



CHEMNITZ UNIVERSITY OF TECHNOLOGY

---

Faculty of Computer Science

# Diploma Thesis

Improving the Performance of Selected MPI Collective  
Communication Operations on InfiniBand Networks

Carsten Viertel

Chemnitz, April 30, 2007

**Supervisor:** Prof. Dr.-Ing. W. Rehm  
**Advisor:** Dipl.-Inf. Torsten Hoefler

# Contents

<b>List of Figures</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 InfiniBand</b>	<b>2</b>
2.1 Introduction . . . . .	2
2.2 Components and Topology . . . . .	2
2.3 Features . . . . .	3
2.3.1 Queuing . . . . .	3
2.3.2 Operations . . . . .	4
2.3.3 Types of Service . . . . .	4
2.3.4 Addressing . . . . .	5
2.4 Verbs API . . . . .	5
<b>3 Open MPI</b>	<b>6</b>
3.1 Introduction . . . . .	6
3.2 Component Architecture . . . . .	6
3.2.1 Point-to-Point Communication Components . . . . .	7
3.2.2 Collective Component . . . . .	7
3.3 Conclusion . . . . .	9
<b>4 Model</b>	<b>10</b>
4.1 Introduction . . . . .	10
4.2 Overview of Models . . . . .	10
4.2.1 PRAM . . . . .	10
4.2.2 BSP . . . . .	10
4.2.3 LogP . . . . .	11
4.2.4 Conclusion . . . . .	11
4.3 LogP Models . . . . .	12
4.3.1 Overview . . . . .	12
4.3.2 Extended Models . . . . .	13
4.3.3 Measuring LogGP Parameters . . . . .	15

<b>5 Algorithms</b>	<b>19</b>
5.1 MPI_Scatter . . . . .	19
5.1.1 Definition . . . . .	19
5.1.2 Linear Algorithm . . . . .	20
5.1.3 Hierarchical Algorithms . . . . .	21
5.1.4 Conclusion . . . . .	26
5.2 MPI_Gather . . . . .	29
5.2.1 Definition . . . . .	29
5.2.2 Algorithms . . . . .	29
5.2.3 Conclusions . . . . .	29
5.3 MPI_Allgather . . . . .	30
5.3.1 Definition . . . . .	30
5.3.2 Algorithms . . . . .	30
5.3.3 Conclusion . . . . .	35
<b>6 Implementation Details</b>	<b>36</b>
6.1 Introduction . . . . .	36
6.2 Initialization . . . . .	36
6.3 Memory Registration . . . . .	37
6.4 Rendezvous Protocol . . . . .	38
6.5 Eager Protocol . . . . .	38
6.6 Known Issues . . . . .	40
6.7 Implemented Algorithms . . . . .	41
<b>7 Benchmark Results</b>	<b>42</b>
7.1 Introduction . . . . .	42
7.2 MPI_Scatter and MPI_Gather . . . . .	42
7.3 MPI_Allgather . . . . .	44
<b>8 Conclusion and Future Work</b>	<b>47</b>
<b>Bibliography</b>	<b>48</b>
<b>A Testing Environments</b>	<b>52</b>
A.1 OSCAR . . . . .	52
A.2 CHiC . . . . .	52
<b>B Glossary</b>	<b>53</b>
<b>C Theses</b>	<b>55</b>
<b>D Acknowledgements</b>	<b>56</b>

## List of Figures

2.1	InfiniBand Channel Adapters and Fabric . . . . .	2
2.2	InfiniBand Consumers and Queues . . . . .	3
3.1	Open MPI Modular Component Architecture . . . . .	6
3.2	Open MPI Point-to-point Component Frameworks . . . . .	7
3.3	Open MPI Collective Component Life Cycle . . . . .	8
4.1	Sending a Message in the LogP Model . . . . .	12
4.2	Sending and Receiving on Multiple Nodes in the LogP Model . . . . .	13
4.3	Sending and Receiving on Multiple Nodes in the LogGP Model . . . . .	14
4.4	Gap Parameters and Resulting Function . . . . .	18
4.5	PRTT(1,0,s) and LogGP prediction . . . . .	18
5.1	Data Distribution before and after Scatter on 6 Processes . . . . .	19
5.2	Scatter using a Recursive Splitting Algorithm on 8 Processes . . . . .	21
5.3	Binomial Tree of Order 3 . . . . .	23
5.4	Tree of Order $i > 2$ and with a per step Fanout of $k > 1$ . . . . .	24
5.5	N-Way Binomial Tree Scatter with a Step Fanout of 2 on 9 Processes . . . . .	24
5.6	Schematic of the Scatter Algorithm on 9 Nodes using a per step Fanout of 2 . . . . .	26
5.7	Scatter LogfP Comparison . . . . .	27
5.8	Scatter LogfP expected Communication Times . . . . .	28
5.9	Data Distribution before and after Gather on 6 Processes . . . . .	29
5.10	Data Distribution before and after Allgather on 6 Processes . . . . .	30
5.11	Allgather using a Ring Algorithm on 5 Processes . . . . .	31
5.12	Allgather using a Modified Ring Algorithm on 5 Processes . . . . .	32
5.13	Allgather using a Neighbor Exchange Algorithm on 6 Processes . . . . .	32
5.14	Allgather using a Recursive Doubling Algorithm on 8 Processes . . . . .	33
5.15	Allgather using the Bruck Algorithm on 6 Processes . . . . .	34
6.1	Usage of the Ranks in MPI_COMM_WORLD to access Queue Pairs . . . . .	37
6.2	Rendezvous Protocol . . . . .	38
6.3	Eager Protocol . . . . .	39
7.1	Gather Performance Scaling with Communicator Size . . . . .	42
7.2	Gather Performance Scaling with Buffer Size . . . . .	43

LIST OF FIGURES

---

7.3	Gather Performance compared to Open MPI and MVAPICH . . . . .	43
7.4	Allgather Performance Scaling with Communicator Size . . . . .	44
7.5	Allgather Performance Scaling with Buffer Size . . . . .	45
7.6	Allgather Performance compared to Open MPI and MVAPICH . . . . .	45
7.7	Allgather Performance compared to Open MPI and MVAPICH . . . . .	46

# 1 Introduction

The performance of collective communication operations is one of the deciding factors in the overall performance of a MPI application. A lot of research has gone into different algorithms, but the implementation of these often has been lacking behind. Open MPI's component architecture offers an easy way to implement new algorithms for collective operations. Current implementations use the point-to-point components to access the InfiniBand network. Therefore it is tried to improve the performance of a collective component by accessing the InfiniBand network directly. This should avoid overhead and make it possible to tune the algorithms to this specific network.

Chapter 2 and 3 give a short overview of the InfiniBand network and Open MPI. In chapter 4 several models for parallel computation are analyzed and a fitting model is chosen. The following chapter deals with the different algorithms for MPI\_Scatter, MPI\_Gather and MPI\_Allgather. The algorithms are analyzed and their theoretical performance is assessed. In Chapter 6 the practical implementation with a focus on the InfiniBand network is described. The practical performance of the algorithms is analyzed and compared to existing implementations in chapter 7. Finally in chapter 8 the results of this work are summarized.

## 2 InfiniBand

### 2.1 Introduction

The InfiniBand Architecture [IBA] was originally designed by the InfiniBand Trade Association<sup>1</sup> as a general I/O technology. Today it is often used as interconnection network for high performance computing.

### 2.2 Components and Topology

The InfiniBand Architecture uses point-to-point connections to transfer data between endpoints which can be anything from I/O devices to host computers. Switches and routers are used to connect the endpoints. The InfiniBand Channel Adapters in the endpoints are called Host Channel Adapters (HCA) if they are in a processor node and Target Channel Adapters (TCA) if they are in an I/O node. Anything connecting the Channel Adapters is referred to as Fabric (see figure 2.1).

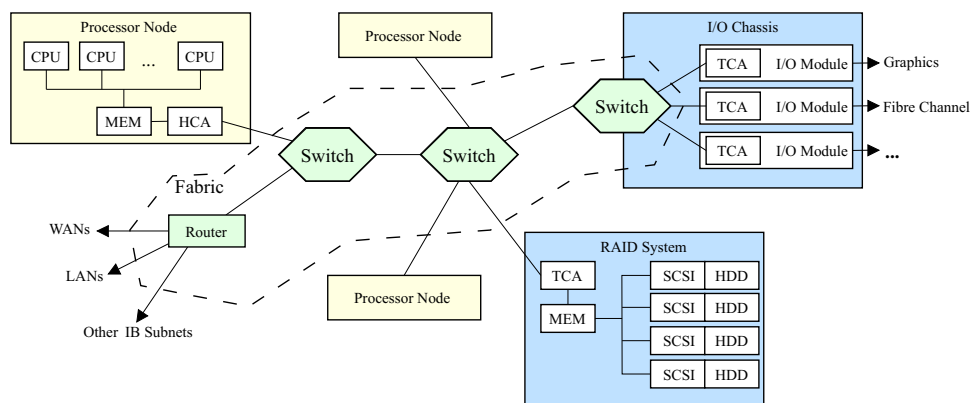


Figure 2.1: InfiniBand Channel Adapters and Fabric

<sup>1</sup><http://www.infinibandta.org/>

## 2.3 Features

### 2.3.1 Queuing

One of the key features of the InfiniBand Architecture is the use of hardware queues. All Channel Adapters have various work queues. This allows a consumer (e.g. an application) to post operations that the Channel Adapter can afterward execute. Some possible operations are Send, RDMA-write and RDMA-read on the Send Queue (SQ) and a receive operation on the Receive Queue (RQ). After the posted operation is finished the Channel Adapter will eventually put a Completion Queue Element (CQE) on a Completion Queue (CQ).

Submitting an operation to the Channel Adapter is achieved by posting a so called Work Request (WR) to one of the Work Queues (WQ). This creates a Work Queue Element in the Work Queue. The Channel Adapter will process the Work Queue Element (WQE) and create a Completion Queue Element (CQE). The Completion Queue Element contains all necessary information for work completion.

A consumer can create multiple Send and Receive Queues. These Work Queues can be attached to one or many Completion Queues where the Completion Queue Elements will be placed. Figure 2.2 shows some Queues and how they relate to each other and different Consumers.

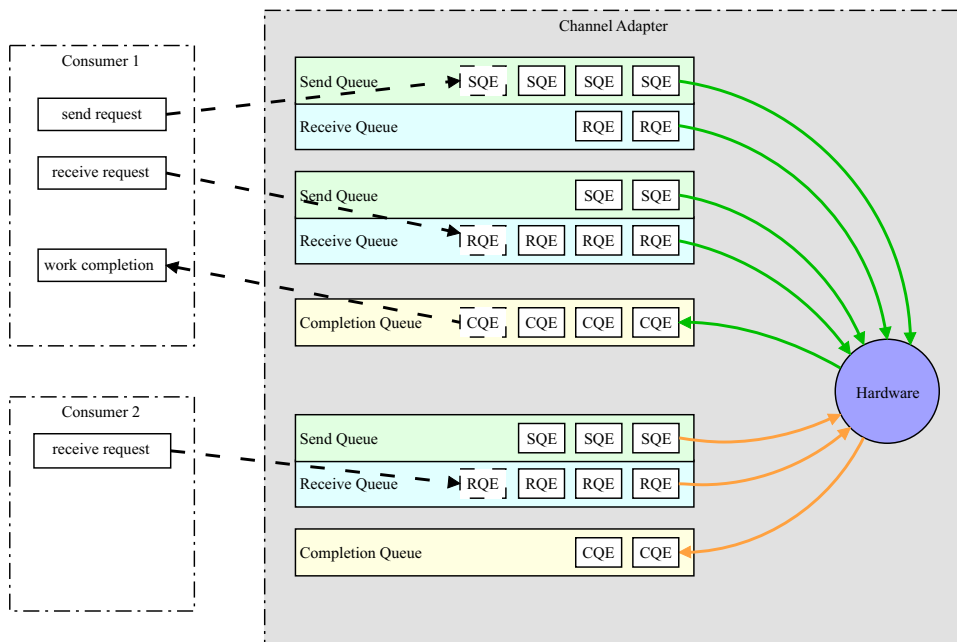


Figure 2.2: InfiniBand Consumers and Queues

### 2.3.2 Operations

As stated earlier there are several possible operations on a work queue. For the Send Queue these operations are Send, RDMA operations and Memory Binding.

The Send operation is used to send a message to a remote consumer. Local address and length of the data to send have to be supplied as arguments. A Send operation makes it always necessary that the remote consumer posts a Receive operation to the appropriate Receive Queue.

The RDMA operations are RDMA-write, RDMA-read and Atomic. As an additional argument the remote address for the operation has to be specified. The RDMA-write operation takes local data and transfers it to the remote address. RDMA-read transfers data from the remote address to the local address. Atomic is used to read a 64bit memory location and eventually update its value (swap if equal, add).

The Memory Binding operation is used to register a memory region with the Channel Adapter. The operation is used to share memory with remote nodes and it returns two keys. The l\_key which is needed by the Channel Adapter for local access to the registered memory and the r\_key which is needed by remote consumers.

For the Receive Queue only the Receive operation is available. It requires a local address and the associated l\_key as arguments and is used to specify where the Channel Adapter puts incoming data from remote Send operations.

### 2.3.3 Types of Service

InfiniBand offers various Types of Service that effect how the Queue Pairs behave. Table 2.1 gives a short overview.

