcm_event)** - acknowledge multicast group join event
17. **ibv_post_recv(rdma_id->qp,wr)** - post receive request with WR containing address of receive buffer
18. **ibv_post_send(rdma_id->qp,wr)** - post send request with WR containing address of send buffer
19. **ibv_poll_cq(rdma_id->qp)** - polls CQ for events
20. **rdma_leave_multicast(rdma_id,rdma_mcast_ip)** - leave the multicast group
21. **ibv_destroy_ah(ah)** - deregister AH
22. **rdma_destroy_qp((rdma_id)** - destroy QP
23. **rdma_destroy_qp((rdma_cq)** - destroy CQ
24. **ibv_dereg_mr(address)** - deregister buffers
25. **ibv_dealloc_pd(prot_domain)** - destory PD

[10] http://lists.openfabrics.org/pipermail/general/2007-January/032303.html

1.4.6 Host Channel Adapter

The aim of this thesis is to optimize the all-reduce operation for the QLogic InfiniPath QLE7140 HCA. The main difference of this HCA to others is the onload architecture which in contrast to the offload architecture proposed in the IBA does not have an ASIC (Application Specific Integrated Circuit) which is responsible for processing work requests. The QLogic InfiniPath QLE7140 HCA uses the host CPU for these tasks and only provides data integrity checks. While this approach introduces more overhead in the CPU, the work requests and their data can be processed more quickly by modern CPUs and thus should lead to a smaller latency. In order to compare the performance of the onload architecture with an offload architecture the Mellanox InfiniHost III Lx HCA is taken as reference in the evaluations. Both HCAs shall be introduced in this section.

QLogic InfiniPath QLE7140 HCA

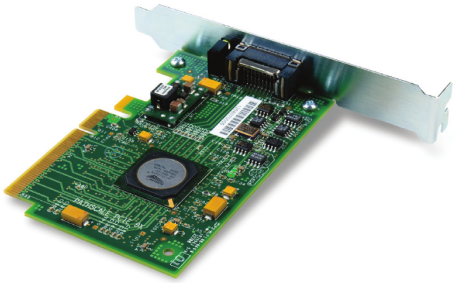

Figure 1.10: QLogic InfiniPath QLE7140 HCA (taken from [2])

The QLogic InfiniPath QLE7140 (see Figure 1.10) is a single data rate (SDR) InfiniBand™ to PCI Express x8 Host Channel Adapter [2]. It is able to transfer up to 10Gbit/s over its port, includes a Serializer/Deserializer (SERDES) and has a configurable MTU with a maximal size of 4096 bytes. It is compatible to standard IBA 1.2 compliant fabrics and cables and is able to communicate with HCAs from different hardware suppliers like Mellanox, Cisco, SilverStorm and Voltaire through the OFED software stack. This HCA does not support the sending of inline messages.

The used onload architecture forces the host CPU to prepare all packages with appropriate headers before placing them in the memory for the HCA to process and send over the fabric. The host CPU is also responsible for error handling. This introduces more overhead to the host CPU which reduces the time for overlapping communication and computation. On the other hand this leads to a reduced latency because of the faster processing capabilities of modern CPUs. The HCA only provides data integrity checks.

There was another version of this HCA (QHT7140) which used the HTX interface to communicate with the host processor and host memory over the HyperTransport link. This version is no longer available from QLogic.

Mellanox InfiniHost III Lx HCA

Figure 1.11: Mellanox InfiniHost III Lx HCA (taken from [3])

The Mellanox InfiniHost III Lx MHES18-X HCA [3] (see Figure 1.11) is also a SDR InfiniBand™ to PCI Express x8 Host Channel Adapter supporting up to 10Gbit/s. It includes a SERDES, has a configurable MTU with a maximal size of 2048 bytes is compatible to IBA version 1.2 and supports the inline sending of 28 bytes. This HCA uses the mem-free feature of the Mellanox ASIC. This technology allows hardware developers to use host memory to save connection context information instead of memory directly on the HCA. This allows for a cheaper HCA but can also lead to a slower performance if the context is not in the HCAs cache [22].

The HCA's ASIC supports up to 16 million QPs and CQs and handles the creation of InfiniBand™ packages, RDMA access and memory protection along with address translation. This frees the host CPU from the communication overhead.

Comparison of both HCAs

Gilad Shainer of Mellanox compared both HCAs under different conditions in [23] and [24] He found that the Mellanox InfiniHost III Lx HCAs outperform the QLogic InfiniPath QLE7140 HCAs. However, he did not state if he used the OFED or the PSM (see Section 1.4.7) API for the QLogic InfiniPath QLE7140 HCAs which greatly influences their performance as Section 2.2.1 and 2.2.2 will show. In [25] and [26] Greg Lindahl of QLogic compared both HCAs. He found that the QLogic InfiniPath QLE7140 HCAs outperform the Mellanox InfiniHost III Lx HCAs under the given conditions. However, the compared clusters were equipped not only with different HCAs but also with different CPUs and other hardware. In [27] QLogic compares the application level performance of the SDR QLogic InfiniPath QLE7140 HCAs to different Mellanox DDR HCAs on similar hardware. In Chapter 2 the results of several microbenchmarks of both HCAs on identical hardware will be compared.

1.4.7 PSM InfiniPath Messaging API

Besides the OFED 1.3 API a consumer can use another API to communicate between hosts equipped with a QLogic InfiniPath QLE7140 HCA: the proprietary PSM InfiniPath Messaging API developed by QLogic. This low-level user-level interface is only available for the InfiniPath family of QLogic HCAs. It implements an intra-node shared memory (SM) and inter-node communication protocol, which are completely transparent to the consumer. Thus, the consumer does not need to know if another process runs locally or on a remote machine to send a message over the PSM interface. Current implementations of the PSM interface (version 2.2.1) support only single threaded applications.

After initializing endpoints the Matched Queue (MQ) interface can be used to send and receive messages. The MQ interface semantics are consistent with those defined by the MPI 1.2 standard for messages passing between two processes. Thus, incoming messages are stored according to their tags to preposted receive buffers. While the "PSM Programmer's Manual" [28] is only available from QLogic under an NDA, the development package can be downloaded from the QLogic homepage[11]. This package includes an example and header files, which are well documented.

The following list shows all (simplified) functions needed in the appropriate order to create all endpoints and send a message to another endpoint using the PSM API and its Matched Queues interface:

1. **psm_init** - initialize the PSM library
2. **psm_uuid_generate(uuid)** - generate unique global job ID
3. **OOB[12] broadcast of the UUID to every process**
4. **psm_ep_open(uuid, ep, myepid)** - open local endpoint returns ep and myepid
5. **psm_mq_init(ep,mq)** - initialize local MQ interface
6. **OOB exchange of epids into rm_eps**
7. **psm_ep_connect(ep,rm_eps,epaddrs)** - connects the local endpoint to endpoints in rm_eps and returns epaddrs
8. **psm_mq_irecv(mq,TAG,in,size,reqr)** - posts non-blocking receive request for message with TAG
9. **psm_mq_send(mq,epaddrs[i],TAG,out,size,reqs)** - send content of out buffer with TAG to epaddrs[i]
10. **psm_mq_wait(reqr)** - wait for receive to finish
11. **psm_mq_finalize(mq)** - close MQ
12. **psm_ep_close(ep)** - close EP
13. **psm_finalize** - close PSM

[11] `http://www.qlogic.com/`
[12] Out Of Band

1.5 Performance Models

In order to analyze, optimize and predict the run-time of collective algorithms, different parallel communication and computation models have been proposed. Good models use the smallest possible number of parameters but also model the complexity of the analyzed system as exact as possible. The performance of collective algorithms depends on several parameters like message size, number of processes, network topology and characteristics, application communication patterns, processor speed, operating system noise[29] and many more. Since collective operations can include computation, both communication and computation aspects have to be considered.

In this section different well known models which can predict the behavior of collective algorithms are introduced.

1.5.1 Hockney Model

The Hockney model [30] was proposed by R. Hockney in 1994 and is used in a slightly extended manner by Rabenseifner et al. [5, 4, 31] to estimate the cost of different collective algorithms. The model defines the time needed to transfer a message between two processes as $\alpha + \beta m$, where α is the latency per message regardless of the size, β the transfer time per byte and m the number of bytes. For collectives which also do computation like the reduce operation γ represents the computation cost per byte to complete the operation locally on any process. Thus, the time needed to transfer a message is $\alpha + n\beta$. A linear reduce in which all processes send data to the root process simultaneously and the root node processes the data needs a time of $\alpha + (P-1)n\beta + Pn\gamma$.

1.5.2 LogP

The LogP model [32] was proposed by Culler et al. in 1993. It is designed for small point-to-point messages in a central switch based network of distributed memory processors and the fact that CPU speed - in terms of bandwidth and latency - is much faster than network speed. Network contention is not considered.

The LogP model abstracts the network through four parameters:

- L - the latency
- o - the communication overhead
- g - the gap between two successive messages
- P - the number of involved processes

The latency is hereby defined as the upper bound of the time needed to send a message from one NIC to another. The communication overhead is differentiated in the send overhead o_s and receive overhead o_r being the time the host CPU is occupied by sending or receiving a message. This is relevant for InfiniBand™ because of different send and receive overheads as is shown in Section 2.6 . Thus, the time needed to send a small message from one node to another according to LogP is $o_s + L + o_r$. The gap indirectly defines the communication bandwidth which is at most $\lfloor L/g \rfloor$. As the gap and send overhead overlap when multiple successive messages are being send only the maximum of both affects the transfer time. The time for a linear reduce in this model is $o_s + L + (P-1)\max\{o_r, g\}$.

The model was tested and verified by different studies [33, 34, 35]. Since it can only model small messages, LogP cannot be used for modelling all aspects of all-reduce algorithms.

1.5.3 P-LogP

The parameterized LogP [36] (P-LogP) model is an extension of LogP to model messages of different sizes. It was proposed by Kielman et al. in 2000. In this model the send and receiver overhead o_s,o_r and the gap g are parameterized by the message size m. So two successive message of size m need a time of $o_s(m) + \max\{o_s(m), g(m)\} + L + o_r(m)$ to be sent from one node to another. The time for a linear reduce in this model is $o_s(m) + L + (P-1)\max\{o_r(m), g(m)\}$.

The goal of Kielman et al. was to create a model which eases the measurement of parameters (see Section 1.5.4 for further details) but in the process complicated the usability and reduced the accuracy of the parameters.

1.5.4 LogGP

As an addition to the LogP Model Alexandrov et al. proposed the LogGP model in 1995 [37]. LogGP addresses the incorrect modeling of big messages in LogP by introducing the new parameter G. G is defined as time per byte for long messages. Thus, the time needed to send a message of m bytes from one node to another is $L + o_s + o_r + (m-1)G$. For successive messages from one node only the maximum of the send overhead o_s and the g overlap need to be considered, since both times overlap. As modern interconnects like InfiniBand™ reach a higher bandwidth for big messages than for small ones (see bandwidth benchmarks in Section 2.2.1) this model is more accurate than LogP for big messages. The time for a linear reduce in this model is $o_s + L + (P-1)(\max\{o_r, g\} + (m-1)G)$.

LogGP Measurement

While the development and evaluation of collective algorithms can be based on the pure model, the decision which algorithm to use for which network at which message size can only be done with the knowledge of the real parameter values of the model. Over the time different measurement methods for LogGP parameters have been proposed. The recent ones will be described in this section.

In 2000 Kielmann et al.[38] proposed a measurement for the P-LogP model which includes redefined LogP parameters. The gap $g(0)$ is determined from the transmission frequency of a large number of zero byte messages. The send overhead o_s can be measured directly by taking the time for the send operation to finish. This send is used to measure the round trip time by waiting until the second process answered. In order to determine the receive overhead o_r the first process sends a message to the second process, waits longer than the round trip time and then measures the time between the call and the return of the receive operation. Thus, o_r reflects the time needed to copy the message from a temporary buffer to the receive buffer. The gap g for a message size of 1 byte and the latency L is finally calculated using the determined parameters.

Bell et al. [39] proposed a benchmark for the LogGP model in 2003 redefining the latency to an end-to-end latency (EEL).The EEL corresponds to $RTT/2$ for small messages. They use a similar approach to measure g. Instead of posting sends one after one the sending process tries to keep non-blocking sends posted until all messages have been send. The average time for sending big messages is taken and cut by g to determine the per-byte gap G. o_s is determined by adding a delay between the start and the completion of a send request. This delay is increased until it effects the communication time, thus revealing o_s.

In 2007 Höfler et al. [40] introduced a new method which avoids network flooding and contention and measures all parameters instead of computing some of them. This avoids the propagation of first level errors. The benchmark introduces a parameterized round trip time (PRTT). The PRTT depends on the number n of successive send operations with one replay after the last message, the delay between each post d and a message size m. The $PRTT(1, 0, m)$ for sending a single message of size m is:

$$PRTT(1, 0, m) = 2(L + 2o + (m - 1)G)$$

The general $PRTT(n, d, m)$ is then:

$$PRTT(n, d, m) = PRTT(1, 0, m) + (n-1) * \max\{o+d, G_{all}\} \text{ with } G_{all} = g + (m-1)G$$

The overhead can now be calculated by determining $PRTT(1, 0, m)$ and $PRTT(n, d, m)$ for a $d > G_{all}$:

$$o = \frac{PRTT(n, d, m) - PRTT(1, 0, m)}{n - 1} - d$$

In order to determine g and G $PRTT(1,0,m)$ and $PRTT(n,d,m)$ are measured for a large number of different values of m resulting in many values for $G_{all} = g+(m-1)G$.

$$G_{all} = \frac{PRTT(n,d,m) - PRTT(1,0,m)}{n-1}$$

Fitting the function $f(m) = g + (m-1)G$ to the measured values for G_{all} finally reveals g and G. While two values would have be enough for the fitting, with the huge number of results anomalies like protocol switches can be detected.

1.5.5 LogfP

As several studies [41, 42] discovered, InfiniBand™ HCAs with offload architecture behave differently from the predictions of LogP models. The reason for this behavior are the HCA processors which show pipelining effects and parallel sending abilities [41]. Thus, Höfler et al. proposed a new model for small messages over InfiniBand™ : LogfP [43]. Instead of using a static overhead parameter like the LogP model, LogfP uses a function to describe the pipeline startup. For this function two parameters have to be measured. o_{min} is determined by measuring the overhead for a reasonable high number of processes and is divided through the number of processes. o_{max} is determined by measuring the overhead for sending one message to one other process. Then the function can be written as:

$$o(P) = o_{min} + \frac{o_{max}}{P}$$

The measured number of processes for which o is the smallest is represented by the parameter f. It is assumed that f multiple messages can be sent in parallel before g affects the transfer time. Thus, the time for sending a message to P nodes can be modeled as:

$$\forall(P \leq f) : T(P) = L + P * o_s(P) + o_r(1)$$
$$\forall(P > f) : T(P) = L + o_s(P) + o_r(1) + \max\{(P-1) * o_s(P), (P-f) * g\}$$

In the linear reduce operation each process sends one message to the root. Thus, this reduce algorithm cannot take advantage of the pipeline startup.

$$T(P) = L + o_s(1) + \max\{(P-1) * o_r(P-1), (P-1)g\}$$

1.5.6 Lop

The Lop model was proposed by Höfler and Rehm [35] as another model for small messages over InfiniBand™ with the limitation that one node can only send one message to another node. Although the number of messages is limited to one, the

number of receiving nodes is not. This model is valid for example for broadcast with small messages, barrier or round-based algorithms. The model introduced the parameter h for the time the processor of the HCA needs to process a message. Since this parameter cannot be measured by the host CPU, it is hidden together with g in the $L(P)$ parameter which is now dependent on the number of participating nodes. The model assumes that $o \ll L(P) \quad \forall P \in N$.

1.5.7 Conclusion

Several studies have shown that the different Log models are capable of predicting the performance of collective algorithms well. The only model which can evaluate the full spectrum of message sizes for all-reduce algorithms is LogGP. Thus, this model is used to analyze the all-reduce algorithms. However, none of the presented models can handle SMP nodes heterogeneous communication channels directly. For SMP nodes with multiple processes on one node, modifications to these models will be necessary.

