CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Diploma Thesis

Evaluation of publicly available Barrier–Algorithms and Improvement of the Barrier–Operation for large–scale Cluster–Systems with special Attention on InfiniBand™Networks

Torsten Höfler

htor@informatik.tu-chemnitz.de

Advisers: Dipl.-Inf. T. Mehlan, Dipl.-Inf. F. Mietke

Supervisor: Prof. Dr.-Ing. W. Rehm

Task of the Thesis

The diploma thesis intends to improve the well–known barrier–operation. This operation is offered by almost every message–passing–system. A wide range of parallel applications depend on the efficiency of the barrier–algorithm. The possibility of providing a barrier of constant time complexity has to be investigated. An optimal barrier–algorithm would need constant time, no matter how much computing nodes are invoked. This research results in suggestions how to provide a well performing implementation of the barrier.

To determine the usability of advanced network services, the performance of special facilities of InfiniBand has to be measured. The main focus lies on the InfiniBand multicast service, the InfiniBand atomic operations and the InfiniBand RDMA capabilities. This evaluation provides basic knowledge to decide which network facilities are suitable to implement barrier–algorithms. Moreover the InfiniBand specification is searched for other facilities supporting barrier–algorithms efficiently.

It is also considered to use hardware support to improve the barrier–operation. Different approaches for an acceleration of the barrier–operation have to be discussed. It shall be determined whether network switches can be modified to improve the barrier. Especially extensions to InfiniBand switches shall be discussed. Furthermore some alternatives shall be considered. Thus an investigation of dedicated barrier– and broadcast–networks and bus–based networks shall be performed.

The diploma thesis has to result in a statement explaining the mechanisms and algorithms that are best suitable to implement the barrier–operation. The main goal is to achieve a barrier with constant time complexity for large numbers of computing nodes. To preserve this behavior some assumptions have to be made. This applies to the processing speed of network devices and computing nodes as well as to the network topology. The assumptions and prerequisites necessary to achieve constant time for barriers have to be explained.

Primary Theses

1. It is possible to create a constant barrier–operation for large–scale cluster systems.
2. The preceding work does not take into account the theoretical foundations properly.

Thesis Declaration

I hereby declare that this diploma thesis was composed by myself and all work included has been done by me.

Chemnitz, 29th March 2005

Torsten Höfler

Abstract

The MPI_Barrier-collective operation, as a part of the MPI-1.1 standard, is extremely important for all parallel applications using it. The latency of this operation increases the application run time and can not be overlaid. Thus, the whole MPI performance can be decreased by unsatisfactory barrier latency. The main goals of this work are to lower the barrier latency for InfiniBand^{TM} networks by analyzing well known barrier algorithms with regards to their suitability within InfiniBand^{TM} networks, to enhance the barrier operation by utilizing standard InfiniBand^{TM} operations as much as possible, and to design a constant time barrier for InfiniBand^{TM} with special hardware support. This partition into three main steps is retained throughout the whole thesis. The first part evaluates publicly known models and proposes a new more accurate model (LoP) for InfiniBand^{TM} . All barrier algorithms are evaluated within the well known LogP and this new model. Two new algorithms which promise a better performance have been developed. A constant time barrier integrated into InfiniBand^{TM} as well as a cheap separate barrier network is proposed in the hardware section. All results have been implemented inside the Open MPI framework. This work led to three new Open MPI collective modules. The first one implements different barrier algorithms which are dynamically benchmarked and selected during the startup phase to maximize the performance. The second one offers a special barrier implementation for InfiniBand^{TM} with RDMA and performs up to 40% better than the best solution that has been published so far. The third implementation offers a constant time barrier in a separate network, leveraging commodity components, with a latency of only 2.5μs. All components have their specialty and can be used to enhance the barrier performance significantly.

Contents

Chapter 1

Introduction

In general, a barrier operation is used to synchronize a number of processes. Thus, it can be seen as a synchronization primitive for parallel systems and can be useful to divide the application into different loosely coordinated phases. For example, a given application could differ between communication and computation steps, synchronized by a barrier operation. This prevents data from being sent before it is valid and overwriting of buffers during the calculation. The semantics for this operation define that each process calls the barrier function and the function does not return until all processes did so. This function is part of the MPI collective framework and therewith regarding to Amdahls Law [Amd00] very time critical. The time to complete a barrier has to be reduced as far as possible. This paper is mainly based on the barrier syntax and semantics defined in the MPI standard (see section 1.2). However, this does not limit the use of the achieved results to only this application domain. The concepts and algorithms which will be developed can be generalized for each synchronization problem in a parallel system. A big part of this paper is bound tightly to InfiniBand™ however parts of the developed methodologies can also be generalized for usage with any other interconnect network.

1.1 Organization/Structure of the Document

The first part of this thesis introduces several basic terms and assumptions which are used throughout the following chapters. The fundamental software to benchmark the results is described in terms of suitability and implementation details in part two of this introduction. Chapter 2 describes different models for parallel computation and their eligibility for modelling the barrier operation for InfiniBand™ . Different published barrier algorithms are analyzed in the context of the chosen model and the correctness of the predictions is validated with benchmarks. Two new barrier algorithms which are able to exploit hardware parallelism are proposed and evaluated in the second part of chapter 2. In addition, a new model will be developed and parametrized with benchmark results at the end of this chapter to satisfy the special needs of the barrier modelling for InfiniBand™ networks. A new hardware solution to incorporate barrier support into InfiniBand™ switches or a separate barrier network is proposed in chapter 3 followed by the practical evaluation of all developed barrier techniques presented in this thesis.

1.2 MPI Standard

MPI stands for Message Passing Interface, which focuses on providing a widely used standard for writing parallel programs. The main target is to unify the conflicting requirements ease-of-use, portability, efficiency and flexibility for all HPC platforms. A complete list of goals stated in the MPI Standard 1.1 [For95] is shown in the following enumeration.

- Design an application programming interface (not necessarily for compilers or a system implementation library)
- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to a communication coprocessor, where available.
- Allow implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface: The user needs not cope with communication failures. Such failures are handled by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provide extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow thread-safety.

List 1.1: MPI 1.1 goals

The design process started officially in 1992 when Dongarra, Hempel, Hey, and Walker proposed a preliminary draft, known as MPI-1, to the community. This ongoing development was promoted by the newly founded MPI Forum[1] which finished the MPI-1 review in February 1993. This first draft was intended as a starting point and covered only point to point communication and some very basic origins for the design of collective operations. An official version of this draft was released to the public at the Supercomputing 93 conference in November 1993.

The MPI Forum continued its work with the participation of over 40 organizations and proposed a final version of the MPI-1.0 standard in May 1994. This standard was swiftly superseeded with the MPI-1.1 standard in June 1995. The standardization process of MPI-2 started in April 1995 and led to a final MPI-2 document in April 1997 as an addition to MPI-1.1. Since then, the community concentrates on implementing the proposed standards in an efficient way.

It is explicitely stated that the design focuses on improving the performance and scalability of parallel computers with specialized interconnect hardware (section 1.3 in [For95]). This thesis focuses on improving the performance of the `MPI_Barrier()` operation stated in the standard by utilizing special features of the InfiniBand™ network or hardware support.

The remainder of this section describes the semantics of the barrier call described in the MPI-1.1 standard and the available Open Source libraries which can be used to test and validate the results.

1.2.1 The MPI_Barrier() Call

The barrier call is standardized as a collective operation in MPI-1.1 as `MPI_Barrier(MPI_Comm comm)`. The communicator `comm` is a logical process group wherein the operation is performed. There are several predefined communicators like `MPI_COMM_WORLD` (includes all processes inside the MPI job) and `MPI_COMM_SELF` (points to the process itself). The semantics of this call are easily described as:

"`MPI_BARRIER` blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call." (chapter 4.3 in [For95])

[1] MPI Forum [http://www.mpi-forum.org]

1.2.2 Available MPI Implementations

Todays most important available open source MPI implementations include MPICH[2] and LAM/MPI[3]. A new approach started by the Open HPC group to develop a new MPI implementation called Open MPI[4], which is mainly a consolidation of different MPI implementations (FT-MPI[5], LA-MPI[6], LAM/MPI[7] and PACX-MPI[8]), has the potential to become an important library in the near future.

The open and well-understood architecture of the emerging Open MPI project is due to its modular design the most suitable implementation to incorporate ideas proposed in this paper. Thus, Open MPI is used to demonstrate and benchmark the results of the new approaches. Architectural and implementation details of the library are presented in chapter 1.4.

1.3 InfiniBand™

The InfiniBand™ Architecture (IBA) is used as an interconnection network. It was mainly intended as a high speed interconnect for servers in a typical data center environment. The development process targeted at offering high bandwidth and high expandability for future computing systems and innovative features like RDMA and message send/receive within the user level without paying the costs for entering kernel routines. This, and the relatively low prices due to the wide spreading make InfiniBand™ also very attractive to HPC vendors. Especially the fact that the design actively supports standard bus-adaptions (e.g. PCI-X, PCI-Express™, HTX) increases the suitability for clusters based on commodity components.

The InfiniBand™ specification [IBA] is actively developed and maintained by the Infiniband Trade Association (IBTA[9]). The first version of the specification was introduced in September 2000 as revision 1.0. Revision 1.1 followed after a subrevision 1.0.a (mainly error corrections) in June 2002 with some new features. The current revision 1.2, available since September 2004, introduces some new features proposed by different vendors (e.g. Mellanox[10]) such as a shared receive queue (see [IBA], section 10.2.9). The legacy revision 1.1 is used as a foundation for implementations and measurements because hardware fully supporting the specification 1.2 is not yet available. This denotes that all approaches shown in this work may be enhanced with features of the new specification.

1.3.1 InfiniBand™ Architecture

A deployed InfiniBand™ architecture is called a System Area Network. It consists of a number of switches which route packets based on virtual point-to-point connections. Nearly every device can act as an end point, beginning with a single I/O Terminal up to very complex systems such as multiprocessor computers. These systems are classified into two fields by their use cases. A TCA (Target Channel Adapter) usually fulfils exactly one task (e.g. a storage controller) while a HCA (Host Channel Adapter) acts together with a more complex system like a computer. The operation mode can additionally be divided into two connection variants:

- module-to-module (inter chip communication, if I/O modules are supported by the systems)
- chassis-to-chassis (classical host interconnect like Ethernet)

This paper assumes that the end-systems are normal computers and it uses the term HCA for the InfiniBand™ connection device. No further assumptions are taken for the connection parameters, both connection variants mentioned above are supported.

[2] MPICH [http://www-unix.mcs.anl.gov/mpi/mpich]
[3] LAM/MPI [http://www.lam-mpi.org]
[4] Open MPI [http://www.open-mpi.org]
[5] FT-MPI [http://icl.cs.utk.edu/ftmpi]
[6] LA-MPI [http://public.lanl.gov/lampi]
[7] LAM/MPI [http://www.lam-mpi.org]
[8] PACX-MPI [http://www.hlrs.de/organization/pds/projects/pacx-mpi]
[9] IBTA [http://www.infinibandta.org]
[10] Mellanox [http://www.mellanox.com]

The IBA offers several features which are helpful for message passing. Some of them are listed in the following:

- reliable transport
- user level communication
- management infrastructure
- native IPv6
- decreased CPU utilization (full offloading)
- send/receive semantics and RDMA/Atomic
- scalability in bandwidth
- request queuing in hardware
- multicast

List 1.2: InfiniBand™ Features

1.3.2 Hardware Queuing

All requests are queued in hardware to ensure that multiple user-level applications can use the HCA to send or receive messages. Due to this fact, the operating system does not need to manage multiple accesses to the device. It acts as a normal entity requesting a service. Quality of Service (QoS), with several virtual lanes ordered by priority, may be used to ensure proper prioritization of data transmissions.

A queue is usually called Work Queue (WQ). Work Queues are created in pairs (Queue Pairs, QP), one for receive operations and one for send operations. Requests to send data (Send Requests, SR) to a remote node are posted to the Send Queue (SQ) and requests to receive data (Receive Requests, RR) with appropriate placement information are posted to the Receive Queue (RQ). The hardware fetches an element from the top of the queue and processes it. It has to be stated that two or more Work Requests (WR) can be processed in parallel, even if they originate from the same queue. The hardware places a notification with status information into a completion queue (CQ) associated with the source QP of the request after processing it. Whereby a completion queue can be shared between different Queue Pairs to simplify the use. Each application on the system creates several Queue Pairs for sending and receiving data, where usually every QP denotes a single communication channel to another system. The whole process is shown in figure 1.1.

A Send Queue can handle three different request types:

1. **SEND**
2. **RDMA**
3. **MEMORY BIND**

A SEND SR specifies a piece of data (address, length) in the local memory to send to a peer. The RDMA SR splits up into three modes:

1. **RDMA Write** - specifies local data (address and length), a remote r_key and a remote address to put the data
2. **RDMA Read** - specifies a remote address, an r_key and a length to fetch the data and a local address to put it
3. **RDMA Atomic** - performs a 64 bit atomic read and a conditional modify in the remote memory

A MEMORY BIND WR instructs the hardware to change memory registration relations. The operation returns also a new r_key which is used by foreign nodes to access local memory (see RDMA). The r_key is a security feature to prevent undesired access to local memory by other nodes.

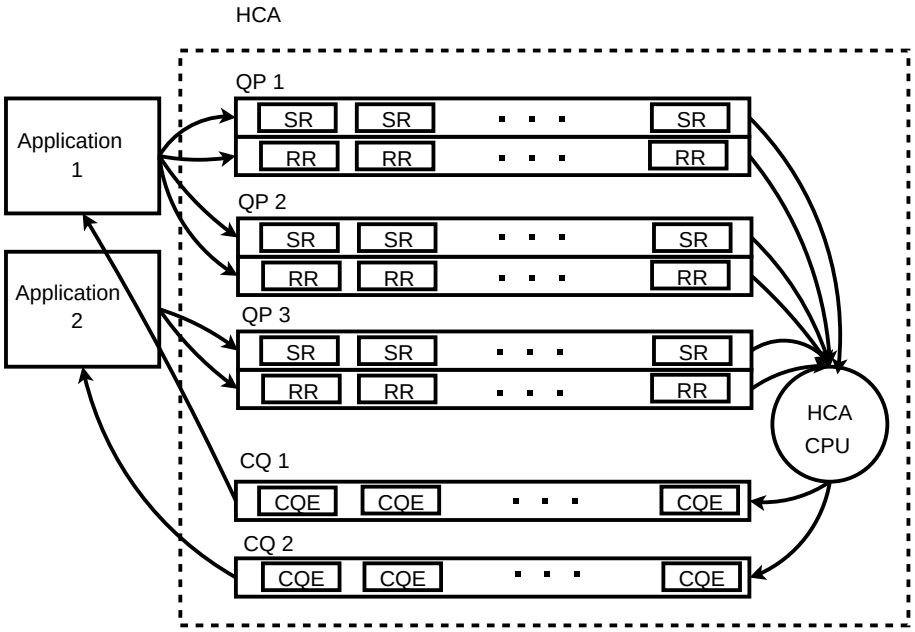

Figure 1.1: Hardware Queuing

A Receive Queue can only handle one type of Receive Request which specifies where to place the data which is received from another send operation on the remote side. Normal RDMA SR are usually not handled by Receive Queue Entries (RQE), except if the RDMA was issued with an 32 bit immediate value which consumes a RQE and puts the value into it.

1.3.3 Connection Management

The IBA supports connected and unconnected operation. The unconnected type uses datagrams to send and receive data and the connected type offers a virtual connection to send and receive (theoretically unlimited) streams of data. The segmentation and reassembly for connected types is done in hardware. To establish a connection, each partner has to create a QP and both have to tie them together. Each QP is uniquely addressed by the HCA LID (Local ID), optionally the GID (Global ID) and the local QP number. In comparison with the TCP/IP[11] protocol, the LID and GID correspond to the IP address (GID = network part, LID = host part) and the QP number correlates with the TCP port number. This information has to be exchanged in advance (e.g. through an out-of-band channel).

Unconnected QPs are not tied to a single remote node. The remote destination is given with each SQE. The same addressing scheme is used as for connected communication. This operation is expected to be generally slower than an already connected QP because most operations which are done during the connection establishment for connected QPs have to be repeated for each single packet.

1.3.4 Options for Message Passing

The IBA specification gives several options for passing a message from one system to another. One can use native IBA transport such as RDMA Read/Write, Send/Receive or other transport types like RAW Datagram encapsulating Ethernet or IPv6. The transport can be connection oriented or connectionless. Another choice is to use reliable transport or to handle the reliability in the application level. Table 1.1 shows a systematic list of service types and their related features.

[11] Transmission Control Protocol/Internet Protocol

Service Type	Connection	Reliable	Send/Receive	RDMA	Transport
Reliable Connection (RC)	y	y	y	y	IBA
Unreliable Connection (UC)	y	n	y	n	IBA
Reliable Datagram (RD)	n	y	y	y	IBA
Unreliable Datagram (UD)	n	n	y	n	IBA
RAW Datagram (RAW)	n	n	y	n	RAW

Table 1.1: InfiniBand™ Service Types

Each SQE has to fit to the QP regarding to the service type or it will be rejected. The following sections describe each of the IBA transport types to give a solid base for modelling the IBA and choosing the right connection type.

1.3.4.1 Reliable Connection (RC)

The steps for sending a message with the RC service type are shown in figure 1.2. It is assumed that the connection has been established in advance. The data is copied directly from the sender to the receiver's provided user-buffers (address, length) exploiting DMA functionality. Each message is acknowledged by transmitting an ACK packet back to the sender.

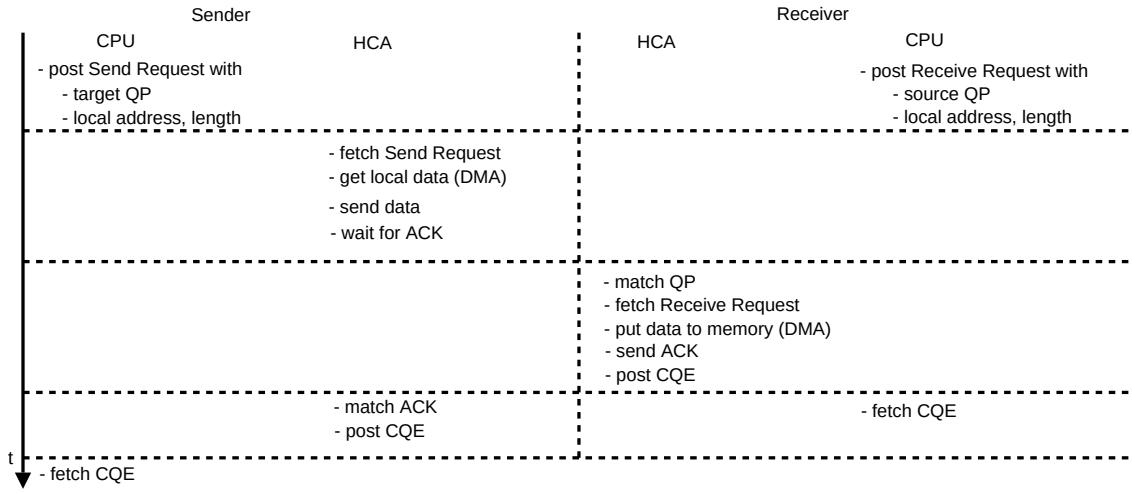

Figure 1.2: Reliable Connection

1.3.4.2 Unreliable Connection (UC)

The process of sending a message with the UC service type is shown in figure 1.3. It is assumed that the connection has been established in advance.

1.3.4.3 Reliable, Unreliable and RAW Datagram (UD/RD/RAW)

The process of sending a message with RD or UD/RAW is nearly the same as for RC (figure 1.2) or UC (figure 1.3) respectively. Only one additional step of finding the route has to be added to the HCA tasks on the sender side before sending a packet. This leads to the conclusion that datagram sending is generally slower than sending packets over an already established connection.

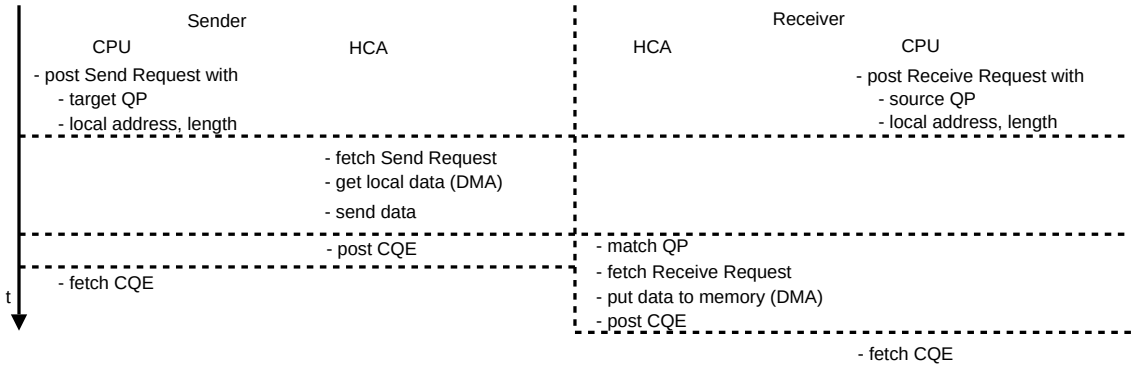

Figure 1.3: Unreliable Connection

1.3.5 Interacting with the HCA

All actions to send or receive data between the user process and the HCA are fully done in the user space of the operating system. The standard defines a so called verbs which is not a programming interface (API), but which defines necessary elements of an API so that the operating system vendor can develop his own version of the verbs API to interact with user programs. This means that verbs is a semantical description of functions which have to be offered to the user. This freedom resulted in mainly three different types of available verbs APIs, the Mellanox verbs API and the SF-IBAL verbs API. Both have been combined into the new emerging standard of the OpenIB verbs API. The OpenIB[12] initiative was founded by different vendors to specify a standard API for accessing the HCA verbs. Due to this standardization efforts, all verbs API examples are shown with the Mellanox VAPI[13] which is fully compatible with the new OpenIB API. More functions have been defined by Mellanox in addition to the standard and are called EVAPI[14]. Some of them are used in the source codes but they are also fully supported by the OpenIB API. The necessary VAPI calls to establish a connection and send a message between two nodes are shown in the following list (simplified). Both nodes perform equal actions for initialization and connection establishment. The only difference resides in the last call to `VAPI_post_sr(QP)` to post a Send Request (on the sender side) or `VAPI_post_rr(QP)` to post a Receive Request (on the receiver side). All memory to send or receive data has to be registered in advance to indicate the HCA the user buffer regions. The register operation locks the pages in memory so that a DMA access from the HCA can be guaranteed at any time.

1. `VAPI_open_hca()` - inits the HCA
2. `VAPI_alloc_pd()` - offers additional memory access rights (if more than one PD[15] is allocated)
3. `VAPI_create_cq()` - creates a new CQ
4. `VAPI_register_mr(ADDR, LEN)` - registers memory to send or receive data
5. `VAPI_create_qp(QP)` - creates a new Queue Pair
6. `VAPI_modify_qp(RST->INIT)` - modifies QP from Reset (RST) to Init
7. `VAPI_modify_qp(INIT->RTR, LID, QP_NUM)` - modifies QP from Init to Ready to Receive (RTR)
8. `VAPI_modify_qp(RTR->RTS)` - modifies QP from RTR to Ready to Send (RTS)
9. `VAPI_post_sr(QP)` - posts a Send Request to SQ (Sender)
10. `VAPI_post_rr(QP)` - posts a Receive Request to RQ (Receiver)
11. `VAPI_poll_cq(CQ)` - polls completion queue for new entries

[12] OpenIB [http://www.openib.org]
[13] verbs API
[14] Extended verbs API
[15] Protection Domain

12. `VAPI_deregister_mr()` - deregisters memory
13. `VAPI_destroy_qp(QP)` - destroys QP in HCA
14. `VAPI_destroy_cq(CQ)` - destroys CQ in HCA
15. `VAPI_dealloc_pd(PD)` - destroys PD
16. `VAPI_close_hca(HCA)` - closes HCA

A full list of VAPI/EVAPI functions is available from Mellanox or the OpenIB project.