Service Type	Connection Oriented	Acknowledged	Transport
Reliable Connection	yes	yes	IBA
Unreliable Connection	yes	no	IBA
Reliable Datagram	no	yes	IBA
Unreliable Datagram	no	no	IBA
RAW Datagram	no	no	RAW

Table 2.1: InfiniBand Types of Service

If the Queue Pairs are connection oriented then each Queue Pair is connected with one other Queue Pair. Datagram Queue Pairs can send to and receive from many other datagram Queue Pairs.

Queue Pairs can be configured to acknowledge transmissions (ACK, NAK or response data) and therefore guarantee data delivery. These Queue Pairs are referred to as reliable. Unreliable Queue Pairs only guarantee that the actually delivered data is uncorrupted.

A Queue Pair may use the transport protocol defined by the InfiniBand Architecture or it can just use RAW packets to send data.

### 2.3.4 Addressing

Every endpoint may have several Channel Adapters which can have several ports. Every port has a Local ID (LID) that is assigned by the Subnet Manager. Each port also has a global ID (GID). A LID is unique in the local subnet and is used by the switches to route the packets within the subnet. The GID is globally unique and is used in an optional Global Route Header (GRH) processed only by routers. Each Queue Pair is identified by a Queue Pair Number (QPN).

As connection oriented Queue Pairs are connected to exactly one other Queue Pair, the remote Queue Pairs LID, QPN and optionally GID have to be supplied as arguments at Queue Pair creation. For unconnected Queue Pairs the addressing information is part of each Send Queue Element.

## 2.4 Verbs API

The InfiniBand Architecture Specification includes Software Transport Verbs. These Verbs define how the user should be able to interact with a Host Channel Adapter. The Verbs describe for example how to post send and receive requests, modify Queue Pair properties or register a memory region. While it isn't an API it strongly influenced the way existing APIs were designed.

The most important Verbs APIs are the OpenFabrics Verbs API<sup>2</sup> (previously OpenIB Verbs API) and the Mellanox Verbs API<sup>3</sup>. To use both APIs without the need to write the code twice a unified API utilizing the C pre-processor [Mos06] is used. In a separate header file for OpenFabrics and MVAPI a common name for each datatype, variable and function is defined. The user only has to include the main header file and a define in his program to choose at compile time which underlying API is used.

---

<sup>2</sup><http://www.openfabrics.org/>

<sup>3</sup><http://www.mellanox.com/>

## 3 Open MPI

### 3.1 Introduction

Open MPI <sup>1</sup> is a relatively new open-source MPI-2 implementation. The developers previously worked on the LAM/MPI, LA-MPI and FT-MPI projects.

Open MPI offers a wide range of features [GFB<sup>+</sup>04, GWS05] like support for various network interconnects (TCP/IP, Myrinet, InfiniBand), network and process fault tolerance and support for heterogeneous networks. One of the key features of Open MPI is the component architecture, which provides component frameworks for point-to-point communication, collective algorithms and other parts of a MPI implementation.

In the following section it is described how the component architecture and existing collective components work.

### 3.2 Component Architecture

The Open MPI Modular Component Architecture (MCA) manages various component frameworks for each major functional area in Open MPI. A component framework offers a single functionality, such as point-to-point data transfer, reduction operations or collective communication. Each component framework can use multiple components (see figure 3.1).

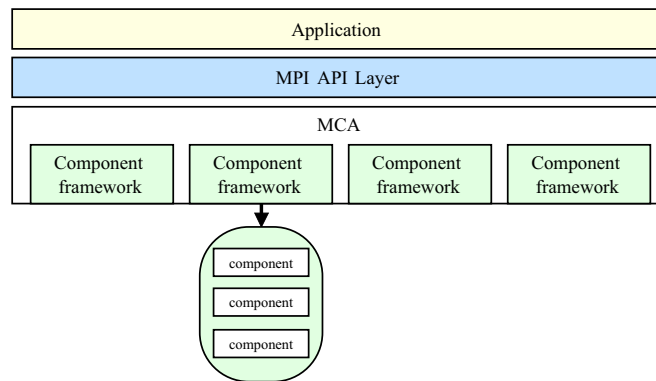


Figure 3.1: Open MPI Modular Component Architecture

---

<sup>1</sup><http://www.open-mpi.org/>

The component framework is responsible for finding, loading, using and unloading the components. Components export a well-defined interface and can be loaded on demand at runtime. Once a component is initialized it is called a module.

While the collective component framework (coll) is the most important framework for this work, existing collective components utilize the components for point-to-point communication. Therefore these are described in the next section.

### 3.2.1 Point-to-Point Communication Components

The point-to-point communication components [SWG<sup>+</sup>06] are the Point-to-point Messaging Layer (PML), the BTL Management Layer (BML) and the Byte Transfer Layer (BTL).

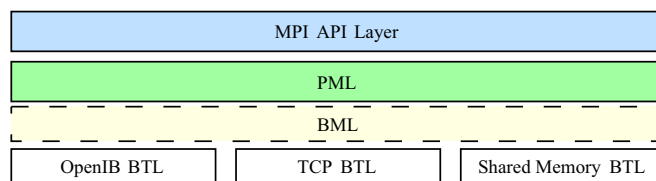


Figure 3.2: Open MPI Point-to-point Component Frameworks

The PML offers all point-to-point functionality needed by higher level MPI functions. It is responsible for the fragmentation and scheduling of messages. The PML also selects which BTL is used to send a message to a specific destination. The BML is mainly responsible for the discovery of peer resources and can afterward be bypassed. A BTL is used for accessing the underlying network. Several BTLs for various networks like Myrinet (GM BTL) and InfiniBand (MVAPI and OpenIB BTL) are currently available.

An analysis of the component architecture overhead [BSL<sup>+</sup>05] showed that it does introduce only very little measurable overhead.

### 3.2.2 Collective Component

The Open MPI collective component framework [SL04] offers an easy way to add new algorithms and technologies for MPI collective functions by just adding a new collective component.

One of the main goals of the coll framework is to offer an interface that makes it possible to implement new collective routines without knowing too much about the internal workings of the MPI implementation. The new collective routines can use MPI point-to-point functions, other collective components or they can access the hardware directly. It can be decided at run-time which of the available collective components are used. Multiple components can be used by one MPI process. An initialized collective component is called a module and it is always tied to a communicator.

The MPI collective functions are implemented in Open MPI as thin wrapper functions, that just perform error checking and then call the function actually implementing the collective operation located in the selected collective component. Therefore the collective component contains a list of function pointers.

It is possible to implement only selected collective functions in a component. For the missing functions the "basic" component can be used. That makes it possible to optimize a few collective functions and still have a fully working MPI implementation.

### Collective Component Lifecycle

A collective component's lifecycle consists of five different phases (see figure 3.3).

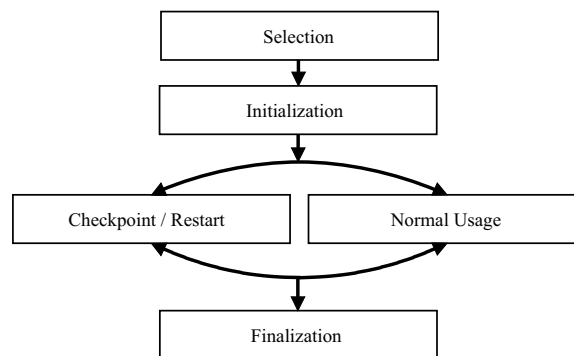


Figure 3.3: Open MPI Collective Component Life Cycle

As a communicator is created (MPI\_INIT, MPI\_COMM\_CREATE, etc.) one of the available collective components is *selected*. Therefore each component's query function is called. In this function the component may analyze various factors that determine its expected performance. The query function then sets a priority value ranging from 0 to 100 which is used by the coll framework to determine which of the collective components should be used by the communicator. The query function returns a module data structure if it wants to be considered for selection. Otherwise it returns NULL. The user may force the selection of a specific collective component by changing the priority with a command line parameter at the start of a MPI application or in a config file.

After a module has been selected its *initialization* function is called. The module now performs all setup operations needed for the supplied communicator. Depending on how the component is implemented that can for example mean setting up network connections for all ranks in the communicator, setting up shared memory regions or pre-computing communication patterns to improve performance. All necessary module data is cached on the communicator itself. Now every time one of the module's functions is called a communicator is supplied as an argument and the module can access its data.

Support of *checkpoint/restart* functionality is optional for a collective component. If the component is using MPI point-to-point functions then the point-to-point components usually take care of it. Components that use their own communication channels need to offer functions which are called to do the necessary cleanup during checkpointing and re-initialize the module during restart. If only one component that may be used by a MPI process isn't supporting checkpoint and restart the MPI process can't be checkpointed.

In the *normal usage* phase the module's collective functions are called for every MPI collective operation on the communicator the module was initialized with.

The last phase in the lifecycle of a coll module is the *finalization*. In it the module has to clean up all previously created resources (e.g free allocated memory, network connections, etc.) for the supplied communicator.

## Components

Various collective components already have been created. Some examples are the *basic*, *tuned* and *smp* components.

The *basic* component offers a simple implementation of all MPI collective functions. It can be used for any communicator regardless of underlying topology. Point-to-point functions are used to implement the collective operations. The *tuned* component offers improved algorithms. The *smp* component offers algorithms to maximize the performance of the collective operations on symmetric multiprocessing architectures by creating sub communicators to separate local and global peers.

## 3.3 Conclusion

Open MPI with its modular component architecture provides a solid base to implement and test new collective algorithms. The point-to-point infrastructure introduces very little overhead and together with the coll framework makes it easy to implement new collective functions layered on top of it.

Accessing the InfiniBand network with the Mellanox or OpenIB Verbs API still can improve the performance of collective operations, because the algorithms can be better tuned to the network. Also it offers the possibility to use special InfiniBand functionality like multicast.

## 4 Model

### 4.1 Introduction

Optimizing an algorithm for a specific problem is a difficult task. A common approach is to assess the performance with a model. This reduces the time for implementing and testing different algorithms.

A wide range of models has been designed for parallel computation including the PRAM [FW78], BSP [Val90] and LogP [CKP<sup>+</sup>93] models. Every model has its advantages and downsides.

A number of works [BHP<sup>+</sup>96, Hoe05, HCM<sup>+</sup>05] showed, that the LogP model is well suited for modeling communication in parallel algorithms. To better understand the advantages of the LogP model the next section gives a short overview of some existing models.

### 4.2 Overview of Models

#### 4.2.1 PRAM

The PRAM model consists of a number of processors communicating over shared memory. Each instruction in the PRAM model is executed synchronously by all processors. The processors read data from the shared memory, compute the data and write the results back to the shared memory.

Various additions to the PRAM model deal with asynchronicity, latency and bandwidth issues.

#### 4.2.2 BSP

The Bulk Synchronous Parallel (BSP) model was designed to be a bridge between hardware and software. The BSP model consists of a number of components for computation and memory access, a network to deliver messages and a mechanism performing a barrier synchronization on a set of components.

A program is processed by executing a number of supersteps. In each superstep every component can send data, compute local data and receive data. Every  $L^1$  time units the components are synchronized and if every component has finished the current superstep

---

<sup>1</sup>periodicity parameter

the next superstep is started. If one or more components aren't finished with the current superstep,  $L$  time units are added to compute the current superstep.

In one superstep only a limited number of fixed size messages can be sent and received by each component. The cost for sending these messages is defined by parameters for network latency and bandwidth.

### 4.2.3 LogP

The LogP model was first proposed by Culler et al in 1993. The model's design was influenced by the growing number of parallel machines that consisted of computing nodes connected by a communication network. The model ignores the network topology and focuses on the performance of the underlying network.

The computing nodes communicate with point-to-point messages. Therefore the LogP model offers parameters for latency, communication overhead and a minimal time interval between consecutive messages. As a result the model rewards overlapping of computation and communication as well as overlapping of communication and communication.

### 4.2.4 Conclusion

The PRAM model offers an easy way to assess the theoretical performance of parallel algorithms. As the communication between processors is free it isn't suited for analyzing collective communication operations. While some of the additions may make it more applicable it still doesn't offer the same range of parameters to characterize the underlying network as the LogP model.

The BSP model has parameters for latency and bandwidth but assumes that there is a mechanism that performs a free barrier synchronization. The InfiniBand network doesn't offer such a mechanism. Also the model's use of supersteps makes it difficult to model the overlapping of communication and computation.

The LogP model is the logical choice to analyze the performance of collective communication on computing nodes connected by the InfiniBand network. It offers parameters that were designed with this scenario in mind and will therefore offer the best performance prediction of the compared models. It is still abstract enough to make it easy to use. The LogP model has also already been used to assess the performance of various collective communication algorithms.

The next section gives a more detailed description of the LogP model, its extensions and the measurement of LogP parameters.

## 4.3 LogP Models

### 4.3.1 Overview

As described in section 4.2.3 the LogP model assumes computing nodes that communicate over a network with point-to-point messages. It offers the following parameters:

- $L$  (latency) - upper bound on the time a message needs to traverse the network from one node to another
- $o$  (overhead) - the time the CPU spends sending or receiving a message, blocking it from performing other operations. The overhead can be divided in separate parameters for send and receive overhead.
- $g$  (gap) - the minimum time between consecutive messages,  $1/g$  is the communication bandwidth
- $P$  (processors) - number of processors

The parameters  $L$ ,  $o$  and  $g$  are measured as multiples of the processor cycle time. The network has a limited capacity of  $L/g$  messages that can be on the network to or from one processor at the same time.