2 Evaluation Benchmarks

This chapter will present the properties found by benchmarking the QLogic InfiniPath QLE7140 HCAs. The used benchmark environments will be presented in the beginning. After that each microbenchmark will be described shortly and its results will be compared to Mellanox InfiniHost III Lx HCAs and in some cases also to Voltaire HCA 410Ex HCAs.

2.1 Benchmark Environment

The performance of the QLogic InfiniPath QLE7140 HCAs was evaluated on two different clusters which will be described shortly below. For a better understanding of the differences of the on- and offload architecture the results are compared to Mellanox InfiniHost III Lx HCAs. The different benchmark suites and the used MPI implementation Open MPI were compiled with the GCC 4.1.2 using the supplied configure and Make files.

2.1.1 Jack

Jack is a local hardware evaluation cluster for different HPC hardware. The benchmarks were conducted on a subset of 8 nodes equipped with two dual core Intel Xeon 5130 2GHz CPUs with 4MiB level-2 cache, 2 GiB RAM, one QLogic InfiniPath QLE7140 HCA and one Mellanox InfiniHost III Lx HCA. A Voltaire ISR 9024 InfiniBand^{TM} switch connects only the HCAs of these 8 nodes with each other. The operating system is Scientific Linux 5.0 based on Red Hat Enterprise Linux 5 using the 2.6.18-8.1.3.el5 kernel with the official QLogic InfiniPath2.2.1-RHEL-x86_64 drivers. These also supply OFED 1.2 which is used for the Open MPI OpenIB interface for both HCA types. MPI Benchmarks were run on Open MPI 1.2.8.

2.1.2 Darwin

Darwin[1] is a 585 node HPC cluster located at the University of Cambridge, which entered production service in February 2007. The benchmarks were executed on up to 50 nodes, which are equipped with two 3 GHz dual core Intel Woodcrest processors, 8 GiB of RAM and connected by QLogic InfiniPath QLE7140 HCAs.

[1] `http://www.hpc.cam.ac.uk/`

2.1.3 CHiC

In order to evaluate the performance of all-reduce algorithms locally in large scale the "Chemnitzer High Performance Linux Cluster" (CHiC)[2] at the Chemnitz University of Technology was used. Each of the 530 processing nodes consists of 2 Dual AMD Opteron 2218 CPUs, at least 4 GiB of RAM, a Voltaire HCA 410Ex (using the Mellanox InfiniHost III Lx ASIC) and two Gigabit Broadcom Corporation NetXtreme BCM5704 Ethernet NICs. OFED 1.2 is used as OpenIB interface for Open MPI. Since each processing node is diskless the critical parts of the Linux operating system (Scientifc Linux 4.4) resides in the main memory. Less critical parts, home and project directories as well as applications are all imported over the parallel filesystem Lustre[3].

2.2 Uni-Directional Latency and Bandwidth Benchmarks

The Ohio State University (OSU) MPI Benchmark 3.1 was used to measure the latency and bandwidth of the HCAs on the Jack cluster. Since it is a synthetic benchmark, real world applications do not necessarily show a comparable behavior. Prior to all measurements, all OSU benchmarks send some warm-up messages to minimize the effects of connection initialization on the results.

2.2.1 Bandwidth

The bandwidth test is carried out by sending 64 messages of the same size with the asynchronous non-blocking point-to-point function MPI_Isend from the sender to the receiver. After the receiver got all messages by using the asynchronous MPI_Irecv function, it sends a 4 byte message back to sender. The bandwidth is then calculated based on the time the sender started to send the first message until it received the reply form the receiver and the number of bytes sent by the sender. This process is repeated several times to get an average value of the bandwidth. The time impact of the 4 bytes reply is included which decreases the measured bandwidth compared to the real bandwidth especially for small messages. On the other hand the usage of MPI_Isend utilizes the pipelining capabilities of InfiniBand™ HCAs which may not be comparable to real world applications but for example optimized collective MPI operations [41].

Figure 2.1 shows the bandwidth between two processes on distinct hosts. One can clearly see, that using the OpenIB interface instead of the PSM interface for the QLogic InfiniPath QLE7140 HCAs results in a massive decrease in bandwidth usage. Compared to the Mellanox InfiniHost III Lx HCAs the InfiniPath HCAs

[2] http://www.tu-chemnitz.de/chic/
[3] http://www.lustre.org/

bandwidth usage over PSM increases fast to its maximum of 956.33 MiB/s with 940MiB/s at a message size of 8KiB. The Mellanox InfiniHost III Lx HCAs bandwidth reaches more than 900MiB/s at a message size of 1MiB with 913MiB/s. The highest bandwidth of 968MiB/s is reached at a message size of 4MiB. For a better comparison of the bandwidth characteristics of different interconnects the N/2 benchmark can be applied. It defines the message size at which the examined interconnect reaches half of its maximum bandwidth. For the PSM interface the QLogic InfiniPath QLE7140 HCAs reach N/2 at a message size of 659 bytes and 4775 bytes when using OpenIB. The Mellanox InfiniHost III Lx HCAs pass N/2 at 4289 bytes.

The default message size at which the PSM libraries switch from the eager to the rendezvous protocol is 64000 bytes. This value can be changed by environment variables [28]. The decreased bandwidth at a message size of 64KiB is most likely caused by the switch between those two protocols. Changing the default switch message size to a different value results in a bandwidth drop at this value. Open MPIs OpenIB BTL switches between eager and rendezvous protocol at 12288 bytes. This is also most likely the reason for the slightly decreased slope of the bandwidth usage for the Mellanox InfiniHost III Lx HCA and decreased bandwidth utilization over OpenIB for the QLogic InfiniPath QLE7140 HCAs.

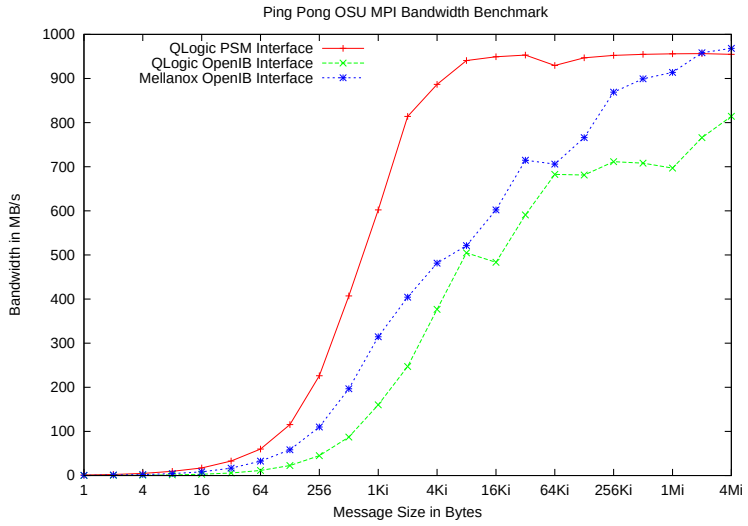

Figure 2.1: OSU 3.1 MPI Bandwidth Benchmark Comparison between QLogic InfiniPath QLE7140 and Mellanox InfiniHost III Lx HCAs

2.2.2 Latency

The OSU latency benchmark performs a ping-pong test using MPI_Send and MPI_Recv. The time the master needs to send a message to the worker process and receive a message of the same size from the worker process is taken and divided by two. Many iterations of this process are carried out for one message size and the average is returned as latency. (Note: In this benchmark the latency is defined as the time needed for a message of a certain size to be transferred from an application on one node to a remote application on another node. Thus, in this case the latency includes the send and receive overhead as opposed to benchmarks for certain performance models (see Section 2.6 below).

Figure 2.2 shows the latency for a message between two processes on distinct hosts. Like in the bandwidth benchmark the PSM interface outperforms the OpenIB interface for QLogic InfiniPath QLE7140 HCAs. In order to transfer a message with a payload of 1 byte the QLogic InfiniPath QLE7140 HCAs need 2.1μs while the Mellanox InfiniHost III Lx HCAs need 3.4μs. Up to a message size of 64KiB the latency of the QLogic InfiniPath QLE7140 HCAs using the PSM interface is 38% smaller than the latency of the Mellanox InfiniHost III Lx HCAs. For messages sizes over 64KiB the advantage drops to 10% for a message size of 1MiB. The bigger the messages the smaller is the impact of the processing time overhead of the Mellanox InfiniHost III Lx HCAs ASIC on the latency. Thus, the advantage of the faster onload CPU processing of the QLogic InfiniPath QLE7140 HCAs vanishes for bigger messages.

2.2.3 Conclusion

The bandwidth and latency benchmarks showed that the PSM interface should be preferred over OpenIB for the communication over QLogic InfiniPath QLE7140 HCAs. Therefore, all following benchmarks, tests and discussions will be based on the PSM interface. They also demonstrated that especially for small and medium messages the QLogic InfiniPath QLE7140 HCAs outperform the Mellanox InfiniHost III Lx HCAs as also indicated by the big difference in the N/2 values of both.

2.3 Bi-directional Bandwidth

The OSU bi-directional bandwidth benchmark uses the non-blocking MPI functions MPI_Isend and MPI_Irecv and performs the same operations on both processes. For each message size, 64 MPI_Irecvs are posted before the corresponding 64 MPI_Isends are invoked on both sides. After that the processes wait for all send and receive operations to finish with MPI_Waitall. The root process measures the time before the first post of MPI_Irecv occurred and after MPI_Waitall returned for all send and

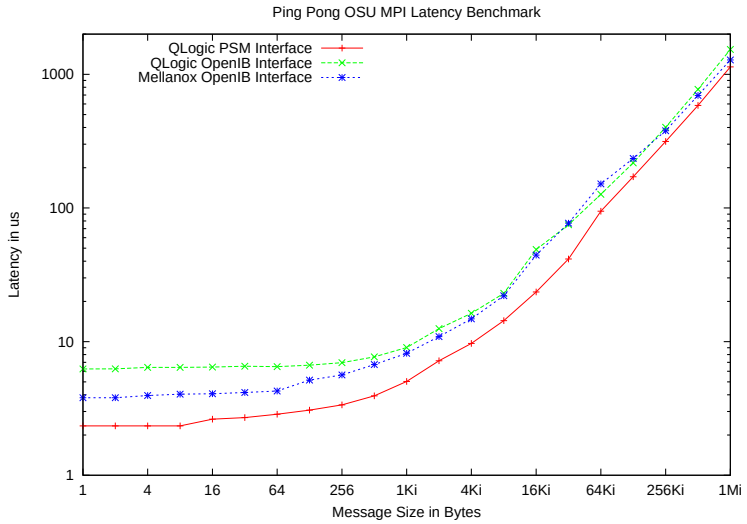

Figure 2.2: OSU 3.1 MPI Latency Benchmark Comparison between QLogic InfiniPath QLE7140 and Mellanox InfiniHost III Lx HCAs

receive operations. The difference of these two times is used to calculate the effective bandwidth. This is done several times for all message sizes to ensure an accurate measurement.

In order to compare bi-directional with uni-directional bandwidth, the results of the uni-directional benchmarks from above are included in Figure 2.3. The QLogic InfiniPath QLE7140 HCAs reach a bi-directional peak performance of 1617MiB/s with a N/2 value of 1090 bytes, profiting from the switch between eager and rendezvous protocol. The Mellanox InfiniHost III Lx HCAs reach a bi-directional peak performance of 1590MiB/s with an N/2 value of 13214 bytes. In general the bi-directional bandwidth never reaches two times of the uni-directional bandwidth but would still outperform two sequential send operations between processes over one interconnection. Comparing the bandwidth of one QLogic InfiniPath QLE7140 HCA in the bi-directional case with one in the uni-directional case the bandwidth loss can reach up to 38% at a message size of 64 bytes. See Figure 2.4 for more details.

2.4 Multiple Process Bandwidth

The OSU multiple bandwidth test measures the bandwidth between multiple pairs of processes. This benchmark uses the same algorithm as the bandwidth benchmark described in Section 2.2.1 for each pair. After all pairs finished the measurement, the longest time taken of all pairs is used by the master process to calculate the total

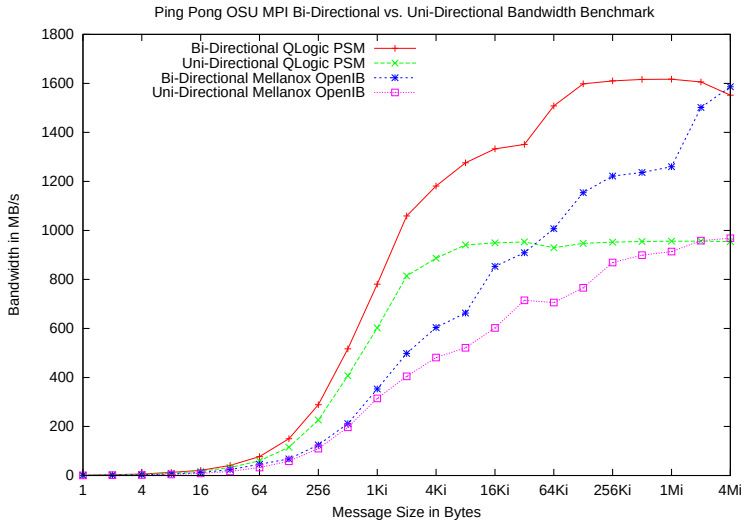

Figure 2.3: OSU 3.1 MPI Bi- and Uni-Directional Bandwidth Comparison

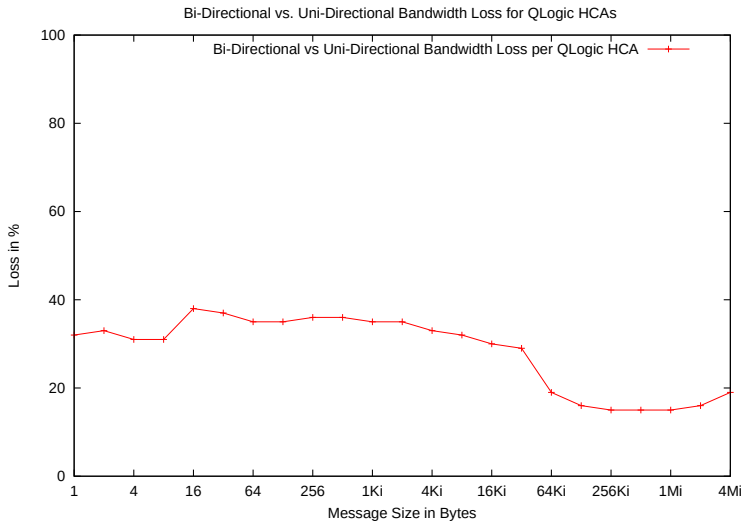

Figure 2.4: Effective Bandwidth Loss in Bi-Directional Case for One QLogic InfiniPath QLE7140 HCA

bandwidth.

In this case all pairs were equally distributed over two nodes sharing one InfiniBand™ connection. The results of the benchmark are shown in Figure 2.5 for QLogic InfiniPath QLE7140 HCAs and in Figure 2.6 for the Mellanox InfiniHost III Lx HCAs. The latter profiteers greatly by using two instead of one process to send data. The N/2 value (see Table 2.1) decreases to a fourth. The decrease of bandwidth utilization at 16KiB is caused by the Open MPI's switch from eager to rendezvous protocol. The bandwidth usage of the QLogic InfiniPath QLE7140 HCAs also benefits from increasing the sending processes. However, using multiple processes the Mellanox InfiniHost III Lx HCAs are still slower for small and medium sized messages compared to the QLogic InfiniPath QLE7140 HCAs using the same amount of processes.