1.4 Open MPI

Open MPI, presented in [GFB+04], is chosen as a framework to incorporate the algorithms, mainly because of its open and extensible framework. Incorporating new collective algorithms is fairly easy. The architecture of Open MPI will be briefly described in the following. However, Open MPI undergoes heavy development including the architecture which had recently major changes and it is not possible to know if more things will be changed in the future. So this description has to be seen to be linked to the current prerelease state of the art.

1.4.1 Component Framework

The architecture of Open MPI, described in [GFB+04] and [SL04] changed slightly and resulted in three distinct software layers:

1. **MPI** - MPI Layer
2. **RTE** - Run Time Environment
3. **MCA** - Modular Component Architecture

The MPI Layer is the adaption layer integrating the MPI standard into the underlying functionality (mainly the RTE and the MCA). The RTE layer provides services at run time (e.g. process startup or output forwarding). These layers do not have to be modified to incorporate new collective algorithms, so there is no need to investigate them further.

The MCA layer is a component framework, called Modular Component Architecture (formerly MPI Component Architecture). This framework manages other layers below by providing several services (e.g. finding components or processing user parameters). Each major functional area has an associated component framework to manage multiple components performing related or identical tasks. Each component is clearly defined by an interface and offers functional services to the upper layers of the framework. An initialized component is called a module and can be seen as an instance of the associated component.

The framework with all layers, some example component frameworks (components A and Z) and managed modules is shown in figure 1.4.

The next listing shows a number of frameworks already implemented in Open MPI. However, the architecture is flexible enough to add arbitrary functionality with new frameworks.

- **PTL** - The Point-to-point Transport Layer consists of network specific modules responsible for low level data transfer. It can be seen as a kind of device driver.
- **PML** - The Point-to-point Management Layer provides several transport services for the MPI Layer (e.g. segmentation and reassembly, striping or reliability).
- **COLL** - The Collective framework provides modules for collective operations.
- **TOPO** - The Topology framework offers processes running within an MPI job a facility which allows the MPI library components to perform optimizations based on locality (e.g. in grid environments).

List 1.3: Available Open MPI Component Frameworks

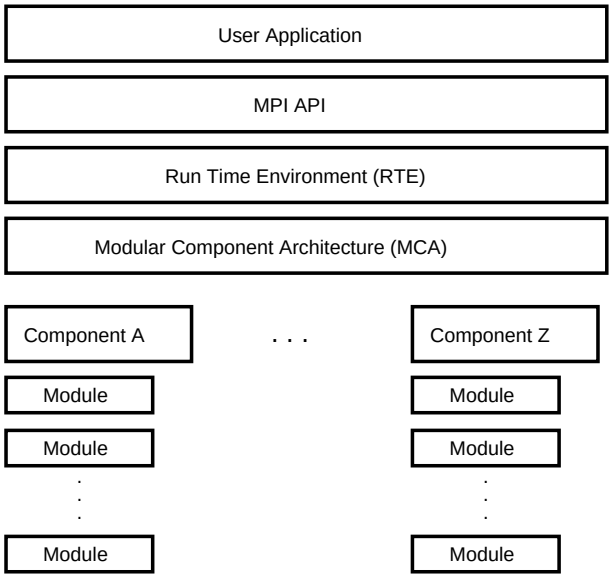

Figure 1.4: Open MPI Architecture

COLL is the most essential framework for this paper. The others are not investigated any further. To understand the structure of a single COLL component, the general structure of a component has to be described.

1.4.2 A Components Lifecycle

As described in [SL04], a component runs through five stages during its existence within the MCA: Selection, initialization, checkpoint/restart, normal operation and finalization. Figure 1.5 shows the order in which these stages are traversed. The COLL component is called module after initialization because each communicator is associated with a single COLL module (but all of these share the same source code). This means that only one instance of the COLL component offering a specific functionality can be active at any time, but each communicator has its own state (comparable to an instance) of this module.

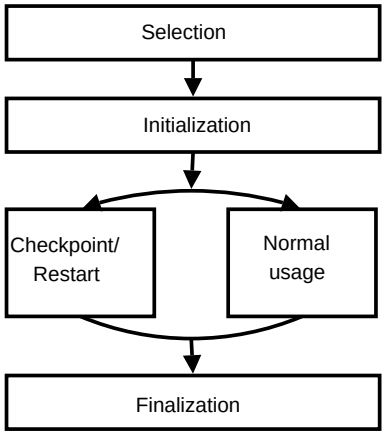

Figure 1.5: A Components Lifecycle

The "selection" is done during the creation of a new communicator (including `MPI_COMM_WORLD` and `MPI_COMM_SELF`), typically triggered inside the MPI API functions `MPI_INIT`, `MPI_COMM_CREATE`, `MPI_COMM_DUP`, `MPI_COMM_COMMIT` or `MPI_COMM_SPLIT`. The `mca_coll_<name>_comm_query` function of each available component is queried by the framework to return a list of function pointers to the offered

functions and a priority (between 0 and 100). The initialization function is also used to test the availability of special features inside the created communicator. If a required feature (e.g. an InfiniBand™ connection to all nodes) is not available, the component can simply disable itself by returning a null pointer. The component returning the highest priority is selected by the COLL framework.

The winning component enters the "initialization" phase and the COLL framework calls the `mca_coll_<name>_module_init` function. The component can also initialize internal data structures, hardware features or any other one-time-setup which is necessary to use the collective functions later on. The initialized data has to be associated with the communicator structure by changing the `c_coll_basic_data` pointer to the beginning of the data structure. The component returns a pointer at the framework which includes a list to all provided functions after the initialization work is done. If several functions are not supported, the function pointers should be null to indicate to the COLL framework that the basic functions (provided by a component named basic) should be used. The proposed module contains only the barrier function (null pointers for all other functions) and uses the initialization phase to perform the barrier setup phase which has to be done only once at startup. The relevant information is stored in a structure which is referenced by the `c_coll_basic_data` pointer.

The "checkpoint/restart" stage has to take care of messages which are currently on the fly and has to drain all queues. This is not required in our case because the component is layered on top of the PTL interface for sending and receiving messages (PTL takes care itself).

"Normal usage" is the state where requested collective operations (e.g. `MPI_Barrier()`) are performed. Therefore previously stored data may be extracted from the communicator. The functions are called by their function pointers which were provided to the COLL during the initialization phase.

The finalization step requests the module to clean up all used data structures and drain the network to unload cleanly. The function `mca_coll_<name>_module_finalize` is called to trigger the cleanup.

1.5 Summary

This chapter gave an introductory description of the basics for this thesis. The MPI Standard, the InfiniBand™ network and the Open MPI framework was described in all facings which are relevant for the reminder of this paper. The following chapter analyzes the possibilities to implement the barrier functionality in software.

Chapter 2

Software Solution

Implementing collective algorithms in hardware or in software are the two fundamental paradigms to optimize them. The latter possibility will be analyzed first. The software algorithm can be layered on top of normal message passing operations, without modifying the hardware design or adding new components. This approach can be generally considered as cheaper in terms of production costs and is more portable across different systems with the same underlying communication architecture. There are several well known algorithms which are currently used to perform the barrier by using normal MPI point-to-point operations on top of any message passing system. Most of them were developed for shared memory systems and have been adopted to distributed memory systems (like InfiniBand^{TM}). These algorithms are evaluated regarding to their running time and scalability (time to complete the barrier operation when all n nodes reach the barrier simultaneously). To perform these evaluations in a proper and accurate way, a model of the underlying network (in our case InfiniBand^{TM}) has to be found or developed if there is no suitable model available yet. The section 2.1 will investigate different models for parallel computation on distributed memory systems. Section 2.2 evaluates all currently known algorithms regarding to the chosen model and draws a conclusion to implement a single algorithm for performing the barrier operation.

2.1 Models for Parallel Computation

2.1.1 Introduction

The different barrier algorithms have to be modeled to find the optimal solution of the problem for the InfiniBand^{TM} network. The used model should reflect the network properties as accurate as possible. The following section describes several models which could be used to analyze barrier algorithms and predict the runtime behavior. The most accurate model is chosen as a result.

Models for parallel programming are often used to develop and optimize time critical sections of algorithms for parallel systems. These models should reflect all relevant parts of real-life-systems for algorithmic design. Several simplifying assumptions are taken to create these models. The next part of this thesis deals with analysis and evaluation of different well known models for their suitability to the InfiniBand^{TM} network architecture. All models are described and rated[1] in this work.

2.1.2 Related Work

Many models have been developed during the past years. Most of them are dedicated to a specific hardware or network architecture [Lei92, Ble87] or the shared memory paradigm [LCW93, GMR97]. There are also some general purpose parallel models which try to remain architecture independent like the PRAM [FW78b, KR90], the BSP [Val90], the C^3 [HK94] or the LogP [CKP$^+$93] model. These generic

[1] rating is done by comparing advantages and disadvantages for modeling InfiniBand^{TM}

programming models are characterized and used as starting point for further work. Several comparative studies and surveys are also available [MMT95, Ham96, BHP96], but they provide a limited view by comparing just a small subset of all available models.

2.1.3 Organization

Each mentioned model is described by its main characteristics. A reference to the original publication is given for additional details (e.g. detailed information in terms of execution time estimation). Each model is analyzed in advantages and disadvantages for modeling the InfiniBand™ architecture, and a conclusion for further usage in the design process of a new model is drawn. Several models have been enhanced by different modifications of third authors. Some of them and their implications for the usability of the underlying model are discussed in a separate subsection. The last section draws a conclusion and proposes a suitable model for the barrier operation over the InfiniBand™ network.

2.1.4 Characterization of available Models

2.1.4.1 The PRAM Model

The PRAM model was proposed by Fortune et al. in 1978 (see [FW78a]). It is the simplest parallel programming model known. But there are some serious defects in its accuracy. It was mainly derived from the RAM model, which bases itself on the "Von Neumann" model [vN45]. It is characterized by P processors sharing a common global memory. Thus it can be seen as a MIMD[2] machine. It is assumed that all processors run synchronously (e.g. with a central clock) and that every processor can access an arbitrary memory location in one step. All costs for parallelization are ignored, thus the model provides a benchmark for the ideal parallel complexity of an algorithm.

2.1.4.1.1 Evaluation

The main advantage is the ease of applicability. To reach this simplicity, several disadvantages have to be accepted. The main drawbacks are that all processors are assumed to work synchronously, the interprocessor communication is free[3] and it neglects the contention when different cells in one memory module are accessed.

Thus, this model is not suitable for any synchronization algorithm because interprocessor communication is free.

2.1.4.1.2 Additions to the PRAM Model

There are numerous additions to the PRAM model addressing its main disadvantages. The Module Parallel Computer (MPC [MV84]) consists of n memory modules with a specific size, where each module can be accessed by only one processor simultaneously. This models the memory bank contention in current multiprocessor systems. Other extensions [CZ89, Gib89] are modeling the natural asynchronity of current systems. The latency of write operations to non-local memory are modeled in the LPRAM [ACS90] or the BPRAM [ACS89] but they do match the properties of today's message passing based clusters[4]. Also the bandwidth is modeled for PRAM in the DRAM model [LM88]. But the problem of the unification of all different models, from which each model addresses a specific disadvantage, is still remaining. Thus, no single addition to the PRAM is suitable to satisfy the needs of modeling the InfiniBand™ architecture sufficiently.

[2] Multiple Instruction Multiple Data
[3] zero latency, infinite bandwidth leads to excessive fine-grained algorithms
[4] e.g. block-transfers are not possible, the latency is charged for each byte

2.1.4.2 The BSP Model

The Bulk Synchronous Parallel (BSP) model was proposed by Valiant in 1990 [Val90]. The BSP model divides the algorithm into several consecutive supersteps. Each superstep consists of a computation phase and a communication phase. All processors start synchronously at the beginning of each superstep. In the computation phase, the processor can only perform calculation on data inside its local memory[5]. The processor can exchange data with other nodes in the communication phase. Each processor may send at most h messages and receive at most h messages of a fixed size in each superstep. This is called a h-relation, further on. A cost of $g \cdot h$ (g is a bandwidth parameter) is charged for the communication.

2.1.4.2.1 Evaluation

Latency and (limited) bandwidth are modeled as well as asynchronous progress per processor. A big disadvantage for modelling barrier operations is the fact that the BSP also assumes special synchronization hardware (barrier is done in $O(1)$). Additionally, each superstep must be long enough to send and receive the h messages[6], resulting in some nodes being idle at the end of a superstep. This leads to the problem that messages received in a superstep cannot be used in the same superstep even if the latency is smaller than the remaining superstep length.

Because of the implicit synchronization, the BSP model is not suitable for modelling barrier algorithms. Each superstep begins in a globally synchronous state.

2.1.4.3 The C^3 Model

The C^3 model, proposed by Hambrusch et al in 1994 [HK94], was also developed for coarse grained supercomputers. The model works also by partitioning an algorithm into several supersteps. Each superstep consists of local computation followed by communication. Supersteps start synchronously directly after the preceding superstep is finished, this implies that a barrier without any costs is necessary (see also the BSP model in chapter 2.1.4.2).

2.1.4.3.1 Evaluation

The C^3 model evaluates complexity of communication, computation and congestion of the interconnect for coarse grained machines. Store-and-forward, as well as cut-through routing can be modeled and the difference between blocking and non-blocking receives is also considered.

But the disadvantages overbalance. The most significant drawback, which prevents the usage of the model for barrier synchronization, is the assumption that a barrier costs nothing and is implicit. Thus, the barrier would be modeled as a single empty superstep which makes no sense (see also section 2.1.4.2). Other drawbacks are that the message exchange can be performed only in fixed length packets and that the clock speed and bandwidth parameters are not included, so that the model is only valid when the processor bandwidth and the network bandwidth are equal[7]. Due to these facts, this model cannot be used for barrier operation over InfiniBand™ networks and is not investigated further on.

2.1.4.4 The LogP Model

The LogP model [CKP+93] was proposed by Culler et al. in 1993. It was developed in addition to the PRAM model (see chapter 2.1.4.1) to cover the changed conditions for parallel computing. It reflects different aspects of coarse grained machines which are seen as a collection of complete computers, each consisting of one or more processors, cache, main memory and a network interconnect[8].

[5] if this is data from remote nodes, it has been received in one of the previous supersteps
[6] the greatest h among all nodes!
[7] e.g. Intel Touchstone Delta
[8] e.g. the Intel Delta or Paragon, Thinking Machines CM-5 ...

It is based on four main parameters:

- L - communication delay (**upper** boundary to the latency for NIC-to-NIC messages from one processor to another)
- o - communication overhead (time that a processor is engaged in transmission or reception of a single message)
- g - gap (indirect communication bandwidth, minimum interval between consecutive messages, $bandwidth \sim \frac{1}{g}$)
- P - number of processors

List 2.4: The four parameters of the LogP model

The parameters of the LogP model can be divided into two layers, the CPU-Layer and the Network-Layer. The o-parameter can also be subdivided into one parameter on the receiver side (o_r) and another on the sender side (o_s). The according visualization of the different parameters for a given network (e.g. Ethernet) can be seen in Figure 2.1.

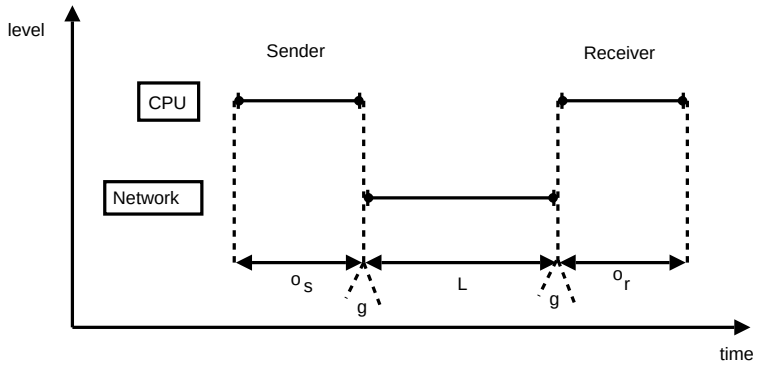

Figure 2.1: Visualization of the LogP parameters

There are several implicit assumptions taken with these four parameters to make the model fully functional:

- $\lceil \frac{L}{g} \rceil$ - count of messages that can be in transmission on the network from one to any other processor in parallel
- L, o and g are measured as multiples of the processor's clock speed

List 2.5: Additional assumptions in the LogP model

It is easy to understand that developing and programming in the PRAM model is easier than in the LogP model, but the bigger accuracy of this model should justify the additional effort. There were also some investigations to find upper bounds to the execution time for PRAM based algorithms on LogP (coarse grained) machines [LZ95] which can be used to simplify the programming again, but they are not suitable for synchronization algorithms (see 2.1.4.1).

An additional study [CLMY96] describes options of assessing the network parameters for real-life supercomputers. This can be very helpful to gain a deeper knowledge about the model's characteristics.

2.1.4.4.1 Evaluation

The LogP model has several advantages over other models. It is designed for distributed memory processors and the fact that network speed[9] is far smaller than CPU speed. It is easily applicable for a flat network model[10]. It encourages careful distribution of computation and overlapping communication as

[9] this means latency as well as bandwidth
[10] central switch based, diameter = 1

well as balanced network operations[11] which is very profitable for determining the running time of many applications accurately.

Some small drawbacks are that the whole communication model consists only of point-to-point messages. This does not respect the fact that some networks (especially InfiniBand™) are able to perform collective operations (e.g. multicast) ideally in $O(1)$. The second drawback is that only short fixed-size messages are modeled, but this can be ignored for the barrier problem.

2.1.4.4.2 Additions to the LogP Model

The LogP model seems to be very promising, thus the additions to the model are examined more in-depth than for the other models.

2.1.4.4.3 LogGP - Long Messages in LogP

The LogGP model was proposed by Alexandrov et al. in 1995 [AISS95]. It was meant as an addition to the original LogP model to address its inaccuracy for bigger messages. The LogP model is only suitable for small fixed size messages, because it uses the inter-frame gap g to express the bandwidth indirectly ($bw \sim \frac{1}{g}$). However, this ignores the fact that many modern interconnect technologies have special support for providing a much higher bandwidth for long messages[12]. The LogGP model incorporates the new parameter G into the original model to pay attention to this fact.

This extension will not be investigated any further because there is no necessity for long messages in the barrier functionality[13].

2.1.4.4.4 LoGPC - Modelling Network Contention

The traditional LogP model is only correct if no network contention occurs. But this assumption is (especially for n-cube networks) not very accurate. Therefore the LoGPC model [MF01] proposes a new technique to incorporate network contention and the pipelining characteristics of the DMA engine into the LogP model. The model was developed for k-ary n-cube networks [Aga91]. The author predicts that the model is easily applicable to other network topologies by changing a single parameter, but this seems not very precise for a central-switch based architecture.

This extension is quite interesting, but seems to be not adaptable to the special needs of a central-switch based InfiniBand™ network, because a k-ary n-cube network is not able to behave like a central switch based architecture.

2.1.4.4.5 LogGPS - Modelling Synchronization

The LogGPS model [IFH01] incorporates an additional delay which is caused by several MPI [For95] libraries due to the rendezvous protocol which is used for sending long messages. The synchronization overhead caused by the rendezvous protocol is modeled with the additional parameter S.

This model is only necessary for long messages sent through the MPI layer. Thus this model will not be investigated any further.

2.1.4.5 Choosing a Model

As described in 2.1, the LogP model is the most accurate model in this specific case. Thus, it is used for all running time estimations in the following sections.

Several simplifying architectural assumptions can be made without lowering the asymptotical accuracy of the model heavily.

[11] no single processor is "flooded"
[12] e.g. by pipelining or bulk transfers
[13] even zero-byte messages are sufficient to notify the status to other nodes

Based on the fact that most clusters operate a central switch connecting all nodes, the properties of this interconnect can be assumed as follows:

- full bisectional bandwidth
- full duplex operation (parallel send/receive)
- the forwarding rate is unlimited and packets are forwarded in a non-blocking manner
- the latency (L from LogP model) is constant above all messages
- the gap between consecutive messages is much smaller than the overhead to process a message on the host system $\Rightarrow$ g from LogP is much smaller than o and so overlaid while sending multiple messages (see section 3.1 in [CKP+93])
- the overhead (o) is constant for single messages (for simplicity: $o_s = o_r = o$)

List 2.6: Interconnect characteristics

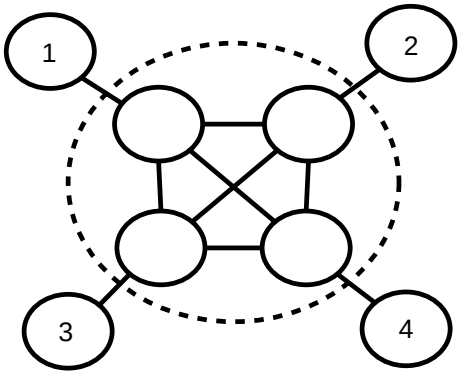

Figure 2.2: The ideal interconnect graph connecting 4 nodes

This model can be described as a graph where all nodes are connected to a fully meshed network. An example is shown in figure 2.2. A nearly ideal interconnect architecture can be manufactured by using the crossbar switch model, depicted in figure 2.3. This model is widely used to produce actually available switches.

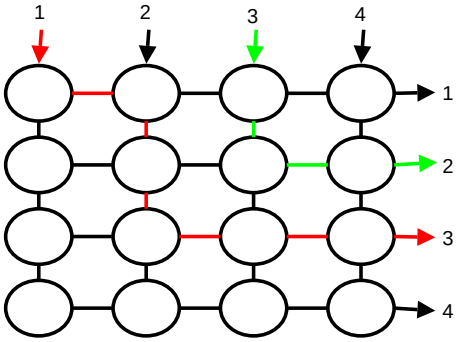

Figure 2.3: A crossbar example connecting 4 nodes

The pedantic communication characteristics are defined as follows: The time to send and receive a single message[14] can be approximated to $o_s + L + o_r$, and the time to send n messages[15] can be estimated as $o_s + (n-1)max\{o_s, g\}$[16]. The time to receive n messages (relative to the first packet sent on the sender side) can be modeled as $o_s + L + o_r + (n-1)max\{o_s, g\}$.

[14] $1:1$ communication
[15] $1:n$ communication with enqueuing
[16] the time o_s and g can run in parallel

Additionally, some constructs show up frequently and are defined as follows:

$$\begin{aligned} f_r &= max\{o_r, g\} \\ f_s &= max\{o_s, g\} \\ t_r &= max\{f_r, o_s + L + o_r\} \\ &= max\{max\{g, o_r\}, o_s + L + o_r\} \\ &= max\{g, o_s + L + o_r\} \\ t_s &= max\{f_s, o_s + L + o_r\} \\ &= max\{max\{g, o_s\}, o_s + L + o_r\} \\ &= max\{g, o_s + L + o_r\} \end{aligned}$$

With the aforementioned assumptions follows

$$\begin{aligned} f_r &= f_s = o \\ t_r &= t_s = 2o + L \end{aligned}$$

This paper distinguishes between t_r and t_s in order to emphasize the semantic properties of the algorithms being analyzed.

2.2 Barrier Algorithms

After defining the model which can be used for analyzing barrier algorithms, the following section describes all presently published algorithms and their asymptotic behavior for increasing processor counts. The LogP model is used to predict the asymptotic runtime behavior. The simplifying assumptions named in section 2.1.4.5 are taken for all further predictions.

The best way to understand each algorithm is to read the description in combination with the given graphical representation. To gain further knowledge about the algorithms, especially on message passing based systems, the reader is encouraged to retrace the proposed pseudo-code.