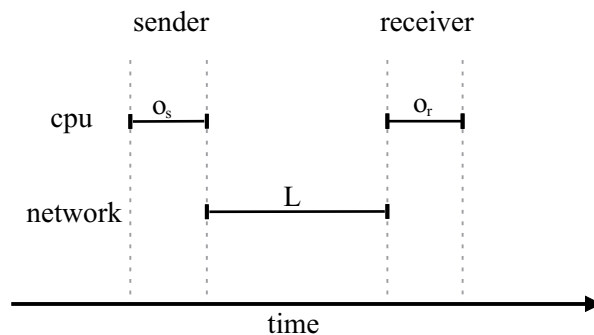


Figure 4.1: Sending a Message in the LogP Model

Figure 4.1 shows the parameters for a communication between two nodes. It takes  $o_s + L + o_r$  cycles until the message is received with  $o_s$  as send overhead and  $o_r$  as receive overhead. The gap doesn't have an effect on the transmission time when sending and receiving only one message.

Figure 4.2 shows how the different parameters can overlap. For this example the following parameter values were chosen:  $L = 20$ ,  $g = 10$ ,  $o_s = 5$  and  $o_r = 6$ . The message from node 1 to node 3 takes the same time as the one in figure 4.1. The following messages from

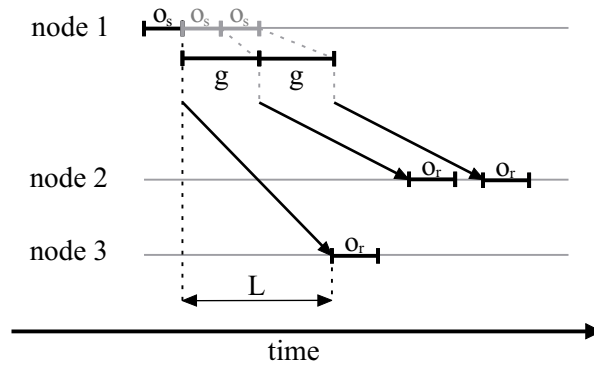


Figure 4.2: Sending and Receiving on Multiple Nodes in the LogP Model

node 1 to node 2 are affected by the gap which is bigger than the send overhead. As gap and overhead overlap only the maximum of both affects the transmission time for multiple messages. The figure also shows that multiple messages can overlap. Node 3 hasn't received the message from node 1 and still node 1 already sent a message to node 2. The time for the whole communication depicted in figure 4.2 is  $o_s + 2g + L + o_r$ .

### 4.3.2 Extended Models

The LogP model only predicts small message performance accurately as most networks achieve a higher bandwidth for larger messages. The LogGP [AISS97] model therefore extends the LogP model with an additional parameter for modeling long messages.

The LogGPS [IFH01] model further extends the LogGP model and deals with the additional time spent for synchronization when sending long messages with high-level communication libraries. Because the InfiniBand network is accessed directly this extension won't be needed.

Another extension to LogGP is LoGPC [MF98]. It models network contention and network interface DMA behavior. The DMA behavior was based on an Alewife network interface and the network contention is based on the behavior of a k-ary n-cube network. While both effects would be interesting to model the different hardware and network topology would require extensive modifications to the model.

LogfP [HMMR06] improves the LogP model's accuracy for small messages on the InfiniBand network.

As LogGP is necessary to model long messages and LogfP offers increased accuracy for modeling small messages on the InfiniBand network both models are further described and analyzed in the following sections.



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

### 4.3.3 Measuring LogGP Parameters

To tune the collective communication algorithms for a special machine it is necessary to measure the LogP parameters. As the LogGP and LogfP models have been chosen it must also be determined for which message sizes each model can be used. This work deals with the LogGP model and in a parallel work by Maik Franke the LogfP model is further analyzed.

#### Existing Measurement Methods

Over the years various measurement methods for LogP parameters have been proposed. Culler et al. [CLMY96] proposed to measure the send overhead by sending a small number of messages and dividing the time through the number of messages sent. The same time for a large number of messages is used to calculate  $g$ . To measure the receive overhead an additional delay  $d$  larger than the round trip time  $RTT = 2 * (o_s + L + o_r)$  is added after sending a message. As the added delay is known  $o_r$  can be calculated by subtracting  $d$  and  $o_s$  from the RTT. The latency  $L$  can finally be calculated by using the previously measured parameters. All parameters were measured using Active Messages with Myrinet. A paper by Ianello et al. [ILM98] describes a similar approach for Fast Messages atop Myrinet.

The measurement method by Kielmann et al. [KBV00] uses a parametrized model with overhead and gap being a function of the message size. Gap is measured by sending a large number of messages to a distant node and receiving one zero byte reply. The measured time divided by the number of messages sent is used as gap. To measure the send overhead a single send operation is used. The receive overhead is measured for a single receive operation after sending an initial message and waiting longer than RTT for the response. The latency is finally calculated using the measured parameters.

Bell et al. [BBC<sup>+</sup>03] proposed a measurement method for LogGP with some model changes. The main difference is the latency parameter that is changed to an end-to-end latency (EEL). The EEL corresponds to  $RTT/2$  for a small message. To measure the RTT a ping pong benchmark is repeated several times and the total time is divided by the number of iterations. The gap is measured similar to the method proposed by Kielmann et al. with the difference that the sender attempts to keep a fixed number of asynchronous sends posted until all messages are issued. The send overhead is measured by adding a delay between initiating and completing a send request. This delay is increased until it has an effect on the communication time. The send overhead now is calculated by subtracting the delay from the gap. The receive overhead is measured in the same way. The per-byte gap  $G$  is calculated by taking the average time for large messages and subtracting  $g$ .

In the most recent work by Hoefler et al. [HLR07] the accuracy of the previous measurement methods is assessed and a new method is proposed and implemented as part of the Netgauge tool [Net06]. As the base for all measurements a parametrized round trip time  $PRTT(n, d, s)$  is defined. The parameters are the number  $n$  of consecutive sends with one reply after the last message, the delay between each sent message  $d$  and the message size  $s$ . The PRTT for sending a single message with reply and the general formula for the PRTT are:

$$\begin{aligned} PRTT(1, 0, s) &= 2 * (L + 2o + (s - 1)G) \\ PRTT(n, d, s) &= PRTT(1, 0, s) + (n - 1) * \max\{o + d, G_{all}\} \quad G_{all} = g + (s - 1)G \end{aligned}$$

The overhead is now calculated by measuring  $PRTT(1, 0, s)$  and  $PRTT(n, d, s)$  for a delay  $d$  greater than  $G_{all}$ :

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

The gap parameters  $g$  and  $G$  are derived by measuring  $PRTT(1, 0, s)$  and  $PRTT(n, 0, s)$  for many different values of  $s$ . The measured values are used to calculate  $g + (s - 1)G$ :

$$g + (s - 1)G = \frac{PRTT(n, 0, s) - PRTT(1, 0, s)}{(n - 1)}$$

The function  $f(s) = g + (s - 1)G$  is then fitted to the calculated values. While a measurement for two different values of  $s$  would have been enough this method was proposed to detect huge differences between the measured values for different message sizes. The latency isn't measured but instead  $PRTT(1, 0, 1)/2$  is used as  $L$ .

## Applied Measurement Methods and Results

To measure the LogGP parameters an InfiniBand micro benchmark is used. The micro benchmark was developed by Torsten Hoefler [Hoe05] and extended in a previous work [HVM<sup>+</sup>06]. The Mellanox Verbs API is used to access the InfiniBand network. To derive the LogGP parameters from the measurements the methods described in the previous section will be used. For the send and receive overhead the methods Kielmann et al. proposed will be applied. The send overhead is measured for a single send operation. For the measurement of the receive overhead a delay greater than RTT is added to make sure a message has arrived. The gap parameters are measured using the method proposed by Hoefler et al. Finally the latency can be calculated with the RTT for a single small message and the measured overhead values:  $L = RTT/2 - o_s - o_r$ . The following listing shows the added communication scenario for the time measurement used to calculate the gap parameters.

```
1  /* prepost receive requests */
   if(sender){
       /* sender needs 1 rr to receive from mirror */
       ret = VAPI_post_rr( );
5  else{
       /* mirror needs n rrs to receive from sender */
       for(j = 0; j < i; j++) {
           ret = VAPI_post_rr( );
       }
10 }

   /* wait until all receive requests are posted */

   start_time_measurement( );
15
   if(sender){
       /* post n send requests to mirror */
       for( j = 0; j < n; j++ ){
           ret = VAPI_post_sr( );
20         }
       /* wait until all messages are sent */
       for( j = 0; j < n; j++ ){
           VAPI_POLL_CQ(<send_completion>);
       }
25 }

   if(mirror){
       /* wait until n messages are received */
       for( j = 0; j < n; j++ ){
30         VAPI_POLL_CQ(<receive_completion>);
       }
       /* now send one message back */
       ret = VAPI_post_sr( );
       /* wait until message is sent */
35     VAPI_POLL_CQ(<send_completion>);
   }

   if(sender){
       /* wait for reply from mirror */
40     VAPI_POLL_CQ(<receive_completion>);
   }

   stop_time_measurement( );
```

The measurement is repeated for several message sizes. The measured gap parameters and the resulting function are shown in figure 4.4.

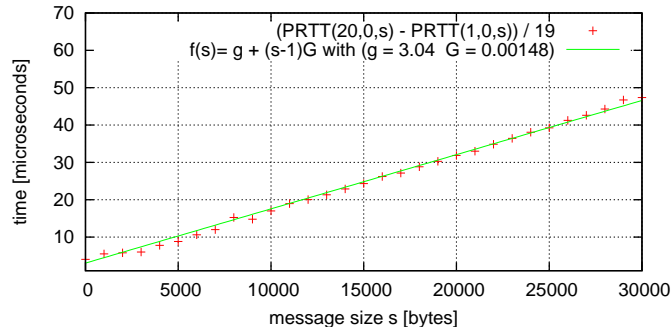


Figure 4.4: Gap Parameters and Resulting Function

The measured overhead values are  $o_s = 2.4\mu s$  and  $o_r = 1.1\mu s$ . The message size doesn't have an effect on the measured overhead values but with a higher number of consecutive messages the overhead per message is decreasing. This is one of the effects that is modeled by the LogGP model. As the sum of gap and gap per byte is usually larger than the overhead for larger messages this behavior shouldn't affect the communication time in the LogGP model.

The resulting value for the latency is  $4.5\mu s$ . Figure 4.5 shows the measured times for a ping pong benchmark ( $PRTT(1,0,s)$ ) and the LogGP prediction.

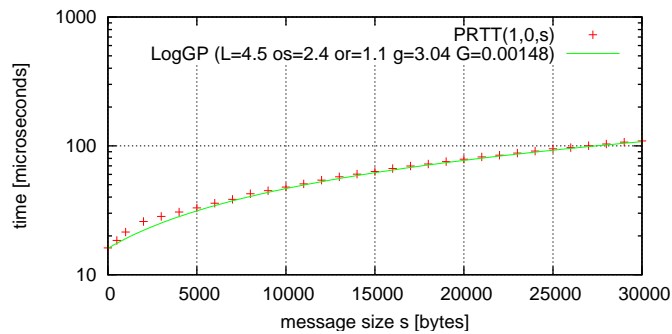


Figure 4.5:  $PRTT(1,0,s)$  and LogGP prediction

The measured parameters are very accurate for messages larger than 5000 bytes. For message sizes between 500 bytes and 4000 bytes the communication time is underestimated by up to 20%.

## 5 Algorithms

### 5.1 MPI\_Scatter

#### 5.1.1 Definition

The MPI\_Scatter operation is used to distribute data from a root process to all other processes in a communicator<sup>1</sup>. The following function call is used:

```
int MPI_Scatter ( void *sendbuf, int sendcnt,  
                 MPI_Datatype sendtype,  
                 void *recvbuf, int recvcnt,  
                 MPI_Datatype recvtype,  
                 int root, MPI_Comm comm )
```

The root process has personalized data for the others processes stored in its send buffer *sendbuf*. The buffer holds *sendcnt* elements of datatype *sendtype* for every destination process. The *sendbuf* isn't required for the other processes. The receive buffer *recvbuf* is required for all processes. It has to offer enough space for *recvcnt* elements of datatype *recvtype*. The send count and receive count have to be equal.

Figure 5.1 shows the data distribution before and after the scatter operation with root 0.

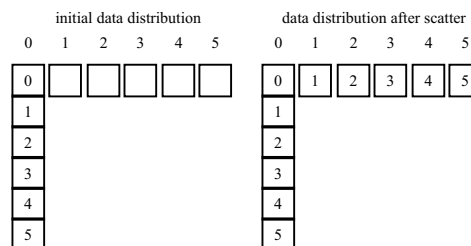


Figure 5.1: Data Distribution before and after Scatter on 6 Processes

The ranks of the different processes are listed at the top. The squares below represent the available buffer space. At the root process each numbered square corresponds to *sendcnt* elements of *sendtype* that have to be sent to the rank with the listed number. After the operation is finished all other processes received their respective data from the root process. The root process is required to copy its data from the send to the receive buffer. Similar figures will be used to visualize the different algorithms.