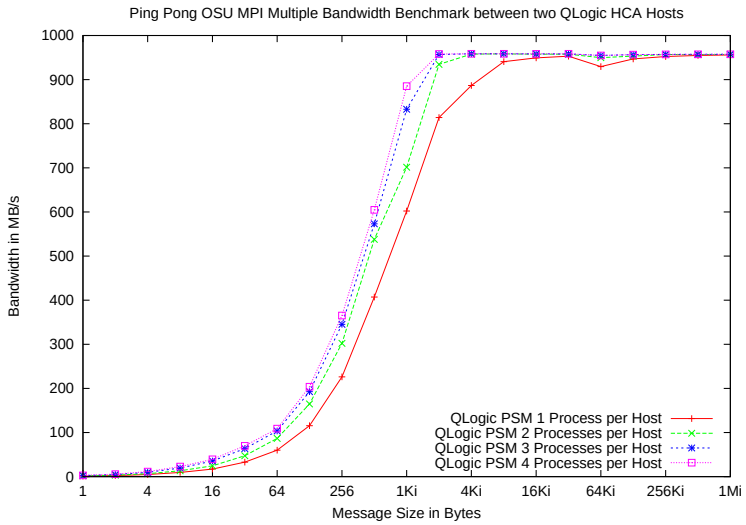

Figure 2.5: OSU 3.1 MPI Multiple Process Bandwidth Benchmark of the QLogic InfiniPath QLE7140 HCA

The benchmarks show that sending out data using multiple cores of a node increases the bandwidth especially for small and medium sized messages, which also results in reduced N/2 values which can be seen in Table 2.1. This knowledge can be used to increase the performance of collective operations by allowing multiple processes on one node to send simultaneously or using multiple threads in the MPI library to send the data of one process. A similar analysis was done by R. Kumar et al. in 2008 [44] for QLogic InfiniPath QLE7140 and Mellanox MT25208 DDR HCAs. They found comparable results for bi-directional bandwidth usage.

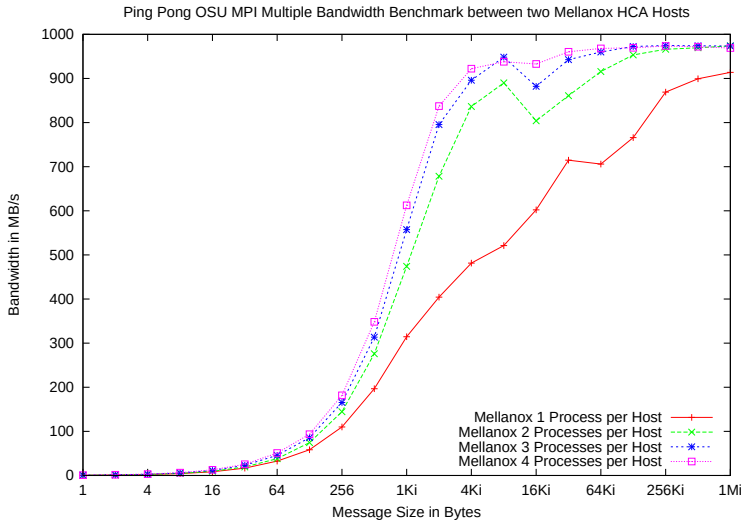

Figure 2.6: OSU 3.1 MPI Multiple Process Bandwidth Benchmark of the Mellanox InfiniHost III Lx HCA

HCA	**Number of Processes**	**N/2 in byte**
Mellanox InfiniHost III Lx	1	4289
	2	1063
	3	829
	4	725
QLogic InfiniPath QLE7140	1	659
	2	426
	3	386
	4	351

Table 2.1: Overview over N/2 for Different Numbers of Sending Processes

2.5 Overlapping Communication and Computation

MPI provides the means (e.g. non-blocking send operations like MPI_Isend) to overlap communication with computation to achieve better application level performance. In order to benchmark the impact on communication performance when parallel computation threads or processes are on the same host, the OSU bandwidth and latency benchmarks were modified. While benchmarking the bandwidth or latency between two processes the modified test also measures the combined memory bandwidth of n OpenMP threads which run the STREAM [45] triad benchmark. The synthetic STREAM benchmark measures the performance of four long vector operations where one is the triad operation: $a(i) = b(i) + q * c(i)$. It was designed to explicitly measure the memory bandwidth of a system and its impact on the processing speed by eliminating the possibility of data re-use in CPU registers or caches.

2.5.1 Overlapping Impact on Bandwidth

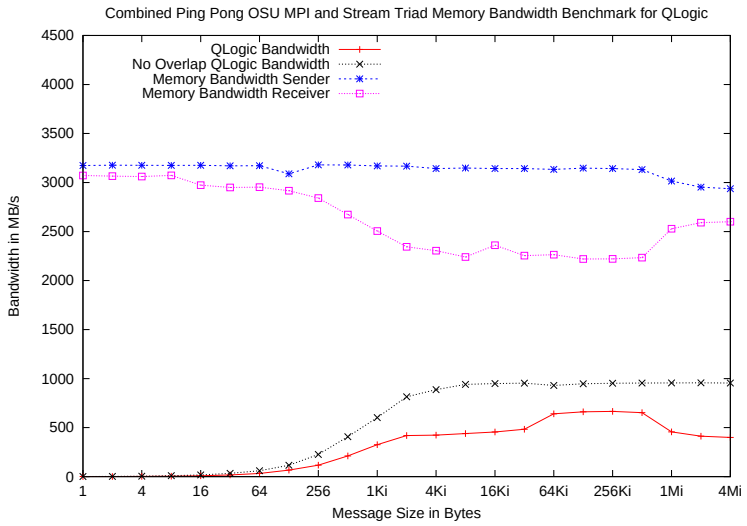

Figure 2.7: QLogic InfiniPath QLE7140 : OSU 3.1 MPI Overlapping 1 Communication Thread and 3 Computation Threads Bandwidth Benchmark

Figure 2.7 and 2.8 show the result of the combined benchmarks for the Jack system with three computation and one communication thread on each node. One additional gray line shows the bandwidth utilization of both HCA types without computational influence on the nodes.

Since the Xeon 5130 CPUs do not have a memory controller on the CPU die and thus the one in the northbridge is shared between two dual core CPUs, the results

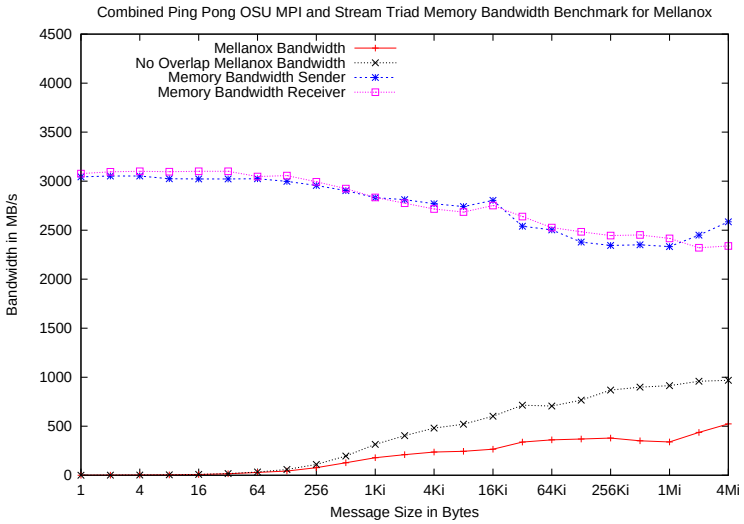

Figure 2.8: Mellanox InfiniHost III Lx : OSU 3.1 MPI Overlapping 1 Communication Thread and 3 Computation Threads Bandwidth Benchmark

of this benchmark are not necessarily comparable to more modern CPUs with an internal memory controller.

While the memory bandwidth of the receiver is directly influenced by the communication for both HCAs, the memory bandwidth of the QLogic InfiniPath QLE7140 HCAs sender does not change much until the message size reaches 1MiB. This can be advantageous in a master/worker scenario where the workers send their results back to the master while computing another work unit. Apart from that the bandwidth of the QLogic InfiniPath QLE7140 HCAs still outperforms the Mellanox InfiniHost III Lx HCAs. A reason for this may be that the host CPU can handle memory access better than the ASIC on the Mellanox InfiniHost III Lx HCAs and the memory bus is less often needed for synchronization between QLogic InfiniPath QLE7140 HCA and CPU memory access.

In order to measure the impact of a system which is fully saturated by computation threads on the communication, the benchmarks were repeated with 4 computation threads and 1 communication thread. The results of these benchmarks are presented in Figure 2.9. Despite utilizing more bandwidth than the Mellanox InfiniHost III Lx HCAs the benchmark results for the QLogic InfiniPath QLE7140 HCAs for messages over 128B were completely random over multiple benchmark runs.

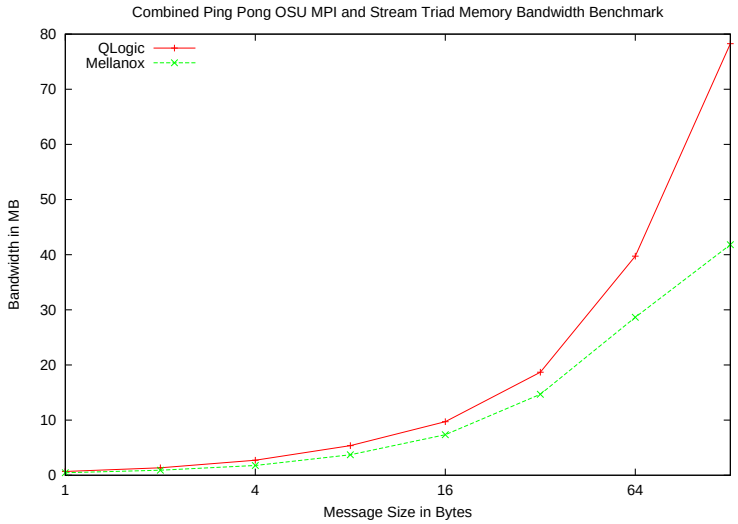

Figure 2.9: OSU 3.1 MPI Overlapping 1 Communication Thread and 4 Computation Threads Bandwidth Benchmark

2.5.2 Overlapping Impact on Latency

Figure 2.10 shows the latency for four computation and one communication thread on each node. As with the bandwidth usage also the latency of the QLogic InfiniPath QLE7140 HCAs was different for each run when the message size exceeded 128B. The computation threads have a clear influence. Compared to the original latency test (see 2.2.2) the latency is much higher.

In 2004 Lui et al. proposed another communication/computation overlap benchmark in [46] based on the latency benchmark. The sender posts non-blocking send and non-blocking receive operations, enters a computation loop and then waits for the send and receive operations to finish. The maximum time in the computation loop which does not increase the latency is then taken as the potential of overlapping communication and computation. Essentially the maximum time in the computation loop t_c corresponds to $t_c = t_{RTT} - o_r$ where t_{RTT} is the round-trip time and o_r the time for the receive overhead, as long as no host intervention (e.g. rendezvous protocol handling by the MPI library) is necessary for the communication.

2.5.3 Summary

As expected, parallel computation and communication activities influence each other. Nevertheless, overlapping should be considered whenever possible to maximize the usage of resources and therefore increase the application level performance or the performance of collective accumulation operations like all-reduce .

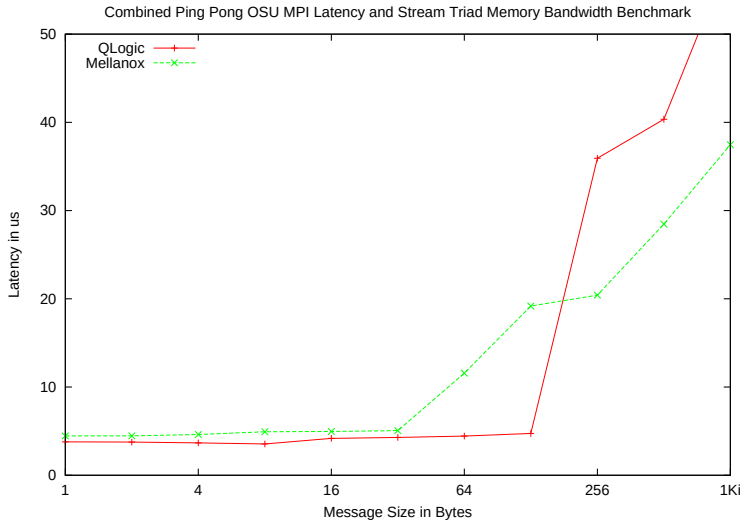

Figure 2.10: OSU 3.1 MPI Overlapping 1 Communication Thread and 4 Computation Threads Latency Benchmark

2.6 Round Trip Time, Send and Receive Overhead

In order to measure the parameters for communication and computation models, different benchmarks have been proposed (see Section 1.5). In this section the parameters determined by different measuring methods will be presented.

In order to be able to predict the performance of MPI_Allreduce() as precisely as possible and since some MPI_Allreduce() uses the MPI functions to send and receive messages instead of using the native operations for the network, the parameters will also be measured using the MPI functions.

The LogGP benchmarks were conducted on Jack and CHiC. The CHiC was chosen to be able to verify the LogGP predictions of the all-reduce algorithms in large scale since the resources on David were limited. The LoP benchmarks were executed on David.

2.6.1 P-LogP Model Benchmarks

In order to measure the P-LogP parameter the MPI P-LogP benchmark[4] from Kielmann and Bal [38] was used. Measurement method and algorithm have been described in Section 1.5.4 above.

Table 2.2 shows the results of the benchmark for a message size of 1 byte. The gap per byte G was calculated by dividing the gap for a reasonable big message

[4] `http://www.cs.vu.nl/albatross/#software`

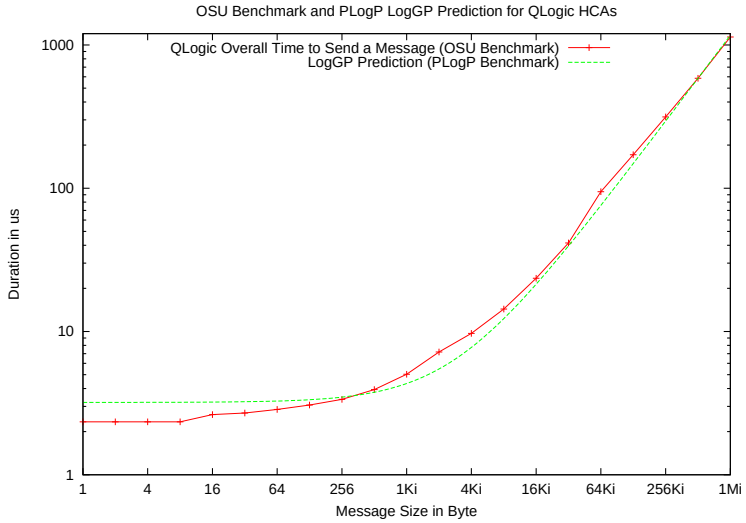

Figure 2.11: Comparison of OSU Latency Benchmark and P-LogP Benchmarks LogGP Prediction for QLogic InfiniPath QLE7140 HCAs

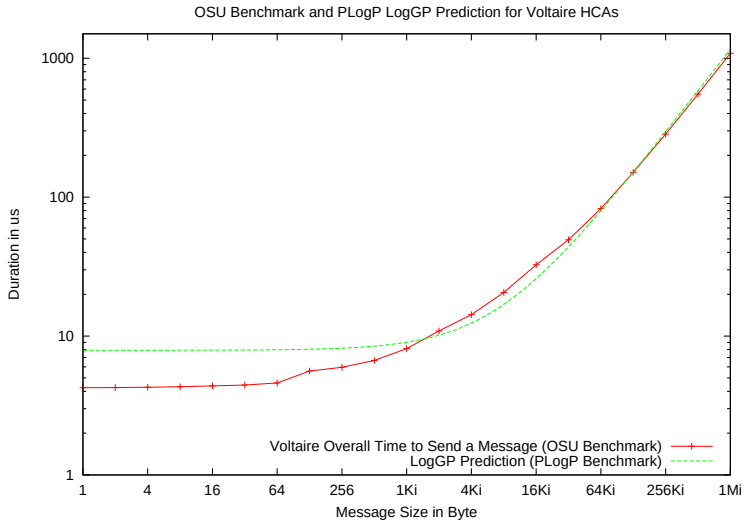

Figure 2.12: Comparison of OSU Latency Benchmark and P-LogP Benchmarks LogGP Prediction for Voltaire HCA 410Ex HCAs

size through the number of bytes. In this case G was calculated of the gap for a message size of 1 MiB. As one can see, the P-LogP benchmark determines 0 as latency and a huge gap of 5μs respectively 6.9μs for both Mellanox InfiniHost III Lx and Voltaire HCA 410Ex HCAs. This is obviously an error due to the unawareness of the benchmark of the offload feature. Nonetheless, using the parameters determined the predicted time to send a message from one node to another in the LogGP model is quite accurate for messages over 1KiB for the Voltaire HCA 410Ex HCAs as shown in Figure 2.12. For the QLogic InfiniPath QLE7140 HCAs (see Figure 2.11), the LogGP model over-estimates the duration for messages smaller than 256KiB by up to 40%, but predicts the time for all other message sizes quite well.