The following sections introduce all currently known barrier algorithms. Each algorithm can be split up logically into three phases. The algorithm is initialized in phase 1 (e.g. reserving shared objects or calculating ranks). So it has to be done only once during initialization or reconfiguration (processors enter or leave) of each communicator. Phase 2, also called "Check-in-Phase" has to be done on each node every time when it calls MPI_Barrier. All nodes communicate with each other until one or all nodes know that every node reached its MPI_Barrier call. A barrier-identifier is often used to distinguish between different MPI_Barrier calls to avoid race conditions when one processor enters the next barrier before all other processors left the last barrier - this is called x in the following chapter. Each barrier number is used once per communicator and incremented for each barrier starting initially with 1. The third and last phase can be referred to as "Notification-Phase" and is only needed when not all processors know that the barrier has been reached by each member of the communicator. The typical case is that one processor knows that the barrier is reached by each member of its communicator and it has to notify all remaining processors. The difference in phase 3 leads to a distinction between two types of algorithms. The first type performs phase 3 as described above (see section 2.2.1) and the second type omits it completely (see section 2.2.2 on page 28). Section 2.2.3 summarizes all algorithms for future analysis and provides simplified information about running time and memory usage relative to the processor count P. A proof of optimality is given in section 2.2.4 followed by a classification of the algorithms in four complexity groups (section 2.3.5), regarding to the LogP model. One representative of each group is analyzed more in detail for the runtime and the asymptotical behavior is compared to a practical benchmark result. Basing on this comprehensive analysis, two new algorithms which efficiently utilize hardware parallelism are proposed in section 2.2.6 and modeled in the LogP model. The running time for all algorithms is assessed under the assumption that the LogP model is accurate for the underlying network and that all nodes arrive simultaneously in their `MPI_Barrier()` call (balanced case).

2.2.1 Algorithms Performing Phase 3

Phase 3, as described in Section 2.2 can efficiently be implemented as a broadcast (e.g. MPI_Bcast). This operation could especially benefit from hardware broadcast or multicast capabilities which perform (ideally) in $O(1)$. If this is not capable with the underlying architecture[17], standard broadcast algorithms could be used, which usually scale with $O(log(P))$ for 1 byte messages. The time which is necessary to perform a broadcast from one to n nodes is modeled as $t_{bc}(P)$ regardless of the implementation and architectural details mentioned above.

2.2.1.1 Central Counter

2.2.1.1.1 Description

This algorithm is quite simple and straightforward. Because of its obvious simplicity and the naive prove for correctness it is implemented quite often. Especially the atomic "fetch-and-Φ"[18] operation is frequently mentioned related to this barrier. This approach is investigated for the fetch-and-increment[19] operation in [FG91] and [GVW89]. One node holds an integer value which is used as central barrier counter. This integer starts with 0 and is increased by each node once (after it entered the barrier) until the node count P is reached. The last node sends a message to all other nodes to activate them.

This barrier consists of the two parts counting and notification. Both parts can be optimized independently. Optimized algorithms for counting and broadcasting a message are evaluated later. We assume the easiest case in the following pseudo-code (see listing 2.1) and graphical representation (see figure 2.4).

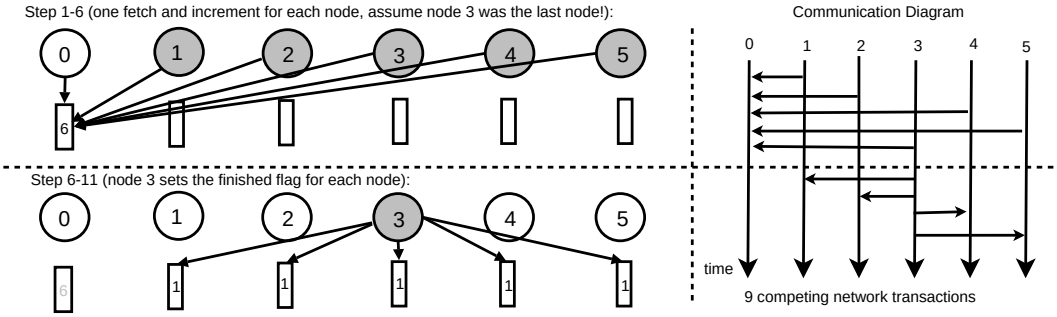

Figure 2.4: A Central Counter barrier between 6 nodes

2.2.1.1.2 Conclusion

As the algorithm splits up into two phases for each MPI_Barrier call, each phase is analyzed apart. Phase one is critical, because the shared counter is altered by each node. This memory location is called a hot-spot [PN85]. $P-1$ competing network transfers are needed to implement the counter and the running time would be $L+P\cdot o$. These operations have to be atomic on the target to prevent lost-update problems, resulting in deadlocks. Phase two is also critical, because one node has to inform all other nodes. Phase one disturbs the transaction scheduling of the memory controller. Regarding to 2.2.1, the possibilities to perform this broadcast are not mentioned here. Thus, the overall running time of this operations can be seen as $L+P\cdot o+t_{bc}(P-1)$. The memory usage is constant (1 byte) per node.

[17] regardless if it's provided by hardware or software

[18] "fetch-and-Φ" is a conceptional term for a collection of atomic operations which change and return a single value in memory - for example fetch-and-add, fetch-and-swap, fetch-and-inc, ...

[19] the fetch-and-increment operation takes a value to increment from its caller, increments its memory value and returns the new value to the caller (some implementations may return the value before incrementing)

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
if rank == 0 then
   // its my counter
   reserve ctr with 1 entry as shared
   set ctr = 1
else
   reserve flag with 1 entry as shared
   set flag = 0
ifend

// phase 2/3 - central barrier
set x = x + 1;
if rank == 0 then
   wait until ctr == p
else
   set localctr = fetch and increment ctr on node 0
   if localctr == p then
      set flag in all nodes to x
   ifend

   wait until flag >= x
ifend
```

Listing 2.1: Central Counter in Pseudocode

2.2.1.2 Combining Tree

2.2.1.2.1 Description

The combining tree barrier was introduced by Yew, Tzeng and Lawrie in [YTL87]. It uses a tree to speed up the central counter barrier. It divides the nodes into subgroups with n members, which synchronize among each other with a simple shared counter. Every first node of each group spins[20] its local counter which is shared to all others until all nodes reach the barrier ($counter == n$). When all nodes in the subgroup reached the barrier, all first nodes form a new group and synchronize among each other. This is repeated until only one group is left and has finished the synchronization. The first node informs all other nodes about the barrier completion. Yew reported a group-count (n) of 4 to achieve the best results. A graphical example as well as pseudocode for this algorithm can be found in figure 2.5 and listing 2.2.

2.2.1.2.2 Conclusion

The combining tree barrier reduces hot spots in memory and network contention. The number of required network operations is naively seen lowered to $log_n P$ steps[21]. But due to enqueuing during the receive, the actual execution time under the assumption of the LogP model is $(L + n \cdot o) \cdot \lceil log_n P \rceil + t_{bc}(P-1)$ and 2 bytes of memory are used per node.

[20] check the counter frequently

[21] this is only valid for a fan-out of n - e.g. in a mesh topology, it has to be seen as a naive approximation for all other cases

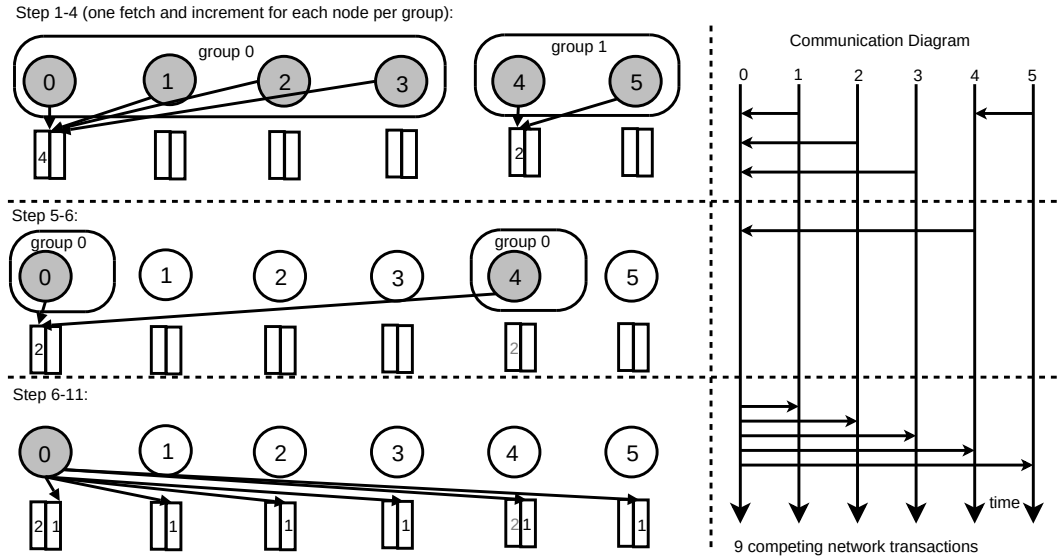

Figure 2.5: A combining tree barrier between 6 nodes

```
set p = number of participating processors
set n = nodes per group // parameter
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter

reserve ctr with 1 entries as shared
set ctr = 1
reserve flag with 1 entries as shared
set flag = 0

set round = 0 // actual round
set relnodeid = 0 // relative nodeid (only active nodes)

// phase 2 - barrier
set x = x + 1;
repeat
   set round = round + 1
   set relnodeid = rank / (n^(round-1))
   set grpnum = relnodeid div n // group number?
   set grprank = relnodeid mod n // my rank in group

   // I am out of the game, when I have no
   // natural number as relnodeid
   if round(relnodeid) != relnodeid then
      wait until flag >= x
   ifend

   if grprank == 0 then
      wait until ctr == n
   else
      set ctr = fetch and increment ctr on node \
         rank-grprank*n^(round-1)
      wait until flag >= x
   ifend
until round == log(n)(p) or flag >= x

// phase 3
if rank == 0 then
   set flag in all other nodes to x
ifend
```

Listing 2.2: Pseudocode for Combining Tree Algorithm

2.2.1.3 Tournament

2.2.1.3.1 Description

The Tournament Barrier, proposed by Hengsen et al. in [HFM88] is mostly suitable for shared memory multiprocessors because it benefits from several caching mechanisms. Nevertheless, the algorithm is analyzed here. As in the Butterfly (see chapter 2.2.2.1) and the Dissemination Barrier (see chapter 2.2.2.3), different rounds are used. The algorithm is similar to a tournament game. In each round two nodes play against each other. The winner is known in advance and waits until the looser arrives. The winners play against each other in the next round. The overall winner (the champion) notifies all others about the end of the barrier. A graphical and pseudo-code representation can be found in figure 2.6 and listing 2.3.

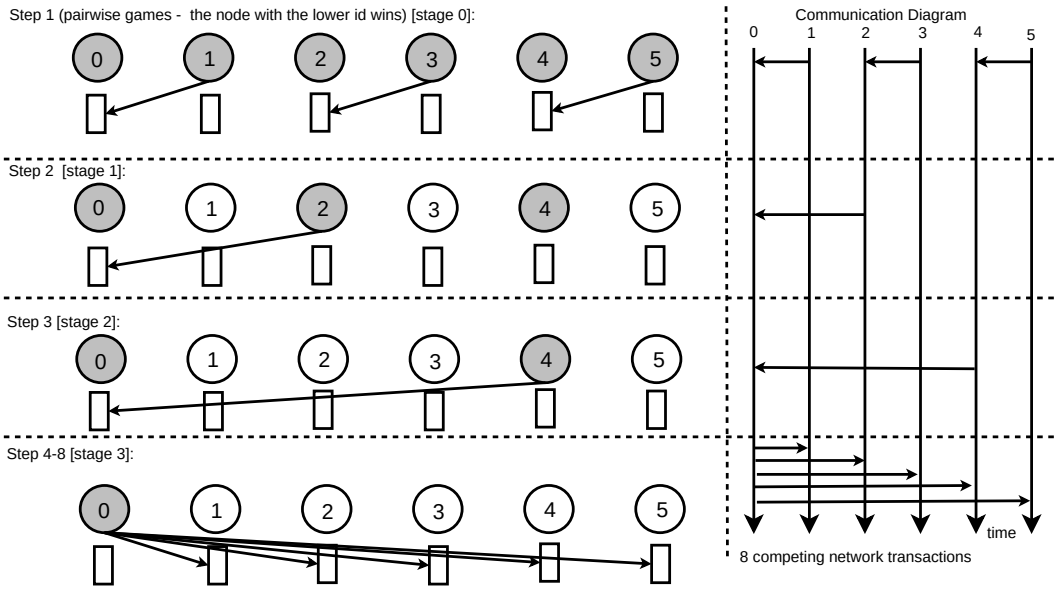

Figure 2.6: Example for the tournament barrier with 6 nodes

2.2.1.3.2 Conclusion

The algorithm is also subdivided into two parts. Part one (the game) scales with $log_2 P$ and uses 1 byte of memory. Part two scales as mentioned in chapter 2.2.1 with $t_{bc}(P-1)$. Thus the entire runtime can be estimated with $(L + 2 \cdot o) \cdot \lceil log_2 P \rceil + t_{bc}(P-1)$.

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
reserve flag with 1 entries as shared
set flag = 0

// phase 2 - done for every barrier
set true = 1
set false = 0
set round = -1
// repeat log(p) times
repeat
   set round = round + 1
   set peer = rank xor 2^round

   // I have no partner -> next round ...
   if peer > p then
      continue
   ifend

   // I am the winner
   if rank > peer then
      wait until flag == true
      set flag = false
   else
      set flag on peer = true
      wait until flag == true
   ifend
until round > ld(p)

// phase 3 - node 0 ever wins
if rank == 0 then
   set flag in all other nodes to true
ifend
```

Listing 2.3: Pseudo Code for Tournament Barrier

2.2.1.4 f-way Tournament

2.2.1.4.1 Description

The f-way Tournament Barrier bases on the same principle as the Tournament Barrier (section 2.2.1.3). It was proposed by Grunwald et al. in 1993 [GV94]. The most important difference is that more than two processors are competing in one game. A graphical representation can be found in figure 2.7. The pseudo-code is nearly identical to the tournament barrier (see listing 2.3), only with more than two nodes.

2.2.1.4.2 Conclusion

This barrier is suitable for special network topologies with a fan-out of more than one (e.g. torus networks). But should not scale better on standard central switching-based networks. The algorithm scales theoretically (with a fan-out of f in each node) with $log_f P$ network transactions and 1 bytes of memory per node, but is practically limited by the network infrastructure which serializes and enqueues concurrent requests. Thus, the predicted runtime within our model is $(L + f \cdot o) \cdot \lceil log_f P \rceil + t_{bc}(P - 1)$.

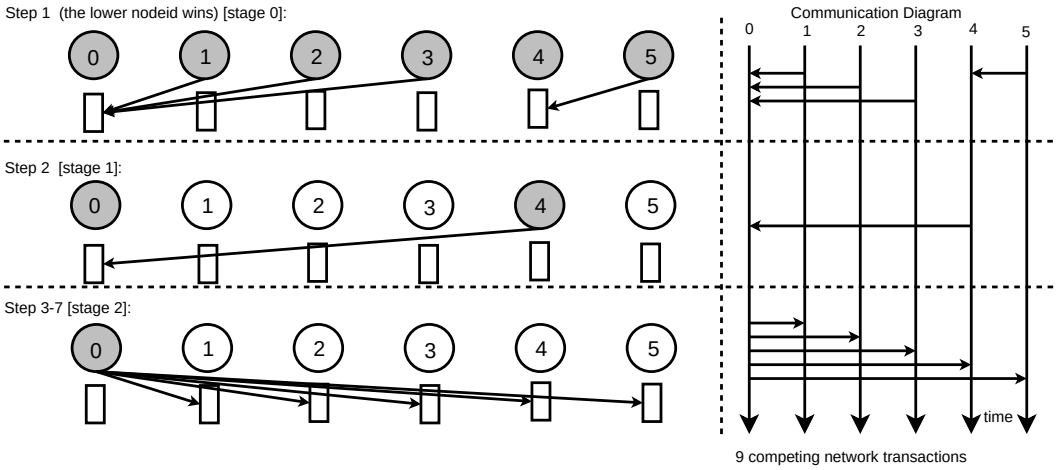

Figure 2.7: Example for the 4-way tournament algorithm between 6 nodes

2.2.1.5 MCS

2.2.1.5.1 Description

The MCS Tree Barrier was proposed by Mellor-Crummey and Scott in 1991 [MCS91a, MCS91b, SMC94]. It uses also a tree structure and is quite similar to the Combining Tree barrier (see chapter 2.2.1.2). Each node is assigned to a tree node. The resulting n-ary tree consists of all nodes, each node has an array of n flags. All, but the top node write to their parent's node flag when all child nodes wrote the flag to them. All nodes, which have no children start with the array initialized with true. When the topmost node's flag array is completely filled, it notifies the others.

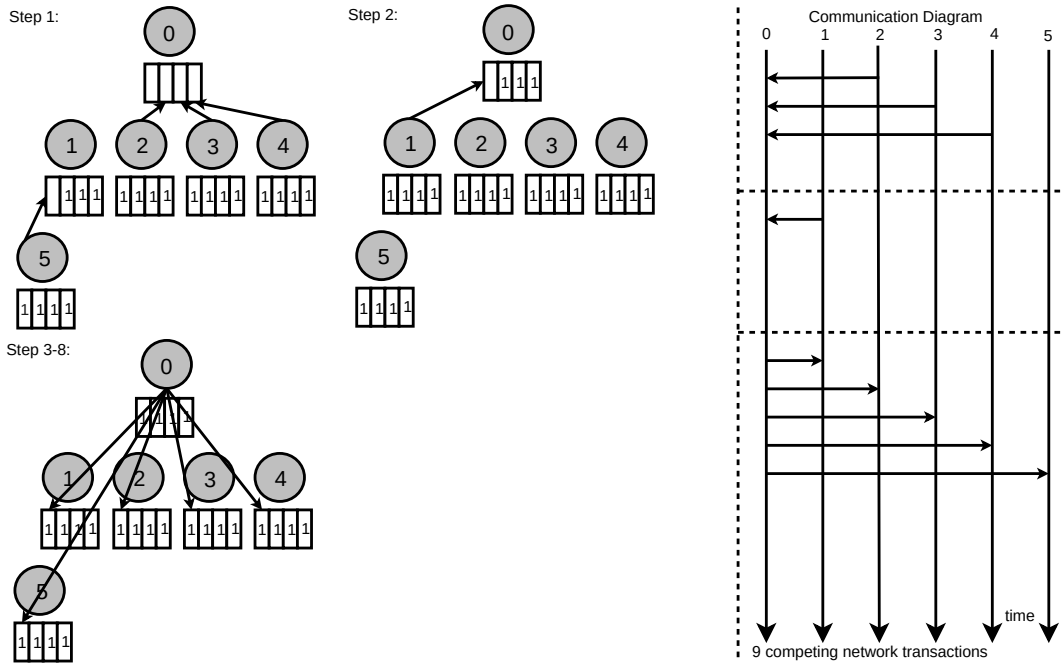

Figure 2.8: Example of the MCS Tree algorithm between 6 nodes

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id
set n = number of childnodes

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
reserve array with n+1 entries as shared
  //-> array[n] acts as barrier_reached flag

// phase 2 - done for every barrier
set x = x + 1

// initialize my flags (flag == 1 if no child is present)
for j in 0..n-1 do
   if p >= (rank * n) + 1 + j then
      set array[j] = 0
   else
      set array[j] = 1
   ifend
forend

set array[n] = 0
repeat
   set parent = (rank-1) div n
   set slot = (rank-1) mod n

   if sum(array[0..n-1]) == 4 then
      if rank == 0 then
         set array[n] = 1
      else
         set array[slot] in parent to 1
      endif
   endif
until array[n] == 1

// phase 3
if rank == 0 then
   set array[n] in all other nodes to 1
ifend
```

Listing 2.4: Example of the MCS Tree algorithm between 6 nodes

2.2.1.5.2 Conclusion

The MCS-Barrier uses a tree structure with a fan-out of n to improve the barrier performance to $log_n P$ concurrent network transactions (only if the network offers a fan-out of n) and n bytes of shared memory per node in the first part. The second notification part depends as usual on the underlying network architecture and scales with $t_{bc}(P-1)$ competing network transactions. The overall execution time for our model can be predicted with $(L + (n+1) * o) * \lceil log_n P \rceil + t_{bc}(P-1)$.

2.2.1.6 BST

2.2.1.6.1 Description

The Binomial Spanning Tree (BST) Barrier was proposed by Tzeng et al. in 1997 - [TK97]. It uses a binomial tree structure[22], which reduces the network contention by its principle. The working principle is quite similar to the MCS Barrier (2.2.1.5) - every processor is assigned to one tree-node and waits until all children reached their barrier (they notify their parent) and then notifies its own parent. A binomial tree is built up recursively, the whole tree of step $j-1$ is appended to the root node in step j. The principle is shown in figure 2.9.

This special characteristic is used to avoid contention on single nodes.[23] To manage the processor-to-tree-node assignment, the following numbering scheme is used:

- each node is numbered in binary digits (from 0 to $P-1$)
- each node calculates it's parent by resetting the leftmost "1" in it's own id to "0"
- each node calculates it's children by adding 2^i to it's own id where $i = \{i \in N \wedge log_2 id < i < \lceil log_2 P \rceil \wedge id + 2^i < P\}$

A numbered binomial tree with 6 nodes is shown in figure 2.10. Pseudocode for the algorithm can be found in listing 2.5.

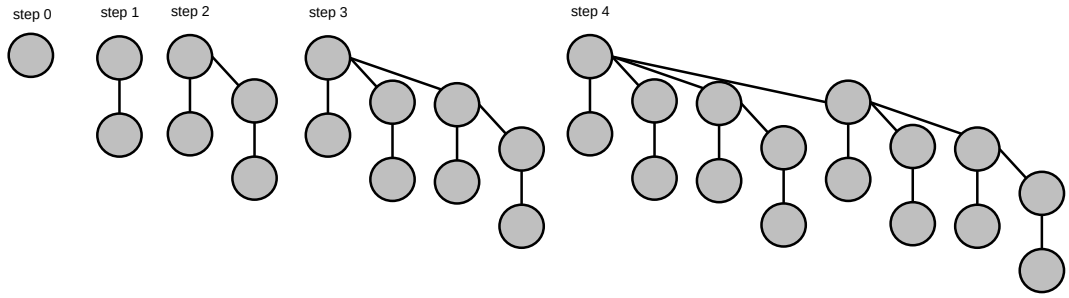

Figure 2.9: Example for building a binomial tree

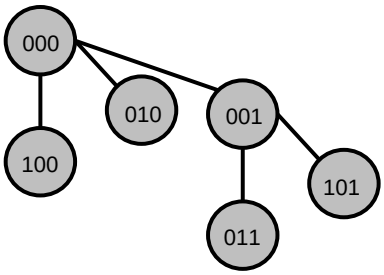

Figure 2.10: A numbered binomial tree with 6 nodes (each processor is assigned to one tree node)

2.2.1.6.2 Conclusion

The binomial spanning tree barrier minimizes the concurrency at the root node. One child of the root node finishes each round. The root node has typically $\lceil log_2 P \rceil$ children, so that the root node knows after $\lceil log_2 P \rceil$ steps that all nodes reached the barrier. So the time for check in scales with $\lceil log_2 P \rceil$. The notification of all nodes scales with $t_{bc}(P-1)$. Thus, the overall execution time is $(L+2\cdot o)\cdot log_2 P + t_{bc}(P-1)$ for all power of two node-counts ($P = 2^x$) and in case of congestion (receiver enqueuing) $(L+3\cdot o)\cdot \lceil log_2 P \rceil + t_{bc}(P-1)$ for all other node-counts. The required memory scales with $log_2 P$ bytes.

[22] which is very similar to a hypercube

[23] due to the distribution of nodes in a binomial spanning tree, each network link is utilized at most once per round if p is a power of two - for all other node-counts, each link is utilized at most twice

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1: initialization
set x = 0 // the barrier counter
reserve array with p entries as shared
// could be shortened to ld(p)

// set all array entries to '1'
for j in 0..p-1 do
   set array[j] = 1
forend

// determine parent (reset leftmost '1')
set j = 1
while j <= rank do
   set j = j * 2
whileend

set parent = rank - j/2

// determine children - unset their array entries
for j=0..ceil(ld(p))-1 do
   // ld(0) is not defined - take all entries for root node
   if rank == 0 or j > ld(rank) then
      set k = rank + 2^j
      // only for rank + 2^j < p
      if k < p then
         array[k] = 0
      ifend
   ifend
forend

// phase 2: check in phase
// wait until all children reached their barrier
for j in 0..p-1 do
   wait until array[rank] == 1
forend

if rank != 0 then
   set array[rank] in node parent to 1
ifend

// phase 3: release phase
// use array[0] as finished indicator,
// because node 0 is the root -
// nobody has it as child node
if rank == 0 then
   set array[0] in all nodes 0;
else
   wait until array[0] == 0
ifend
```

Listing 2.5: Pseudocode for BST Barrier

2.2.2 Algorithms Omitting Phase 3

2.2.2.1 Butterfly

2.2.2.1.1 Description

The Butterfly Barrier was proposed by Brooks in 1986 [Bro86]. The original algorithm uses a single shared array of flags (shared memory) and performs several stages of pairwise synchronization. The used algorithm can be described easily in the following way:

1. wait until previous stages finished (until my flag is false)
2. set my flag to true (I am currently synchronizing)
3. wait for the partner's flag to become true (the partner is ready)
4. set the partners flag to false (done)

After the initial synchronization finished the whole process is repeated $w = log_2 P$ times, each time is called a stage. The stages (s) are numbered ascending, the very first stage starts with 0. Each node p_i synchronizes in each stage with node p_j where $j = i \ XOR \ 2^s$ (see figure 2.11). This method only works for $p = 2^x;\ x \in N$ (p = power of two). For all other number of nodes, the necessary pairs are represented virtually by the other nodes (e.g. to synchronize 6 nodes, 2 additional virtual nodes are necessary). Thus this algorithm performing worst with any number of nodes, slightly bigger than a power of two.