<sup>1</sup>a group of processes each identified by a unique id

### 5.1.2 Linear Algorithm

The trivial algorithm for the scatter operation is to send all data elements from root to their respective destinations. To utilize the higher bandwidth for large messages all the elements for one destination process should be sent in a single message. The size of *sendcnt* elements of *sendtype* in bytes is from now on referred to as *s*.

If the communicator size is *P* this algorithm requires *P* – 1 messages. The following formulas represent the communication time in the LogGP and LogfP model. The LogfP model is only used for small messages and therefore the message size is ignored.

$$\begin{aligned} \text{LogGP}(P) &= o_s + (P-1) * (\max\{o_s, g\} + (s-1) * G) + L + o_r \\ \forall((P-1) \leq f): \text{LogfP}(P) &= (P-1) * o_s(P-1) + L + o_r(1) \\ \forall((P-1) > f): \text{LogfP}(P) &= o_s(P-1) + L + o_r(1) + \\ &\quad + \max\{(P-2) * o_s(P-1), (P-1-f) * g\} \end{aligned}$$

The listing below shows the simple algorithm using C and the unified Verbs API briefly described in section 2.3.4. Only the important operations are shown and some operations and details are replaced with pseudo code statements to provide a better overview.

```

1  if (my_rank == root) {
    for( i = 0; i < P; i++){
        if(my_rank != i){
            /* send data from position sendbuf+i*s to rank i */
5      iba_post_send( <address = sendbuf+i*s> );
        }
    }
    <copy own data to receive buffer>
    /* wait until data has been sent */
10  for( i = 0; i < P; i++){
        if(my_rank != i){
            do {
                iba_poll_cq( );
            } while <completion queue empty>
15      }
    }
}
if (my_rank != root){
    /* post a receive request */
20  iba_post_recv( <address = recvbuf> );
    /* wait until data is received */
    do {
        iba_poll_cq( );
    } while <completion queue empty>
25 }

```

The linear algorithm doesn't require additional buffer space. For large messages a rendezvous protocol is used. Therefore an additional message is required to signal that a receive request has been posted. The root process then has to check if a destination process is ready to receive before posting a send request. This adds  $o_s + L$  to the complete communication time. No receive overhead is added because RDMA is used for this message.

A simple improvement for the rendezvous algorithm is to perform the ready to receive test in a non-blocking way. This enables the root node to send to any destination that is ready instead of always waiting for the destinations with a lower rank.

### 5.1.3 Hierarchical Algorithms

A common approach to improve the performance of the scatter algorithm is the use of a hierarchical structure to distribute the data. The idea behind it is simple. The root process groups the data it has to send and forwards it to other processes. These then can help distribute the data and the root node has to send less messages. Due to the fact that the bandwidth for larger messages is higher this lowers the communication time.

#### Recursive Splitting

In [vdGPSW96] a recursive splitting algorithm for scatter on hypercubes and power of two linear arrays is described. The basic idea is to embed a minimum spanning tree in the hypercube from the root node to all other nodes. Therefore at each step of the communication the processes that got data toggle a bit in their binary address to determine the destination process. Starting at the root node and with the most or least significant bit the communication takes  $\log_2(P)$  steps for a communicator size of  $P$ . Figure 5.2 shows how the algorithm works on 8 processes with root 0 and starting with the most significant bit.

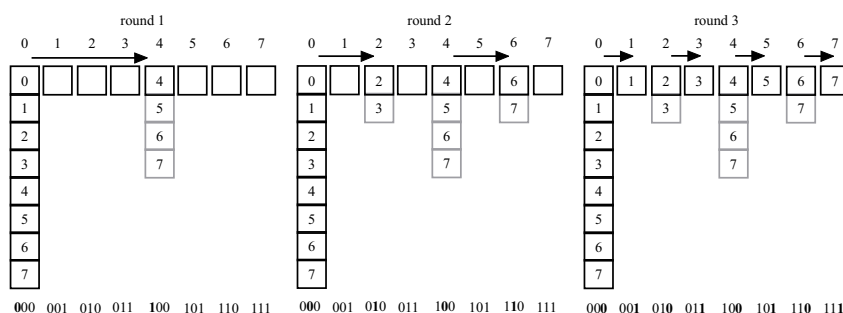


Figure 5.2: Scatter using a Recursive Splitting Algorithm on 8 Processes

For non power of two linear arrays the recursive algorithm divides the nodes in two partitions. It is determined in which partition the process with data is and the destination in the

other partition is chosen. The data is sent to the destination and the algorithm is started again for the initial process and the destination with their respective partitions. This is done until the partitions consist only of one process. For hypercubes this algorithm behaves the same as the initial recursive splitting algorithm if the fitting destinations are chosen.

For a communicator size of  $P$  and  $L > g > o_s$  the following maximum communication times are expected:

$$\begin{aligned} \text{LogGP}(P) &= \lceil \log_2 P \rceil * (o_s + L + o_r) + (P - 1) * (s - 1) * G \\ \text{LogfP}(P) &= \lceil \log_2 P \rceil * (o_s(1) + L + o_r(1)) \end{aligned}$$

For the LogfP model only the first message sent by each process defines the communication time as it takes  $o_s + L + o_r$  cycles until the message is received. The second message sent by each process increases the communication time only by  $o_s(2)$  or  $g$ . In the LogGP model the situation is similar as the amount of data left to sent after each step is equal for all nodes with data. When using a rendezvous protocol  $o_s + L$  has to be added to the complete communication time. If  $P$  isn't a power of 2 the communication time may be reduced by up to  $o_r + L - g$  cycles.

### LogGP Optimal Algorithm

In [AISS97] a scatter algorithm that is optimal in the LogGP model is described. The algorithm basically works in the same way as the recursive splitting algorithm with the difference that the amount of data sent at each step is varied.

The LogGP prediction for the recursive splitting algorithm was based on the time the last node finishes the communication operation. The first node that isn't root and finishes the communication may need considerably less time. The following LogGP formulas model the different communication paths if  $P$  is a power of 2 and  $(o_s + L + o_r) = L' > g$ :

$$\begin{aligned} \min \text{LogGP}(P) &= (\log_2 P - 1) * g + L' + (P - 1) * (s - 1) * G \\ \max \text{LogGP}(P) &= (\log_2 P - 1) * L' + L' + (P - 1) * (s - 1) * G \end{aligned}$$

So the difference between the fastest and slowest path is  $(\log_2 P - 1) * (L' - g)$ . If half the items are sent at each step the communication takes always longer on the receiving nodes.

The algorithm now tries to even out the time difference by adjusting the amount of data that is sent at each step. If  $P$  data items are remaining the following formula is used to determine how many data items  $n$  of size  $s$  are sent at each step.

$$\begin{aligned} t(1) &= 0 \\ t(P) &= \min_{0 < n < P} \{ n * (s - 1) * G + \max\{L' + t(n), g + t(P - n)\} \} \end{aligned}$$

The root node now starts by sending out  $n$  data items that arrive after  $L'$  cycles at their destination. Then the root node calculates the optimal  $n$  for the remaining  $P - n$  items and starts

sending them after  $g$  cycles. The first destination node continues distributing the received data after  $L'$  cycles using the optimal algorithm for the number of received items.

The algorithm has the most effect for big communicator sizes and small item sizes. It is only optimal for single item scatter.

### N-Way Binomial Tree

The spanning tree created for the recursive splitting algorithm on a hypercube is a binomial tree. A binomial tree is defined recursively. The binomial tree of order 0  $B_0$  consists of a single node. A binomial tree of order  $i$  is defined as a root node and  $i$  binomial subtrees  $B_0$  to  $B_{i-1}$ . An example is shown in figure 5.3.

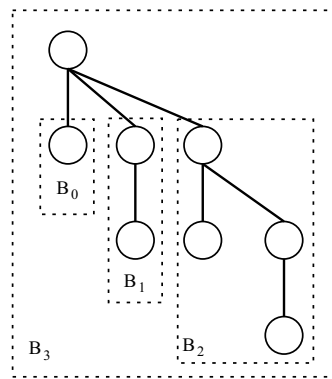
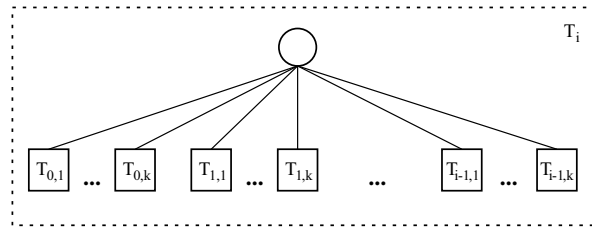


Figure 5.3: Binomial Tree of Order 3

As described in section 4.3.2 the LogfP model suggests that a number of messages can be sent for free with the InfiniBand network. With this information an improved algorithm can be designed.

The number of messages the algorithm sends at each communication step is determined by the step fanout  $k$ . As the base of the algorithm a more general binomial tree is defined. The tree of order 0  $T_0$  as the binomial tree consists of 1 node. A tree of order  $i$  consists of  $k$  subtrees of each order  $T_0$  to  $T_{i-1}$  resulting in a number of  $k * i$  subtrees. Figure 5.4 shows the general layout for a step fanout greater than 1.  $T_{i,j}$  is the  $j$ th subtree of order  $i$ . A tree of order  $i$  has a number of  $(k + 1)^i$  nodes.

If  $P = (k + 1)^i$  is the communicator size the root process will have a number of  $n = P$  data items at start. Each data item contains the data for one process in the communicator. In the first communication step the root process sends  $(k/(k + 1)) * n$  of these items to  $k$  target processes. This reduces the number of remaining data items for the next step to  $n = (k + 1)^{i-1}$ . Now at each following step the root process with the initial data and each process who received data continue distributing it, always sending a part of  $(k/(k + 1))$  of the remaining

Figure 5.4: Tree of Order  $i > 2$  and with a per step Fanout of  $k > 1$ 

items. This continues until only the item containing a processes own data is left. As the communicator size was defined as  $P = (k + 1)^i$  the whole algorithm needs  $i$  communication steps to finish. Figure 5.5 shows how the algorithm works an 9 processes with root 0.

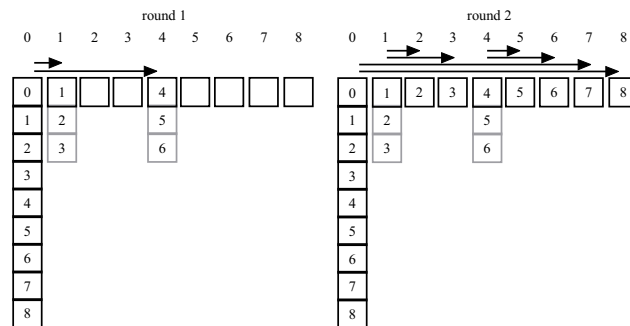


Figure 5.5: N-Way Binomial Tree Scatter with a Step Fanout of 2 on 9 Processes

To construct the tree for any communicator size the following recursive function is used. To offer a better overview only the operations are included that are relevant for the algorithm. The data for each node of the tree is saved in a structure that includes the ranks of all child processes for each round, the rank of the parent process, the size of the subtree and the number of communication rounds.

```

1  int create_bin_sched_rec( int root, int size, int commsize,
                           int rounds, struct node_data_t *node){
    /* set startnode for round 0 */
5  tree_startnode = root;
    /* set rounds for root */
    node[root].rounds = rounds;

10  for( i = 0; i < rounds; i++){
        /* determine the number of nodes per subtree */
        nsubtree = (size + 1) / (step_fanout + 1);
        modsubtree = (size + 1) % (step_fanout + 1);
    }

```

```

15     /* if the size of the remaining tree is smaller than
    * step_fanout then add remaining nodes as childs */
    if( size < step_fanout ){
        /* number of childs (remaining nodes instead
        * of step_fanout) - number of childs can be 0 */
20     node[root].child_count[i] = size;

        <add all remaining nodes as childs in round i>

        /* we are done building the tree structure */
        size = 0;
25     }
    /* size is larger than step_fanout so we send to
    * step_fanout nodes */
    else {
        /* root sends to step_fanout nodes */
30     node[root].child_count[i] = step_fanout;

        for(j = 0; j < step_fanout; j++){
            /* determine child node */
            child = tree_startnode + nsubtree * j + 1;
35     /* if size wasn't a multiple of (step_fanout+1)
            * and nsubtree add remaining nodes */
            if((j > 0) & (modsubtree >= j)){
                child = child + j;
40     } else if ((j > 0) & (modsubtree > 0)){
                child = child + modsubtree;
            }
            /* determine last node in subtree */
            endnode = child + nsubtree;
            if(addstride > j){
45     endnode = endnode + 1;
            }
            /* check if the subtree exceeds the highest
            * communicator rank */
            if (endnode > commsize ){
50     endnode = commsize;
            }
            /* calculate the size of the subtree */
            count = endnode - child;
            /* child data: root is parent
            * subtree consists of count nodes */
55     node[child].parent = root;
            node[child].tree_size = count;
            /* current node data: add child j in round i */
            node[root].child[i][j] = child;
60     /* now call function for child node */
            create_bin_sched_rec( child, count-1, commsize,
                                rounds-i-1, node );
        }
        /* startnode for next round is last node
        * of this round - 1 */
65     size = size + tree_startnode - endnode + 1;
        tree_startnode = endnode - 1;
    }
70 }
    return 0;
}

```

At communicator creation the function is called with the following parameters:

```

1 /* get communicator size */
   size = ompi_comm_size(comm);
   /* calculate needed communication rounds */
   rounds = (int) ceil((log(size)/log(step_fanout+1)));
5 /* create tree with rank 0 as root */
   create_bin_sched_rec(0, size-1, size, rounds, node);

```

The tree structure is built with rank 0 as the root node. If another process is the root in the scatter algorithm its position and data is exchanged with that of rank 0. The actual scatter algorithm just uses the node data of the tree structure and sends and receives accordingly. Figure 5.6 shows two examples for a communicator size of 9 and a step fanout of 2.