Parameter	**QLogic**	**Mellanox**	**Voltaire**
L	1.7μs	0.0μs	0.0μs
o_s	0.8μs	0.6μs	7.2μs
o_r	0.7μs	0.5μs	0.6 μs
g	1.8μs	5μs	6.9μs
G	0.0012μs	0.00128μs	0.0011μs

Table 2.2: P-LogP Parameter Results for the P-LogP Model Benchmark over Open MPI for a Message Size of 1 byte

2.6.2 LogGP Model Benchmarks

The microbenchmark used to determine the LogGP parameters was developed by Höfler et al. [40] and is available as part of the Netgauge[5] tool. The used algorithm is described in detail in 1.5.4.

Table 2.3 shows the measured LogGP values for the different HCAs. Using these parameters in the LogGP model for the QLogic InfiniPath QLE7140 HCAs as shown in Figure 2.13 reveals a very accurate prediction for message sizes between 2048 and 64000 bytes. For messages over 64000 bytes the transfer needs more time than predicted because of the rendezvous protocol. The determined parameters for the Voltaire HCA 410Ex HCAs show good accuracy for messages bigger than 32KiB (see Figure 2.14). Over the whole spectrum of message sizes the parameters determined by the P-LogP benchmark show greater accuracy even though the measured parameter for the Voltaire HCA 410Ex HCAs cannot be right. The accuracy at greater message sizes is caused by the dominance of the gap per byte.

The high deviations for small messages are caused by different measurement methods used by Netgauge, the P-LogP benchmark and the OSU MPI latency benchmark and overlapping of the send overhead, the latency and the receive overhead. So does

[5] `http://www.unixer.de/research/netgauge/`

Netgauge measure an RTT of 6.087092μs for a message size of 1 byte over the QLogic InfiniPath QLE7140 HCAs while the determined LogGP parameters for this message size add up to an RTT of $2 * (3.0434\mu s + 0.7797\mu s + 0.9587\mu s) = 9.5636\mu s$.

Parameter	QLogic	Mellanox	Voltaire
L	3.0434μs	4.74134 μs	5.3157 μs
o_s	0.7797μs	0.96 μs	7.08μs
o_r	0.9587μs	1.38 μs	6.8493μs
g	0.9μs	1.4 μs	0.00265μs
G	0.001154μs	0.021143 μs	0.001024μs

Table 2.3: LogGP Parameter Results for the LogGP Model Netgauge Benchmark over Open MPI

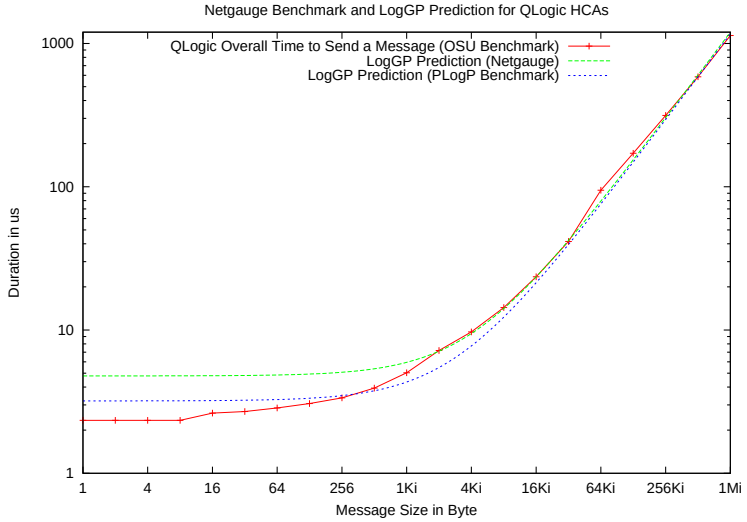

Figure 2.13: Comparison of OSU Latency Benchmark and LogGP Predictions for QLogic InfiniPath QLE7140 HCAs

2.6.3 LoP Model Benchmarks

The benchmarks to measure the parameters for the LoP model were developed by T. Höfler [41]. For this thesis, these were adapted for the PSM interface, OpenIB and MPI. Since the purpose of this thesis is the optimization of an MPI collective function which may use MPI_Send and MPI_Recv, the overhead of the MPI library must be considered as well. It has been shown [41] that pipelining effects of the HCAs can greatly reduce the time needed for communication between multiple processes.

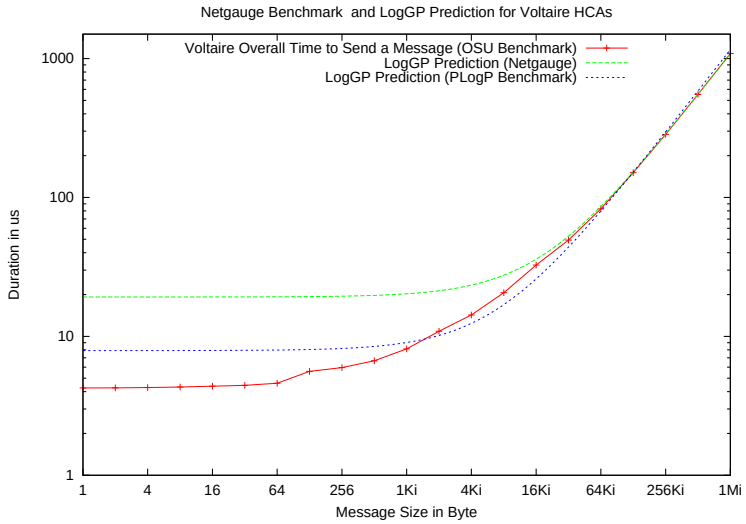

Figure 2.14: Comparison of OSU Latency Benchmark and LogGP Prediction for Voltaire HCA 410Ex HCAs

Thus, the parameters are not only measured for 1:1 communication but also for 1:n and n:1.

Figure 2.15 and 2.16 presents the result of the send and receive post overhead measurements for different numbers of combined posts. Compared with the native PSM posts, the MPI library overhead adds 0.15μs. Also the onload architecture of the QLogic InfiniPath QLE7140 HCAs does benefit from posting multiple requests simultaneously, the offload architectures of the Mellanox InfiniHost III Lx HCAs outperforms it for send requests of 1 byte greatly.

In order to measure the impact of the message size on the send and receive overhead the benchmarks were extended. The results for 10 combined posts are shown in Figure 2.17 and 2.18. The switch between eager and rendezvous protocol by the PSM library drastically reduces the measured time needed for posting a send request for the QLogic InfiniPath QLE7140 HCAs. Nonetheless, the CPU is still involved in the transfer.

Figure 2.19 shows the MPI RTT for different message sizes of the QLogic InfiniPath QLE7140 HCAs. The benchmarks were performed on the David cluster. Up to 10 processes the RTT per process shrinks steadily to about 38% of the 1:1 RTT.

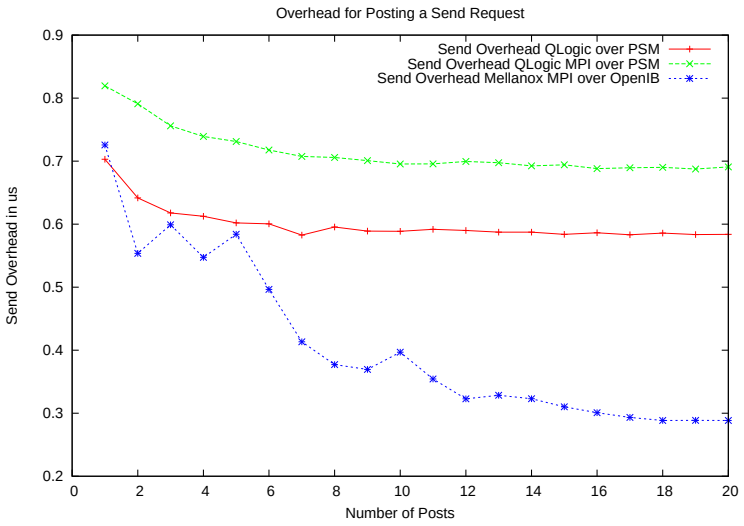

Figure 2.15: Average Send Post Overhead for a Message of 1 byte

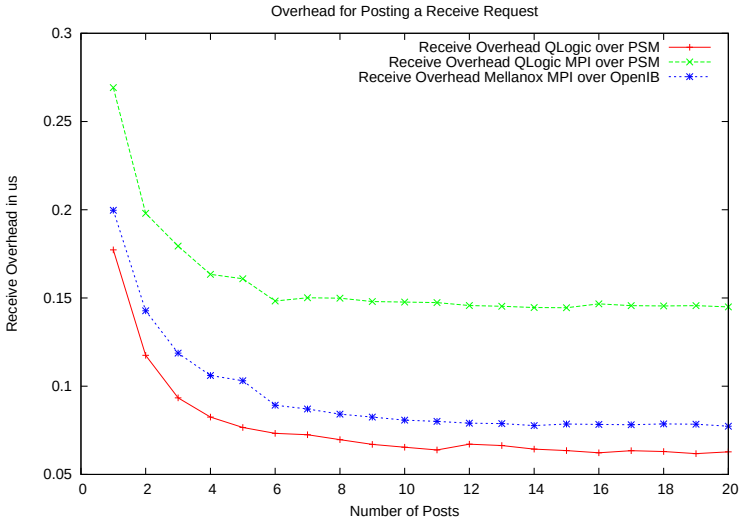

Figure 2.16: Average Receive Post Overhead for a Message of 1 byte

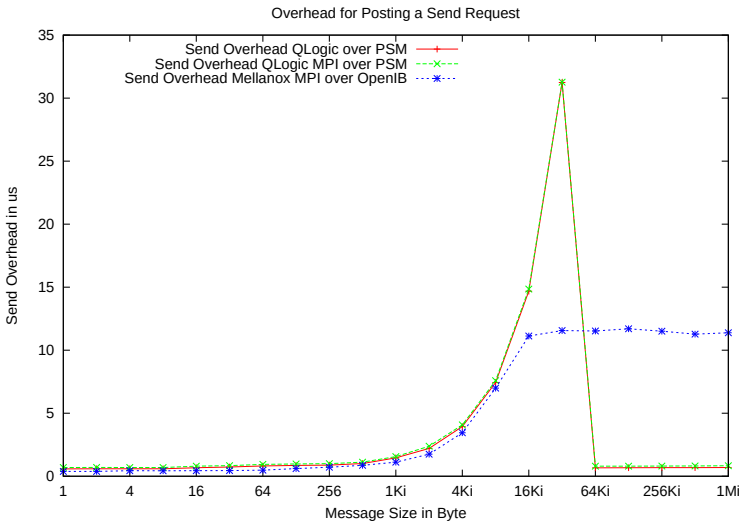

Figure 2.17: Average Send Post Overhead for 10 Chained Posts

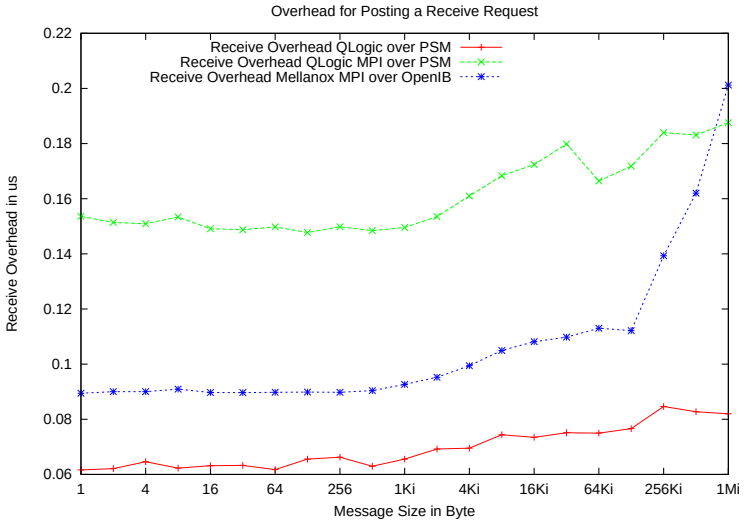

Figure 2.18: Average Receive Post Overhead for 10 Chained Posts

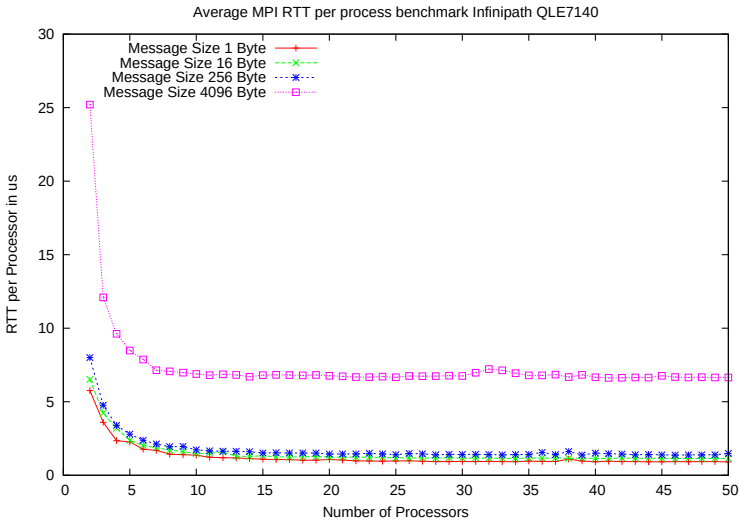

Figure 2.19: Average RTT per Process for QLogic InfiniPath QLE7140 HCAs

2.7 Conclusion

The benchmarks revealed that the onload architecture of the QLogic InfiniPath QLE7140 HCAs can strongly outperform the Mellanox InfiniHost III Lx HCAs in bandwidth usage and latency for small and medium sized messages. The downside are higher send and receive overheads which on the other hand can be reduced by faster CPUs and faster buses. Using more than one process on multi-core systems to send data increases the bandwidth utilization and therefore should be considered for collective communication in particular for small and medium message sizes. While overlapping communication with computation has side effects on both, the overall application level performance should still be higher than their serial execution. An optimal all-reduce algorithm for QLogic InfiniPath QLE7140 HCAs would therefore use the PSM interfaces non-blocking operations to send small and medium messages to as many peers simultaneously by as many processes of the same node as possible.

3 MPI_Allreduce() Algorithms

The all-reduce operation has been extensively studied in the past and still is an object of interest to several groups. Different hardware, networks and application level usage patterns require different all-reduce algorithms to optimally run MPI_Allreduce() . Thus, there exist many implementations for different requirements. In this chapter the most commonly used and some promising proposed algorithm will be presented and analysed.

As Patarasuk and Yuan proved (see [6]) in 2009, the minimum number of communicated atomic data items needed to complete an all-reduce operation is $2 * (P - 1)$, where P is the number of participating processes. Here, an atomic data item is the biggest possible payload of one message. For more details see Section 3.5. For big messages, the bandwidth usage of an all-reduce algorithm is of great importance. The algorithm must ensure that only the lowest amount of data possible is transferred over the network. This may be achieved by sending multiple medium sized messages as long as the latency and overhead impact on the overall transfer time is small. The algorithm should also be designed to avoid network contention. Such algorithms can be called *bandwidth optimal*. The algorithm presented in Section 3.5 is such a bandwidth optimal algorithm for tree network topologies. For small messages, an all-reduce algorithm should be *latency optimal*. In the case of small messages the latency may have a great impact on the overall transfer time of a message. Thus, the algorithm should send only the minimum number of messages needed to finish the all-reduce operation. Another important parameter to consider is the time needed for the computation of the accumulation operation. For complex operations or big operands the computation should be divided over all processes to compute the result as fast as possible. This is in some cases achieved by Rabenseifners algorithm presented in Section 3.3.