The array mentioned above has to have the dimensions $P \cdot log_2 P$. One column per processor and one row for each round.

This implementation does not scale very well on a message passing based system (because of the shared array). After applying all modifications to ensure scalable operation on message passing based systems, the algorithm looks very similar to the Pairwise Exchange (section 2.2.2.2). Thus, this paper does not propose a pseudo code.

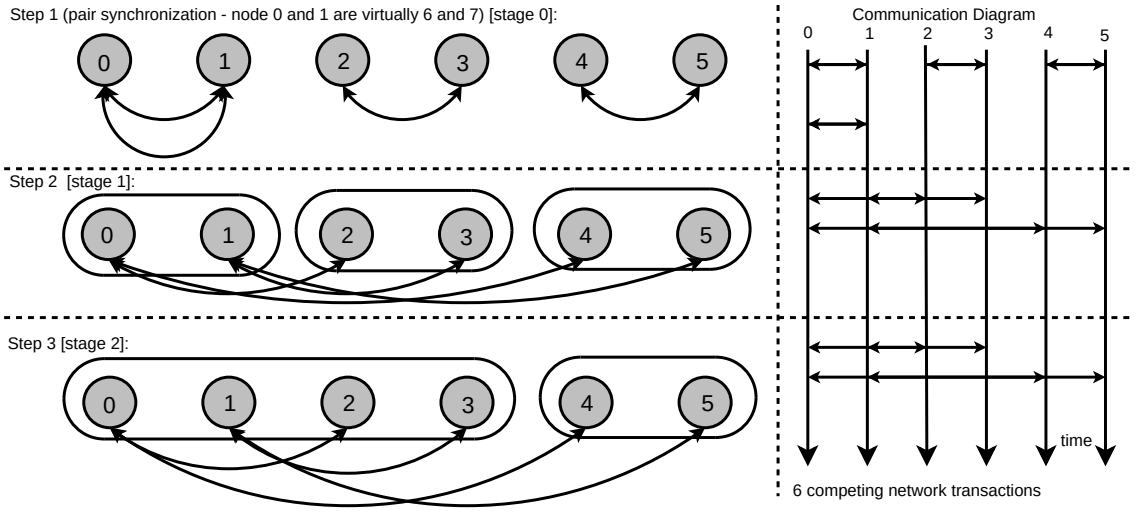

Figure 2.11: The Butterfly algorithm - the shared array was left out to improve the clearness

2.2.2.1.2 Conclusion

The barrier's competing network operations scale best with processor numbers which are a power of two with $log_2 P$. The worst case is when the processor number is slightly higher than a power of two with $2 \cdot log_2 P$ because nearly all processors must synchronize twice. Thus the overall performance is $(L + 2 \cdot o) \cdot log_2 P$ for $P = 2^x$ and $(2 \cdot (L + 2 \cdot o)) \cdot \lceil log_2 P \rceil$ for all other P (worst case). The memory used by the shared array of flags scales with $P \cdot log_2 P$ in size. Due to the above mentioned problems, the Pairwise Exchange Barrier (chapter 2.2.2.2) should be implemented in message passing based systems instead of the Butterfly Barrier.

2.2.2.2 Pairwise Exchange With Recursive Doubling

2.2.2.2.1 Description

This algorithm was proposed in [GTNP02] and will be discussed in the following section. All nodes group themselves in pairs (node 0 and node 1 for each pair) during the first part of the pairwise exchange algorithm. The barrier-identifier x, described in chapter 2.2 is used to avoid several race conditions.

In the first part, all nodes write their value of x to the corresponding peer. After node 0 and node 1 of each pair have received the correct barrier value[24] from their peer, they continue and enter the next stage. Each group peers with another group of two processors and each member of the group writes the barrier number to its corresponding peer in the other group. This procedure is recursively repeated until all nodes form one big group. So this algorithm uses $\lfloor log_2 P \rfloor$ network write operations per node.

Thus this works only for power of two nodes. For all other node counts P, the biggest power of two $y = 2^z, z \in \mathbb{N}$ is calculated which is smaller than P. This creates two groups, namely group A including y nodes and group B with the remaining nodes. Every single node in group B pairs with another node in group A. When a node of group B reaches the barrier it writes the barrier number to its peer node in the group A. Each of these nodes in group A waits until it receives the barrier number from the second's group partner before it starts the normal pairwise exchange algorithm. When the barrier is finished, each peer node in group a notifies its partner that the barrier is finished. This extension for non power of two node counts increases the latency to $\lfloor log_2 P \rfloor + 2$ network write operations.

Figure 2.12 gives a graphical representation of a barrier with 6 nodes. After step 4, node 0 has all necessary information (that all nodes entered the barrier already) - node 1,2 and 4 communicated directly with node 0 and the other nodes finished before node 1,2 or 4.

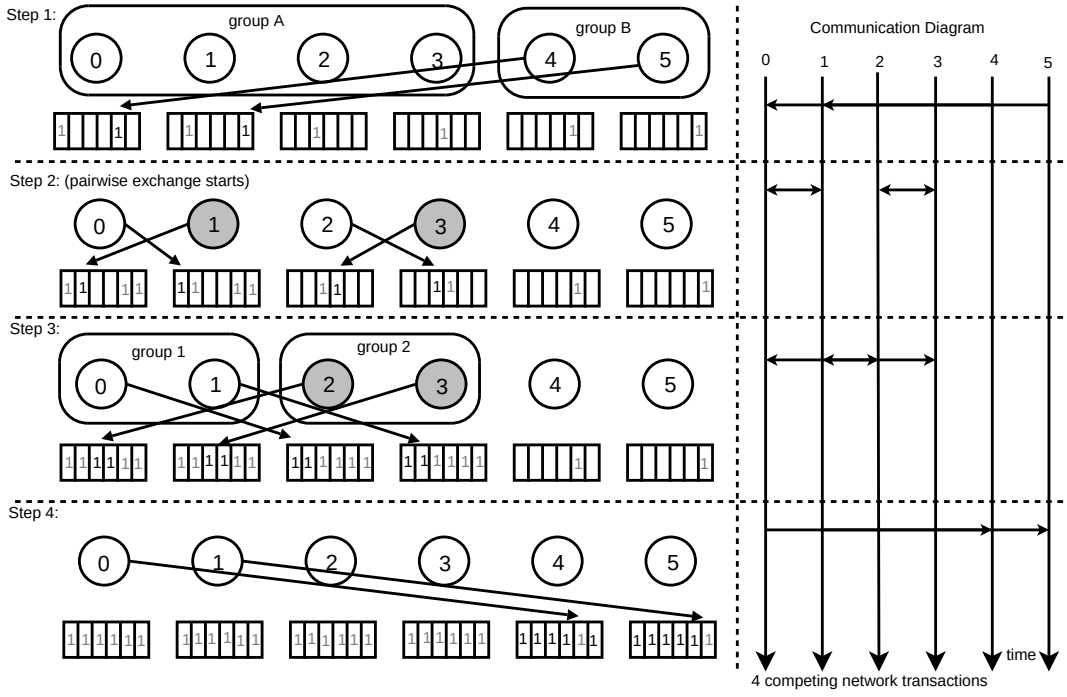

Figure 2.12: Example for the pairwise exchange algorithm between 6 nodes

2.2.2.2.2 Conclusion

The algorithm uses a maximum of $\lfloor log_2 P \rfloor + 2$ network writes and P bytes memory per node. The overall runtime can be estimated with $(L + 2 \cdot o) \cdot log_2 P$ for all $P = 2^x$ and with $(L + 2 \cdot o) \cdot \lfloor log_2 P \rfloor + 2 \cdot (L + 2 \cdot o)$ for other values of P (worst case). Thus, the algorithm can be used to exploit the advantages of RDMA operations efficiently.

[24] the currently active barrier number or each number higher than this

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
reserve array with p entries as shared
for i in 0..p-1 do
   set array[i] = 0
forend
set x = 0 // the barrier counter
y = 2^floor(ld(p)) // the 2^z count

// barrier - done for every barrier
set x = x + 1
if rank >= y then
   // I am in group b, my partner is node i-y in group a
   set array[rank] in node rank-y to x
   // wait for notification from partner
   wait until array[rank] >= x
else
   // I am in group a
   if p-y > rank then
      // I have a partner in group b
      // wait for partner
      wait until array[rank+y] >= x
   ifend

   // the pairwise exchange algorithm
   set round = -1

   // repeat log(p) times
   repeat
      set round = round + 1

      set peer = rank XOR 2^round

      set array[rank] in node peer to x
      wait until array[peer] >= x
   until round == (log(y)-1)

   if p-y > rank then
      // I have a partner in group b
      // notify partner
      set array[rank+y] in node rank+y to x
   ifend
ifend
```

Listing 2.6: Pseudocode for the pairwise exchange barrier

2.2.2.3 Dissemination

2.2.2.3.1 Description

The Dissemination Barrier, introduced by Hengsen, Finkel and Manber in 1988 [HFM88], is mostly an improvement of the Butterfly Barrier for non power of two processor counts. In every round s, each processor p_i synchronizes with p_j where $j = (i + 2^s)\ mod\ P$. Each processor sets the flag in the cyclically

next processor to x and waits for the circular previous processor to set its own flag to a value greater than x. The algorithm is basically the same as used in the butterfly barrier but with different partners.

The implementation with a central shared array does not scale very well on a message passing based system. Thus this paper proposes a more suitable solution for message passing systems.

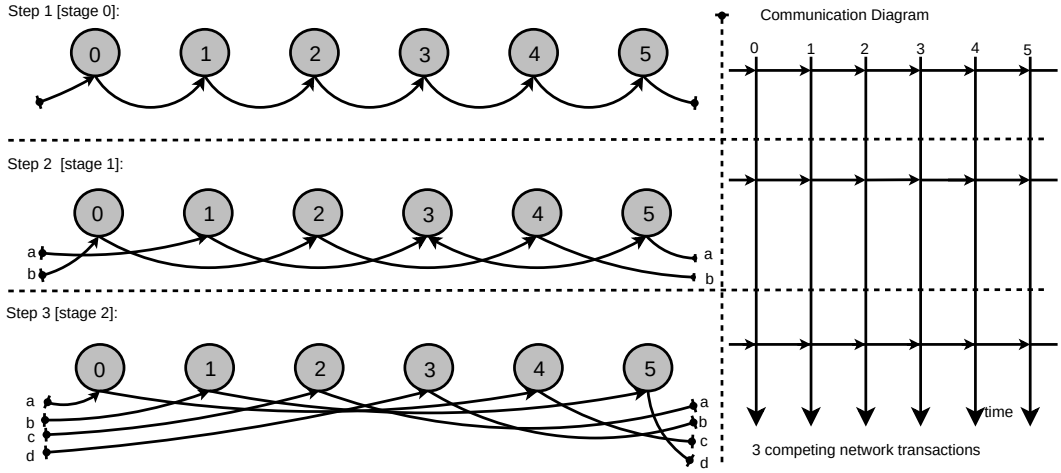

Figure 2.13: Dissemination Barrier with 6 processors

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
reserve array with p entries as shared
for i in 0..p-1 do
   set array[i] = 0
forend

// barrier - done for every barrier
set round = -1
set x = x + 1
// repeat log(p) times
repeat
   set round = round + 1

   set sendpeer = (rank + 2^round) mod p
   set recvpeer = (rank - 2^round) mod p

   set array[rank] in node sendpeer to x
   wait until array[recvpeer] >= x
until round >= log(p)-1
```

Listing 2.7: Pseudocode for the Dissemination Barrier

2.2.2.3.2 Conclusion

The Dissemination Barrier scales better than the Butterfly Barrier also for non power of two processor counts with $\lceil log_2 P \rceil$ competing network transactions. The overall runtime can be predicted as $(L + 2 \cdot o) \cdot \lceil log_2 P \rceil$. The algorithm uses P bytes of memory per node.

2.2.3 Summary of Algorithms

Table 2.1: Summary Table

Algorithm	**Network operations**	**Complexity**	**Memory**	**Complexity**
Central Counter	$L + Po + t_{bc}(P-1)$	$O(P)$	1	$O(1)$
Combining Tree	$(L+no)\lceil log_n P\rceil + t_{bc}(P-1)$	$O(nlog_n P)$	2	$O(1)$
Tournament	$(L+2o)\lceil log_2 P\rceil + t_{bc}(P-1)$	$O(log_2 P)$	1	$O(1)$
f-way Tournament	$(L+fo)\lceil log_f P\rceil + t_{bc}(P-1)$	$O(flog_f P)$	1	$O(1)$
MCS	$(L+(n+1)o)\lceil log_n P\rceil + t_{bc}(P-1)$	$O(nlog_n P)$	P	$O(P)$
BST	$(L+3o)\lceil log_2 P\rceil + t_{bc}(P-1)$	$O(log_2 P)$	$\lceil log_2 P\rceil$	$O(log_2 P)$
Butterfly	$(2(L+2o))\lceil log_2 P\rceil$	$O(log_2 P)$	$P\lceil log_2 P\rceil$	$O(Plog_2 P)$
Pairwise Exchange	$(L+2o)\lfloor log_2 P\rfloor + 2(L+2o)$	$O(log_2 P)$	P	$O(P)$
Dissemination	$(L+2o)\lceil log_2 P\rceil$	$O(log_2 P)$	P	$O(P)$

The running time of an algorithm depends mainly on the number of network send operations[25] and the network congestion. Another limiting constraint is the consumed memory per node. Thus, table 2.1 summarizes the different algorithms and puts the number of performed network operations and the required amount of memory per node in the context. The t_{bc} parameter is explained in section 2.2.1.

According to our model (see section 2.1), the theoretically best algorithms with regards to the number of network sends[26] are the Dissemination Barrier and the Pairwise Exchange Barrier. Both nearly reach a complexity of $L \cdot log_2 P$. The optimal complexity for a barrier algorithm in our model is deducted in the next section.

2.2.4 Proof of Optimality

The optimality of algorithms scaling with $O(log_2 P)$ is proven by induction. All preconditions are already defined in our model (see section 2.1). The algorithms use several (limited) rounds to achieve their targets to synchronize all nodes. Each node can exactly perform one send and one receive per round. To find a lower bound to this problem, a discovery algorithm is modeled. The information, that one node reached the barrier has to be transported to all other nodes. Each discovered node has the information that the root (starting) node entered its barrier function. So this problem gives a lower border to the problem that all nodes discover that one node reached the barrier function. This is also a lower bound to the general barrier problem (all nodes know that all other nodes reached their barrier function). The discovery process is shown in figure 2.9 and creates a binomial tree. This leads to Lemma 2.1.

Lemma 2.1: A maximum of 2^k nodes are discovered in round k.

An induction over k is used to proof Lemma 2.1. n_k is the number of discovered nodes in round k. Each node can send one notification to another node and receive one notification per round.

Claim: $n_k = 2^k$
Proof:
$k = 0$: $n_0 = 2^0 = 1 \rightarrow$ correct!
$k \rightarrow k+1$: $n_{k+1} = 2^{k+1}$
$n_{k+1} = 2 \cdot 2^k \rightarrow$ correct! (see model)

$\rightarrow$ Lemma 2.1 is correct and $log_2 P$ is a lower bound to the barrier problem.

[25] in our model, the CPU is expected to be much faster than the network and the performed calculations in the algorithms are slight

[26] which is expected to dominate the whole operation in time

2.2.5 Evaluating the LogP Predictions for TCP/IP

All barrier algorithms have been subdivided into four groups categorized by their runtime complexity inside the LogP model. This section shows benchmark results for each of the application classes. The four groups are:

1. $O(P) \Rightarrow$ Central Counter [FG91, GVW89]
2. $O(n \cdot log_n P) \Rightarrow$ Combining Tree [YTL87], f-way Tournament [GV94] and MCS [MCS91a]
3. $O(log_2 P)$ with broadcast $\Rightarrow$ Tournament [HFM88] and BST [TK97]
4. $O(log_2 P)$ without broadcast $\Rightarrow$ Butterfly [Bro86], Pairwise Exchange [GTNP02] and Dissemination [HFM88]

All algorithms have been implemented in a new COLL component within the Open MPI framework as described in section 1.4.1. The selection of different algorithms and the passing of parameters (mainly the group size n for the Combining Tree Barrier) was realized by utilizing the `mca_parameter` functions, which can be used to parametrize the modules during runtime.

The resulting code was executed on our local cluster (CLiC). It consists of 528 Pentium III 800 MHz nodes interconnected with an Extreme Black Diamond 6x96-Port Fast Ethernet switch. This switch satisfies nearly all the requirements stated in section 2.1.4.5.

The measurements were taken using the PMB2.2.1 [Pal00]. All results and the corresponding evaluation in terms of the LogP model are shown in the following sections.

2.2.5.1 Central Counter

The central counter is already implemented in the Open MPI framework because the run-time for small sets of processors is extremely low, even if the scaling with processor count is suboptimal. Thus, the Open MPI framework defines a threshold processor number (as MCA parameter) to change the used algorithm from the central counter to another logarithmic implementation. This was disabled during the tests to ensure that only the central counter is used. The results and the LogP prediction are shown in figure 2.14. Phase 1 is finished after

$$rt_{phase1} = o_s + L + (P-2)f_r + o_r$$

whereby $f_r = max\{o_r, g\}$ (as stated in section 2.1.4.5). Phase 2 starts at $T = rt_{Phase1}$ and the runtime until the last node is notified can be predicted with

$$rt_{phase2} = o_s + (P-2)f_s + L + o_r$$

whereby Node 0 finished the barrier after:

$$\begin{aligned} rt_{node0} &= rt_{phase1} + o_s + (P-2)f_s \\ &= 2o_s + o_r + L + (P-2)f_r + (P-2)f_s \end{aligned}$$

Node $x(\{x > 0; x < P-1\})$ finishes after:

$$\begin{aligned} rt_{nodex} &= rt_{phase1} + o_s + L + (x-1)f_s + o_r \\ &= 2(o_s + L + o_r) + (P-2)f_r + (x-1)f_s \end{aligned}$$

The last Node $(P-1)$ and the whole barrier finishes after:

$$rt = 2(o_s + L + o_r) + (P-2)f_r + (P-2)f_s$$

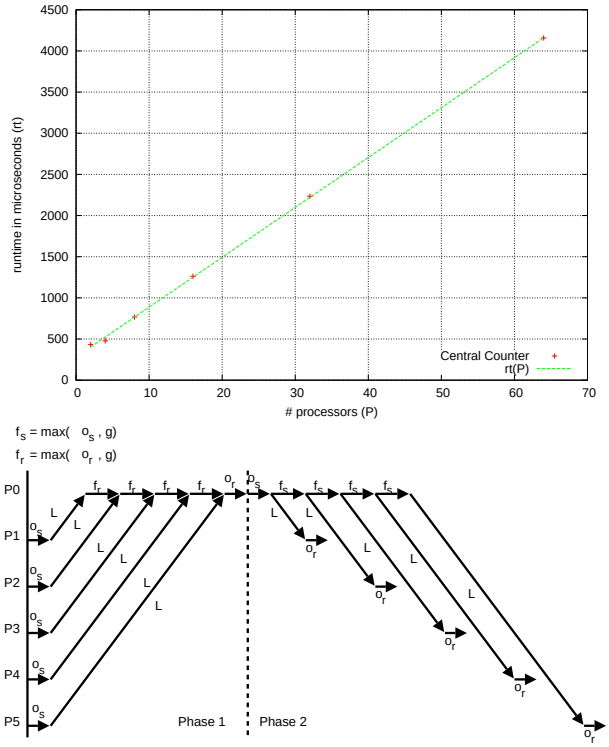

Figure 2.14: Central Counter

2.2.5.1.1 Influence of the Benchmark Loop

The Pallas Benchmark uses a loop (default repetitions: $b = 0; b < 1000; b++$), to measure the time spent inside the barrier operation. This loop could influence the results, because some nodes may enter the next barrier $(b+1)$ before all nodes finished the barrier (b). This occurs in the central counter, because P1 finishes the barrier after:

$$rt \quad = \quad 2(o_s + L + o_r) + (P-2)f_r$$

and sends its packet for barrier $b+1$ to P0, but P0 is still sending packets within barrier b. The first packet arrives and is enqueued by the MPI library because no matching receive was posted yet. P0 posts the first receive after it finishes barrier b. The operating system processed the message already and MPI stored it in a buffer, so f_r has been paid already (by delaying barrier b) for the first messages, when P0 enters barrier $b+1$. This adds a constant overhead to barrier b in each round (processing o_r for messages of barrier $b+1$), but does not change the asymptotic behavior.

rt can be simplified under the assumptions taken in 2.1.4.5 and a large processor count P:

$$\begin{aligned} rt &= 2(o_s + L + o_r) + (P-2)f_r + (P-2)f_s \\ o &= o_r = o_s \\ f_r &= f_s = o \ (o \gg g) \\ rt &\approx 2(2o + L) + 2o(P-2) \\ &\approx 2(L + Po) \end{aligned}$$

Thus, the graph for the runtime (rt) should depict as a linear function.

2.2.5.2 Combining Tree

The Combining Tree algorithm, described in section 2.2.1.2, is used to represent an $O(n \cdot log_n P)$ algorithm. The runtime for the first phase in the LogP model can be predicted as shown in figure 2.15 with:

$$rt_{phase1} = (o_s + L + f_r(n-2) + o_r) \cdot \lceil log_n P \rceil$$

as an upper bound. $\forall P \neq n^x$ $(x \in \mathbb{N})$ the running time for phase 1 is slightly smaller.

The second part, denoted as $t_{bc}(P-1)$, uses the binomial tree algorithm in Open MPI and is analyzed in figure 2.15.