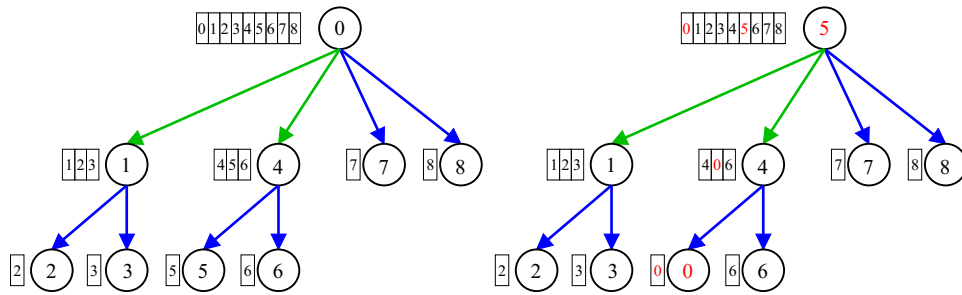


Figure 5.6: Schematic of the Scatter Algorithm on 9 Nodes using a per step Fanout of 2

The running time for the whole communication operation is determined by the last message of the first round. The same maximum amount of data is sent to each node in one round. So the first destination node may already start distributing the received data while the following nodes still have to receive their data that takes the same time to distribute. For a communicator size of  $P$ , a step fanout of  $k \leq f$  and  $L > g > o_s$  the following maximum communication times are expected:

$$\begin{aligned}
 \text{LogGP}(P) &= \lceil \log_{k+1} P \rceil * (o_s + (k-1) * g + L + o_r) + (P-1) * (s-1) * G \\
 \text{LogfP}(P) &= \lceil \log_{k+1} P \rceil * (k * o_s(k) + L + o_r(1))
 \end{aligned}$$

If the communicator size isn't a power of  $(k+1)$  the communication time may be reduced by up to the time needed for the last step due to overlapping effects.

#### 5.1.4 Conclusion

The most important scatter algorithms have been described and analyzed and an algorithm that utilizes the special properties of the InfiniBand network has been proposed. The communication time for all algorithms depends on the communicator and message size.

## Small Messages

For small messages the LogfP model is used to compare the algorithms. Figure 5.7 shows the LogfP communication behavior for the linear, recursive splitting and n-way binomial tree algorithms with the following parameters:  $o_{s_{min}} = 0.5$ ,  $o_{s_{max}} = 2.0$ ,  $g = 3.0$ ,  $L = 5.0$ ,  $o_{r_{max}} = 1.5$ ,  $o_{r_{min}} = 0.5$ ,  $k = f = 2$ .

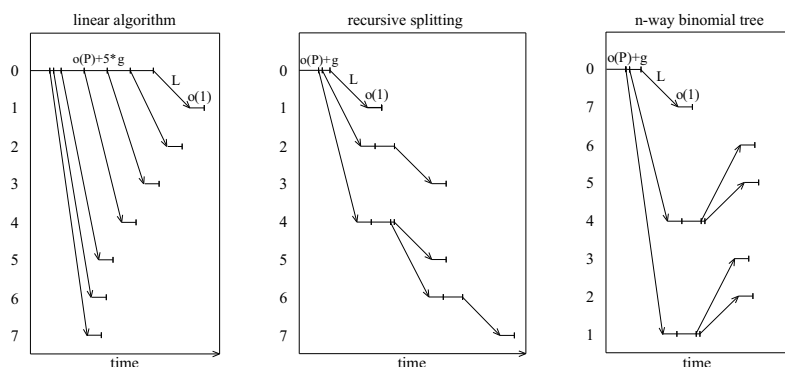


Figure 5.7: Scatter LogfP Comparison

The simple linear algorithm performs well for very small communicator sizes and the communication time scales linearly with the communicator size.

The recursive splitting algorithm is slower for small communicator sizes as the added overhead and latency for each level of the tree have a big effect on the communication time. Additionally the algorithm creates a load imbalance due to the fact that only a single message is sent at each step. So there is a big difference between communication times for the fastest and slowest communication path. The recursive splitting algorithm with its logarithmic growing communication time outperforms the linear algorithm for bigger communicator sizes.

The n-way binomial tree algorithm sends multiple messages at each step and therefore avoids some of the load imbalances. Therefore the overall communication time is reduced compared to the recursive splitting algorithm. Additionally the n-way binomial tree's depth grows slower compared to the recursive splitting algorithm. Latency and overhead for an additional level are added later and the algorithm outperforms the linear algorithm for smaller communicator sizes.

Figure 5.8 shows the expected communication times for the different algorithms in the LogfP model with the parameters already used in figure 5.7 and  $k = f = 4$ .

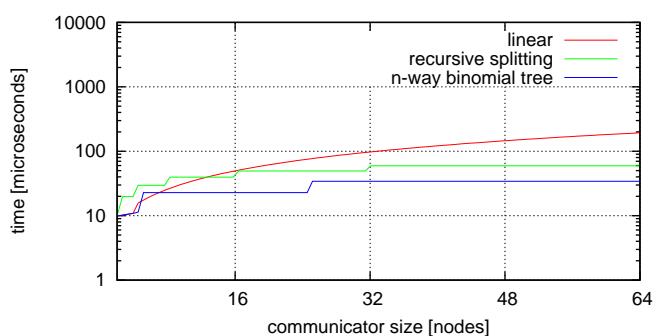


Figure 5.8: Scatter LogP expected Communication Times

### Large Messages

For large messages in the LogGP model an optimal single item scatter algorithm exists. The algorithm is also the fastest algorithm for the general scatter problem. The speedup compared to the recursive splitting algorithm is a maximum of  $\frac{(\log_2(P)-1) * ((o_s + L + o_r) - g)}{2}$  for a communicator size of  $P$ . The actual speedup depends on the size and number of data items. Only data items that are small enough can be used to overlap the overhead and latency with the additional gap per byte. The biggest percentual speedup is achieved for very small messages and large communicator sizes. For this case the n-way binomial tree algorithm also reduces the load imbalance.

As message sizes grow the performance of the algorithms is depending more and more on the gap per byte. The performance advantage of the hierarchical algorithms is shrinking while the additional network traffic they produce is growing. Therefore the linear algorithm is actually the best choice for very large messages as it reduces the network load.

### Memory Usage

The recursive splitting, optimal LogGP and n-way binomial tree algorithms require additional buffer space. For recursive splitting and optimal LogGP up to  $\frac{P * s}{2}$  bytes are required. For the n-way binomial tree the additional buffer space depends on the step fanout  $k$ . Up to  $\frac{P * s}{k}$  bytes are needed.

## 5.2 MPI\_Gather

### 5.2.1 Definition

The MPI\_Gather operation is used to receive data from all other processes in a communicator on a root process. The following function call is used:

```
int MPI_Gather ( void *sendbuf, int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcnt,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm )
```

Each process has data for the root process in its send buffer. The buffer holds *sendcnt* elements of datatype *sendtype*. The receive buffer is only required at the root process and can hold *recvcnt* elements of type *recvtype* from every process in the communicator.

Figure 5.9 shows the data distribution before and after the gather operation with root 0.

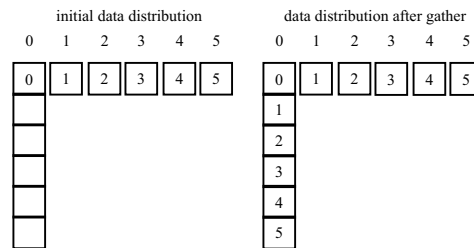


Figure 5.9: Data Distribution before and after Gather on 6 Processes

### 5.2.2 Algorithms

The gather operation is the counterpart to scatter. All algorithms that have been described for scatter can be used to realize the gather operation simply by exchanging the send and receive operations.

### 5.2.3 Conclusions

The overall communication times for the gather and scatter operation only differ by the overhead as the send overhead may be replaced by receive overhead.

## 5.3 MPI\_Allgather

### 5.3.1 Definition

The MPI\_Allgather operation is used to receive data from all other processes in a communicator on every process. The following function call is used:

```
int MPI_Allgather ( void *sendbuf, int sendcnt,
                   MPI_Datatype sendtype,
                   void *recvbuf, int recvcnt,
                   MPI_Datatype recvtype,
                   MPI_Comm comm )
```

Each process has data for the other processes in its send buffer. The buffer holds *sendcnt* elements of datatype *sendtype*. The receive buffer can hold *recvcnt* elements of type *recvtype* from every process in the communicator.

Figure 5.10 shows the data distribution before and after the allgather operation.

initial data distribution						data distribution after allgather					
0	1	2	3	4	5	0	1	2	3	4	5
0						0	0	0	0	0	0
	1					1	1	1	1	1	1
		2				2	2	2	2	2	2
			3			3	3	3	3	3	3
				4		4	4	4	4	4	4
					5	5	5	5	5	5	5

Figure 5.10: Data Distribution before and after Allgather on 6 Processes

In the following algorithm descriptions a data item means *sendcnt* elements of type *sendtype*. A data item's size in bytes is *s*.

### 5.3.2 Algorithms

#### Utilizing Gather and Broadcast

The most trivial algorithm for allgather is to first perform a gather operation on a chosen root process and then use broadcast to distribute the data to all other nodes in the communicator. The communication time is the summation of the time for a single gather operation and a broadcast of the complete receive buffer.

#### Ring Algorithm

The ring algorithm is performed in several steps. In each round a process sends the data at position  $rank - round$  in the receive buffer to  $rank + 1$  and receives data at position  $rank -$

$round - 1$  from  $rank - 1$ . All ranks and positions are computed by adding the communicator size and then using a modulo function with the communicator size. For a communicator of size  $P$  the algorithm takes  $P - 1$  communication rounds. Figure 5.11 shows how the algorithm works on 5 processes.

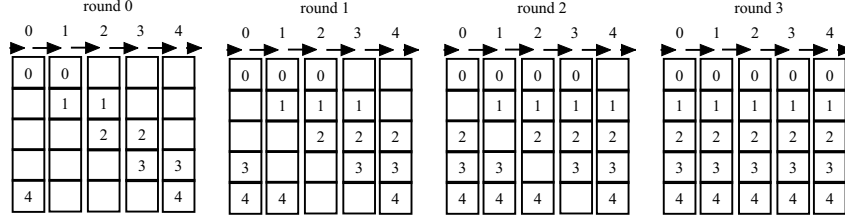


Figure 5.11: Allgather using a Ring Algorithm on 5 Processes

A process can only start the next round if it received the data of the previous round. This limits the number of messages send to one node and may avoid network congestion. The  $\text{LogfP}$  and  $\text{LogGP}$  communication times for a communicator size of  $P$ , a message size of  $s$  and  $(o_s + o_r + L) > g$  are:

$$\begin{aligned} \text{LogGP}(P) &= (P - 1) * (o_s + L + o_r + (s - 1) * G) \\ \text{LogfP}(P) &= (P - 1) * (o_s(1) + L + o_r(1)) \end{aligned}$$

### Modified Ring Algorithm

Instead of constructing a ring on the communicator processes and sending different data items at each step the modified algorithm changes the source and destination processes at each step. A process always sends its own data to the destination  $rank + round + 1$  and receives data at position  $rank - round - 1$  from  $rank - round - 1$ . All ranks and positions are again computed by adding the communicator size and then using a modulo function with the communicator size. Figure 5.12 shows how the algorithm works.

A process can send data even if it didn't receive the data from the previous round. The  $\text{LogfP}$  and  $\text{LogGP}$  communication times for a communicator size of  $P$ ,  $g > o_s > o_r$  and a message size of  $s$  are:

$$\begin{aligned} \text{LogGP}(P) &= o_s + o_r + (P - 1)(s - 1) * G + \max\{(P - 2) * g + L, \\ &\quad (P - 2) * (o_s + o_r)\} \\ \forall((P - 1) \leq f) : \text{LogfP}(P) &= o_s(P - 1) + o_r(P - 1) + \max\{(P - 2) * o_s(P - 1) + L, \\ &\quad (P - 2) * (o_s(P - 1) + o_r(P - 1))\} \\ \forall((P - 1) > f) : \text{LogfP}(P) &= o_s(P - 1) + o_r(P - 1) + \max\{(P - 1 - f) * g + L, \\ &\quad (P - 2) * (o_s(P - 1) + o_r(P - 1))\} \end{aligned}$$

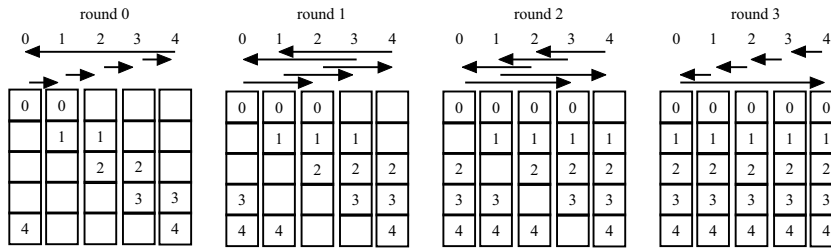


Figure 5.12: Allgather using a Modified Ring Algorithm on 5 Processes

Even with the restrictive assumption that the gap is always larger than the overhead the resulting formulas are complicated. This is due to the fact that the overhead may overlap with gap and latency. The overhead may even become the only relevant parameter for the communication time. The overlap effects may reduce the overall communication time compared to the original ring algorithm.