The following algorithms will be evaluated in the LogGP model which represents the time needed for *all* processes to finish. Therefore, the model is extended to also incorporate the computation time C per byte. The size of a message will be represented by m. Full bi-directional bandwidth is assumed as well as a contention free network. To reduce the complexness of the time functions, we introduce some new parameters. The parameter α represents the maximum of o_s and g since both parameters can overlap when sending consecutive messages:

$$\alpha = \max\{o_s, g\}$$

The parametrized $\gamma(m)$ represents the maximum of $o_s + mC$ and g to reflect the overlapping of g by messages which are sent after the computation of the accumulation operation:

$$\gamma(m) = \max\{o_s + mC, g\}$$

3.1 Linear

The linear algorithm is the most basic MPI_Allreduce() implementation and is used as fallback function in Open MPI. The algorithm calls the reduce function, which reduces the data of all participating processes to the root process, followed by a broadcast of the result from the root to all other processes.

By using optimized reduce (see [4, 42, 47]) and broadcast (see [48]) algorithms this can be the preferred method for a small amount of processes and small messages, even though the root process implies a bottle-neck in terms of communication and computation.

In [49] Mamidala et al. proposed an adaptive version of all-reduce for InfiniBand™ based on the combining tree algorithm for the reduce phase. The algorithm handles the situation where not all processes start the all-reduce operation at the same time and thus may have to wait valuable time for other late processes.

LogGP Prediction If the reduce and the broadcast functions are implemented with a linear algorithm, the following formulas give a measure for the communication time. For the reduce, all processes expect of the root send their data in parallel to the root. Thus the time for the reduce is affected by o_s and L only once. The root must compute the accumulation operation for the data of all processes. After this is done it broadcasts the data to all other processes. In the linear case this is done by sending the result to each processes one after another. Thus, the last process receives the results after the root sent the message to the first process in $o_s + (m-1)G$ and to all others in $o_r + L + (P-2)(\alpha) + (P-2)(m-1)G$.

$$\begin{aligned} LogGP_{reduce}(P) =& o_s + L + (P-1)(o_r + (m-1)G) + PmC \\ LogGP_{broadcast}(P) =& o_s + o_r + L + (P-2)\alpha + (P-1)(m-1)G \end{aligned}$$

Since the broadcast follows right after the reduce and the reduction operation, the first send of the broadcast operation is influenced by the computation of the accumulation operation and g. Therefore, it needs a time of $\gamma(Pm) + (m-1)G$. This leads to the following LogGP function for the linear case:

$$LogGP(P) = o_s + o_r + 2L + \gamma(Pm) + (P-2)\alpha + (P-1)(o_r + 2(m-1)G)$$

3.2 Recursive Doubling

In order to solve linear equations, Recursive Doubling techniques have already been used by the English mathematician J.J. Sylvester in 1853 [50]. As an algorithm for collective MPI operations it was analyzed by Thakur et al. in [4]. For Recursive Doubling the number of processes participating in the all-reduce operation is of importance. One can distinguish between the "power-of-two" and the "non-power-of-two" case.

3.2.1 Power-of-Two Case

Recursive Doubling can be run in $\log_2 P$ steps if P, the number of participating processes, is a power-of-two. Basing on the distance, in each step pairs of processes are created, which exchange their data and apply the reduce operation on it. Figure 3.1 shows an example for 8 processes. In the first step processes with a distance of 1 exchange their data and perform the reduction on it. In the second step processes that are a distance of 2 apart exchange the reduced data from the last and reduce it. In the third step processes with a distance of 4 exchange their data reduced in the second step and apply the reduce operation on it. Thus, after three steps all processes own the same result of the reduction operation. In the power-of-two case $log_2 P * P$ full sized data messages have to be communicated using bi-directional transfers, which can reduce the available bandwidth for both processes as shown in Section 2.3. This algorithm is used by MVAPICH2 1.0.2 for small and medium sized messages when the number of participating processes equals a power-of-two.

LogGP Prediction In the first step $s = 1$ each process can immediately start to send without computing the reduce operation or waiting for g. Sending and receiving can be done in parallel. Therefore, each process must wait only for one full message transfer:

$$LogGP_{s=1}(P) = o_s + (m-1) * G + o_r + L$$

After that, each process must compute the reduction of the data vector received in the last step, send the result and receive a new data vector. Each process can start sending after processing the data vectors and the send overhead or after waiting for g, whichever takes longer. Thus, one intermediate step can be described in LogGP as:

$$LogGP_{1<s<=\log_2 P}(P) = \gamma(m) + (m-1) * G + o_r + L$$

After $(\log_2 P) - 1$ intermediate steps each process finishes the all-reduce operation by calculating the reduction of the received data:

$$LogGP_{s=(\log_2 P)+1}(P) = mC$$

Therefore, the complete algorithm can be described in the LogGP model with the following function:

$$LogGP(P) = (\log_2 P - 1)\gamma(m) + o_s + \log_2 P(o_r + L + (m-1)G) + mC$$

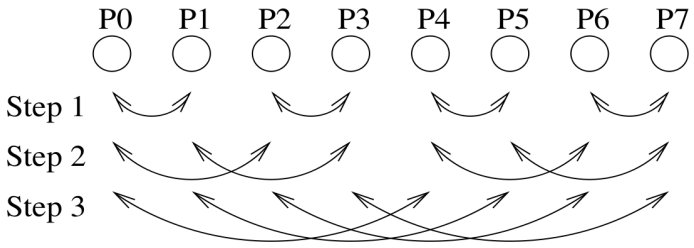

Figure 3.1: Recursive Doubling (image taken from [4])

3.2.2 Non-Power-of-Two Case

If the number of processes does not equal a power-of-two the total number of steps is bounded by $2\lfloor \log_2 P \rfloor$, if the algorithm compensate for the non-power-of-two processes in each step [4]. Another method of handling this case is to eliminate the non-power-of-two processes before the start of the process and send the result of the reduction to them after it finished. This technique was proposed by Rabenseifner for his algorithm (see Section 3.3). Therefore, the number of processes is reduced to a power-of-two value: $P' = 2^{\lfloor \log_2 P \rfloor}$ and the rest $r = P - P'$ is the number of processes which are removed in the first step. This is done by letting the first $2r$ processes send pairwise from each odd rank to the even $(rank - 1)$ their data. The even ranks reduce the received data with their own and perform the algorithm of the power-of-two case together with the $P - 2r$ ranks. After they have finished, the even ranks send the result to the odd.

LogGP Prediction If one uses the elimination technique, the first send of the recursive-doubling algorithm is influenced by the computation and g. In the additional last step, the send is also influenced by the computation. Thus, the LogGP function in the non-power-of-two case is:

$$\begin{aligned} LogGP(P) = & o_s + o_r + L + (m-1) * G \\ & + ((\log_2 P) + 1)(\gamma(m) + (m-1) * G + o_r + L) \end{aligned}$$

3.3 Rabenseifner

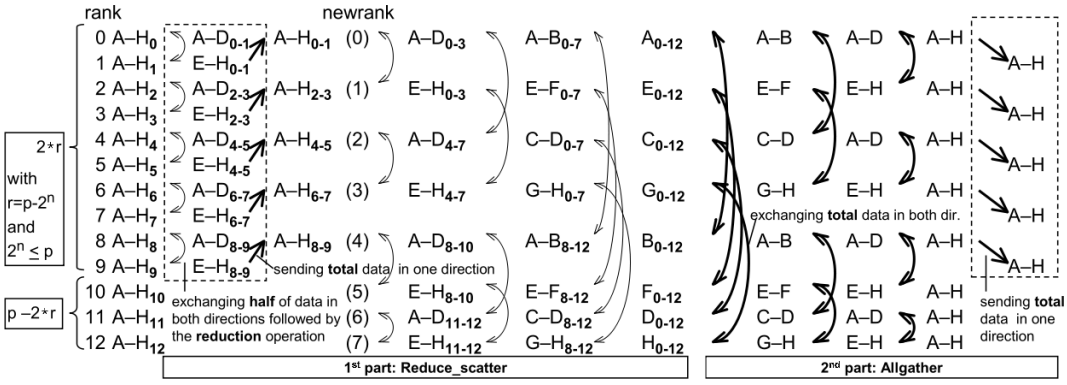

Figure 3.2: Example for Rabenseifners Algorithm (image taken from [5])

Rabenseifner proposed this algorithm in 2004 (see [5]) for an optimized reduce load balance in communication and computation. It is similar to an algorithm developed by Geijn in 1994 for reduce [51]. Rabenseifners method uses a reduce-scatter implemented as recursive input vector halving and distance doubling followed by an all-gather operation using recursive output vector doubling and distance halving. The purpose of the reduce-scatter operation is to reduce the elements of the input vector of all processes and scatter the element i to the process i. The all-gather function places the ith element sent by process i in the receive vector of all participating processes at the ith position.

3.3.1 Preparation

Like Recursive Doubling this only works for a number of processes which equals power-of-two. So in the first step the number of processes P must be cut down to a power-of-two value $P' = 2^{\lfloor \log_2 P \rfloor}$. The rest $r = P - P'$ must be removed for the time being. So the first $2r$ processes pairwise send from each even $rank$ to the odd $(rank + 1)$ the second half of their input vector and each odd $rank$ sends its first half to the even $(rank - 1)$. Each process of the $2r$ reduces its half of the vector and the odd ranks send the result back to the even ranks. The remaining $P - 2r$ and the first r even processes are then assigned new $ranks'$ and the recursive input vector halving and distance doubling can be performed.

LogGP Prediction The exchange of the $\frac{m}{2}$ data vectors can be done in parallel. After that, the even ranks can reduce their half and have to wait for the odd ones to finish the reduction and the transfer. Thus, if the number of processes is not a power

of two the needed time in the LogGP model for the preparation is:

$$LogGP = (o_s + (\frac{m}{2} - 1)G + o_r + L) + (\gamma(\frac{m}{2}) + (\frac{m}{2} - 1)G + o_r + L)$$

3.3.2 Reduce-Scatter: Recursive Vector Halving and Distance Doubling

In the first step, the even $ranks'$ send the second half of their buffer to the $rank' + 1$ and the odd send the first half of their buffer to $rank' - 1$. Each process performs the reduction operation between their local buffer and the receive buffer. The reduced half is then used in the next step. For the next $(\log_2 P') - 1$ steps, the buffer is halved and the distance between the processes is doubled in each step. After that, each of the P' processes has the result for $\frac{1}{P'}$ of the global reduction vector.

LogGP Prediction In each step the number of transferred and reduced data items is halved. Thus, the number of data items to process in step s is $\frac{m}{2^s}$. The number of items to process in all steps is then $\sum_{s=1}^{\log_2 P'}(\frac{m}{2^s})$. In each step, each process sends and receives data in parallel. For the power-of-two case, the first step is not influenced by g or computation. For the non-power-of-two case, the first step of the reduce-scatter function is not influenced by both either. The reason is that the r even ranks had to wait for the r odd ranks to transfer the result of their accumulation back to the even ranks. This transfer already included $\gamma(\frac{m}{2}$. Thus, the first step of the reduce-scatter operation in the LogGP model is:

$$LogGP_{s=1} = o_s + o_r + L + \frac{m}{2}G$$

All following steps are influenced by the computation of the last step. The combined LogGP function for the steps $s = 2$ to $s = \log_2 P'$ is then:

$$LogGP(P') = \sum_{s=2}^{\log_2 P'} (\gamma(\frac{m}{2^{s-1}}) + o_r + L + (\frac{m}{2^s} - 1) * G)$$

In the last step of the reduce-scatter function, only the accumulation must be computed:

$$LogGP_{s=(\log_2 P')+1}(P') = \frac{m}{2^{(\log_2 P')}}C$$

3.3.3 All-Gather: Recursive Vector Doubling and Distance Halving

Now all processes need to exchange their $\frac{1}{P'}$ result with the others to get the complete reduction result. So in each of the $\log_2 P'$ steps the processes of distance 2^{step-1} exchange 2^{step-1} results, leading to the complete result vector after $\log_2 P'$ steps.

LogGP Prediction Like in the reduce-scatter part, the processes exchange $\frac{m}{2^s}$ data in each step in parallel. The first step of the all-gather operation is influenced by g or the computation of the last reduce-scatter step:

$$LogGP_{s=1}(P') = o_r + L + \gamma(\frac{m}{2^{\log_2 P'}}) + \frac{m}{2^{\log_2 P'}}G$$

All following steps are only influenced by the maximum of o_s and g. The combined LogGP function for the steps $s = 2$ to $s = \log_2 P'$ is then:

$$LogGP(P') = \sum_{s=2}^{\log_2 P'} (\alpha + o_r + L + (\frac{m}{2^s} - 1) * G)$$

3.3.4 Finalization

If the number of processes had to be reduced in the first step the complete result vector is now send to the r removed processes.

LogGP Prediction This causes another delay of one send done by r processes in parallel:

$$LogGP(P) = \alpha + (m-1)G + o_r + L$$

3.3.5 Summary

While neither being bandwidth nor latency optimal, for the power-of-two case at the moment there exists no a more balanced algorithm for all-reduce which involves all processes in the communication and computation process more equally. Figure 3.2 shows an example for 13 processes. The vectors are divided in P' sections in this case A,B,...,H for each $rank$ as indicated by $A - H_{rank}$.

LogGP Prediction In the power-of-two case, Rabensteiners all-reduce algorithm needs a time of:

$$\begin{aligned}
LogGP_{p2}(P) =& o_s + o_r + L + \frac{m}{2}G \\
&+ \sum_{s=2}^{\log_2 P} (\gamma(\frac{m}{2^{s-1}}) + o_r + L + (\frac{m}{2^s} - 1) * G) \\
&+ o_r + L + \gamma(\frac{m}{2^{(\log_2 P)}}) + \frac{m}{2^{(\log_2 P)}}G \\
&+ \sum_{s=2}^{\log_2 P} (\alpha + o_r + L + (\frac{m}{2^s} - 1)G) \\
\\
=& o_s + 2(\log_2 P)(o_r + L) + \frac{m}{2}G \\
&+ \sum_{s=2}^{\log_2 P} (\alpha + \gamma(\frac{m}{2^{s-1}}) + 2(\frac{m}{2^s} - 1)G) \\
&+ \gamma(\frac{m}{2^{(\log_2 P)}}) + \frac{m}{2^{(\log_2 P)}}G \\
\\
=& o_s + 2(\log_2 P)(o_r + L) + \frac{m}{2^{(\log_2 P)}}G + \frac{m}{2}G + \gamma(\frac{m}{2^{(\log_2 P)}}) \\
&+ \sum_{s=2}^{\log_2 P} (\alpha + \gamma(\frac{m}{2^{s-1}}) + 2(\frac{m}{2^s} - 1)G)
\end{aligned}$$

For the non-power-of-two case :

$$LogGP_{np2}(P) = LogGP_{p2}(P') + o_s + \alpha + \gamma(\frac{m}{2}) + 3((\frac{2}{3}m - 1)G + o_r + L)$$

3.3.6 Optimizations

In [31] more efficient algorithms to reduce the number of processes to a power-of-two number were presented by Rabenseifner and Träff. Nonetheless, decreasing the number of processes participating in the all-reduce process introduces load imbalances in communication and computation between the processes. This excludes valuable communication and processing resources which could have led to a faster execution. The Binary Blocks algorithm tries to minimize the load imbalances and should therefore be described in the next section.

3.4 Binary Blocks

If the number of processes is not a power-of-two the algorithm described above leads to load imbalances by excluding processes from the communication and computation. Therefore, Thakur et al. developed an extension to Rabenseifners method called Binary Blocks [4].

3.4.1 Preparation

Regardless of the number, the processes are decomposed into a minimum number of blocks with power-of-two processes (see Figure 3.3 for an example).

3.4.2 Reduce-Scatter: Recursive Vector Halving and Distance Doubling

Each block runs the reduce-scatter as described above for Rabenseifners approach. The blocks finish after $\log_2 P_{block}$ steps and, starting with the smallest block, the results of the reduce-scatter is split into n segments where n is the number of processes in the next bigger block. These segments are then sent to the processes in the next bigger block which reduce them.

3.4.3 All-Gather: Recursive Vector Doubling and Distance Halving

Before the smaller blocks can start their all-gather operation, they have to receive the results of the reduction from the next larger blocks as shown in the example in Figure 3.3. After each block has finished the all-gather phase, all processes possess the same result of the all-reduce operation.