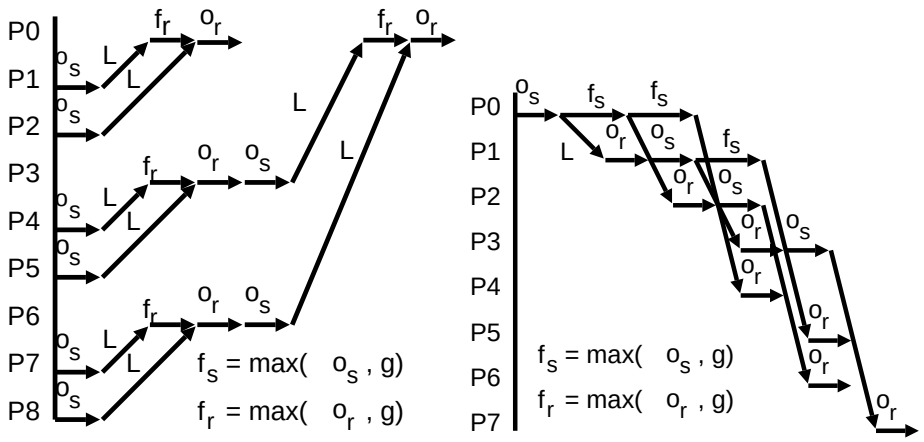

Figure 2.15: LogP model for Combining Tree and Binomial Broadcast ($n = 3$)

The runtime can be predicted as:

$$t_{bc} = o_s + \lceil log_2 P \rceil \cdot max\{f_s, o_s + L + o_r\} + L + o_r$$

Thus, the whole runtime rt can be seen as:

$$rt = (o_s + L + f_r(n-2) + o_r) \cdot \lceil log_n P \rceil + o_s + \lceil log_2 P \rceil \cdot max\{f_s, o_s + L + o_r\} + L + o_r$$

rt can be simplified under the assumptions taken in 2.1.4.5 and a large processor count P:

$$\begin{aligned} rt &\approx (L + no) \cdot \lceil log_n P \rceil + \lceil log_2 P \rceil \cdot (2o + L) \\ &\approx log_2 P \cdot (2o + L) \end{aligned}$$

There are different possibilities to assess the n (group size) parameter. To evaluate the influence on runtime, measurements were taken in the range 2..5. The benchmark results and the fitted functions are shown in figure 2.16. Another interesting result (already mentioned in [YTL87]) is that $n = 4$ seems to be the best choice for n. This is confirmed by our model and the fitted parameters as a global minimum, thus a group size of 4 is chosen for all future comparisons.

2.2.5.3 Tournament Barrier

The Tournament Barrier which was described in section 2.2.1.3 is chosen to represent the class of $O(log_2 P)$ algorithms, which perform the last step of notifying all other nodes, typically by broadcasting to them.

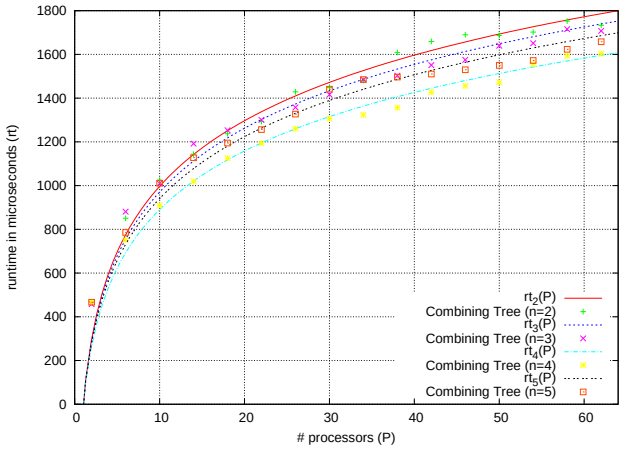

Figure 2.16: Measured rt Values

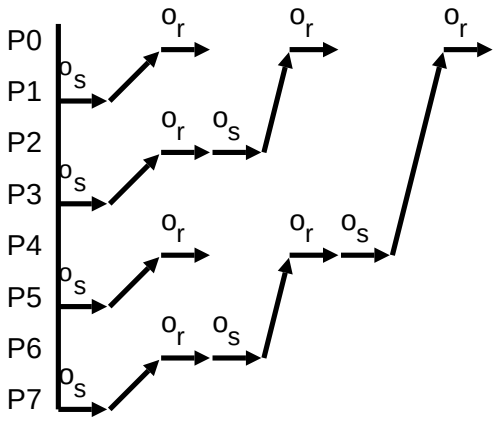

Figure 2.17: LogP for the Tournament Barrier

The LogP prediction depicted in figure 2.17 can be described as ($\forall P = 2^x$ $(x \in \mathbb{N})$, for all other P, the running time is slightly lower):

$$rt \quad = \quad max\{f_r, o_s + L + o_r\} \cdot \lceil log_2 P \rceil + t_{bc}$$

The Binomial Tree is used again for broadcasting at the end:

$$t_{bc} \quad = \quad o_s + \lceil log_2 P \rceil \cdot max\{f_s, o_s + L + o_r\} + L + o_r$$

Assuming the usual simplifications:

$$\begin{aligned} rt &= o_s + L + o_r + 2 \cdot max\{f_r, o_s + L + o_r\} \cdot \lceil log_2 P \rceil \\ &\approx 2o + L + 2(2o + L) \cdot log_2 P \\ &\approx (4o + 2L) \cdot log_2 P \end{aligned}$$

The measured values and the accordingly matched functions are shown in 2.18.

2.2.5.4 Dissemination Barrier

The Dissemination Barrier serves as an example for barrier algorithms of the complexity class $O(log_2 P)$ which do not have to perform a broadcast phase at the end, was previously proven to give an optimal

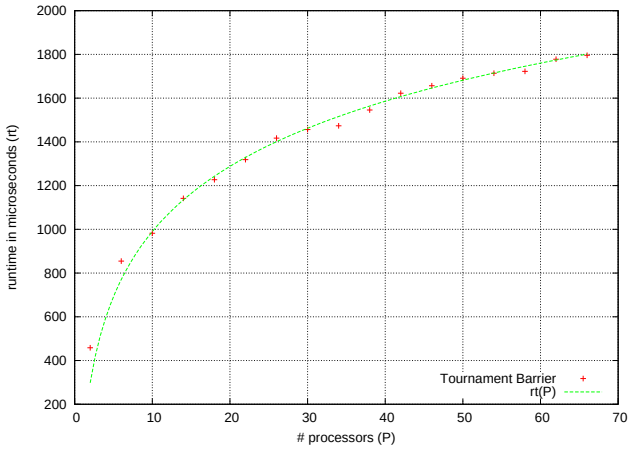

Figure 2.18: Tournament Barrier

solution to the barrier problem [HMMR04]. So we expect this barrier to achieve the best results. The algorithm was described in section 2.2.2.3. The LogP model is shown in figure 2.19 and the runtime can

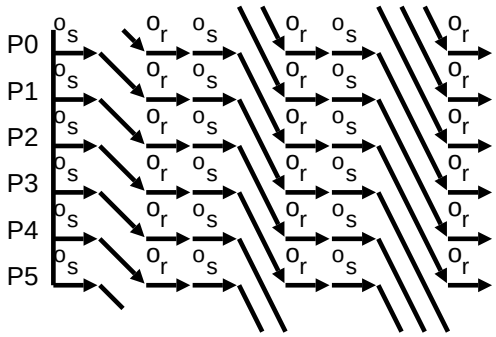

Figure 2.19: LogP for the Dissemination Barrier

be predicted with

$$rt \quad = \quad max\{f_r, f_s, o_s + L + o_r\} \cdot \lceil log_2 P \rceil$$

With the usual simplifications, the runtime behaves asymptotically as follows

$$rt \quad = \quad (2o + L) \cdot \lceil log_2 P \rceil$$

No broadcast step is needed to notify all participating nodes. The benchmark results and a fitted function of the upper bound are shown in figure 2.20. The runtime increases in steps, where each step starts at a power of two processor count.

2.2.5.5 Comparison of the Different Algorithms

In sections 2.2.1.1 to 2.2.5.4 it was shown that the LogP model is very accurate to predict the asymptotic behavior of barrier algorithms for systems which comply with the assumptions taken by the model[27]. The Dissemination algorithm seems to be the best solution for the barrier problem on LogP compliant systems. All measured data up to 66 nodes are shown in figure 2.21. Some results of the benchmarks

[27] the asymptotic standard error has been less than 5%

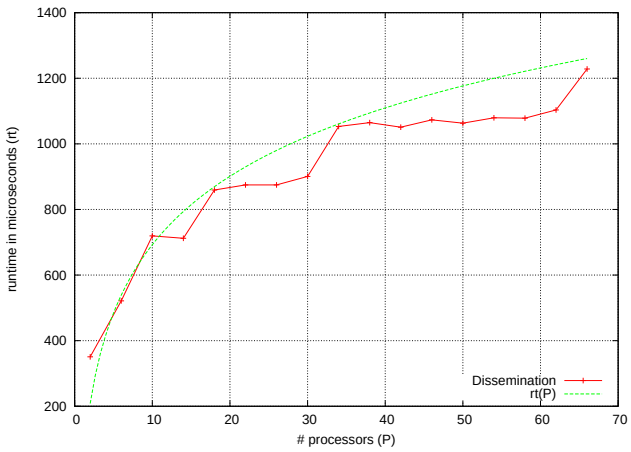

Figure 2.20: Dissemination Barrier

Table 2.2: Results for big Numbers of Processors

Algorithm	**128 nodes**	**256 nodes**
Central Counter	$4594.50\mu s$	$4909.67\mu s$
Combining Tree	$4009.79\mu s$	$4343.63\mu s$
Tournament	$3642.54\mu s$	$4378.77\mu s$
Dissemination	$1904.57\mu s$	$1977.12\mu s$
Open MPI	$3559.88\mu s$	$4226.88\mu s$

conducted on 128 and 256 nodes are summarized in table 2.2. The results for Open MPI were measured using the standard implementation switching automatically between central counter and Binomial Tree.

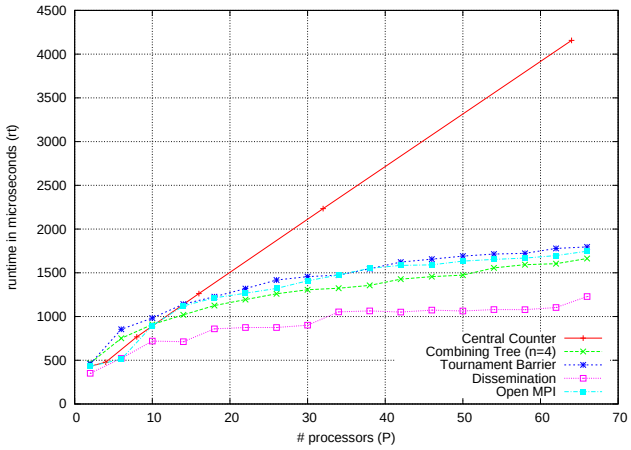

Figure 2.21: Comparison of all Barrier Algorithms

2.2.6 Two new Algorithms for Barrier Synchronization

The following two new algorithms are proposed to leverage the implicit hardware parallelism of InfiniBand and to benefit from the availability of multiple network interfaces to perform the barrier. A barrier operation can gain advantage from multiple interfaces as defined in the InfiniBand™ standard[28]. The n-way Dissemination and the n-wise exchange algorithm are described, analyzed and compared in the following sections.

[28] the InfiniBand™ standard defines two interfaces per HCA

Table 2.3: Peer Hosts for the 2-way Dissemination

Node	**Round**	$speer_1$	$speer_2$	$rpeer_1$	$rpeer_2$
0	0	1	2	8	7
1	0	2	3	0	8
⋮	0	⋮	⋮	⋮	⋮
8	0	0	1	7	6
0	1	3	6	6	3
1	1	4	7	7	4
⋮	1	⋮	⋮	⋮	⋮
8	1	2	5	5	2

2.2.6.1 The n-way Dissemination Algorithm

The n-way Dissemination Barrier is related to the Dissemination Barrier, proposed by Hengsen et al. in 1988 [HFM88]. It enhances the Dissemination Barrier to be more flexible in different environments. An additional parameter n defines the number of communication partners in each round. Thus, the original algorithm typifies the 1-way Dissemination Barrier ($n = 1$).

In each round every node p sends n packets to notify n other nodes that it reached its barrier function and is waiting for the notification of n other nodes. At the beginning of a new round r, node p calculates the peer nodes for all $\{i \in \mathbb{N}; 0 < i < n\}$ as follows:

$$speer_i = (p + i \cdot (n+1)^r) \ mod \ P \tag{2.1}$$

whereby P is the number of nodes participating in the barrier. The peers to wait for are also determined each round:

$$rpeer_i = (p - i \cdot (n+1)^r) \ mod \ P \tag{2.2}$$

An example for $n = 2$ and $P = 9$ is given in figure 2.22. The communication pattern is shown in table 2.3 for clearness.

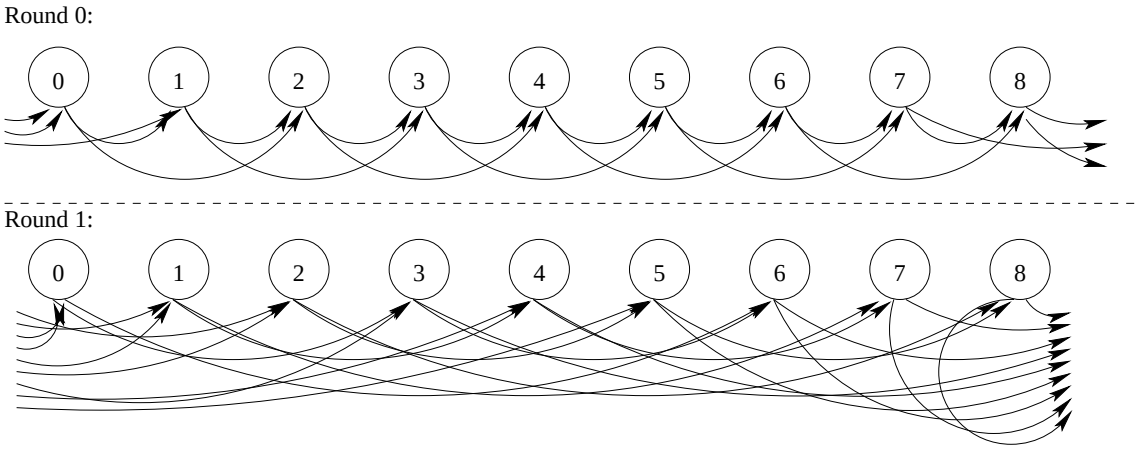

Figure 2.22: Example of the 2-way Dissemination Barrier

A possible pseudo-code for an RDMA-based or shared memory implementation is given in listing 2.8.

The communication behavior is analyzed with the LogP model and shown in figure 2.23. This figure assumes that $g > o_s + L + o_r$ and the send or receive operations have to wait for the network. $\forall g < o_s + L + o_r$, the gaps in the figure would vanish. Accordingly, the overall running time of this algorithm

```
// parameters (given by environment)
set n = 3 // parameter
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
reserve array with p entries as shared
for i in 0..p-1 do
   set array[i] = 0
forend

// barrier - done for every barrier
set round = -1
set x = x + 1

// repeat log_n(p) times
repeat
   set round = round + 1

   for i in 1..n do
      set sendpeer = (rank + i*(n+1)^round) mod p
      set recvpeer = (rank - i*(n+1)^round) mod p
      set array[rank] in node sendpeer to x
      wait until array[recvpeer] >= x
   forend
until round = ceil(log(p)/log(n))
```

Listing 2.8: Pseudocode for the n-way Dissemination Barrier

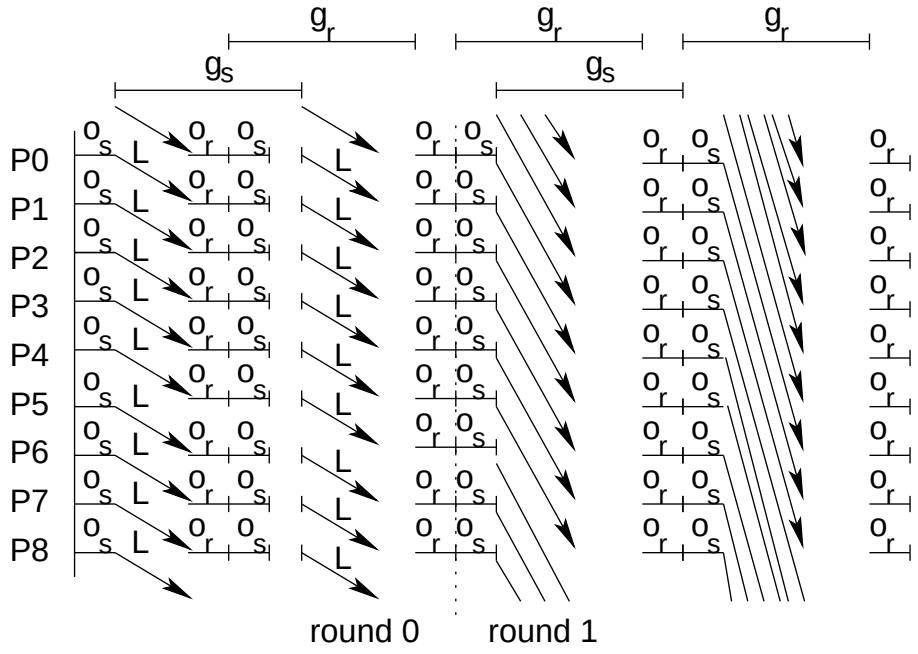

Figure 2.23: LogP Analysis of the 2-way Dissemination Barrier

with respect to the LogP model can be estimated with[29]:

$$rt_n(P) \quad = \quad n \cdot max\{g, o_s + L + o_r\} \cdot \lceil log_{n+1} P \rceil$$

and the asymptotic behavior can be assessed as

$$rt_n(P) \quad = \quad O(n \cdot \lceil log_{n+1} P \rceil)$$

Assuming, that m network paths[30] exist and are used to simultaneously send a message to each of the peer nodes p, the parameters g and L can be divided by m ($m \leq n$) and the runtime (by neglecting the scheduling overhead) changes to:

$$rt_n^m(P) \quad = \quad n \cdot max\left\{\frac{g}{m}, o_s + \frac{L}{m} + o_r\right\} \cdot \lceil log_{n+1} P \rceil$$

The influence of the network specific constants g and L declines with increasing m while the host dependent send and receive overhead $o_{\{r,s\}}$ remains constant. For multiple networks i ($i \in \mathbb{N}; 0 < i < m$), offering different LogP parameters g_i and L_i, the biggest factors $g = max\{g_i\}$ and $L = max\{L_i\}$ have to be used for modeling (e.g. using Fast Ethernet and InfiniBand™ , the values for Fast Ethernet have to be used).

This equation can be simplified for a high network parallelism (big m) in the following way

$$rt_m(P) \quad = \quad n \cdot (o_s + o_r) \cdot \lceil log_{n+1} P \rceil$$

and the aforementioned asymptotical behavior of $O(n \cdot \lceil log_{n+1} P \rceil)$ remains valid also for massively parallel networks.

2.2.6.1.1 Choosing the Parameter n

The influence of the parameter n can be assessed by the already mentioned equation of rt for a constant P and variable n. If $n < m$, only n links are used and $m - n$ links remain idle.

$$rt_m^P(n) \quad = \quad n \cdot max\left\{\frac{g}{min\{m,n\}}, o_s + \frac{L}{min\{m,n\}} + o_r\right\} \cdot \lceil log_{n+1} P \rceil$$

To exploit the whole network parallelism, the n should be chosen equal or bigger than m. The runtime $\forall n \geq m$ is given by the strictly monotone increasing function

$$rt_m^P(n) \quad = \quad max\left\{g \cdot \left\lceil \frac{n}{m} \right\rceil, n \cdot o_s + L \cdot \left\lceil \frac{n}{m} \right\rceil + n \cdot o_r\right\} \cdot \lceil log_{n+1} P \rceil$$

which has its global minimum for $n = m$. Therefore the ideal group size n for a system offering m independent LogP compliant links is m.

2.2.6.2 The n-wise Exchange Algorithm

The n-wise Exchange Barrier is an extended version of the pairwise exchange barrier with recursive doubling [GTNP02]. It enhances the algorithm regarding to its flexibility by adding a new parameter n. n constitutes the number of communication partners in each round of the algorithm. The original pairwise exchange algorithm used $n = 1$ to achieve an ideal solution inside the LogP model. If any network parallelism is assumed[31], this algorithm can be enhanced by adjusting n to the given environment.

The algorithm for all nodes $p \in P$ splits up into three phases. Each node calculates its position in the barrier, which results in two groups [GTNP02], one acting in the main game, and one waiting for notification. The main group consists of $P_m = max\{(n+1)^i\}$ ($i \in \mathbb{N}; (n+1)^i \leq P$) nodes and the $P_r = P - P_m < \frac{P}{n+1}$ remaining nodes form the waiting group.

The whole process for $n = 2$ is depicted in figure 2.24. Every node $p_j \in P_r$ contacts an appropriate node

[29] note: the occurrence of a gap before sending/receiving the first packet is assumed to simplify the equations
[30] e.g. devices, physical lanes
[31] e.g. multiple network interfaces

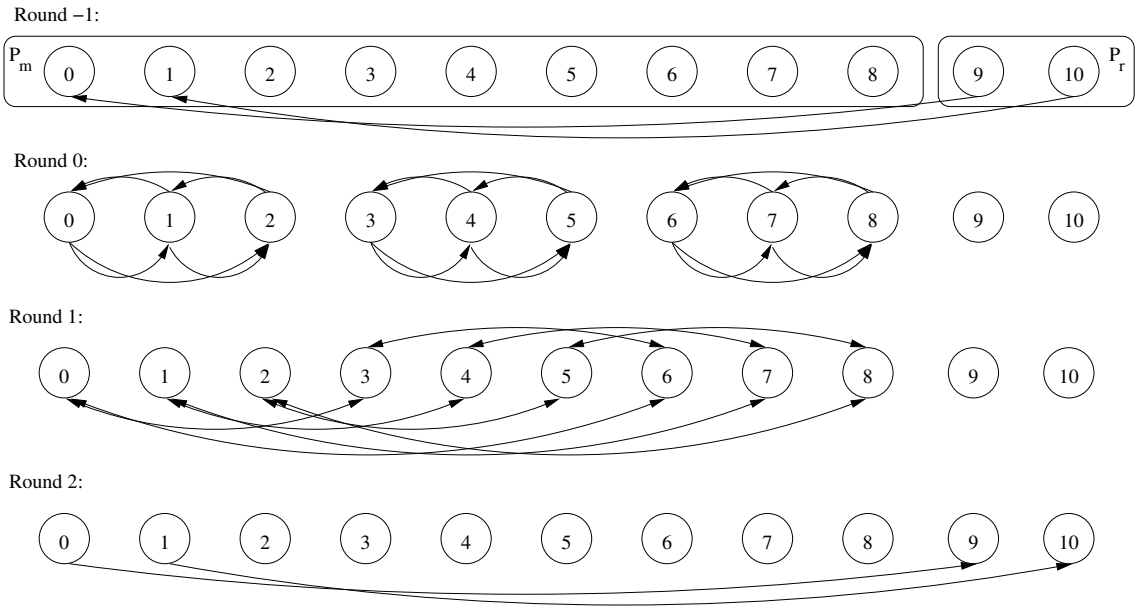

Figure 2.24: Example of a 2-wise Exchange barrier

$p_i \in P_m$ $(0 \leq i < j < P)$ to notify it that the barrier call is reached (round -1). After a node in the main group was notified by its peer, the node starts with the main algorithm (round 0 in figure 2.24). This main part is subdivided into $log_{n+1}P_m$ rounds. Assuming round number k, $(n+1)^k$ nodes form a group in each round k and notify the other group members (round 0 and 1 in the figure). After the main phase is finished, the nodes P_r are notified and leave the barrier call (round 2 in the figure). A pseudocode to realize this functionality is given in listing 2.9.