### Neighbor Exchange Algorithm

The neighbor exchange algorithm [CZZY05] was proposed by Chen et al. and only works on an even number of processes. Each process determines two neighbors in the same way it was done in the ring algorithm. In the first round every rank exchanges one block of data with one of its neighbors. In every following round every process exchanges two blocks of data with one of its two neighbors and selects the other neighbor for the next round. Figure 5.13 shows how the algorithm works.

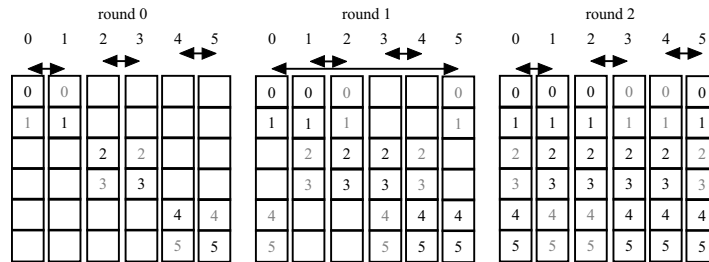


Figure 5.13: Allgather using a Neighbor Exchange Algorithm on 6 Processes

For a communicator size of  $P$  the algorithm needs  $\frac{P}{2}$  communication rounds. Every new round can only be started if the data from the previous round has been received. The  $\text{Log}P$

and LogGP communication times for  $(o_s + o_r + L) > g$  and a message size of  $s$  are:

$$\begin{aligned} \text{LogGP}(P) &= \frac{P}{2} * (o_s + L + o_r) + (P - 1) * (s - 1) * G \\ \text{LogfP}(P) &= \frac{P}{2} * (o_s(1) + L + o_r(1)) \end{aligned}$$

### Recursive Doubling Algorithm

The recursive doubling algorithm [TG03] works only for communicator sizes that are a power of two. The algorithm takes  $\log_2 P$  rounds for a communicator size of  $P$ . In each round  $r$  a process exchanges  $2^r$  data items with a process at distance  $2^r$ . Figure 5.14 shows an example for 8 processes.

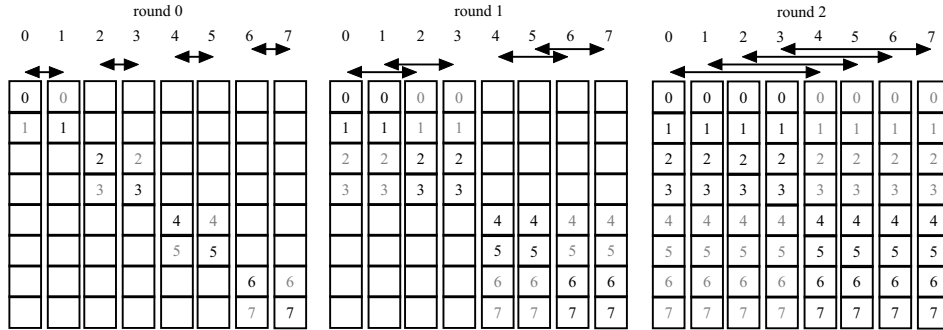


Figure 5.14: Allgather using a Recursive Doubling Algorithm on 8 Processes

A process can only start a new round if it received data from the previous round. For  $(o_s + o_r + L) > g$  the following communication times are expected in the LogGP and LogfP model.

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

The algorithm can be extended to work on non power of two communicator sizes. Additional communication during each round or after the algorithm finishes on the power of two part is necessary. The algorithm then takes  $2 * \lceil \log_2 P \rceil$  steps.

### Bruck Algorithm

The Bruck algorithm [BHU<sup>+</sup>97] is a multi port allgather algorithm. If  $k$  is the number of ports or in our case the number of simultaneous messages we can send under the LogfP model then the algorithm needs  $r = \lceil \log_{k+1} P \rceil$  communication rounds for a communicator

of size  $P$ . In round  $0 \leq i < r$  a process sends  $k$  messages to the processes  $rank - y * (k + 1)^i$  for  $0 < y \leq k$ . Each message contains  $(k + 1)^i$  data items.

For the last round the original algorithm transforms the problem in a table partitioning problem and tries to solve it. This results in an optimal schedule for the last round for a selected number of parameters. The modified version of the algorithm just calculates the number of the missing data items on every process and sends them with as few messages as possible.

Before the actual algorithm starts, the contents of the send buffer have to be copied to the start of the receive buffer. Figure 5.15 shows the modified algorithm with a step fanout of 2.

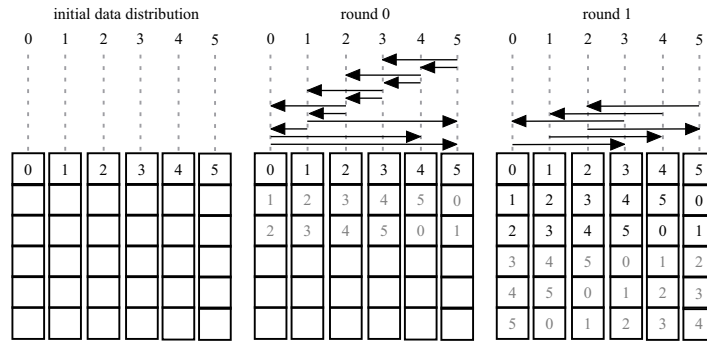


Figure 5.15: Allgather using the Bruck Algorithm on 6 Processes

As can be seen the data has to be shifted to the right position after the communication part of the algorithm is finished. Therefore an additional buffer of size  $\lfloor \frac{P}{2} \rfloor$  is required. The first part of the data items starts at position  $rank$  in the receive buffer and the second part at position 0. The communication time for a communicator of size  $P$ ,  $g > o_s > o_r$  and  $k \leq f$  is the following:

$$\begin{aligned}
 \text{LogGP}(P) &= \lceil \log_{k+1} P \rceil * (o_s + o_r + (s - 1) * G + \max\{(k - 1) * g + L, \\
 &\quad (k - 1) * (o_s + o_r)\}) \\
 \text{LogfP}(P) &= \lceil \log_{k+1} P \rceil * (o_s(k) + o_r(k) + \max\{(k - 1) * o_s(k) + L, \\
 &\quad (k - 1) * (o_s(k) + o_r(k))\})
 \end{aligned}$$

### Dissemination Algorithm

In [BCHC07] an algorithm based on the dissemination barrier is proposed. Therefore it is the single port equivalent of the Bruck algorithm. It introduces more data fragmentation due to the fact that the data is sent to the process  $rank + 2^i$  compared to  $rank - 2^i$  for the Bruck

algorithm. The overall communication time is the same with either the addition of a local copy operation or an additional message in each round.

### 5.3.3 Conclusion

Various allgather algorithms have been described. The expected performance for the different algorithms varies with the communicator and message size.

#### Small Messages

To analyze small message performance the LogfP model is used again. The communication times for recursive doubling and the Bruck algorithm have a logarithmic growth and should perform well for medium to large communicator sizes. The Bruck algorithm is expected to perform better under the LogfP model, but it also introduces additional overhead for data copying after the communication is finished. The overhead increases with growing communicator and message sizes. As the recursive doubling algorithm only works if the communicator size is a power of two, the Bruck algorithm should be the best choice for non power of two sizes.

For small communicator sizes the modified ring algorithm should perform best as all communication is overlapping. With growing communicator sizes the performance is expected to degrade due to the linearly growing amount of gap and overhead. Additionally the algorithm may lead to network congestions.

#### Large Messages

For medium sized messages the Bruck and recursive doubling algorithm should still perform well but the additional overhead introduced by the Bruck algorithm should become even more visible.

For large messages the ring or neighbor exchange algorithms should perform best. Due to the synchronized send and receive operations with one or two data items at each step they limit the generated network traffic. As the bandwidth becomes the limiting factor the additional overhead and latency for more messages compared to the recursive doubling or Bruck algorithms shouldn't notably increase the communication time.

#### Memory Usage

Only the Bruck algorithm requires additional buffer space of size  $\frac{P*s}{2}$  for a communicator of size  $P$  and data item size of  $s$ .

## 6 Implementation Details

### 6.1 Introduction

The collective communication algorithms are implemented as part of a collective component for Open MPI. As the base for the component an implementation of the n-way dissemination barrier by Torsten Hoefler [Hoe05] was used. In the next sections the changes and additions to this component are described.

### 6.2 Initialization

As described in chapter 3 a collective component is selected at communicator creation and its initialization function is called. If the init function is called for the first time it sets up all necessary data for the component. The original component used the Mellanox Verbs API. All functions therefore have been ported to the unified Verbs API described in section 2.3.4.

Before the InfiniBand network can be used a HCA handle has to be retrieved. Next a Protection Domain used for memory access control has to be created. Afterwards the reliable connection queue pairs for the rendezvous protocol are set up. To exchange the needed LIDs and QP numbers the `alltoall` function of the basic component is used. Finally a local buffer used to avoid memory registration for small messages is allocated and registered so the HCA can access it. This completes the component initialization. All the necessary data to use the InfiniBand network and the created queue pairs is saved in a global data structure.

The following initialization operations are done for every created communicator for that the collective component is selected. First a memory region used for 'ready to receive' (RTR) flags in the rendezvous protocol is allocated and registered. The addresses and access keys for the memory regions are exchanged with the `alltoall` function of the basic component. With the exchanged data the send requests used for RTR are created. The ranks of each process in the `MPI_COMM_WORLD` communicator are saved in an array and later used to access the queue pairs created for the component. See figure 6.1 for an example.

If the use of the eager protocol is activated additional reliable connection queue pairs connecting all ranks in the current communicator as well as memory regions with a similar function as RTR are created. An additional buffer is allocated and registered. The size of the buffer depends on the requested eager size, the number of requested pre-posted receive requests and the communicator size. Further details are explained in the eager protocol section.

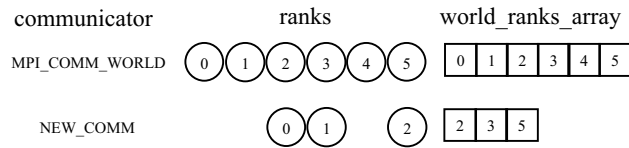


Figure 6.1: Usage of the Ranks in MPI\_COMM\_WORLD to access Queue Pairs

Finally if the implemented algorithms require additional initialization functions these are called. All data is stored in a structure that is attached to the communicator and can therefore be retrieved in any collective function that is called later.

### 6.3 Memory Registration

The InfiniBand Architecture doesn't utilize system buffers and can directly access data in user level buffers. Therefore a memory registration is required that introduces a large overhead.

One way to avoid this overhead is the creation of a preregistered buffer. Now every time a collective operation wants to register a buffer it is first checked if the preregistered buffer is big enough. If that is the case, the preregistered buffer will be returned. As all collective operations are implemented as blocking operations a single buffer is enough. It can be guaranteed that the buffer is exclusive for the operation. The restrictions are that a collective operation may only register one buffer with this method and additional overhead for copying the data in or out of the preregistered buffer is introduced. Therefore the buffer size shouldn't be too big.

Another method to avoid memory registration costs is the use of a memory region handle cache to store registration information. A very simple cache mechanism has been implemented. Every time a registration bigger than the preregistered buffer is requested it is compared to the previous registration. If the requested memory region is part of the previous registration it is reused. Otherwise the old memory region is deregistered, the new memory region registered and stored in the memory region cache.

For the eager protocol an additional preregistered buffer is available. This buffer is used to receive eager messages of a limited size and it is exclusive to a communicator.

The registration information is stored in a data structure that contains the base address and size of the memory region as well as the memory region handle and memory keys.

While most of the implemented methods aren't very sophisticated they are enough to eliminate the effect of memory registration in most benchmarks and therefore make it possible to focus on the communication performance. The MVAPI and OpenIB BTLs of Open MPI use preregistered buffers as well as a searchable cache where multiple memory registrations are stored.

## 6.4 Rendezvous Protocol

The rendezvous protocol is used for large messages, because preregistered buffers to receive large messages from every node would require too much memory. Therefore a synchronization is used to signal that a process has registered the user buffer and is ready to receive a message. Figure 6.2 shows how the protocol works on a communicator of size 4.

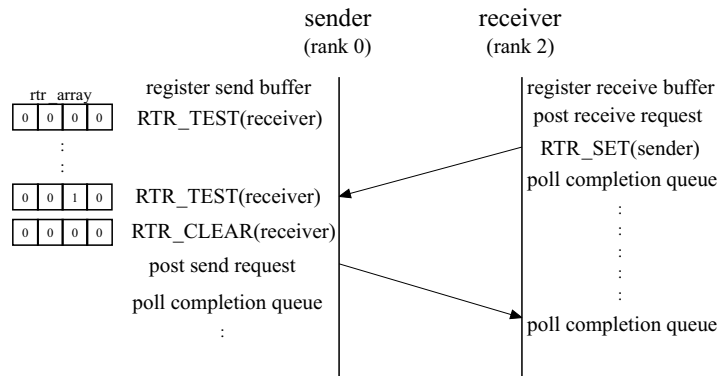


Figure 6.2: Rendezvous Protocol

The protocol is receiver driven. The receiver first registers the receive buffer and then posts a receive request to the appropriate receive queue. Then a RDMA write operation is used to set a flag in the sender's memory. The receiver then waits until the message has arrived by polling the completion queue.