3.4.4 Summary

Although the load imbalance is smaller than in Rabenseifners algorithm for the non-power-of-two case, it is still created by different sizes of blocks. The bigger the difference in the number of processes between two successive blocks the bigger the imbalance is - especially in the low range of exponents. Smaller blocks have to wait for bigger ones until they finish the reduction, which wastes valuable computation and communication resources. Thus, if the maximum difference between the number of processes of two successive Binary Blocks is small this algorithm can perform well.

Example In the example the Binary Blocks algorithm is run with 12 processes (numbered from 1 to 12) and 8 data elements (A to H) per process. The 12 processes are separated in blocks of 8,4 and 1 processes. Block 3 and 2 start with the reduce-scatter operation by pairwise exchanging half of their data items (A to D and E to H respectively) and accumulate them. Thus after the first step, process 0 owns

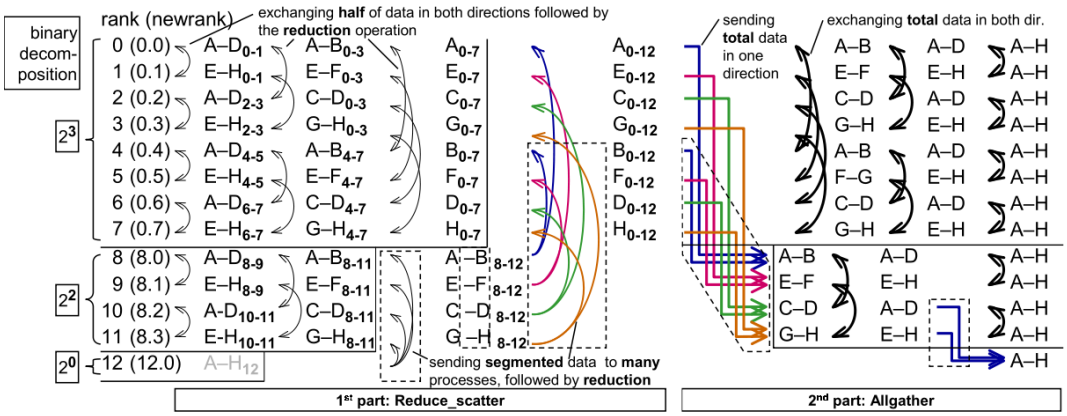

Figure 3.3: Example for Binary Blocks Algorithm (image taken from [4])

the result of the reduction of the data items A to D of rank 0 and 1 (indicated by $A-D_{0-1}$). Each block further halves the data and exchanges it with a rank which has twice the distance of the rank from the last step. The number of steps s is determined by the number of processes in the block: $s_{block} = \log_2 P_{block}$). Thus, in block 3 rank 0 owns the result of the accumulation of data element A of all processes in block 3 (A_{0-7}). In block 2 rank 8 owns the result of the accumulation of data element A and B of the 4 processes in this block ($A - B_{8-11}$) after two steps. This rank is now ready to receive the data elements A and B from rank 12 in block 3. After receiving both elements it reduce them and sends the result of A to rank 0 and the result of B to rank 4 in block 3. Rank 0 and 4 accumulate the received element with their local element. Thereby both ranks finish the reduce-scatter phase by owning the result of the reduction over all processes for their element.

In the all-gather phase, all processes pairwise exchange the results of the reduction in s_{block} steps similar to the reduce-scatter algorithm. The synchronization in the all-gather phase is done so that always two processes of a block send half of the data each to one process in the smaller block.

LogGP Prediction The communication and computation time in LogGP for the single blocks is the same as for the power-of-two case in Rabenseifers approach. Since communication and computation are done separately in parallel in each block, the run-time of the biggest block of size $P_{bmax} = 2^{\lfloor \log_2 P \rfloor}$ dominates the overall run-time of the all-reduce operation.

The additional time for the synchronization inside the reduce-scatter operation can vary very strongly. In the worst case one process has to send its data to $\log_2(P-1)$ other processes. In the best case one process only needs to transfer its data to two other processes. Thus, the additional time needed for the synchronization in

the reduce-scatter part depends on the maximum difference between the number of processes of two successive blocks and the number of blocks b. The synchronization time in the reduce-scatter phase is limited by the upper bound of:

$$LogGP_{rs-sync}(P) = \lceil (b-1)(P'-1)(\gamma(m) + (m-1)\alpha + (m-1)G + o_r + L) \rceil$$

where m and P' depend on the block sizes. The first send to another process may be influenced by the last accumulation of the sender. The synchronization time in the all-gather phase is always done by two processes which send the result of the reduce of their block to one process of the next smaller block. Thus the synchronization time in the all-reduce phase is limited by the upper bound of

$$LogGP_{ag-sync}(P) = \lceil \gamma(mC) + (b-2)\alpha + b - 1(L + o_r + 2(\frac{m}{2} - 1)G) \rceil$$

where m depends on the block sizes. The send from the biggest block is influenced by the last reduce-scatter accumulation operation.

Thus, the LogGP function for the Binary Blocks algorithm is:

$$\begin{aligned} LogGP(P) =& LogGP_{Rabenseifner}(P_{bmax}) \\ &+ \lceil (b-1)(P'-1)(\gamma(m) + (m-1)\alpha + (m-1)G + o_r + L) + \gamma(mC) \\ &+ (b-2)\alpha + b - 1(L + o_r + 2(\frac{m}{2} - 1)G) \rceil \end{aligned}$$

For the example shown in Figure 3.3, the schematic run-time for each block is shown in Figure 3.4 and the exact LogGP time of the synchronization will be discusses below.

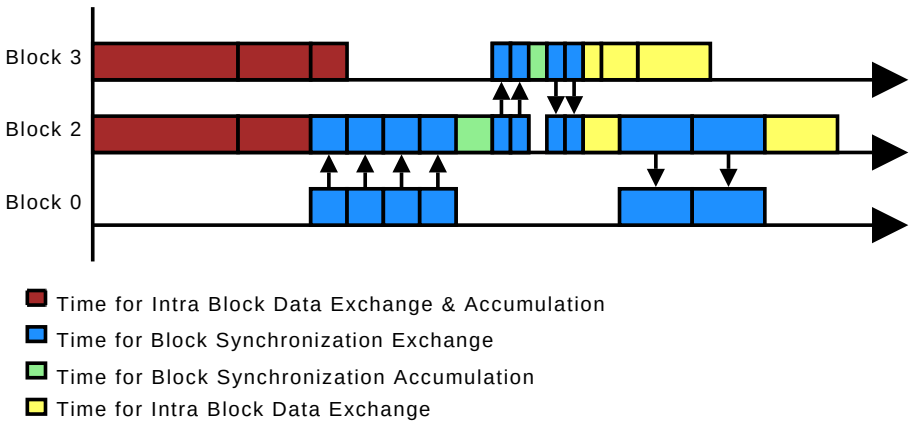

Figure 3.4: Time needed for the Binary Blocks Example in 3.3

A red square symbolizes the time needed for a intra block exchange and the accumulation operation of one data element. Thus computation and communication

time for one data element are the same in this example. In each step the number of data elements is halved, which halves the communication and computation time. The accumulation operation of the reduce-scatter synchronization (green) can be done in parallel, therefore it influences the run-time only once for each block. The four blue squares in block 0 represent the time for the sending of two data elements each. After a process has received and accumulated its data element(s) it can immediately start synchronizing with the next bigger (reduce-scatter state) or next smaller (all-gather state) block. Thus the time showed in Figure 3.4 is valid for the slowest process. The LogGP time for the synchronization in the reduce-scatter part for this example is:

$$LogGP(12) = 4 * (\alpha + (\frac{m}{4} - 1)G) + o_r + L + \gamma(\frac{m}{4}) + \alpha + 2(\frac{m}{8} - 1)G + o_r + L$$

For the all-gather part:

$$LogGP(12) = \gamma(\frac{m}{8}) + 2 * (\frac{m}{8} - 1)G + o_r + L + \alpha + 2 * (\frac{m}{2} - 1)G$$

3.5 Bandwidth Optimal Algorithm

In 2009 Patarasuk and Yuan proposed an bandwidth optimal all-reduce algorithm for tree network topologies [6]. Therefore, they combined Rabenseifners approach of realizing the algorithm by a reduce-scatter followed by an all-gather operation, both using logical ring based algorithms, with a method proposed by Faraj et al. in [52] to construct a contention-free logical ring. While the algorithm is bandwidth optimal even on SMP clusters, it is not optimal in the latency term and thus should not be used for small messages.

3.5.1 Finding a contention-free logical ring

In order to find a contention-free logical ring, the network topology must be provided for the algorithm as a tree graph $G = (S \cup M, E)$ where M is the set of machines and S the set of switches. Let $G' = (S, E')$ be a subgraph of G which only contains the switches and the links between them. Using Depth First Search the switches are denoted as $s_0, s_1, ..., s_{|S|-1}$ where s_i is the ith switch discovered in G' as demonstrated in Figure 3.5 . Let the X_i machines connected to switch s_i be numbered as $n_{i,0}, n_{i,1}, ..., n_{i,|X_i|-1}$. If no machine is connected to switch s_i then $|X_i|$ shall be 0. The contention-free logical ring is now $n_{0,0} \rightarrow ... \rightarrow n_{0,|X_0|-1} \rightarrow n_{1,0} \rightarrow ... \rightarrow n_{1,|X_1|-1} \rightarrow ... \rightarrow n_{|S|-1,0} \rightarrow ... \rightarrow n_{|S|-1,|X_{|S|-1}|-1} \rightarrow n_{0,0}$. [52] If the machines run only one process each, X can be directly mapped to the processes. SMP machines can be approximated as another switch connected to the processes only if the processes were assigned consecutive ranks by the MPI library.

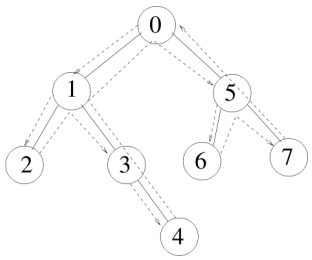

Figure 3.5: Contention-free Ring of Switches (image taken from [6])

3.5.2 Reduce-Scatter

For the reduce-scatter operation, each process splits the input vector in P segments $seg_0, seg_1, ..., seg_{P-1}$. In the first step $rank_i$ sends $seg_{(i-1) \mod P}$ to $rank_{(i+1) \mod P}$. Each process reduces the received segment with its local segment, overwriting the local segment with the result. For the next j steps, $rank_i$ sends $seg_{(i-j) \mod P}$ to $rank_{(i+1) \mod P}$. After $P-1$ steps $rank_i$ has the result of the complete reduction in seg_i of its vector. Figure 3.6 shows an example for 3 processes.

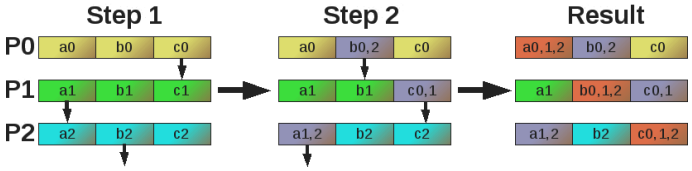

Figure 3.6: Example for a Reduce-Scatter Operation Implemented as Ring

LogGP Prediction In each of the $P-1$ steps each process sends and receives $\frac{m}{P}$ data segments in parallel and reduces $\frac{m}{P}$ data segments. In the first step, all processes can immediately send their segment and receive another in parallel:

$$LogGP_{s=1}(P) = (o_s + o_r + L + (\frac{m}{P} - 1)G)$$

After that, each processes must compute the reduction of the data segment received in the last step and the local data, send the result and receive a new data segment. Each process can start sending after processing the data segment and the send overhead or after waiting for g whichever takes longer. Thus, one intermediate step can be described in LogGP as:

$$LogGP_{1<s<P-1}(P) = \gamma(\frac{m}{P}) + o_r + L + (\frac{m}{P} - 1)G$$

After $P-2$ intermediate steps each process finishes the reduce-scatter by computing the reduction on the last segment:

$$LogGP_{s=P-1}(P) = \frac{m}{P}C$$

Thus, the time needed for the reduce-scatter is:

$$\begin{aligned} LogGP_{red-scat}(P) =& (P-2)(\gamma(\frac{m}{P}) + o_r + L + (\frac{m}{P}-1)G) \\ &+ (o_s + o_r + L + (\frac{m}{P}-1)G + \frac{m}{P}C) \end{aligned}$$

3.5.3 All-Gather

The all-gather algorithm distributes the results of each segment in the same fashion: Starting with $j = 0$ in step j $rank_i$ sends $seg_{(i+j) \mod P}$ to process $rank_{(i-1) \mod P}$. After $P-1$ steps each process obtained the result of the all-reduce operation.

LogGP Prediction The time for the all-gather LogGP is similar to that of the reduce-scatter since it uses the same ring algorithm to gather the results of the reduction. The first step is influenced by the last step of the all-gather part:

$$LogGP_{s=1}(P) = (\gamma(\frac{m}{P}) + o_r + L + (\frac{m}{P}-1)G)$$

All other $P-2$ steps can be carried out in the time:

$$LogGP(P) = (P-2)(\alpha + o_r + L + (\frac{m}{P}-1)G)$$

Thus, the time needed for the all-gather part in the LogGP model is:

$$LogGP_{all-gather}(P) = \gamma(\frac{m}{P}) + (P-2)(\alpha) + (P-1)(o_r + L + (\frac{m}{P}-1)G)$$

3.5.4 Summary

As shown in Section 2.2.1 the latency per byte is smaller for bigger message sizes, thus sending only the segments increases the latency in comparison to algorithms like Recursive Doubling for small and medium messages. For big messages however, the latency is affected much less by splitting up the message in smaller segments. Therefore, this algorithm should be used for big messages.

LogGP Prediction Since the all-gather starts right after the reduce-scatter the fist send of it is still affected by the computation time of the last reduce. Thus, the time needed for the bandwidth optimal algorithm is the sum of the first and intermediate steps of the reduce-scatter algorithm, the first step of the all-gather operation influenced by the last computation and the rest of the all-gather steps: :

$$\begin{aligned} LogGP(P) =& o_s + o_r + L + (\frac{m}{P} - 1)G \\ &+ (P-2)(\gamma(\frac{m}{P}) + o_r + L + (\frac{m}{P} - 1)G) \\ &+ \gamma(\frac{m}{P}) + (P-2)(\alpha) + (P-1)(o_r + L + (\frac{m}{P} - 1)G) \\ \\ =& o_s + \gamma(\frac{m}{P}) + 2(P-1)(o_r + L + (\frac{m}{P} - 1)G) + (P-2)(\alpha + \gamma(\frac{m}{P})) \end{aligned}$$

3.6 Hierarchical and Heterogeneous Algorithms

Modern clusters are built of many SMP machines. This leads to heterogeneity of the message transfer medium. Processes on the same machine share a common memory while processes on two different machines need to communicate over a considerably slower network. For this reason hierarchical algorithms have been proposed. These algorithms use different methods for local and global communication and reduction by combining fast shared memory algorithms with fast network algorithms. For big messages a leader based scheme should perform better than a method which sends from all processes of one node to all processes of another. In the leader based scheme all local processes reduce their buffers before one process per node exchanges the data with other nodes. For small messages the benchmarks in Chapter 2 showed that simultaneous transfers increase the effective bandwidth and cut the latency by up to 38% and thus may reduce the time needed for the all-reduce function.

In 1999 Sistare et al. [53] proposed a shared memory all-reduce algorithm for large-scale SMP machines. Another hierarchical algorithm was developed by Tipparaju et al. [54]. Their shared memory all-reduce is a combination of a shared memory reduce followed by a broadcast.

3.7 Verification of the LogGP Prediction

Since only the Recursive Doubling algorithm is implemented in Open MPI 1.2.8 it was chosen to verify its LogGP model function:

$$LogGP(P) = (\log_2 P - 1)\gamma(m) + o_s + \log_2 P(o_r + L + (m-1)G) + mC$$

In order to measure the time for different message and process sizes the IMB[1] 3.2 was used. After two MPI_Barriers IMB simply measures the time for several consecutive MPI_Allreduce() calls and divides it by the number of performed operations. The used operation is MPI_SUM on double values. The used method leads to shorter execution times in comparison to real applications since the occurrence of process skew is very unlikely and processes may be able start a new all-reduce operation while others have not yet finished the last one. IMB uses the MPI_SUM operation on MPI_FLOAT data. Measurements with the STREAM [45] benchmark compiled with GCC 4.1.2 and $-O3$ show a performance of 2844.44MiB/s on the CHiC and 2942.81MiB/s on the Jack cluster for a summation operation of double values. Thus, the computation time per byte C for the Jack is 0.000324μs and for the CHiC 0.000335μs.