The running time behavior of the n-wise Exchange algorithm for the worst case, if $P_m \neq P$ is shown in figure 2.25, and can be predicted with the following equation[32]:

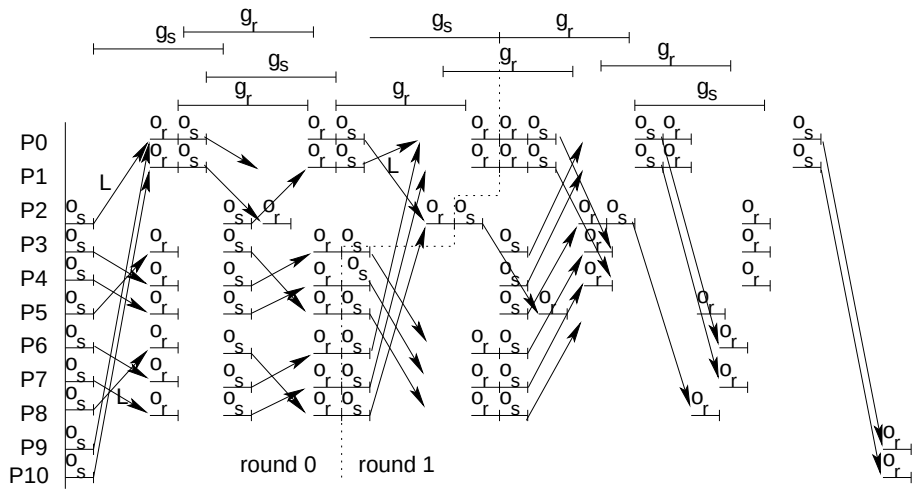

Figure 2.25: LogP Analysis of the 2-wise Exchange Barrier

$$rt_n(P) \quad = \quad (n+2) \cdot max\{g, o_s + L + o_r\} \cdot \lfloor log_{n+1}P \rfloor$$

This shows directly that the n-wise Exchange Barrier performs slightly worse than the n-way Dissemination because:

$$n \cdot \lceil log_n P \rceil \quad \leq \quad (n+1)\lfloor \cdot log_n P \rfloor < (n+2) \cdot \lfloor log_n P \rfloor$$

[32]note: the occurrence of a gap before sending/receiving the first packet is assumed to simplify the equations

```
// parameters (given by environment)
set n = 3 // parameter
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
reserve array with p entries as shared
for i in 0..p-1 do
   set array[i] = 0
forend

// barrier - done for every barrier
set round = -1
set x = x + 1

// border -> biggest ((n+1)^x)-1<=p (for all x in N)
set border = 1
set nn = n+1
repeat
   set border = border + 1
   set nn = nn * (n+1)
until nn > p
border = (n+1)^(border - 1)

if rank >= border then
   // I am in group 2
   set array[rank] in node rank-border to 1
else
   // I am in group 1
   if rank + border < p then
      // I have a partner in group 2
      wait until array[rank+border] = 1
   ifend

   repeat
      set round = round + 1
      // actual group size
      set grpsize = (n+1)^(round+1)
      // my group number
      set grpnum = rank div grpsize
      // the maximum rank in my group
      set maxgrp = (grpnum + 1)*grpsize-1
      // the minimal rank in my group
      set mingrp = grpnum*grpsize
      for i in 1..n do
         offset = i*(n+1)^round
         set peer = rank + offset
         // violated group borders?
         if peer > maxgrp
            set peer = mingrp + (peer - maxgrp - 1)
         ifend
         set array[rank] in node peer to round
         wait until array[peer] >= round
      forend
   until round = floor(log(p)/log(n))
ifend
```

Listing 2.9: Pseudocode for the n-wise Exchange Barrier

The optimal assessment of parameter n is identical to the n-way Dissemination Barrier and can be found in section 2.2.6.1.1.

2.2.6.3 Proof of Optimality

The optimality in running time of the newly introduced algorithms for m available parallel interfaces and $n = m$ is proven by induction. Again, the same preconditions, described in section 2.1 are assumed. The algorithms are divided into a limited number of distinct rounds for modeling purposes. Each node can exactly issue m sends and receives in parallel in each round, o_s and o_r are neglected.

To find a lower bound to this problem, a discovery algorithm is modeled. The information that one node reached the barrier has to be transported to all other nodes. Each discovered node has the information that the root (starting) node entered its barrier function. So this problem gives a lower border to the problem that all nodes discover that one node reached the barrier function. This is also a lower bound to the general barrier problem (all nodes know that all other reached their barrier function). The discovery process for $m = 2$ is shown in figure 2.26 and creates a binomial tree with a fan-out of m. This leads to

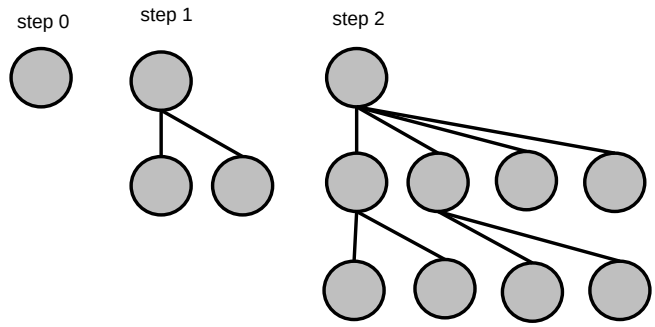

Figure 2.26: Binomial Discovery Tree for a fan-out of 2

Lemma 2.2.

Lemma 2.2: A maximum of $(m+1)^k$ nodes is discovered in round k.

An induction over k is used to proof Lemma 2.2. p_k is the number of discovered nodes in round k. Each node can send m notifications to m other nodes and receive m notifications per round.

Claim: $p_k = (m+1)^k$
Proof:
$k = 0$: $p_0 = (m+1)^0 = 1 \rightarrow$ correct!
$k \rightarrow k+1$: $p_{k+1} = (m+1)^{k+1}$
$p_{k+1} = (m+1) \cdot (m+1)^k \rightarrow$ correct! (see model)

$\rightarrow$ Lemma 2.2 is correct and $log_{m+1}P$ is a lower bound to the barrier problem.

2.3 Proposal of a Model for InfiniBand™

As described in the analysis of the different models (see 2.1.4), the LogP model reflects the needs to model an InfiniBand™ architecture quite well. The main assumptions, that each node consists of a complete "Von Neumann" computer with its processor, cache, memory, and network interconnect and that the computing power is much higher than the network throughput are completely conformed by the

InfiniBand™ architecture[33]. Unfortunately no presently known addition to the LogP model seems to be helpful for modeling InfiniBand™ very accurate because non of the mentioned architectural details are incorporated, so the original LogP model has to be modified to fit our special needs.

The LogP model was designed for general purpose environments, therefore it avoids architectural details. Thus, it is mostly suitable for a high level abstractions and algorithmic design, for example above the MPI [For95] layer. This work has already been done [EM04]. But the MPI layer may hide architectural details and when optimizing collective operations for specialized networks, each detail in this network architecture which can be used to speed up the operation has to be mentioned in the model. Thus, the model described here is more detailed than the original. It is not meant as a general purpose programming model competing with the the LogP, however it can be utilized to design highly specialized algorithms to exploit the InfiniBand™ architecture for optimal performance. The following architectural details which all have been described in section 1.3 are considered to design an accurate model.

2.3.1 Message Passing Options

For modeling all architectural details, especially the different possibilities for sending and receiving data (described in section 1.3.4), the model has to represent each of the following communication options separately. All options in brackets are currently not supported by available HCAs.

- (Reliable Datagram)
- Reliable Connection
- (Atomic Operation)
- RDMA Read
- RDMA Write
- Unreliable Datagram (also with Multicast)
- Unreliable Connection
- (RAW Ethernet)
- (RAW IPv6)

List 2.7: Modeled Communication Techniques

2.3.2 The HCA Processor

The HCA[34] is used to process previously posted work requests and participates actively in the communication relieving the host processor (see section 1.3.1), which strictly lowers the o parameters. This implies an additional level of parallelism and introduces new possibilities for overlapping computation and communication. This is modeled as part of the latency (L) parameter in the standard LogP model, which can be very accurate in most circumstances. But if the HCA is slower than the host CPU (the CPU is able to post more work requests than the HCA can process), contention will occur at the HCA[35] and the latency will vary from packet to packet (according to the frequency of previously occured posts).

2.3.3 Hardware Parallelism

The InfiniBand™ standard (see section 1.3.2) proposes implicit hardware parallelism or pipelining to the vendors, therefore the easy idea of using a gap as time to wait between consecutive messages cannot be very accurate in this architecture. The HCA can send two messages nearly in parallel until a single message fills the whole bandwidth. Thus the linear model of LogP is not accurate enough. It is assumed

[33] at least with 4x links
[34] InfiniBand™ Host Channel Adapter
[35] the Queue Pairs in InfiniBand™ will fill up

that the gap is now part of the latency which now depends on the number of previously issued send operations (denoted as L(p)). To reflect this behavior correctly, the model has to pay attention to the following send-receive scenarios:

- 1:1 communications
- 1:n communication
- n:1 communication

List 2.8: Send/Receive Scenarios

The latter two can be implemented either by a consecutive post of single work requests or by a single post of a list of work requests. The performance implications have to be modeled as well. The approximation function should behave like a normal pipeline startup function with $t = a + \frac{b}{x}$. The parameters a and b have to be measured for each vendor specific InfiniBand™ solution.

The new model introduces the new parameter h expressing the time which the HCA spends to process a message (it can be subdivided into h_s and h_r for sender and receiver). The h parameters cannot be measured directly because all actions are performed in hardware without notifying the host CPU. Thus, the model hides the h and g parameter inside the $L(p)$ parameter which varies depending on the number of hosts addressed (p). This limits the model to be used only if one node does never send more than one packet to another node because the $L(p)$ does only depend on the number of addressed hosts and not on the number of sent or received messages. This is given by the barrier problem, thus this model has to be seen as barrier-specific. Further enhancements of the model to allow general use are part of future work.

The traditional LogP would be a linear function like: $L(p) = h_s(p) + L + (p-1) \cdot g + h_r(p)$. Due to parallelism and pipeline effects, this function is assumed to be non-linear. The $L(p)$ parameter is measurable and can be used to find the best algorithm for barrier implementations with InfiniBand™ . The new model, named LoP, is depicted in figure 2.27. It has to be mentioned that $L(p)$ and o_s/o_r overlay each other because they are processed on different CPUs. Thus the HCA starts immediately to process messages after the CPU posted the first one. It is assumed that $o << L(p)\ \forall p \in \mathbb{N}$. The only exception is the VAPI call to post a list of requests, where the HCA has to wait until all requests have been posted, because all are posted at once.

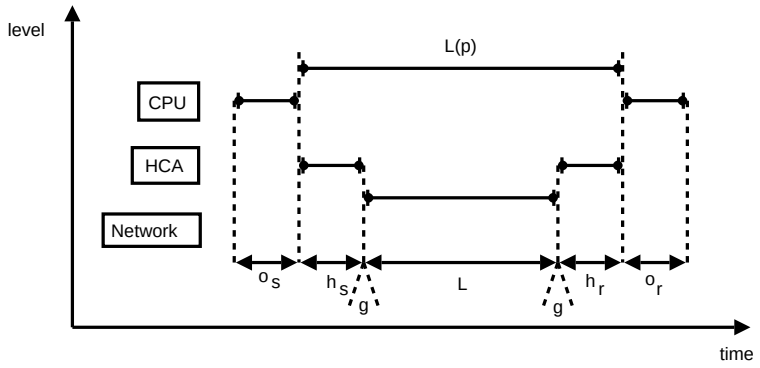

Figure 2.27: A new Model of InfiniBand™

2.3.4 Measuring the Parameters

All parameters mentioned in the previous section are hardware specific and have to be measured for each machine. The only possibility to measure the parameters is to evaluate the running time of different operations performed by the HCA. The following statements are based on a typical interaction with the HCA, shown in section 1.3.5.

The parameters can be measured as follows:

- $o_s(p)$ - time to complete the call `VAPI_post_sr()` or `EVAPI_post_sr_list()`
- $o_r(p)$ - time to complete the call `VAPI_post_rr()` or `EVAPI_post_rr_list()`
- $L(p) = \frac{RTT(p)}{2} - (o_s(p) + o_s(1))$ (sending to p processors and receiving from p processors - the HCA starts processing after the first request arrived - the exception for posting a list of send requests is found below, $o_r(p)$ does not matter because Receive Requests can be posted in advance)
- $L^{list}(p) = \frac{RTT(p)}{2} - (p \cdot o_s(p) + o_s(1))$ (sending to p processors and receiving from p processors for posting a list of send requests)

List 2.9: LoP Parameter Measuring

A possible benchmark is shown in figure 2.28 for the reliable connection type (see also 1.3.4.1), where

$$\begin{aligned} o_s(p) &= \frac{t_2 - t_1}{p} \\ o_r(p) &= \frac{t_1 - t_0}{p} \\ L_{mea}(p) &= \frac{t_3 - t_2}{2 \cdot p} \\ RTT(p) &= \frac{t_4 - t_1}{p} \end{aligned}$$

$o_r(p)$, $o_s(p)$, $L(p)$ and $RTT(p)$ values should be affected by sending multiple consecutive small messages (see section 2.3.3) and have to be measured for n:1 and 1:n communication. $RTT(p)$ is used to measure $L(p)$ because $L_{mea}(p)$ does not include eventual memory contention effects on the receiver side.

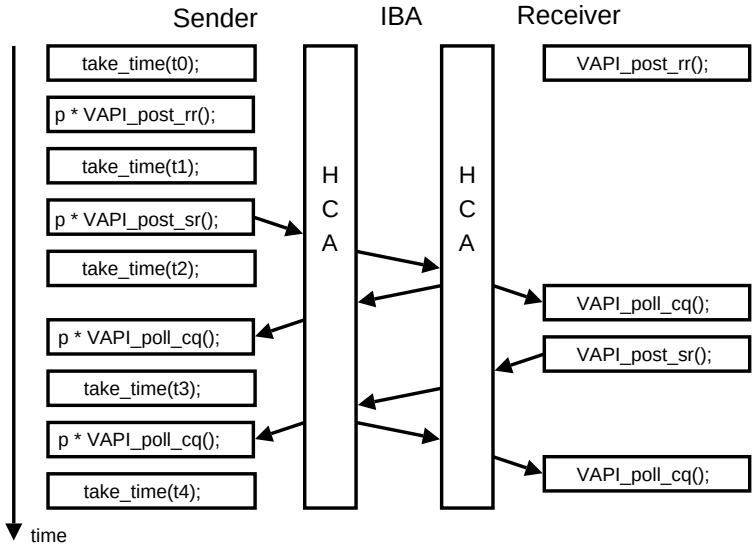

Figure 2.28: A Possible LoP Benchmark

2.3.5 A Benchmark of the LoP Model

The only way to verify the model and to measure the parameters defined in the LoP model is to benchmark the actual hardware. The used benchmark, written in C[36] with MPI function calls is presented in the following section. The benchmark implements the scheme described in section 2.3.4. It uses two different scenarios to measure all necessary parameters. Scenario 1 is used to measure all overheads for sending a

[36] 1290 SLOC, estimates to 3.1 person-months in the basic COCOMO model

single message, while scenario 2 measures ping-pong times for 1:n and n:1 communications. The source code is structured as follows:

- `mpi_iba_bench.c` - implements all administrative tasks (establishing connections, warming up the HCA, printing the results)
- `mpi_iba_bench.h` - is used to parametrize the whole benchmark before compiling (all parameters are compiled in for performance reasons)
- `hr_timer.c` and `hr_timer.h` - implements the high resolution timer functionality
- `create_qp.c` - functions to create new QPs
- `scenario_1.sub.c` - Scenario 1 (single message overhead)
- `scenario_2.sub.c` - Scenario 2 (ping-pong test)
- `query.c` - queries the HCA for most features (not part of the benchmark)
- `send.c` - simple send example for IBA (implements all functionality described in 1.3.5)

List 2.10: Structure of the LoP benchmark

The benchmark is described as pseudocode in the following listings. Listing 2.10 shows the preparation phase, where most parts can be found in `mpi_iba_bench.c` and `query.c`. Listing 2.11 describes scenario 1 (`scenario_1.sub.c`) which is chosen for measuring o_s and o_r, while scenario 2 (`scenario_2.sub.c`) is shown in listing 2.12.

Several implicit assumptions are taken to ensure the correctness of the pseudocode in all cases:

- `vapi_poll_cq_wait(cq = recv_cq)` returns immediately when RDMA is used
- `vapi_post_rr()` does not post a receive request if RDMA is used
- all measured times are summed up and divided by P, and REPETITIONS → all measured times are per single message sent to and received from P hosts (RTT for 1:P and P:1 communication)
- the number of hosts participating in the run are correctly returned by the MPI library

List 2.11: Assumptions to the benchmark pseudocode

The behavior of the benchmark is modified by changing the defines in `mpi_iba_bench.h`. The following defines can be changed in order to manipulate the benchmark:

- `MODE` - describes the operation used for transporting the messages
  - `MODE_SEND` - normal Send/Receive Operation
  - `MODE_RDMAW` - RDMA Write Operation
- `MEASURE` - selects the times to measure
  - `MEA_POST_SR_CPU_OVERHEAD` - CPU time consumed for posting a single SR (o_s) (scenario 1)
  - `MEA_POST_RR_CPU_OVERHEAD` - CPU time consumed for posting a single RR (o_r) (scenario 1)
  - `MEA_RTT_TIME` - RTT (Round Trip Time) for a single message in Ping-Pong (scenario 2)
- `SEND` - options for posting a SR (send a message)
  - `SEND_NORMAL` - post a normal SR (`VAPI_post_sr()`)
  - `SEND_LIST` - post a list of SRs (`EVAPI_post_sr_list()` - only in scenario 1)
  - `SEND_INLINE` - post an inline send request (`EVAPI_post_inline_sr()`)
- `RECV` - options for posting a RR
  - `RECV_NORMAL` - post a normal RR (`VAPI_post_rr()`)
  - `RECV_LIST` - post a list of RRs (`VAPI_post_rr_list()`)
- `REPETITIONS` - overall number of tests (to ensure accuracy)

```
initialize_mpi()
initialize_timers()

set p = number of participating processors
set rank = my local id

set addr_send = allocate_memory(MEMSIZE*p-1)
set addr_recv = allocate_memory(MEMSIZE*p-1)

vapi_create_pd()
set recv_cq = vapi_create_cq()
set send_cq = vapi_create_cq()

// create QPs to all peers
if rank == 0 then
   for i in 1..p-1 do
      set qp[i] = create_qp(src = 0, dst = i)
   forend
else
   set qp[i] = create_qp(src = rank, dst = 0)
ifend

vapi_register_mem(addr_recv)
vapi_register_mem(addr_send)

// post 1000 RRs for warmup
for i in 0..1000 do
   if rank == 0 then
      for j in 0..p-2 do
         vapi_post_rr(addr = addr_recv, peer = j+1)
      forend
   else
      vapi_post_rr(addr = addr_recv, peer = 0)
   ifend
forend

MPI_Barrier()

// send 1000 packets to peers
for i in 0..1000 do
   if rank == 0 then
      for j in 0..p-2 do
         vapi_post_sr(addr = addr_recv, peer = j+1)
         vapi_poll_cq_wait(cq = send_cq)
         vapi_poll_cq_wait(cq = recv_cq)
      forend
   else
      vapi_poll_cq_wait(cq = recv_cq)
      vapi_post_sr(addr = addr_recv, peer = 0)
      vapi_poll_cq_wait(cq = send_cq)
   ifend
forend
```

Listing 2.10: Pseudocode of the LoP benchmark - preparation

```
for k in 0..REPETITIONS do
   wait_us(time = 1000)
   for i in 1..MAXPOST+1 do
      set_mem(val = 0, addr = addr_send)
      set_mem(val = 0, addr = addr_recv)

      // rank 1 receives
      if rank == 1 then
         for j in 0..i do
            take_time(t0)
            vapi_post_rr(addr = addr_recv, peer = 0)
            take_time(t1)
         forend
      else
         take_time(t1)
      ifend

      MPI_Barrier()

      // rank 0 sends
      if rank == 0 then
         for j in 0..i do
            vapi_post_sr(addr = addr_send, peer = 1)
            take_time(t2)
            vapi_poll_cq_wait(cq = send_cq)
            take_time(t3)
         forend
      else
         for j in 0..i do
            vapi_poll_cq_wait(cq = recv_cq)
         forend
      ifend
   forend
forend
```

Listing 2.11: Pseudocode of the LoP benchmark - scenario 1

```
set state = 0
for k in 0..REPETITIONS
   wait_us(time = 1000)
   for i in 1..p do
      if rank == 0 then
         for j in 0..i do
            vapi_post_rr(addr = addr_recv[i], peer = j+1)
         forend
      ifend
      if rank > 0 and rank <= i then
         vapi_post_rr(addr = addr_recv[0], peer = 0)
      ifend

      MPI_Barrier()
      set state = state + 1
      take_time(t1)
      set_mem(state, addr_send, MEMSIZE)

      if rank == 0 then
         for j in 0..i do
            vapi_post_sr(addr = addr_send, peer = j+1)
         forend
         for j in 0..i do
            vapi_poll_cq_wait(cq = send_cq)
         forend
      ifend

      if rank > 0 and rank <= i then
         vapi_poll_cq_wait(cq = recv_cq)
         wait until addr_recv[0] = state
         vapi_post_sr(addr = addr_send, peer = 0)
         vapi_poll_cq_wait(cq = send_cq)
      ifend

      if rank == 0 then
         for j in 0..i do
            vapi_poll_cq_wait(cq = recv_cq)
            wait until addr_recv[i] = state
         forend
      ifend

      take_time(t4)
   forend
forend
```

Listing 2.12: Pseudocode of the LoP benchmark - scenario 2

2.3.6 Benchmark Results

All benchmarks are extremely implementation specific. The measured values highly depend on the given architecture and circumstances. All following benchmark results have been gaged on a 64 node InfiniBand™ cluster, interconnected with a 64 port switch (the hardware has been described in section 4.2.1). The general architecture to assess the parameters L and o of the LoP model for offloading based systems is modeled in the following section.

The benchmarks have been conducted for Send/Receive and RDMA Write without immediate operation. RDMA Read and RDMA Write with immediate have not been considered because the architectural design and several studies, such as [LWP04, LJW+04] show that these operations are generally slower than RDMA Write without immediate. Atomic Operations are not available on the used HCAs.

2.3.6.1 Modelling the Architecture

A general model of the Round Trip Time (RTT) and overhead times for offloading based networks will be described to generalize the results of this thesis (e.g. to Myrinet, Elan). This model will be parametrized to fit to the test cluster mentioned in section 2.3.6.

2.3.6.1.1 A Model of the RTT

The RTT model consists of three sections: The warmup section for the NIC (e.g. pipelining or cache effects), the maximum performance section (NIC CPU is fully saturated) and the network saturation section. This model assumes that 1:n and n:1 communications are equal in terms of costs. All λ values have to be seen as variables changing the shape of the curve and have to be fitted to give the best approximation to the measured values. The first section is typically represented by a pipeline startup function of the shape:

$$t_{pipeline} = \frac{\lambda_1}{\lambda_2 + p}$$

The second section is only defined by the maximum $CPU \rightarrow NIC \rightarrow NIC \rightarrow CPU$ throughput (packet processing rate), and is thus defined as constant:

$$t_{processing} = \lambda_3$$

The third section reflects the network saturation which typically behaves like an exponential function:

$$t_{saturation} = \lambda_4 \cdot (1 - e^{\lambda_5 \cdot (p - \lambda_6)})$$

λ_4 and λ_5 influence the signature of the function and λ_6 introduces a p-offset.

Altogether the RTT can be described with the following abstract model, which is depicted in figure 2.29:

$$\begin{aligned} t_{rtt}(\lambda_{1\ldots6}) &= t_{saturation} + t_{processing} + t_{pipeline} \\ &= \frac{\lambda_1}{\lambda_2 + p} + \lambda_3 + \lambda_4 \cdot (1 - e^{\lambda_5 \cdot (p - \lambda_6)}) \end{aligned}$$

2.3.6.1.2 A Model of the Overhead

The send and receive overheads are modeled as pipeline startup functions. This is due to several cache effects and pipelining effects at the host CPU. The HCA should not be involved into this process, because the data is written into memory mapped registers inside the HCA memory.

The function can be described as:

$$t_{ov}(\lambda_{1\ldots3}) = \lambda_1 + \frac{\lambda_2}{\lambda_3 + p}$$

and is depicted in figure 2.30.