The sending process registers the send memory and then checks if the receiver is ready to receive. If that is the case the sender clears the RTR flag of the receiver in its memory and then posts a send request. Afterwards the send queue completion queue is polled to finish the operation.

As the RTR memory is unique for every communicator and the implemented MPI operations are blocking the protocol is deadlock free.

## 6.5 Eager Protocol

For small messages an eager protocol is used. The protocol requires a set of additional queue pairs connecting all ranks in a communicator. A preregistered buffer and the queue pairs are set up in the init function that is called during communicator creation.

Each receive queue has a unique completion queue to allow polling for messages from a specific source process. A number of receive requests are posted to every receive queue and an array of counters for local and remote receive requests is initialized. Only messages

smaller than an adjustable eager size can be received with the pre-posted receive requests. Each message is received at a unique address in the preregistered buffer. The complete size of the buffer depends on the number of pre-posted receive requests, the maximum size of the eager messages and the communicator size.

An example for a communicator size of four and two pre-posted receive requests is shown in figure 6.3.

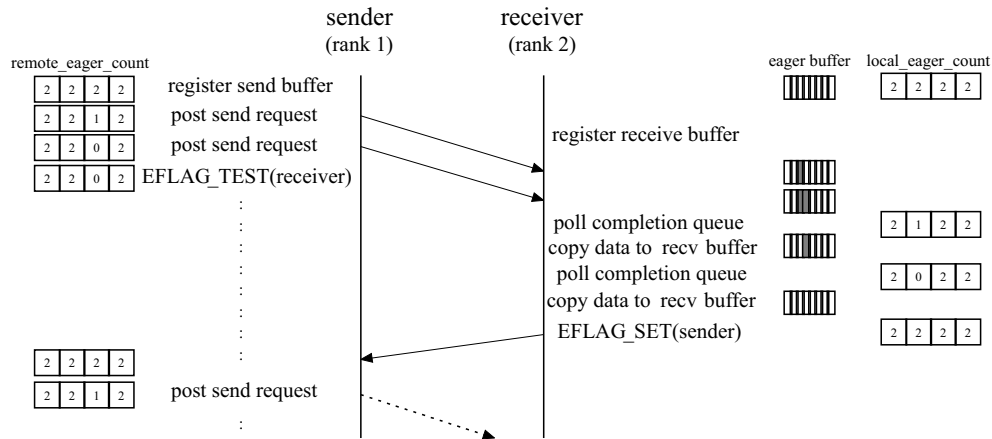


Figure 6.3: Eager Protocol

If a message smaller than the eager size has to be send to another process, it is first checked if the remote eager count of the destination process is larger than zero. If thats the case, there is at least one receive request left at the destination. The sender then reduces the remote eager count and posts a send request. If the eager count is zero the sender waits until the receiver has posted new receive requests. This is realized in the same way as the rendezvous protocol's RTR functionality. If new receive requests have been posted the remote eager count is set to the maximum number of receive requests.

To receive an eager message the receiver polls the appropriate completion queue, reduces the local eager count for the source process and copies the received message in the user buffer. If the local eager count reaches zero new receive requests are posted, the local count is set to the maximum number of receive requests and a flag is set at the source process.

As every receive request has a unique address in the preregistered buffer the eager protocol can easily be changed to use InfiniBand's RDMA write functionality. Therefore the addresses and memory keys for the buffer have to be exchanged between all ranks in a communicator. Afterwards eager messages can be written directly to the destination memory. To signal message reception it is enough to set and check the last bit of a buffer fragment as the InfiniBand Architecture guarantees in order delivery of a message. Another advantage of RDMA write is that one additional set of queue pairs for all communicators would be

sufficient.

MVAPICH and Open MPI use RDMA write for their eager protocols. As the memory requirement with growing communicator sizes becomes a problem, Open MPI can also use a shared receive queue with pre-posted receive requests.

## 6.6 Known Issues

There are still various unresolved problems with the component. The most notable problem is the huge startup time. During initialization the component uses alltoall operations to exchange information. A single alltoall operation takes up to seven seconds on only four nodes of the OSCAR test system. This may be a problem with Open MPI's connection establishment. The BTL endpoints are only connected if a communication between them occurs. The alltoall operations during the initialization of the collective component for `MPI_COMM_WORLD` don't seem to trigger this mechanism. The problem doesn't occur for subsequent communicators.

Another issue with the component was the opal progress function. Open MPI caches message transmissions. This resulted in deadlocks as the collective component was busy waiting for messages while the BTL hadn't finished a previous communication. A simple example would be `MPI_Broadcast` using the basic component followed by `MPI_Gather` using the new collective component. The root process finishes the broadcast operation and enters the gather operation waiting for the other processes. The remaining processes still wait for the broadcast message and a deadlock occurs. Therefore the opal progress function now is called when busy waiting in the component. If this completely resolves the issue is unclear as some deadlocks occurred after adding the function. With the latest Open MPI revisions the problem has disappeared.

Another problem is the memory usage of the eager protocol especially if multiple communicators are used. Some possibilities to reduce the required resources have been described in the eager section. Currently the eager protocol is deactivated if the memory requirements exceed an adjustable limit.

The component currently isn't thread safe.

## 6.7 Implemented Algorithms

A wide range of collective algorithms has been implemented with the component.

The n-way dissemination barrier of the original component was ported to the new framework.

The linear and n-way binomial tree algorithms have been implemented for MPI\_Gather and MPI\_Scatter. For MPI\_Allgather the ring, recursive doubling and Bruck algorithms have been added.

A MPI\_Broadcast algorithm [Sie06] utilizing the multicast capabilities of the InfiniBand Architecture has been implemented by Christian Siebert and Torsten Hoefler.

In a parallel work by Maik Franke the MPI\_Reduce, MPI\_Allreduce, MPI\_Scan and MPI\_Reduce\_scatter functions are added.

## 7 Benchmark Results

### 7.1 Introduction

All benchmarks have been conducted on the CHiC (see appendix A) with NBCBench [NBC06]. The benchmark measures the time for a single collective operation on every node. The maximum of the measured times then is used as the result for one benchmark repetition. Finally the median of 50 repetitions is used to compare the performance of the different algorithms.

The benchmarks were conducted for different communicator sizes  $P$  and with varying sizes  $s$  for one data item.

### 7.2 MPI\_Scatter and MPI\_Gather

MPI\_Scatter and MPI\_Gather should have a very similar performance. Therefore the following comparisons only deal with the MPI\_Gather performance. First the different implemented algorithms are compared to each other.

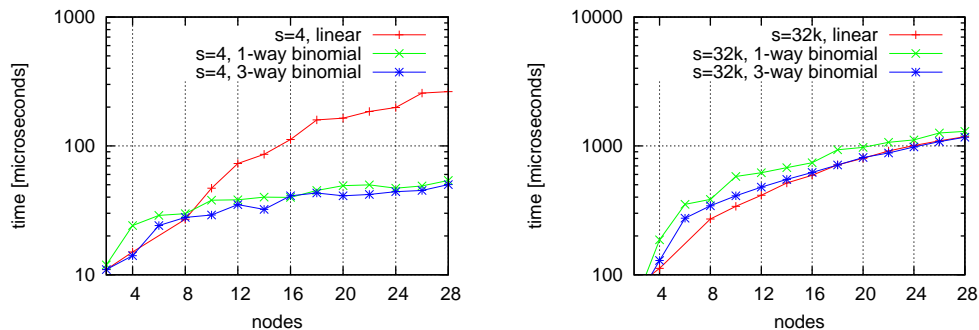


Figure 7.1: Gather Performance Scaling with Communicator Size

The graphs in figure 7.1 show how the algorithms scale with the communicator size. The left graph shows the small message performance ( $s = 4$  byte) and the right graph the large message performance ( $s = 32k$  byte).

First the small message performance is compared. As expected the linear algorithm is fast for small communicator sizes. The  $n$ -way binomial tree algorithm has a better performance

for growing communicator sizes. With a step fanout of one it is equivalent to a recursive doubling algorithm. A higher step fanout improves the performance for small messages. The measured values for the binomial tree algorithms clearly reflect the addition of the first level to the tree. As the communicator size is increased by two this happens at  $P = 4$  for a step fanout of one and at  $P = 6$  for a step fanout of three. The benchmark results fit the LogfP predictions. As shown in the right graph of figure 7.1 the linear algorithm has the best overall performance for larger messages.

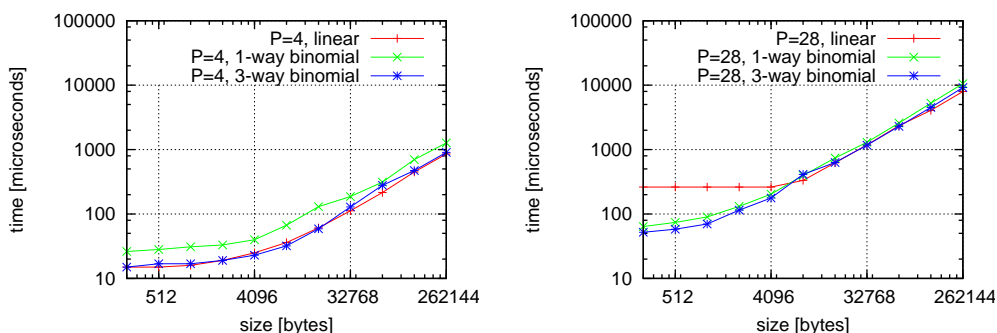


Figure 7.2: Gather Performance Scaling with Buffer Size

Figure 7.2 shows how the algorithms scale with increasing data item size. The right graph shows that on 28 nodes the binomial tree algorithm performs better for a data item size of up to 4 kbyte. For larger data items all algorithms have a similar performance.

Next the component's performance is compared to Open MPI and MVAPICH. Therefore a combination of the n-way binomial tree algorithm with a step fanout of 3 and the linear algorithm is used.

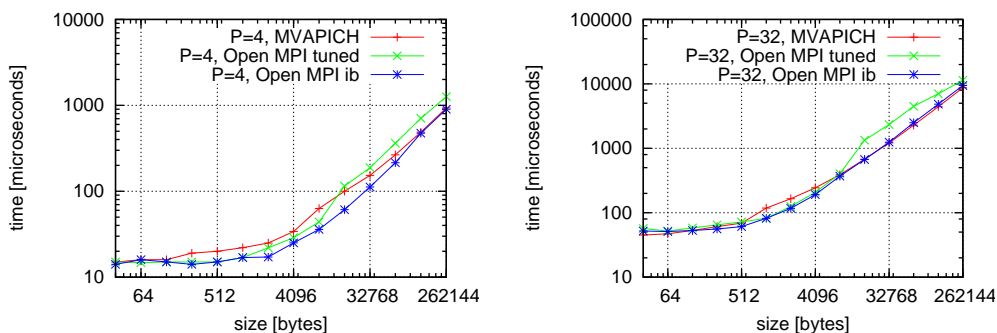


Figure 7.3: Gather Performance compared to Open MPI and MVAPICH

The left graph in figure 7.3 shows the performance on a communicator of size four. For this communicator size the linear algorithm is used. It outperforms both the MVAPICH and the Open MPI tuned component for most message sizes. For larger message sizes MVAPICH has an equal performance. The new component is up to 50% faster than MVAPICH and the tuned component that overall has a decreasing performance for data item sizes between 16 kbyte and 512 kbyte.

The right graph in figure 7.3 shows the performance for a larger communicator of size 32. For small data item sizes the n-way binomial tree algorithm is used and for larger sizes the linear algorithm. The new component doesn't perform as well as MVAPICH for data item sizes below 128 byte. While the new component outperforms MVAPICH and Open MPI's tuned component for a range of item sizes, MVAPICH has a very similar performance.

### 7.3 MPI\_Allgather

The recursive doubling, modified ring (linear) and Bruck algorithms have been implemented.

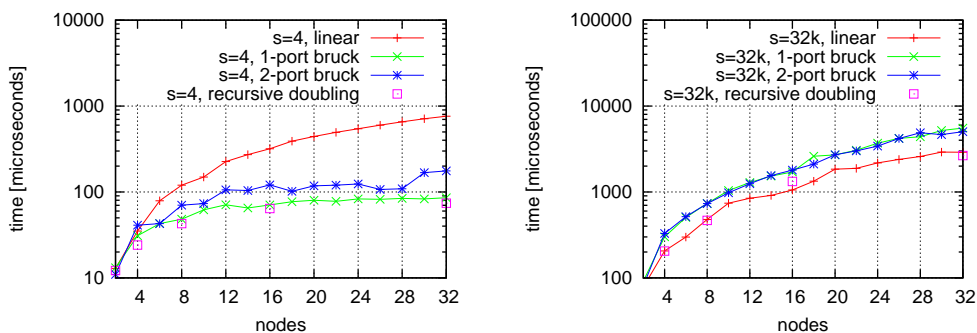


Figure 7.4: Allgather Performance Scaling with Communicator Size

Figure 7.4 shows how the algorithms scale with the communicator size. For small data item sizes the 1-port Bruck algorithm and the recursive doubling algorithm scale best. The linear algorithm is already slower for six nodes. Contrary to the LogfP prediction additional messages in each round increase the communication time for the Bruck algorithm. This may be an implementation problem. The algorithm first sends its messages and then blocks waiting for messages from other processes. If the message from the first source process in the schedule is arriving later than consecutive messages, the algorithm blocks until the first message is received. Afterwards the received data has to be copied to the user buffer before the next round can be started.