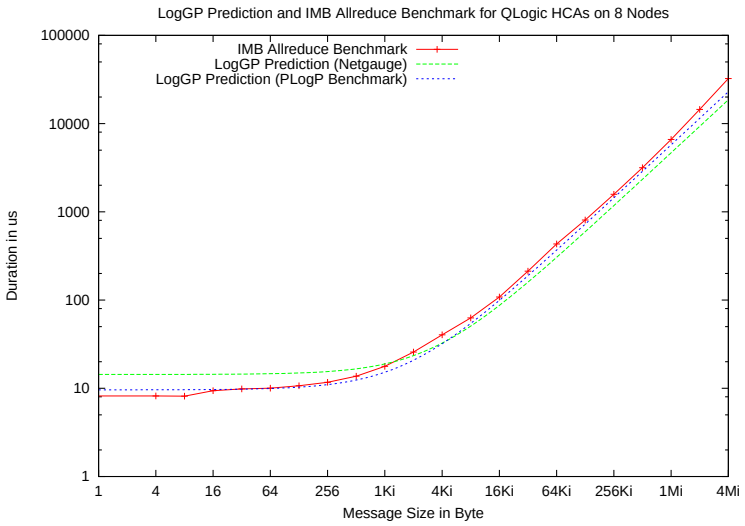

Figure 3.7: Recursive Doubling LogGP Prediction and IMB Benchmark for 8 Processes on 8 Nodes using QLogic InfiniPath QLE7140 HCAs

Figure 3.7 and 3.8 show the LogGP prediction of the Recursive Doubling algorithm for different message sizes between 8 processes on 8 nodes in comparison to the measured run-times by IMB 3.1. The parameters determined by the P-LogP benchmark predict the runtime accurately for message sizes of up to 128 bytes for the QLogic InfiniPath QLE7140 HCAs and up to 64 bytes for the Voltaire HCA 410Ex HCAs. The Netgauge parameters on the other hand lead to a high overassessment of the runtime for messages smaller than 1KiB and 8KiB respectively. For bigger messages the predicted run-time is up to 50% lower than the measured. One big influence in the case of Recursive Doubling is the bi-directional bandwidth. As shown in Chapter

[1]Intel MPI Benchmark `http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm`

2, the bandwidth for one HCA while sending data bi-directionally can be up to 38% smaller than the bandwidth reached while only one HCA is sending. In order to reflect the lower bandwidth the gap per byte G can be increased by 20%. This leads to a more accurate prediction of the run-time.

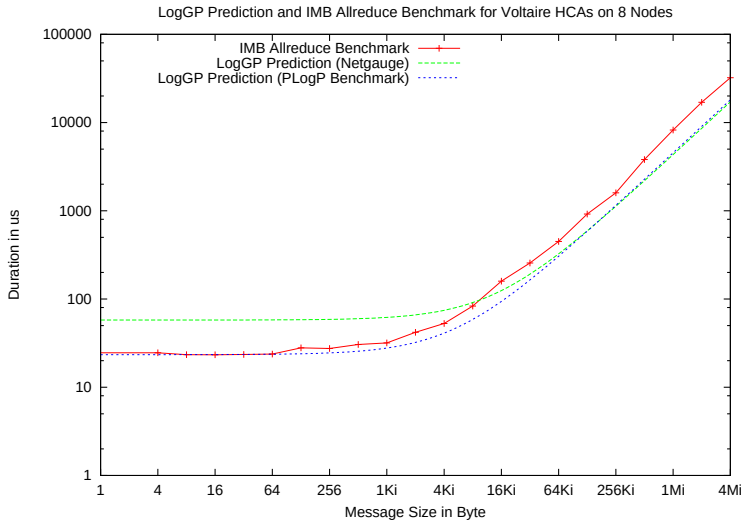

Figure 3.8: Recursive Doubling LogGP Prediction and IMB Benchmark for 8 Processes on 8 Nodes using Voltaire HCA 410Ex HCAs

In Figure 3.9 the prediction and measurement for a message size of 4KiB and different numbers of processes on the CHiC can be seen. 4KiB were chosen since the Netgauge over- and the P-LogP parameters underestimated the run-time for 8 nodes for this message size. As more nodes participate in the operation the prediction of Netgauge becomes more accurate but tends to underestimate the time for more processes. The P-LogP underestimate the time for all numbers of processes. One reason for this is most likely the architecture of the CHiCs InfiniBand^{TM} fabric as a fat-tree. Another reason may have been network contention created by applications of other users on the CHIC.

Summary

While the predicted run-time is not accurate for all message sizes the LogGP function gives a very good description of the behavior of the Recursive Doubling algorithm on two different clusters with different HCAs.

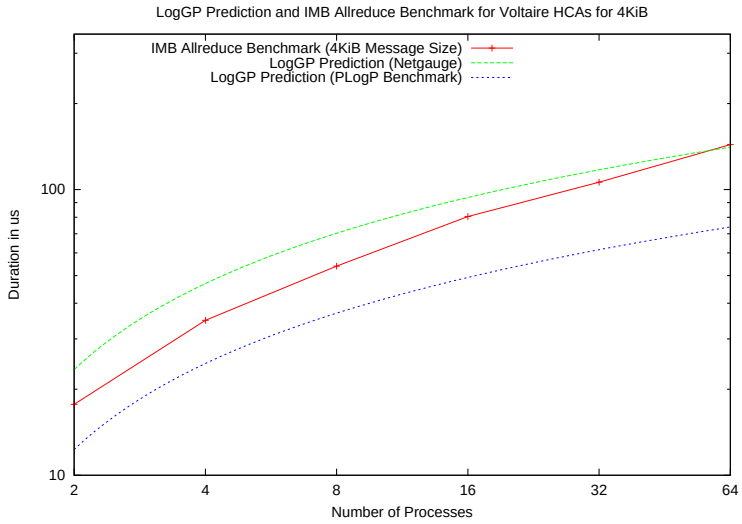

Figure 3.9: Recursive Doubling LogGP Prediction and IMB Benchmark for a Message Size of 4096 bytes using Voltaire HCAs

3.8 Conclusion

Algorithms like Recursive Doubling or Rabenseifners cannot profit from the pipeling of the InfiniBand^{TM} HCAs because of the distributed reduction, which has to be done in each step before another message can be sent. The linear algorithm and in some cases the Binary Blocks algorithm may be able to benefit form the pipeling effects when it synchronizes different blocks.

Table 3.1 gives an overview over the presented all-reduce algorithms and their time in the extended LogGP model. As stated in the beginning of this section there exists no algorithm which is optimal for all message sizes and number of processors. Using the time functions and the LogGP parameters, one can predict the optimal algorithm for all pairs of processes and message sizes. Figure 3.10 shows the optimal algorithm for message sizes between 1 byte and 1MiB, 1 to 550 processes and an multiplication of double values. The Binary Blocks algorithm needs to much time to synchronize in this scenario but is the optimal algorithm in some cases if the accumulation operation is more expansive. As assumed, the linear algorithm is the fastest for a very small number of processes and messages. For message sizes smaller than 1024 bytes, the Recursive Doubling algorithm is the best in most cases. However, with increasing message sizes and process numbers Rabenseifners algorithm is often faster then Recursive Doubling. For message sizes over 1024 bytes Rabenseifners algorithm is slowly superseded by the Bandwidth Optimal algorithm. With bigger message sizes the later is faster for larger numbers of processes. Rabenseifners algorithm is the

fastest in nearly all cases where the number of processes equals a power-of-two.

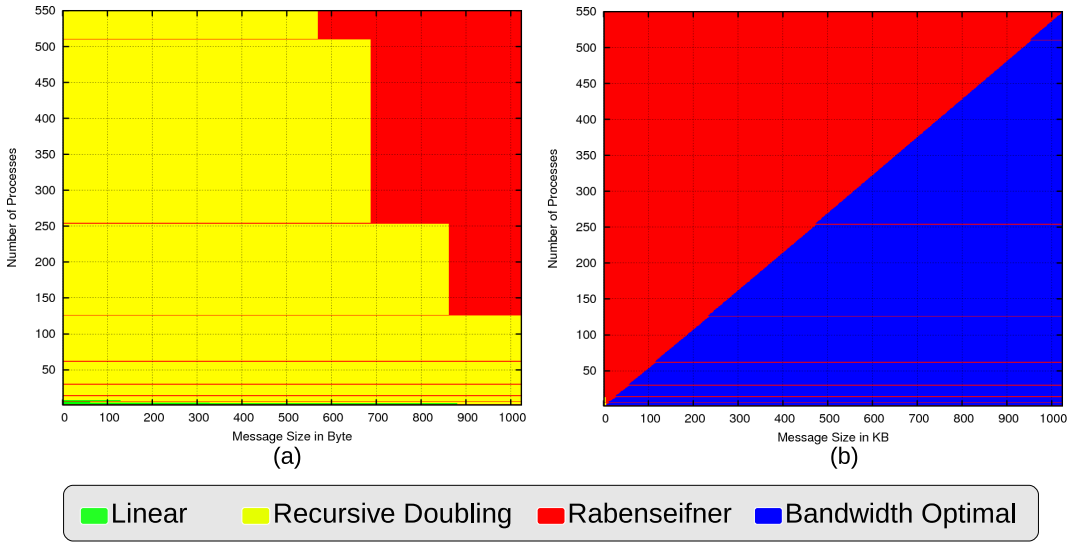

Figure 3.10: LogGP Prediction for Message Sizes between 1 byte and 1KiB (a) and 1KiB and 1MiB (b)

Thus, the MPI implementation should choose the best algorithm for the given parameters. For this purpose Faraj and Yuan proposed an automatic generation and tuning system for MPI collective communication routines [55]. Open MPI supports several tuned algorithms and can choose an optimal one at the time of creation of a communicator or by user interference [56].

Algorithm	$LogGP(P)$
Linear	$o_s + o_r + 2L + \gamma(Pm) + (P-2)\alpha$ $+ (P-1)(o_r + 2(m-1)G)$
Recursive Doubling*	$(\log_2 P - 1)\gamma(m) + o_s$ $+ \log_2 P((m-1)G + o_r + L) + mC$
Rabenseifner*	$o_s + 2(\log_2 P)(o_r + L) + \frac{m}{2^{(\log_2 P)}}G + \frac{m}{2}G$ $+ \gamma(\frac{m}{2^{(\log_2 P)}})$ $+ \sum_{s=2}^{\log_2 P}(\alpha + \gamma(\frac{m}{2^{s-1}}) + 2(\frac{m}{2^s} - 1)G)$
Binary Blocks	$LogGP_{Rabenseifner}(P_{bmax})$ $+ \lceil (b-1)(P'-1)(\gamma(m) + (m-1)\alpha$ $+ (m-1)G + o_r + L) + \gamma(mC)$ $+ (b-2)\alpha + b - 1(L + o_r + 2(\frac{m}{2} - 1)G)\rceil$
Bandwidth Optimal	$o_s + \gamma(\frac{m}{P}) + 2(P-1)(o_r + L + (\frac{m}{P} - 1)G)$ $+ (P-2)(\alpha + \gamma(\frac{m}{P}))$

Table 3.1: LogGP Time of All-reduce Algorithms (* power-of-two case)

4 Reduce Followed by a Practical Constant-Time Broadcast

In 2006 Siebert proposed a practical O(1) broadcast in [48, 57] using the multicast feature of the underlying network. Utilizing this broadcast can result in a very fast all-reduce operation because all processes are ready to receive when the root process finished the reduction operation and thus eliminate the major problem of the proposed multicast broadcast: late processes. The reduction to the root can be implemented in different ways depending on the message size, number of processes and complexity of the reduction. Various reduce algorithms have been proposed and analysed in several studies in the past [4, 42, 47].

4.1 Practical Constant-Time Broadcast

In the past, several broadcast algorithms based on multicast have been proposed [58, 59]. If the used multicast implementation was unreliable the authors proposed acknowledging (ACK) messages to be sent from the peers back to the sender after successfully receiving the message. The sender waits for all ACKs and retransmits the message using reliable P2P communication to peers which did not respond after a certain timeout. Another method are NACK messages which are sent from the receivers to the sender if no message was received in a certain amount of time. Both methods significantly increase the overhead for the broadcast operation and may introduce performance bottlenecks. Another problem is the timeout. Choosing a timeout which is too small results in sending unnecessary message retransmissions. A too big value leads to bad performance.

Therefore, Siebert proposed a broadcast algorithm based on multicast which does not rely on ACK or NACK messages. Confronted with the problem that multicast datagrams often get lost when the receiver is not ready to receive them, Siebert split the broadcast in two steps. A common reason for single processes to not be ready to receive is process skew caused by process scheduling or interrupts [29]. In the first step the root process is delayed by a predetermined amount of time (e.g. user controlled MCA parameter for Open MPI or adaptively adjusted at run-time) to ensure that most of the receiving processes are ready to receive. Another method to ensure that all receiving processes are ready to receive is a barrier operation. However, Sieber found that a barrier would result in a slower broadcast than using a delay[48].

After the delay the root process uses multicast to send the data to all other involved processes. Each process keeps a record of successful received data fragments. The second step insures that all processes which may have missed some fragments receive them now. This is done by the fragmented chain broadcast algorithm which defines a predecessor and successor in a virtual ring topology. Each process sends successfully received fragments to its predecessor as soon as possible. Whether those fragments were received during the multicast stage or from the successor through P2P transfer in the second step does not matter. After all fragments have been received and sent to the predecessor the process can finalize the broadcast and cancel or ignore any outstanding receive requests.

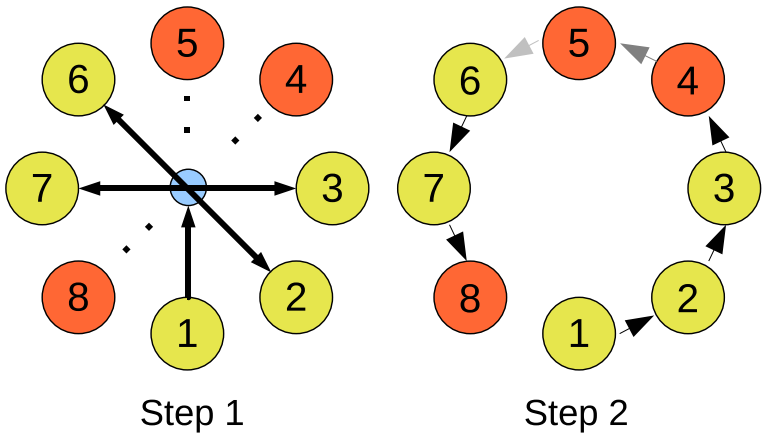

Figure 4.1: Steps of the Broadcast: 1. Multicast 2. Fragmented Chain Broadcast

If a process did not receive the data in the first step it has to wait until its predecessor sent the data in the second step. This operation can be called a penalty round. If the predecessor did also not receive data in the multicast step the number of penalty rounds increases. In the worst case the number of penalty rounds scales with the number of processes. For the all-reduce algorithm this is possible but highly unlikely since all processes must have sent their data to the root process and therefore must already be waiting for the multicast of the result. Thus, the only reasons for the data to not reach single processes is network contention or an error on the hardware level.

The two steps of the algorithm are shown for 8 processes in Figure 4.1. In the first multicast step rank 4,5 and 8 do not receive the multicast. Thus,in the second chained step rank 4 and 8 receive the message in the first penalty round from their predecessor and rank 5 receives the message in the second penalty round from rank 4.

In an all-reduce operation which uses a reduce and this broadcast one can skip the delay of the root process since all other processes must have send a message in

the reduce operation. In this case, the reduce operation acts also as a barrier. A prototype has been implemented and should be described in detail below.