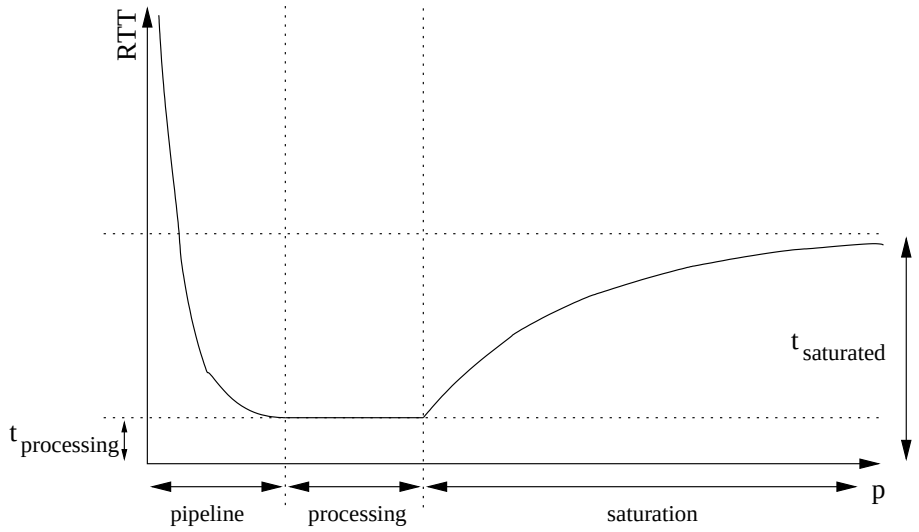

Figure 2.29: The RTT Model

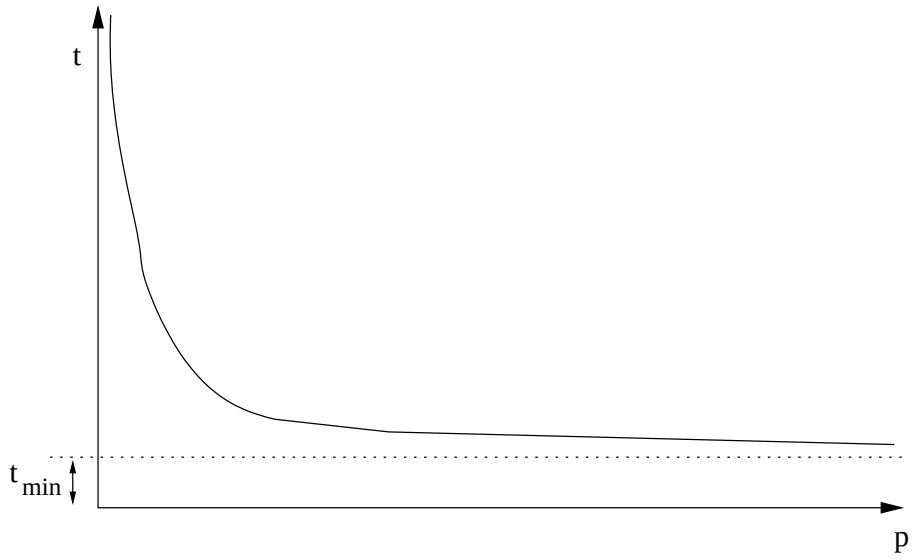

Figure 2.30: The Overhead Model

2.3.6.1.3 Parametrizing the Model

The least squares method is used to find an optimal parametrization for all $\lambda_{1...6}$. This method calculates the sum of the squared deviations of the measured values to the functional prediction for all available data-points and tries to minimize it. The following steps are performed for all predicted $\lambda_{1...6}$ ($y(m)$ represents the value of data-point m, M is the number of available data-points, $1 \leq m \leq M$).

$$\begin{aligned} d_m(\lambda_{1...6}) &= (y(m) - t_{rtt})^2 \\ &= \left(y(m) - \left(\frac{\lambda_1}{\lambda_2 + p} + \lambda_3 + \lambda_4 \cdot (1 - e^{\lambda_5 \cdot (p - \lambda_6)})\right)\right)^2 \\ d(\lambda_{1...6}) &= \sum_{m=1}^{M} d_m(\lambda_{1...6}) \end{aligned}$$

$d(\lambda_{1...6})$ represents the least squares deviation of the actual $t_{rtt}(\lambda_{1...6})$ from the measured values $y(m)$ $\forall 1 \leq m \leq M$. It is easy to see that d has to be minimized in a 6-dimensional space. To perform this task, the Nelder-Mead simplex search method, proposed in [LRWW98] was used to find a local minimum.

The search represents a k-dimensional input vector ($\lambda_{1...k}$) with $k+1$ vectors which form a simplex. For example, a two-dimensional space is described by a triangle, and a three dimensional space by a tetrahedron (a special pyramid). Each step creates a new point in or near the current simplex, calculates the values and compares the solution to the original y. This is repeated until a given tolerance or a maximum loop-count is reached.

The approximation scheme for t_{ov} is omitted because it can be easily derived from the scheme shown above.

Most fits are optically not extremely accurate, this is mainly due to the fact that the Nelder-Mead method converges against a local minimum which is highly dependent on the starting values of $\lambda_{1...6}$. All values have been optimized by hand to fit the graph as accurate as possible, but a single calculation takes up to three hours and this method is in fact very uncomfortable. Future steps should include finding a better approximation by leveraging the hints given at Optimization Software[37] and and calculating the λs with a parallelized version of the given algorithms.

2.3.6.2 Send/Receive Results

The minimal RTT results of Send/Receive InfiniBand™ operations can be seen in figure 2.31. The

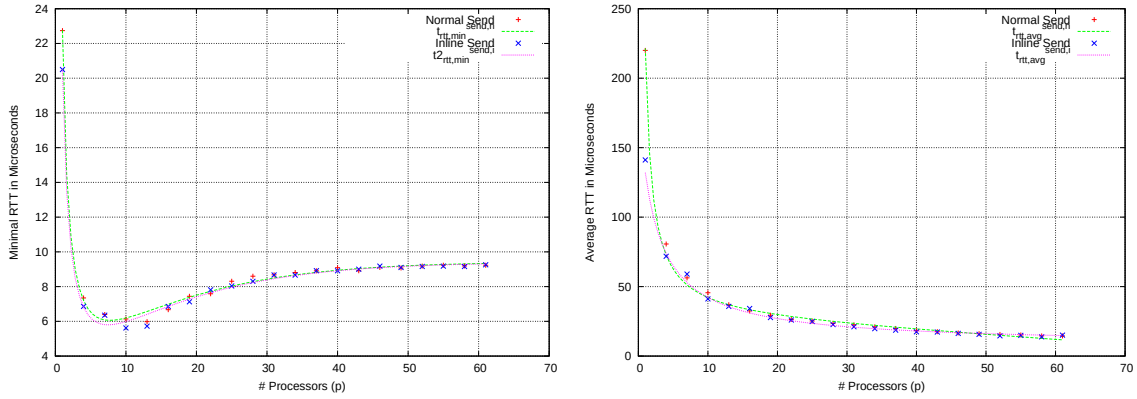

Figure 2.31: Minimal and Average Send/Receive RTT Times

depicted function describes t_{rtt} as described in section 2.3.6.1.1. The measured ($y(m)$) and fitted (parametrized t_{rtt}) functions are shown in figure 2.31 and mathematically described in the following:

$$
\begin{aligned}
t^{send,n}_{rtt,min}(p) &= 9.1637 + \frac{22.4558}{-0.0140+p} + 0.0174 \cdot \left(1 - e^{-0.0625\cdot(p-101.3065)}\right) \\
t^{send,i}_{rtt,min}(p) &= 9.0502 + \frac{23.3204}{0.1081+p} + 0.0895 \cdot \left(1 - e^{-0.0636\cdot(p-74.6471)}\right) \\
t^{send,n}_{rtt,avg}(p) &= 35.7292 + \frac{191.6019}{-0.0232+p} + 63.578 \cdot \left(1 - e^{0.0034\cdot(p+43.7788)}\right) \\
t^{send,i}_{rtt,avg}(p) &= 8.4888 + \frac{424.1248}{2.4364+p} - 0.4132 \cdot \left(1 - e^{-1.6407\cdot(p-1.1077)}\right)
\end{aligned}
$$

The difference between normal (marked with n) and inline (marked with i) send is modeled quite accurate. It is constantly about $1\mu s$ for small processor counts and vanishes when the network begins to saturate ($p \approx 30$).

[37] Optimization Software [http://plato.la.asu.edu/topics/problems/nlolsq.html]

The measured send (o_s) and receive (o_r) overheads are shown in figure 2.32. The measured values fit to

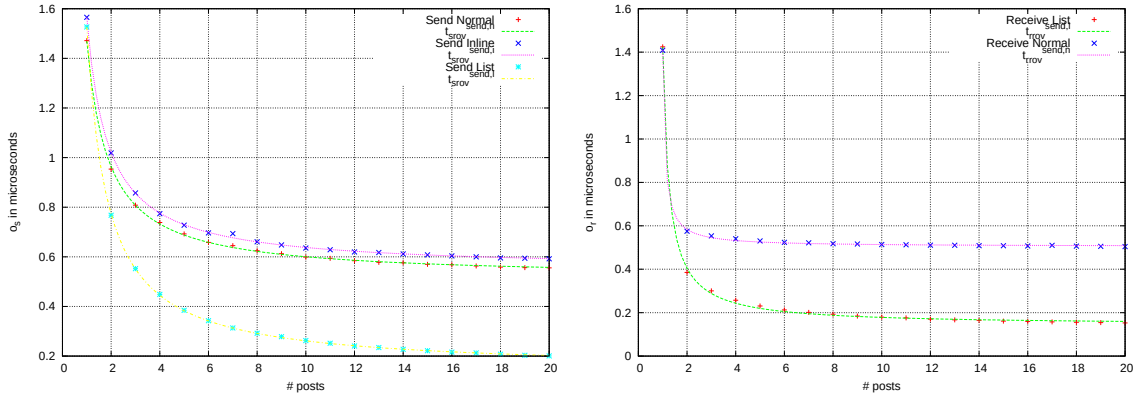

Figure 2.32: SR and RR Times

the model function as given by the equations:

$$
\begin{aligned}
t_{sr,ov}^{send,n}(p) &= 0.5150 + \frac{0.8443}{-0.1160 + p} \\
t_{sr,ov}^{send,i}(p) &= 0.5483 + \frac{0.8830}{-0.1316 + p} \\
t_{sr,ov}^{send,l}(p) &= 0.1443 + \frac{1.1528}{-0.1657 + p} \\
t_{rr,ov}^{send,n}(p) &= 0.5042 + \frac{0.0871}{-0.9037 + p} \\
t_{rr,ov}^{send,l}(p) &= 0.1427 + \frac{0.3284}{-0.7437 + p}
\end{aligned}
$$

The fastest method to post more than two send requests is generally to post a list (marked with l) of send requests. All other methods could be beneficial with special send operations (inline send). The overall results are calculated in section 2.3.7.

2.3.6.3 RDMA Write Results

The minimal and average RTT results of RDMA Write InfiniBand™ operations can be seen in figure 2.33. The shown function depicts t_{rtt} as described in section 2.3.6.1.1 for RDMA Write operation. The

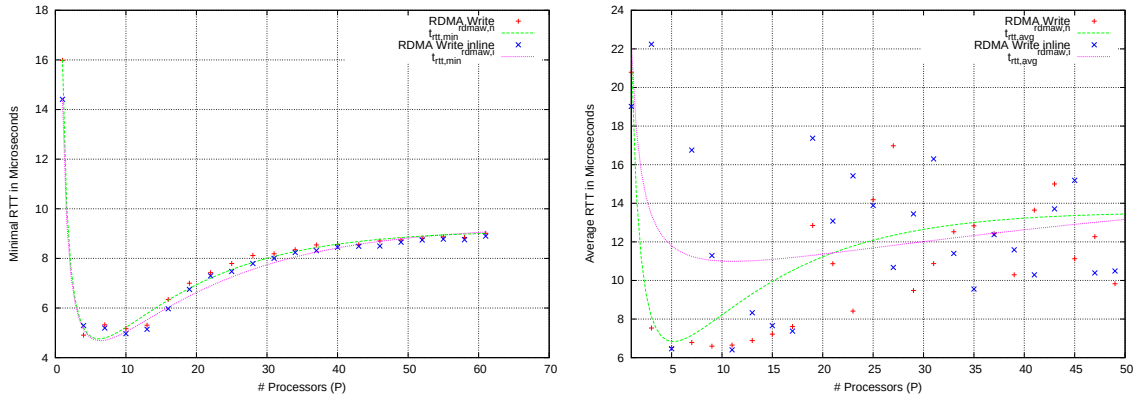

Figure 2.33: Minimal and Average RDMA Write RTT Times

average functions show a big deviation, and are only plotted and fitted up to 50 processors. The t_{rtt} values

raise quickly up to $700\mu s$ for bigger processor counts, which could lead to the conclusion that memory or bus contention occurs. The plotted deviation may be caused by memory contention and blocking/arbiting effects of single RDMA write operations and varies extremely between different measurements.

The inline send is again about $1\mu s$ faster than the normal send for small processor counts p and this difference vanishes during the network saturation ($p > 30$). The normal send seems to be much better and even more "stable" in the average case than the inline send. The functions for the average case are also quite accurate, even if the measured values oscillate a lot. This is guaranteed by the least squares method, punishing bigger deviations more than smaller ones.

The fitted functions for all described data-sets are given in the following:

$$\begin{aligned}
t_{rtt,min}^{rdmaw,n}(p) &= 4.4642 + \frac{16.7937}{0.0058+p} + 4.4751 \cdot \left(1 - e^{-0.0642\cdot(p-12.9209)}\right) \\
t_{rtt,min}^{rdmaw,i}(p) &= 3.0074 + \frac{14.9630}{0.0446+p} + 6.1891 \cdot \left(1 - e^{-0.0531\cdot(p-8.2665)}\right) \\
t_{rtt,avg}^{rdmaw,n}(p) &= 13.1499 + \frac{26.2100}{0.0596+p} + 0.0317 \cdot \left(1 - e^{-0.0841\cdot(p-75.1048)}\right) \\
t_{rtt,avg}^{rdmaw,i}(p) &= 11.4724 + \frac{14.0689}{0.0117+p} + 4.3843 \cdot \left(1 - e^{-0.0186\cdot(p-29.1184)}\right)
\end{aligned}$$

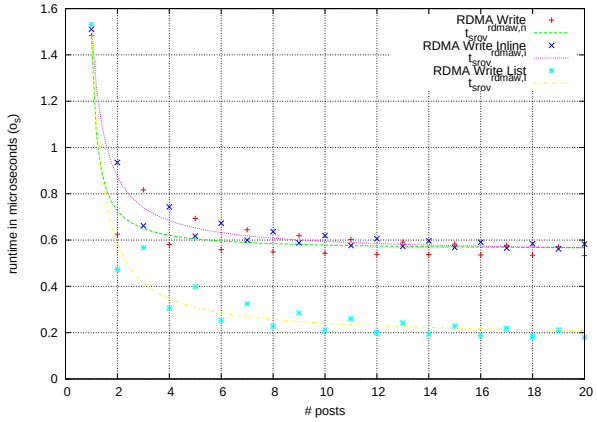

Figure 2.34: RDMA o_s overhead

Figure 2.34 shows the send overhead (o_s) for RDMA Write operations. Posting a list of send requests is again the fastest method of sending multiple packets, but to send the data inline could lower the latency in the best case. This means that the send overhead is the lowest for list send but the RTT is lowered by the use of inline send.

$$\begin{aligned}
t_{srov}^{rdmaw,n}(p) &= 0.5557 + \frac{0.2103}{-0.7728+p} \\
t_{srov}^{rdmaw,i}(p) &= 0.5417 + \frac{0.5003}{-0.4951+p} \\
t_{srov}^{rdmaw,l}(p) &= 0.1803 + \frac{0.5726}{-0.5746+p}
\end{aligned}$$

2.3.6.4 Comparison of Send/Receive and RDMA Write

Figure 2.35 shows a direct comparison of RDMA Write and Send/Receive communication. The exact LoP parametrization is derived in the next section.

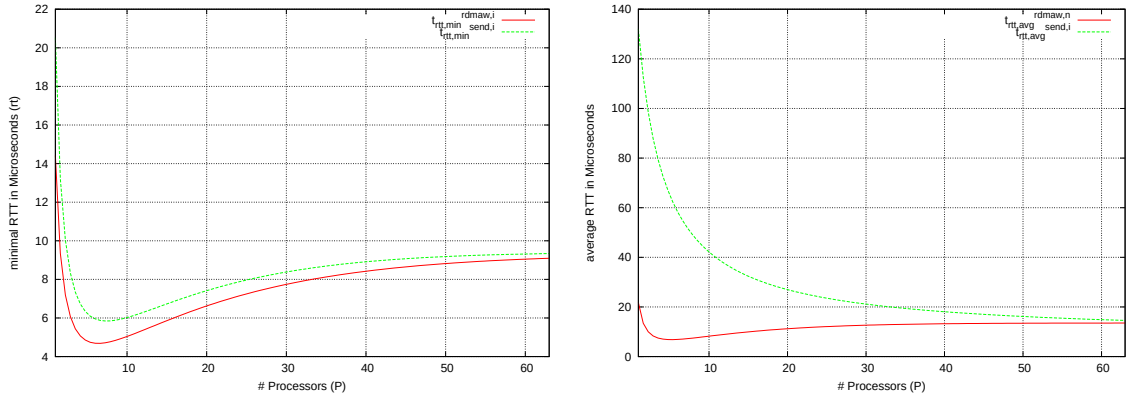

Figure 2.35: RDMA Write and Send/Receive comparison

2.3.7 Choosing the Optimal Solution to the Problem

The three possibilities for RDMA Write or Send/Receive semantics to pass a message to a remote system with InfiniBand^{TM} have been investigated. The RTT was measured for inline send and normal send. List send was not measured because it should behave exactly like normal send regarding to $L(p)$, only the time to post a single message (o_s) is reduced (but the operation on the HCA starts after posting all requests). The $L(p)$ should be affected by sending a message inline or not, thus it has to be modeled for these two different scenarios. However, the following equations are mainly given for completeness and show up to be extremely system dependent. Thus, it is not recommended to reproduce every single calculation, but the reader should be able to perform a similar analysis for another system. (all values have been rounded for convenience):

$$
\begin{aligned}
L_{min}^{send,n}(p) &= \frac{t_{rtt,min}^{send,n}(p)}{2} - \left(t_{sr,ov}^{send,n}(1)\right) - \left(t_{sr,ov}^{send,n}(p)\right) \\
&= 4.58 + \frac{11.23}{-0.01+p} + 0.01 \cdot \left(1 - e^{-0.06\cdot(p-101.31)}\right) - \left(0.52 + \frac{0.84}{-0.12+1}\right) - \left(0.52 + \frac{0.84}{-0.12+p}\right) \\
&= 2.59 + \frac{-1.34 + 10.39p}{0.13p + p^2} + 0.01 \cdot \left(1 - e^{-0.06\cdot(p-101.31)}\right) \\
L_{min}^{send,i}(p) &= \frac{t_{rtt,min}^{send,i}(p)}{2} - \left(t_{sr,ov}^{send,i}(1)\right) - \left(t_{sr,ov}^{send,i}(p)\right) \\
&= 4.53 + \frac{11.66}{0.11+p} + 0.04 \cdot \left(1 - e^{-0.06\cdot(p-74.65)}\right) - \left(0.55 + \frac{0.88}{-0.13+1}\right) - \left(0.55 + \frac{0.88}{-0.13+p}\right) \\
&= 2.42 + \frac{-1.61 + 10.78p}{-0.01 - 0.02p + p^2} + 0.04 \cdot \left(1 - e^{-0.06\cdot(p-74.65)}\right) \\
L_{avg}^{send,n}(p) &= \frac{t_{rtt,avg}^{send,n}(p)}{2} - \left(t_{sr,ov}^{send,n}(1)\right) - \left(t_{sr,ov}^{send,n}(p)\right) \\
&= 17.86 + \frac{95.8}{-0.02+p} + 31.79 \cdot \left(1 - e^{0.003\cdot(p+43.78)}\right) - \left(0.52 + \frac{0.84}{-0.12+1}\right) - \left(0.52 + \frac{0.84}{-0.12+p}\right) \\
&= 15.87 + \frac{-11.51 + 94.96p}{-0.14p + p^2} + 31.79 \cdot \left(1 - e^{0.003\cdot(p+43.78)}\right) \\
L_{avg}^{send,i}(p) &= \frac{t_{rtt,avg}^{send,i}(p)}{2} - \left(t_{sr,ov}^{send,i}(1)\right) - \left(t_{sr,ov}^{send,i}(p)\right) \\
&= 4.24 + \frac{212.06}{2.44+p} - 0.21 \cdot \left(1 - e^{-1.641\cdot(p-1.11)}\right) - \left(0.55 + \frac{0.88}{-0.13+1}\right) - \left(0.55 + \frac{0.88}{-0.13+p}\right)
\end{aligned}
$$

$$
\begin{aligned}
&= 2.13 + \frac{-29.72 + 211.18p}{-0.32 + 2.31p + p^2} - 0.21 \cdot \left(1 - e^{-1.641 \cdot (p-1.11)}\right) \\
L_{min}^{rdmaw,n}(p) &= \frac{t_{rtt,min}^{rdmaw,n}(p)}{2} - \left(t_{sr,ov}^{rdmaw,n}(1)\right) - \left(t_{sr,ov}^{rdmaw,n}(p)\right) \\
&= 2.23 + \frac{8.39}{0.01 + p} + 2.24 \cdot \left(1 - e^{-0.064 \cdot (p-12.92)}\right) - \left(0.56 + \frac{0.21}{-0.77 + 1}\right) - \left(0.56 + \frac{0.21}{-0.77 + p}\right) \\
&= 0.20 + \frac{-6.46 + 8.18p}{-0.01 - 0.76p + p^2} + 2.24 \cdot \left(1 - e^{-0.064 \cdot (p-12.92)}\right) \\
L_{min}^{rdmaw,i}(p) &= \frac{t_{rtt,min}^{rdmaw,i}(p)}{2} - \left(t_{sr,ov}^{rdmaw,i}(1)\right) - \left(t_{sr,ov}^{rdmaw,i}(p)\right) \\
&= 1.5 + \frac{7.48}{0.04 + p} + 3.09 \cdot \left(1 - e^{-0.053 \cdot (p-8.27)}\right) - \left(0.54 + \frac{0.50}{-0.49 + 1}\right) - \left(0.54 + \frac{0.50}{-0.49 + p}\right) \\
&= -0.56 + \frac{-3.69 + 6.98p}{-0.02 - 0.45p + p^2} + 3.09 \cdot \left(1 - e^{-0.053 \cdot (p-8.27)}\right) \\
L_{avg}^{rdmaw,n}(p) &= \frac{t_{rtt,avg}^{rdmaw,n}(p)}{2} - \left(t_{sr,ov}^{rdmaw,n}(1)\right) - \left(t_{sr,ov}^{rdmaw,n}(p)\right) \\
&= 6.57 + \frac{13.11}{0.06 + p} + 0.02 \cdot \left(1 - e^{-0.08 \cdot (p-75.10)}\right) - \left(0.56 + \frac{0.21}{-0.77 + 1}\right) - \left(0.56 + \frac{0.21}{-0.77 + p}\right) \\
&= 4.54 + \frac{-10.1 + 12.9p}{-0.05 - 0.71p + p^2} + 0.02 \cdot \left(1 - e^{-0.08 \cdot (p-75.10)}\right) \\
L_{avg}^{rdmaw,i}(p) &= \frac{t_{rtt,avg}^{rdmaw,i}(p)}{2} - \left(t_{sr,ov}^{rdmaw,i}(1)\right) - \left(t_{sr,ov}^{rdmaw,i}(p)\right) \\
&= 5.74 + \frac{7.03}{0.01 + p} + 2.19 \cdot \left(1 - e^{-0.02 \cdot (p-29.12)}\right) - \left(0.54 + \frac{0.50}{-0.49 + 1}\right) - \left(0.54 + \frac{0.50}{-0.49 + p}\right) \\
&= 3.68 + \frac{-3.45 + 6.53p}{-0.01 - 0.48p + p^2} + 2.19 \cdot \left(1 - e^{-0.02 \cdot (p-29.12)}\right)
\end{aligned}
$$

All functions for the different possibilities to send or receive 1 byte packets using the send-receive and RDMA semantics are shown in figure 2.36. The results show that RDMA write with immediate is the best solution for small messages.