For larger messages the linear and recursive doubling algorithms perform best. The Bruck algorithm introduces additional overhead, because the received data has to be shifted to the right position in the receive buffer after the communication is finished.

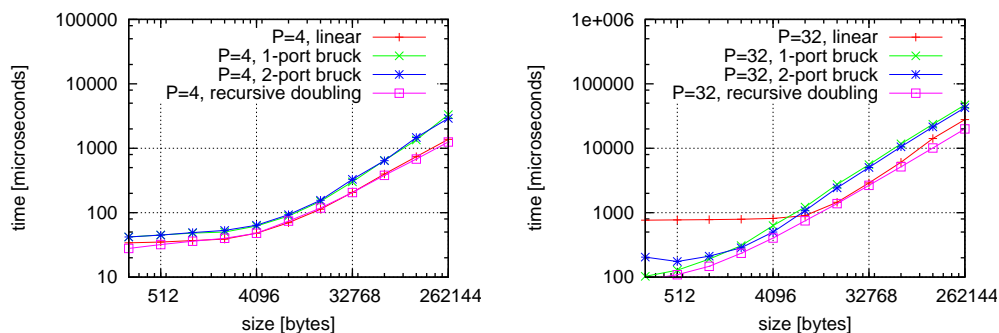


Figure 7.5: Allgather Performance Scaling with Buffer Size

The growing overhead and how the algorithms scale with the buffer size is shown in figure 7.5. For a communicator size of four the linear and recursive doubling algorithm perform best for all buffer sizes. For a larger communicator size the recursive doubling algorithm performs best. For data item sizes below 4 kbyte the Bruck algorithm also outperforms the linear algorithm.

Finally the component's performance is compared to MVAPICH and the tuned components of Open MPI. For comparison reasons the performance of the basic component is shown. The basic component uses MPI\_Gather and MPI\_Broadcast to realize an allgather operation

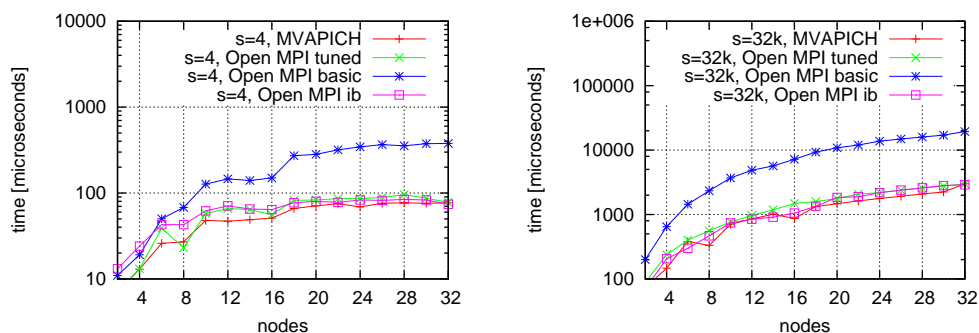


Figure 7.6: Allgather Performance compared to Open MPI and MVAPICH

Figure 7.6 shows how the performance scales with the communicator size. For small message sizes MVAPICH has the best performance especially for communicator sizes below 16. The new and tuned component are slightly behind for bigger communicators. For larger message sizes the different implementations have a very similar performance.

The graphs also show how different algorithms are used. This is especially notable for small message sizes and the tuned component. Like the new component it also uses a recursive doubling algorithm for power of two communicator sizes and a single port Bruck algorithm for other communicator sizes.

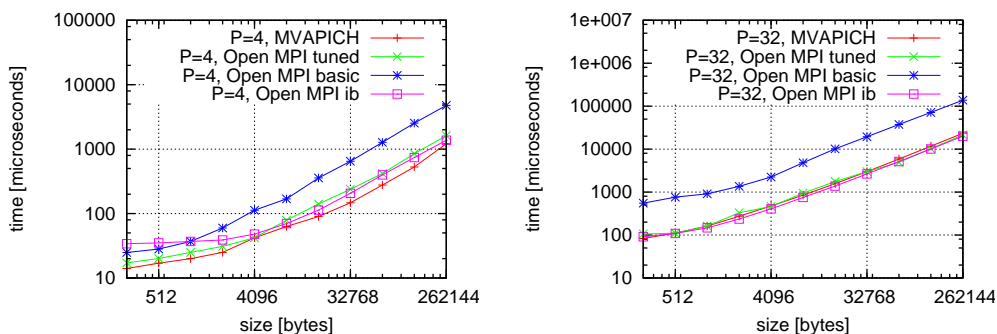


Figure 7.7: Allgather Performance compared to Open MPI and MVAPICH

Figure 7.7 shows how the performance changes with different data item sizes. For small communicator and message sizes the component isn't as fast as MVAPICH or the tuned component. With growing message sizes the new and the tuned component have a similar performance.

For larger communicator sizes the new component slightly outperforms MVAPICH and the tuned component. The new component, the tuned component and MVAPICH all use a recursive doubling algorithm for the chosen communicator size.

## 8 Conclusion and Future Work

In this work a new collective component has been developed. The included algorithms for MPI\_Scatter, MPI\_Gather and MPI\_Allgather have been theoretically analyzed with the LogfP and LogGP models. To assess the performance for a selected machine a LogGP parameter measurement method, based on different existing approaches, has been implemented as part of an existing micro benchmark. Finally the new component has been compared to other existing implementations.

The performance of the implemented algorithms has been improved for several message and communicator sizes. For instance the MPI\_Gather operation is up to 50% faster than existing implementations for small communicator sizes. But overall the performance of all implementations doesn't differ much for the tested communicator and message sizes. The new component does outperform the other implementations, if these use an algorithm that doesn't have the best performance for a selected communicator and message size. Additionally the new component does outperform the other implementations for some message size ranges. This may be due to protocol changes. Accessing the InfiniBand network directly doesn't offer as big an advantage as expected. The use of point-to-point functions doesn't seem to introduce much overhead.

The LogGP and LogfP models were very helpful for implementing and optimizing the algorithms as they encourage overlapping. The predictions for the different algorithms set a lower boundary for the measured values. The LogfP effect could be seen best for the n-way binomial tree MPI\_Gather algorithm. The effect wasn't notable for the Bruck algorithm. This has to be further investigated.

For future work there is still optimization potential for the new component. Especially the eager protocol and memory registration should be improved and the required resources reduced.

## Bibliography

- [AISS97] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.
- [BBC<sup>+</sup>03] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current highperformance networks, 2003.
- [BCHC07] Gregory D. Benson, Cho-Wai Chu, Qing Huang, and Sadik G. Caglar. A Comparison of MPICH Allgather Algorithms on Switched Networks, 2007.
- [BHP<sup>+</sup>96] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul G. Spirakis. BSP vs LogP. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, 1996.
- [BHU<sup>+</sup>97] Jehoshua Bruck, Ching-Tien Ho, Eli Upfal, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(11):1143–1156, 1997.
- [BSL<sup>+</sup>05] 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.
- [CKP<sup>+</sup>93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
- [CLMY96] David E. Culler, Lok T. Liu, Richard P. Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.
- [CZZY05] Jing Chen, Linbo Zhang, Yunquan Zhang, and Wei Yuan. Performance Evaluation of Allgather Algorithms On Terascale Linux Cluster with Fast Ethernet. In *HPCASIA '05: Proceedings of the Eighth International Conference on*

- High-Performance Computing in Asia-Pacific Region*, page 437, Washington, DC, USA, 2005. IEEE Computer Society.
- [FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM Press.
- [GFB<sup>+</sup>04] 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.
- [GWS05] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.
- [HCM<sup>+</sup>05] Torsten Hoefler, Lavinio Cerquetti, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A Practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 562–569, June 2005.
- [HLR07] Torsten Hoefler, André Lichei, and Wolfgang Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. 03 2007. Accepted for publication at the PMEOPDS 2007 in conjunction with IPDPS 2007.
- [HMMR06] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. LogfP: A Model for small Messages in InfiniBand. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006*, 2006.
- [Hoe05] Torsten Hoefler. Evaluation of publicly available Barrier-Algorithms and Improvement of the Barrier-Operation for large scale Cluster-Systems with special Attention on InfiniBand Networks. Diploma Thesis, Chemnitz University of Technology, 2005.
- [HR05] T. Hoefler and W. Rehm. A Communication Model for Small Messages with InfiniBand. In *PARS Mitteilungen*, pages 32–41. PARS, 06 2005. (Awarded with the PARS Junior Researcher Prize).

- [HVM<sup>+</sup>06] T. Hoefler, C. Viertel, T. Mehlan, F. Mietke, and W. Rehm. Assessing Single-Message and Multi-Node Communication Performance of InfiniBand. In *Proceedings of IEEE International Conference on Parallel Computing in Electrical Engineering, PARELEC 2006*, pages 227–232. IEEE Computer Society, 9 2006.
- [IBA] Infiniband Trade Association. *InfiniBand Architecture Specification Volume 1, Release 1.2*.
- [IFH01] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 133–142, New York, NY, USA, 2001. ACM Press.
- [ILM98] G. Iannello, M. Lauria, and S. Micolino. LogP performance characterization of Fast Messages atop Myrinet, 1998.
- [KBV00] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1176–1183, London, UK, 2000. Springer-Verlag.
- [MF98] Csaba Andras Moritz and Matthew I. Frank. Logpc: modeling network contention in message-passing programs. *SIGMETRICS Perform. Eval. Rev.*, 26(1):254–263, 1998.
- [Mos06] Marek Mosch. Integration einer neuen InfiniBand-Schnittstelle in die vorhandene InfiniBand MPICH2 Software, 2006.
- [NBC06] NBCBench. <http://www.unixer.de/research/nbcoll/perf/>, 2006.
- [Net06] Netgauge. <http://www.unixer.de/research/netgauge/>, 2006.
- [Sie06] Christian Siebert. Efficient Broadcast for Multicast-Capable Interconnection Networks. Diploma Thesis, Chemnitz University of Technology, 2006.
- [SL04] Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In Vladimir Getov and Thilo Kielmann, editors, *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.
- [SWG<sup>+</sup>06] Galen M. Shipman, Tim S. Woodall, Rich L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. InfiniBand Scalability in Open MPI. In *Proceedings of IEEE Parallel and Distributed Processing Symposium*, April 2006.

## BIBLIOGRAPHY

---

- [TG03] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH, 2003.
- [Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [vdGPSW96] R. van de Geijn, D. Payne, L. Shuler, and J. Watts. A Streetguide to Collective Communication and its Application., 1996.

# A Testing Environments

## A.1 OSCAR

The Oscar Cluster was used to test the MVAPI functionality of the component. The Cluster consists of 4 nodes. A single node has the following configuration:

- 2x2,4GHz Xeon Processor
- 2GB Memory
- Mellanox "Cougar" HCA (MTPB 23108)

## A.2 CHiC

The Chemnitz High Performance Linux Cluster (CHiC) was used to test the Open Fabrics Verbs API functionality. Additionally all benchmarks have been conducted on the CHiC. It consists of 538 nodes connected by 288 port InfiniBand switches. A single computing node has the following configuration.

- 2x2,6GHz Dual Core Opteron Processor 2218
- 4GB Memory
- Voltaire HCA 400-EX-C (InfiniBand 4x, 10GB/s)

## **B Glossary**

**ACK** Acknowledged

**API** Application Programming Interface

**BML** BTL Management Layer

**BSP** Bulk Synchronous Protocol

**BTL** Byte Transfer Layer

**CQ** Completion Queue

**CQE** Completion Queue Element

**DMA** Direct Memory Access

**GID** Global Identification

**HCA** Host Channel Adapter

**IBA** InfiniBand Architecture

**LID** Local Identification

**MCA** Modular Component Architecture

**MPI** Message Passing Interface

**MVAPI** Mellanox Verbs API

**NAK** Not Acknowledged

**PML** Point-to-point Messaging Layer

**PRAM** Parallel Random Access Machine

**PRTT** Parametrized Round Trip Time

**QP** Queue Pair

**QPN** Queue Pair Number

**RTR** Ready to Receive

**RTT** Round Trip Time

**RDMA** Remote Direct Memory Access

**SQ** Send Queue

**TCA** Target Channel Adapter

**WQ** Work Queue

**WQE** Work Queue Element

**WR** Work Request

## C Theses

- I Open MPI offers an easy way to implement new collective algorithms.
- II The performance of collective communication operations can be improved.
- III Direct access to the InfiniBand network can improve the performance of collective communication operations.
- IV The LogfP model accurately models the behavior of the InfiniBand network for small messages.
- V The LogGP model is accurate enough for large messages.
- VI Accurate LogGP parameters can be measured.

## **D Acknowledgements**

I want to thank my advisor Torsten Hoefler for the technical support. I also want to thank my supervisor Prof. Rehm for the useful hints. I want to thank Frank Mietke for administration and support of the used test systems and Maik Franke for his help with testing the component. Last but not least I want to thank my family and friends for their social support during this work.

## 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, April 30, 2007

---

Carsten Viertel