4.2 Prototype Implementation

The prototype is implemented on top of MPI, using the MPI library's MPI_Reduce and non-blocking MPI_Isend and MPI_Irecv functions. Therefore, it can be used with any MPI implementation. The multicast MPI_Bcast code of Höfler and Siebert [57] was used as a starting point and modified for RDMA-CM functions. The prototype provides a struct *rdma_qp_ctx* which serves as context for all operations. The prototype also exports three functions which can be used by any program:

- **RMC_Init(struct rdma_qp_ctx *mc_qp_ctx, MPI_Comm comm)** connects the calling process to a new multicast group as described in Section 1.4.5. The root acts as server. The function initialises and allocates all necessary buffers and preposts receive requests. The receive requests must be posted with a buffer size 40 bytes larger than the actual multicast payload since the GRH is added to the buffer in front of every payload for UD QPs. Thus, the first 40 bytes will be ignored when data in the receive buffer is copied to its destination.
- **RMC_Finalize(struct rdma_qp_ctx *mc_qp_ctx, MPI_Comm comm)** leaves the multicast group, frees the allocated memory and destroys the QPs and CQs.
- **RMC_Allreduce(void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,struct rdma_qp_ctx *mc_qp_ctx)**
  executes the actual all-reduce algorithm. Beside *rdma_qp_ctx*, all parameters are identical to the MPI definition of MPI_Allreduce() (e.g. MPI_IN_PLACE is possible for **sendbuf*). RMC_Allreduce calls MPI_Reduce which returns the result in *recvbufand* of the root. The result is then broadcasted to all other processes using **RMC_Bcast**.

The function **RMC_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm,struct rdma_qp_ctx *mc_qp_ctx)** executes the multicast followed by a fragmented chain algorithm. Beside *rdma_qp_ctx*, all parameters are identical to the MPI definition of MPI_Bcast. Since the size of the send buffer may be bigger than the MTU supported by the used HCA, the algorithm supports sending and receiving of fragments. Each fragment has a status which indicates if the fragment is owned by the process, if it has been received via reliable send and if

it has been sent to the successor. They also have a sequence number and broadcast id attached in order to identify the position and to ensure that no multicast messages from previous broadcasts corrupt the buffer. The root process sends each fragment after attaching the sequence number and broadcast id to the buffer to the multicast group, while all other ranks call the function **recv_send_raw_message**. This function polls for received multicast messages. It also calls MPI_Irecv and, if a fragment has already been received, calls MPI_Isend to send this fragment to the next process. If a fragment has been received through multicast, the sequence number and broadcast id are analyzed and the fragment is then handled accordingly. After the root has finished the multicast send, it also calls **recv_send_raw_message** to send the fragments reliable to rank 1.

4.2.1 Complications

In some cases it is possible that the root node sends faster than a receiving process is able to copy the data of the multicast receive buffer to a local one. In this cases the data of the last sent operation is lost or corrupted and must therefore be received using reliable message transfer. Using multiple multicast receive buffers should help avoid these problems.

4.3 Testing

The prototype was tested using OFED 1.3 and Open MPI 1.2.8 which was configured to use the PSM library to send messages over the QLogic InfiniPath QLE7140 HCAs. Apart from the complications mentioned above, the tests where successful.

5 Conclusion and Future Work

This thesis analysed the QLogic InfiniPath QLE7140 HCA and its onload architecture and compared the results to the Mellanox InfiniHost III Lx HCA which uses an offload architecture. As expected, the QLogic InfiniPath QLE7140 HCA can outperform the Mellanox InfiniHost III Lx HCA in latency and bandwidth terms on our test system in various test scenarios. The benchmarks showed, that sending messages with multiple threads in parallel can increase the bandwidth greatly while bi-directional sends cut the effective bandwidth for one HCA by up to 30%. Thus, collective operations for InfiniBand™ should be made aware of SMP machines and none full bi-directionally bandwidth.

Different all-reduce algorithms where evaluated and compared with the help of the LogGP model. The comparison showed that new all-reduce algorithms can outperform the ones already implemented in Open MPI for different scenarios. Especially the bandwidth optimal algorithm proposed by Patarasuk and Yuan shows great potential for big messages and should therefore be tested and implemented in Open MPI.

The thesis showed that one can implement multicast algorithms for InfiniBand™ easily by using the RDMA-CM API. However, the author was not able to find a complete documentation of all functions and properties of the RDMA-CM API. Thus, for a better understanding and faster implementation of algorithms based on the RDMA-CM API a documentation needs to be written. The implemented prototype needs to be further optimized and analyzed.

Throughout this work some Open MPI all-reduce COLL component algorithms were developed but in the end discarded, since they did not proof to be faster than currently available algorithms.

Acronyms

ACK Acknowledged

AH Address Handler

API Application Programming Interface

ASIC Application Specific Integrated Circuit

BML Byte Management Layer

BTL Byte Transfer Layer

CM Connection Manager

COLL Collective Component

CQ Completion Queue

CQE Completion Queue Element

DDR Double Data Rate

EEL End-to-End Latency

EP End Point

GID Global ID

HCA Host Channel Adapters

HPC High Performance Computing

IBA InfiniBand^{TM} Architecture

IMB Intel MPI Benchmark

LID Local ID

MPI Message Passing Interface

MQ Matched Queue

MTL Matched Transport Layer

NACK Not Acknowledged

NAK Not Acknowledged

NDA Non-Disclosure Agreement

OFA OpenFabrics Alliance

OFED OpenFabrics Enterprise Distribution

OMB OSU MPI Benchmark

OMPI Open MPI layer

OPAL Open Portability Access Layer

ORTE Open Run-Time Environment

OSU Ohio State University

P2P Point to Point

PD Protection Domain

PML Point-to-point Messaging Layer

PRTT Parameterized Round Trip Time

QPN Queue Pair Number

QP Queue Pair

RAW Raw Datagram

RC Reliable Connection

RDMA-CM RDMA Connection Manager

RDMA Remote Direct Memory Access

RST Reset

RTR Ready to Receive

RTS Ready to Send

RTT Round Trip Time

SDR Single Data Rate

SERDES Serializer/Deserializer

SM Shared Memory

TCA Target Channel Adapters

UC Unreliable Connection

UD Unreliable Datagram

WRE Work Queue Element

WR Work Request

Bibliography

[1] Infiniband Trade Association. InfiniBand Architecture Specification Volume 1 Release 1.2.1. 2008.

[2] QLogic Corporation. InfiniPath QLE7149 Data Sheet. White paper,http://www.qlogic.com/SiteCollectionDocuments/Education%20and%20Resource/Datasheets/HPC/InfiniBand%20HCAs/QLE7140_datasheet.pdf.

[3] Mellanox Technologies. InfiniHost III Lx Data Sheet. White paper,http://www.mellanox.com/pdf/products/silicon/InfiniHostIII_Lx.pdf.

[4] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005.

[5] Rolf Rabenseifner. Optimization of Collective Reduction Operations. In *Computational Science - ICCS 2004, Springer-Verlag LNCS 3036*, pages 1–9, 2004.

[6] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117 – 124, 2009.

[7] Rolf Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. In *PVM/MPI*, pages 35–42, 1999.

[8] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *www.mpi-forum.org*, 1995.

[9] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. *www.mpi-forum.org*, 1997.

[10] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse Non-Blocking Collectives in Quantum Mechanical Calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 55–63. Springer, Sep. 2008.

[11] Jeffrey S. Vetter. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *In International Parallel and Distributed Processing Symposium*, 2002.

[12] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[13] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *In Proceedings, 10th European PVM/MPI Users' Group Meeting, number 2840 in Lecture Notes in Computer Science*, pages 379–387. Springer-Verlag, 2003.

[14] Richard L. Graham, Sung eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski Introduction. A Network-Failure-tolerant Message-Passing system for Terascale Clusters. In *International Journal of Parallel Programming*, pages 77–83, 2003.

[15] Graham E. Fagg, Edgar Gabriel, Zizhon Chen, Thara Angskun, George Bosilca, Antonin Bukovsky, and Jack J. Dongarra. Fault tolerant communication library and applications for high perofrmance. In *In Los Alamos Computer Science Institute Symposium*, pages 27–29, 2003.

[16] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. The open run-time environment (openrte): A transparent multi-cluster environment for high-performance computing. In *In Proceedings 12th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Sorrento(Naples)*. Springer-Verlag, 2005.

[17] Brian Barrett, George Bosilca, Rich Graham, Galen Shipman, Tim Woodall, and Jeff Squyres. Open MPI Developer's Workshop. April 17-20, 2006.

[18] Richard L. Graham, Brian Barrett, Galen M. Shipman, Timothy S. Woodall, and George Bosilca. Open MPI: a High Performance, Flexible Implementation of MPI Point-to-Point Communications. *Parallel Processing Letters*, 17(1):79–88, 2007.

[19] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the Component Architecture Overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[20] Ron Brightwell and Keith Underwood. Evaluation of an eager protocol optimization for MPI. In *In Proceedings of EuroPVM/MPI*, pages 327–334, 2003.

[21] Tom Shanley. *InfiniBand Network Architecture*. MindShare Inc, 2002.

[22] Mellanox Corporation. InfiniHost III Ex MemFree Mode Performance. White paper, http://www.mellanox.com/pdf/whitepapers/PCIxVsMemfree_WP_100.pdf.

[23] Gilad Shainer. Cluster Interconnects: Single Points of Performance . White paper,http://clustermonkey.net/download/whitepapers/Single-points-of-performance.pdf, 2007.

[24] Gilad Shainer. Cluster Interconnects: Real Application Performance and Beyond. White paper,http://clustermonkey.net/download/whitepapers/RealApplicationPerformanceandBeyond.pdf, 2007.

[25] Greg Lindahl. A Preliminary Report on Application Performance using Intel Woodcrest Processors and QLogic InfiniPath InfiniBand Adapters. White paper,http://www.qlogic.com/SiteCollectionDocuments/Education_and_Resource/whitepapers/whitepaper1/Performance_Report.pdf, 2006.

[26] Greg Lindahl. Real Application Performance with the QLogic InfiniPath InfiniBand Interconnect. White paper,http://www.qlogic.com/SiteCollectionDocuments/Education_and_Resource/whitepapers/whitepaper1/RealApplications-paper.pdf, 2005.

[27] QLogic Corporation. QLogic InfiniPath SDR Adapters Outperform Mellanox DDR Adapters. White paper,http://www.qlogic.com/SiteCollectionDocuments/Education_and_Resource/whitepapers/whitepaper1/HSG-WP07005.pdf, 2007.

[28] QLogic Corporation. Psm programmer's manual version 1.05. Available from QLogic Corporation under NDA, 2008.

[29] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. *Cluster Computing, IEEE International Conference on*, 0:1–12, 2006.

[30] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.*, 20(3):389–398, 1994.

[31] Rolf Rabenseifner and Jesper Larsson Träff. More Efficient Reduction Algorithms for Non-power-of-two Number of Processors in Message-Passing Parallel Systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface,Springer-Verlag LNCS 3241*, pages 36–46, 2004.

[32] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP:

towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[33] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. BSP vs LogP. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 25–32, New York, NY, USA, 1996. ACM.

[34] Bernaschi Massimo and Ianello Giulio. Collective communication operations: experimental results vs. theory. *Concurrency: Practice and Experience*, 10(5):359–386, 1998.

[35] Torsten Hoefler and Wolfgang Rehm. A Communication Model for Small Messages with InfiniBand. In *PARS Proceedings*, 2005.

[36] Thilo Kielmann, Henri E. Bal, and Sergei Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. In *In Proc. International Parallel and Distributed Processing Symposium (IPDPS 2000), Cancun*, pages 492–499. IEEE, 2000.

[37] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, New York, NY, USA, 1995. ACM.

[38] Thilo Kielmann and Henri E. Bal. Fast Measurement of LogP Parameters for Message Passing Platforms. In *Lecture Notes in Computer Science*, pages 1176–1183. Springer-Verlag, 2000.

[39] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks, 2003. In IPDPS 2003.

[40] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[41] T. Hoefler. Evaluation of publicly available Barrier-Algorithms and Improvement of the Barrier-Operation for large-scale Cluster-Systems with special Attention on InfiniBand Networks, Apr. 2005.

[42] M. Franke. Optimierte Implementierung ausgewaelter kollektiver Operationen unter Ausnutzung der Hardwareparallelitaet des InfiniBand Netzwerkes, Apr. 2007.

[43] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. LogfP - a model for small messages in InfiniBand. *Parallel and Distributed Processing Symposium, International*, 0:367, 2006.

[44] R. Kumar, A. Mamidala, and D. K. Panda. Scaling Alltoall Collective on Multi-core Systems. *Workshop on Communication Architecture for Clusters, in conjunction with IPDPS '08, Miami, USA*, 2008.

[45] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream/`.

[46] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2003. IEEE Computer Society.

[47] C. Virtel. Improving the Performance of Selected MPI Collective Communication Operations on InfiniBand Networks, institution=Technical University of Chemnitz, Apr. 2007.

[48] Christian Siebert. Efficient Broadcast for Multicast-Capable Interconnection Networks, 09 2006.

[49] Amith R Mamidala, Jiuxing Liu, and Dhabaleswar K Panda. Efficient Barrier and Allreduce InfiniBand Clusters using Hardware Multicast and Adaptive Algorithms. In *In Proceedings of Cluster Computing*, 2004.

[50] P.M. Kogge and H.S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. In *IEEE Trans. Computers, Vol. C-22, No. 8*, pages 786–793, 1973.

[51] Robert A. van de Geijn. On global combine operations. *J. Parallel Distrib. Comput.*, 22(2):324–328, 1994.

[52] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. Bandwidth Efficient All-to-All Broadcast on Switched Clusters. *International Journal of Parallel Programming*, 36(4):426–453, 2008.

[53] Steve Sistare, Rolf vandeVaart, and Eugene Loh. Optimization of MPI collectives on clusters of large-scale SMP's. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 23, New York, NY, USA, 1999. ACM.

[54] Dhabaleswar Panda Vinod Tipparaju, Jarek Nieplocha. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *In International Parallel and Distributed Processing Symposium*, 2003.

[55] Ahmad Faraj and Xin Yuan. Automatic Generation and Tuning of MPI Collective Communication Routines. In *19th ACM International Conference on Supercomputing (ICS05*, pages 393–402. ACM Press, 2005.

[56] Graham E. Fagg, Jelena Pjesivac-grbovic, George Bosilca, Jack J. Dongarra, and Emmanuel Jeannot. Flexible collective communication tuning architecture applied to open MPI. In *In 2006 Euro PVM/MPI*, 2006.

[57] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, page 232. IEEE Computer Society, Mar. 2007.

[58] Benoit Hudzia and Serge Petiton. Reliable multicast fault tolerant MPI in the Grid environment, 2004.

[59] Jiuxing Liu, Amith R Mamidala, and Dhabaleswar K Panda. Fast and Scalable MPI-Level Broadcast using InfiniBands Hardware Multicast Support. In *In Proceedings of IPDPS*, 2004.

Appendix A

Benchmarks

A.1 Implementation Notes for the Benchmarks

For native PSM or OpenIB benchmarks the out of band MPI_Barrier synchronization (in this case over ethernet) proved to influence the benchmarks heavily because of high latency of the Ethernet network. Therefore, the synchronization has been made within the benchmarked network itself.

Using preposted receive requests for OpenIB reduced the RTT significantly compared with the RTT passed when the receive requests where posted only when a incoming send was expected. For the Mellanox InfiniHost III Lx HCA using the inline send feature of small messages reduced the RTT further.

Acknowledgements

I want to tank my advisor Frank Mietke for his support and commitment. I also want to thank my supervisor Professor Dr.-Ing. Wolfgang Rehm for his support. I want to thank Torsten Mehlan, René Oertel, Torsten Höfler, Christian Siebert and Simon Wunderlich for the useful hints and Dr. Stuart Rankin for running my benchmarks on David. Last but not least I want to thank my friends for there social support and hints as well as my family for their social support, help and financial support during my study.

Thesis Declaration

I hereby declare that this diploma thesis is my own work and has not been submitted in any form for another degree or diploma at any other institution. Information derived from the work of others has been acknowledged and referenced.

Chemnitz, March 31, 2009

Nico Mittenzwey