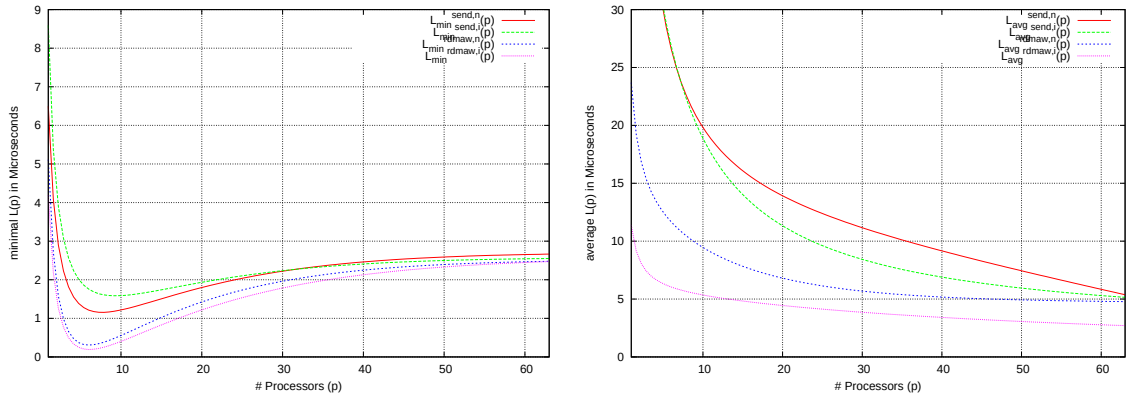

Figure 2.36: Average and minimal L(p) for RDMA and Send

Thus, the time to send one message to n hosts for each possible post send request / send type combination can be assessed with:

$$t_{1:n} = o(n) + n \cdot L(n)$$

... for posting a list of send requests:

$$t_{1:n}^{list} = n \cdot o(n) + n \cdot L(n)$$

the time to send 1 messages from n hosts with:

$$t_{n:1} = o(1) + n \cdot L(n)$$

and the time to send 1 messages from 1 host with:

$$t_{1:1} = o(1) + L(1)$$

It is assumed that $L(n) \gg o(n)\ \forall n \in \mathbb{N}$.

2.3.7.1 Barrier Algorithms inside the LoP Model

As already done for the LogP model, in section 2.2.5 and 2.2.6, a representative of each barrier-algorithm group will now be analyzed inside the LoP model. This process is more complicated because of the influence of the time (the "cooling down"). A new parameter for the number of performed barrier operations (cnt) has to be introduced to reflect this. The following predictions assume a balanced state of the algorithm (all nodes can send without waiting for the other nodes).

2.3.7.1.1 Central Counter

The running time of the central counter algorithm, shown in figure 2.37, can be predicted as:

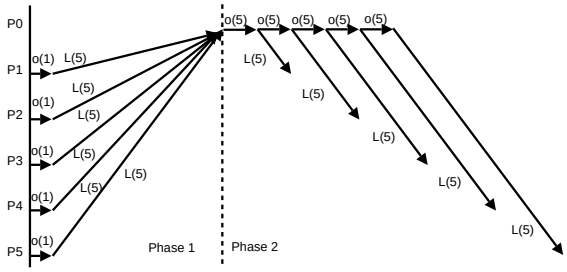

Figure 2.37: LoP for the Central Counter

$$rt = o(1 \cdot cnt) + (p-1) \cdot L\left((p-1) \cdot cnt\right) + o((p-1) \cdot cnt) + (p-1) \cdot L\left((p-1) \cdot cnt\right)$$

2.3.7.1.2 Combining Tree

The running time of the combining tree algorithm for a given n can be predicted as:

$$\begin{aligned} rt &= \lceil log_n p \rceil \cdot (o(1 \cdot cnt \cdot \lceil log_n p \rceil) + (n-1) \cdot L\left((n-1)\right) \cdot cnt \cdot \lceil log_n p \rceil)) + t_{bc} \\ t_{bc} &= \lceil log_2 p \rceil \cdot o(cnt \cdot \lceil log_2 p \rceil) + \lceil log_2 p \rceil \cdot L(cnt \cdot \lceil log_2 p \rceil) \end{aligned}$$

The describing figures have been omitted, compare the according LogP figures in 2.15 for rt and t_{bc} (the longest path - for t_{bc} starting at $P0$).

2.3.7.1.3 Tournament Barrier

The running time prediction for the Tournament Barrier is given by the following equation:

$$rt = \lceil log_2 p \rceil \cdot (o(1 \cdot cnt \cdot \lceil log_2 p \rceil) + L\left(1 \cdot cnt \cdot \lceil log_2 p \rceil\right)) + t_{bc}$$

2.3.7.1.4 n-way Dissemination Barrier

The n-way Dissemination barrier should perform as follows:

$$rt \quad = \quad \lceil log_{n+1}p \rceil \cdot (n \cdot o(n \cdot cnt \cdot \lceil log_{n+1}p \rceil) + n \cdot L(n \cdot cnt \cdot \lceil log_{n+1}p \rceil))$$

Note: the 1-way Dissemination barrier behaves exactly like the normal Dissemination Barrier.

2.3.7.1.5 Comparison inside the LoP Model

Figure 2.38 shows the predicted runtime curves for all algorithms using RDMA Write with inline send. As mentioned in the previous section 2.3.7.1, the algorithm is assumed to be balanced and no node has

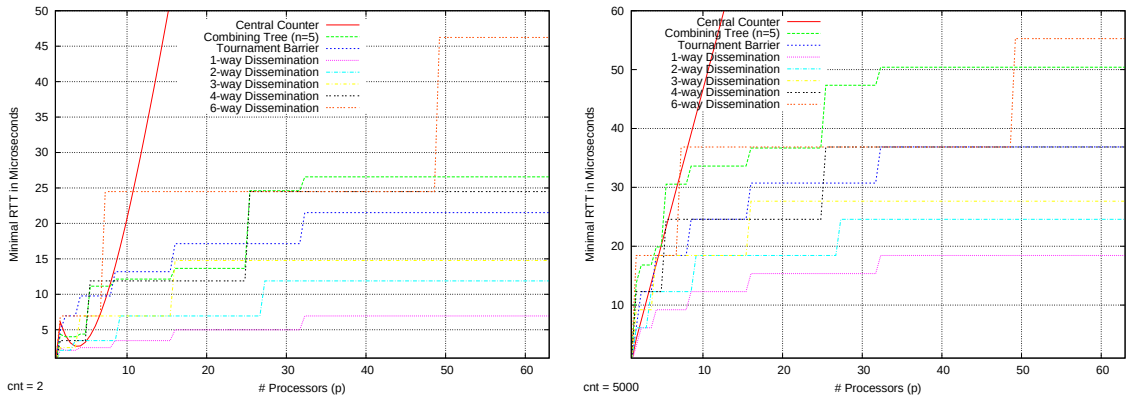

Figure 2.38: Minimal LoP Predictions for RDMA-Write inline (left: $cnt = 2$, right: $cnt = 5000$)

to wait for the others to arrive. Additionally, the best case scenario is used to predict the performance. Both assumptions are only taken to simplify the model and one should be aware that they influence the accuracy of the model. Thus, the benchmark results are expected to be bigger than the predicted results, especially for big node-counts. Additionally, the conclusion that the 1-way Dissemination Barrier seems to be the best is only valid within this simplified model without memory contention. Therefore, the benchmark in section 4.4.3 is conducted for different values of n. Figure 2.39 shows the estimated number

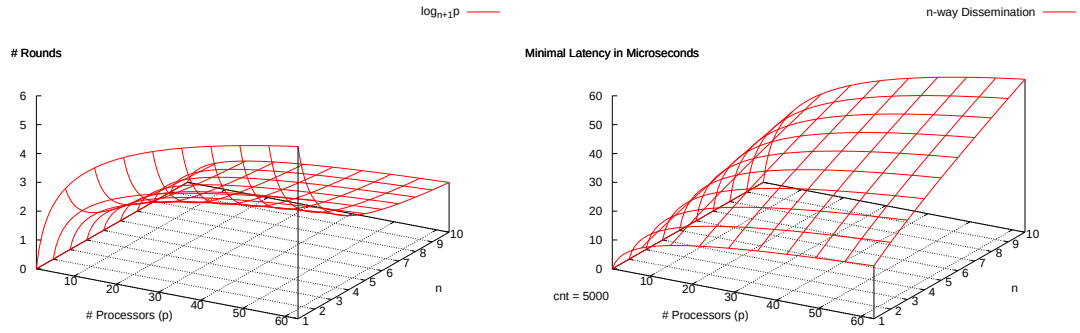

Figure 2.39: Round-count and appropriate predicted run time of the n-way Dissemination Barrier

of rounds for the n-way Dissemination Barrier to synchronize all nodes in the left and the predicted overall runtime with the parameters given by the LoP model for RDMA Write inline in the right. The number of rounds, which is the main indicator for the overall runtime declines rapidly with ascending n, but the predicted running time increases. This phenomena can be explained with the fact that more work has to be done per round and the "low-region" for sending a message (see figure 2.36) is left for 5000 consecutive barriers. But due to the simplifications done in this model, especially to neglect the memory contention and synchronization wait times, the benchmark result is quite unpredictable. All these theoretical models assume the best-case latency and give a lower bound to the barrier problem. The memory contention

and synchronization overhead can be assessed with the difference between the predicted and measured values.

2.4 Summary

This chapter analyzes several models regarding to their suitability for small messages within the InfiniBand network. Several well known barrier algorithms have been described and analyzed with regards to the number of issued network operations and their total running time inside the LogP model. A new model based on empirical benchmarks was proposed for small messages on the InfiniBand^{TM} network to increase the accuracy of the running time assessment. After that, a theoretically optimal solution was derived from all modelings and it was shown that the defined operations of the InfiniBand^{TM} architecture can not be used to reach a constant time barrier. Thus, the next chapter introduces several possibilities to achieve this task by leveraging hardware support.

Chapter 3

Hardware Solution

Software solutions as described in chapter 2 can be implemented in a portable and scalable way by sending 0 byte messages for synchronization. But at least the logarithmic complexity and a significant startup overhead per message remain and result in a high barrier latency. To reduce the impact of the logarithmic complexity and the overhead, the barrier operation can be totally offloaded or partially supported by the interconnect hardware. The three main models for offloading barrier are to add support in the existing data-network, deploy an extra synchronization network or to offload the operation to the NIC hardware. Several supercomputing vendors like Cray and Thinking machines and universities like the Purdue University added hardware support for global synchronization to their machines. Some of them use a dedicated network [KS93, LAD+96] and some try to leverage the existing data network [BP91, GGK+98, Pan, Sco96]. Additionally, several studies [ABP92, Sco96, OD89] present the benefits which can be gained by using hardware synchronization. The following sections propose hardware support schemes within the data network and an explicit synchronization network. NIC offloading is not contemplated because the currently available InfiniBand™ HCAs do not offer a user programmable NIC-CPU. Several practical studies have shown that the barrier performance can be improved [YBGP04] but only the o_s and o_r parameters are lowered and the influence of the machines PCI bus is eliminated, the Latency L and the gap g remain constant with their logarithmic complexity. Thus this technique can only offer a constant degree of improvement.

3.1 Barrier Support in the Data Network

A former study [SSP97] proposed a reliable synchronization scheme for switches. A new hardware scheme to improve interconnect switches, mainly focused on easy silicon design, high scalability and low latency will be proposed based on this paper. The following proposal is based on a single switch system, if multiple switches are connected to form a single network, the barrier operation is first performed within a single switch and propagated hierarchically between multiple switches. Therefore the switches could act as clients performing the most suitable barrier synchronization scheme for their topology (see chapter 2) in software. A hardware scheme is not proposed here because the additional costs would not pay off. A Mag-Pie [KHB+99] like software synchronization scheme seems more beneficial and can be implemented easier to avoid deadlocks.

3.1.1 Single Switch

It is assumed that P nodes are interconnected by a switch with N ports and the switch is built up as a crossbar (cmp. section 2.1.4.5). A new packet-type is defined for the low level switch protocol (e.g. a special LID in InfiniBand™ or a special MAC address in Ethernet) for all barrier related packets. These packets are generally very small and never routed between switches, they are only used for switch to host and host to switch communication. The crossbar logic inside the switch identifies these packets and passes them to the barrier logic incorporated in each switch (see figure 3.1).

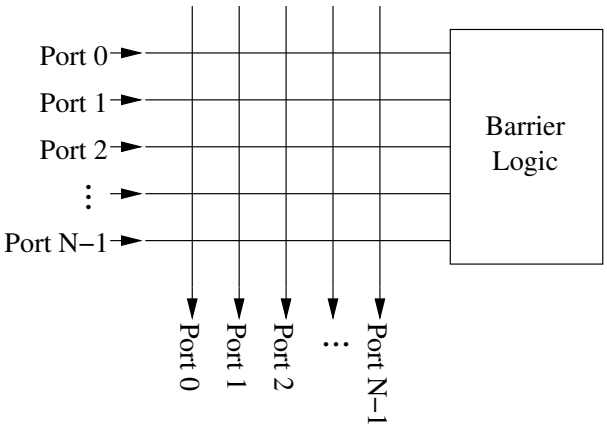

Figure 3.1: Barrier Logic inside the Crossbar

Each barrier unit can serve up to I barriers concurrently. To relate each barrier packet to a specific barrier in the logic, an ID i ($i \in \mathbb{N}; 0 \leq i < I$) is assigned.

The barrier logic implements a bit field of size $N + 1 \times 2 \cdot I$, where each line i and $i + I$ represents status information for barrier number i. Each bit n ($n \in \mathbb{N}; 0 \leq n < N$) in line i represents the current barrier status of the host connected to port n ($n = 1$ means barrier reached and $n = 0$ not). The bits n in line $i + I$ indicate whether the host connected to port n is participating in barrier i ($n = 1$ means no, $n = 0$ means yes). Each bit in the array can be addressed as (line,column), for example the bit 4 (port 4) in line 6 (barrier 6) can be written as $(6, 4)$. Bit N in line i is defined as follows:

$$(i, N) = ((i, 0) \vee (i + I, 0)) \wedge ((i, 1) \vee (i + I, 1)) \wedge \ldots \wedge ((i, N - 1) \vee (i + I, N - 1))$$

Bit N in line $i + I$ is used to indicate the status of barrier i (0 = unused, 1 = used). The whole bit-array and the functionality is shown in figure 3.2.

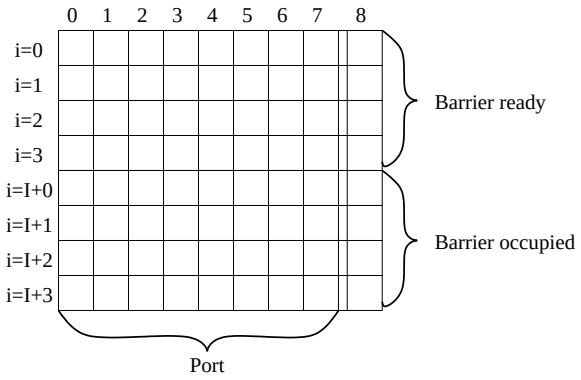

Figure 3.2: Bit Array in the Barrier Logic ($I = 4, N = 8$)

The protocol to initialize a barrier operation is performed after creating a new communicator (e.g. `MPI_INIT`, `MPI_COMM_DUP`).

1. rank 0 sends a `getId()` packet to the switch
2. the switch searches for bit $(i + I, N) \equiv 0$ ($0 \leq i < I$), sets the selected $(i + I, N) = 1$, initializes the barrier array and returns i to the requestor (as a `getIdAck(ID)` packet)
   - I is returned to indicate that all available barriers are used
   - the test for $(i + I, N) \equiv 0$ has to be performed as an uninterruptable operation (atomic) to prevent race conditions

- initialization is done by setting all (i, n) to 0 and all $(i + I, n)$ to 1 $(0 \leq n < N)$

3. rank 0 broadcasts the received ID to all ranks
4. all ranks p $(0 \leq p < P)$ send a `wantParticipate(ID)` packet to the switch
5. the barrier logic sets bit $(ID + I, p)$ to 0 and responses `wantParticipateAck(ID)` to the sender
6. each rank waits in a software barrier until all ranks received their `wantParticipateAck(ID)`

The initialization phase ends after all nodes have registered for participating in the barrier. The normal barrier operation protocol consists mainly of single send/receive operations which are processed in parallel at the switch.

1. rank p sends a `barrierReached(ID)` packet to the switch
2. if the switch receives a `barrierReached(ID)` packet at port n, it sets bit (ID, n) to 1 and responds with `barrierReachedACK(ID)`
3. if $(ID, N) \equiv 1$, then all nodes reached the barrier and the switch sends a `barrierReady(ID)` packet to all ports n where $(ID + I, n) \equiv 0$ after setting all bits (ID, j) $(0 \leq j < N)$ to 0
4. all ranks receive the `barrierReady(ID)` packet, answer with `barrierReadyACK(ID)` and leave the barrier

After finishing the parallel job, the barrier ID has to be deregistered at the switch.

1. rank 0 sends a `freeBarrier(ID)` packet
2. the switch sets $(ID + I, N)$ to 0 and answers `freeBarrierACK(ID)`

An example for operation is shown in figure 3.3.

Initial:

port 0: rank 1
port 1: –
port 2: rank 0
port 3: rank 2

	0	1	2	3	4
i=0	0	0	0	0	0
i=1	0	0	0	0	0
i=I+0	0	0	0	0	0
i=I+1	0	0	0	0	0

After Register:

port 0: rank 1
port 1: –
port 2: rank 0
port 3: rank 2

	0	1	2	3	4
i=0	0	0	0	0	0
i=1	0	0	0	0	0
i=I+0	0	1	0	0	1
i=I+1	0	0	0	0	0

Barrier 0 Ready:

port 0: rank 1
port 1: –
port 2: rank 0
port 3: rank 2

	0	1	2	3	4
i=0	1	0	1	1	1
i=1	0	0	0	0	0
i=I+0	0	1	0	0	1
i=I+1	0	0	0	0	0

Figure 3.3: Bit Array Operation $(I = 2, N = 4)$

3.1.1.1 Packet Format

The protocol uses 12 different packet types and can be encoded in 4 bits. The barrier id length depends on the biggest available barrier number, we assume 128 (7 bit) for this example. An example packet is shown in figure 3.4 and needs only 11 bit of payload. A possible protocol encoding is given in the

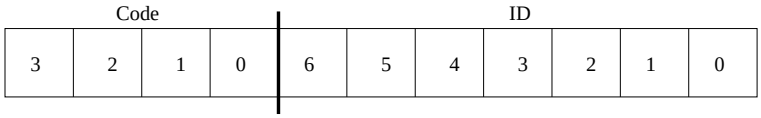

Figure 3.4: Barrier Packet Format

following table:

Code	Packet Type
0001	getId
0010	getIdACK
0011	wantParticipate
0100	wantParticipateACK
0111	barrierReached
1000	barrierReachedACK
1001	barrierReady
1010	barrierReadyACK
1011	freeBarrier
1100	freeBarrierACK

Table 3.1: Barrier Protocol Encoding

3.1.1.2 Reliability

Reliability is an important aspect, because network packets can be corrupted or get lost during transmission and cause deadlocks. Each packet has a corresponding ACK packet, which is used to notify the sender about successful reception. The sender repeats the request if no ACK packet is received within a specific time interval. The ACK packets are also assiciated with a specific barrier number to distinguish between different barriers on both sides.

3.1.1.3 Runtime and Scalability

The running time can be predicted with the following parameters:

- o_s overhead at the sending host to send a barrier message
- o_r overhead at the receiving host to receive and process a barrier message
- $o_p(P)$ maximal time to process a barrier in the hardware
- L network latency
- P number of hosts

A single barrier can be performed in:

$$t_b = o_s + 2 \cdot L + o_p(P) + o_r$$

Thus, the only parameter which depends on the number of hosts is the processing time $o_p(P)$. The asymptotic behavior of o_p is logarithmic ($O(log(P))$) and depends on the specific architecture.

The system is extremely scalable, even though the required number of Flip-Flops grows linear with the number of nodes. Only two additional Flip-Flops are needed per node and barrier. A 512 port switch offering 128 concurrent barriers would only need 128 000 Flip-Flops and the logic to process the barrier.

3.2 Barrier Support in a dedicated Network

An extra synchronization network, based on commodity components which leverages the parallel port has been implemented by the Purdue University and is described in [CDS94], [DHM96] and [DCMM95]. The achieved barrier latency for 4 nodes is only $2.5\mu s$. The architecture which is described in the following section is only tidly related to the main focus of this thesis. Thus, only a short description for a proof of concept design is given, which tries to rate the applicability to our problem domain and the upcoming problems introduced by such a solution.

3.2.1 Proof of Concept Design

This proof of concept design is used to build a barrier network between 4 nodes. The parallel port is used to exchange all information. The parallel port offers 1 byte of outgoing data and some additional flags, altogether 5 bits can be received and 12 bits can be sent in parallel. The parallel port is represented by three bytes, which can be accessed directly via the *inb*() and *outb*() 80x86 processor instructions relative to a `BASE` port (usually `0x378` for the first parallel port). The pin assignment is shown in figure 3.5. A C-

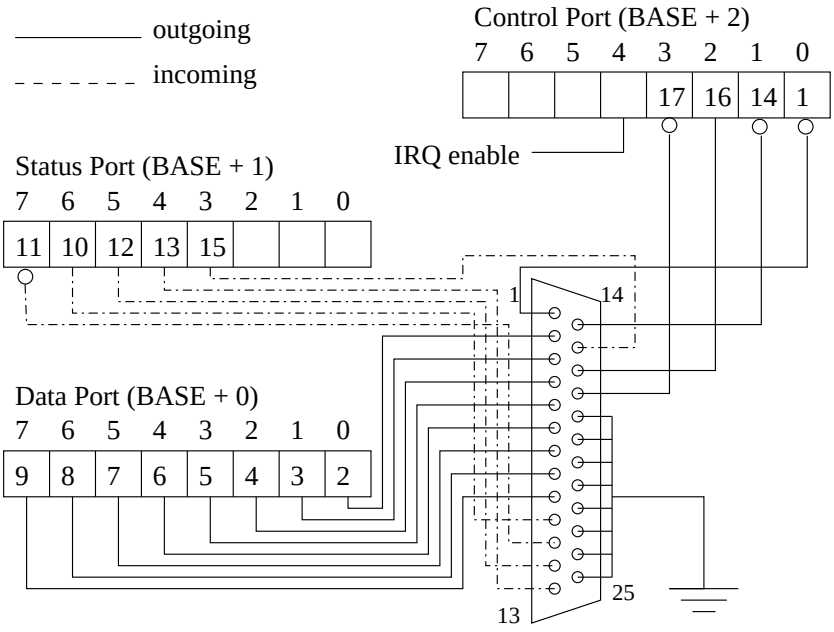

Figure 3.5: Parallel Port Pin Assignment (back side)

Source code to read from the port and write to it is given in listing 3.1. However, pin 14 (CONTROL[1]) was used as incoming line and pin 2 (DATA[0]) as outgoing to implement a single barrier for 4 nodes. A two-state machine, shown in figure 3.6 is used to implement the barrier mechanism on an FPGA board (ij is DATA[0] from node j, o is CONTROL[1] to node j ($\forall\ 0 \leq j \leq 4$)).

```
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378

int main()
{
  /* Get access to the ports - only as root! */
  if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

  /* Set the data signals (D0-7) of the port to all low (0) */
  outb(0, BASEPORT);

  /* Read from the status port (BASE+1) and print the result */
  printf("status: %d\n", inb(BASEPORT + 1));
}
```

Listing 3.1: Accessing the Port in C

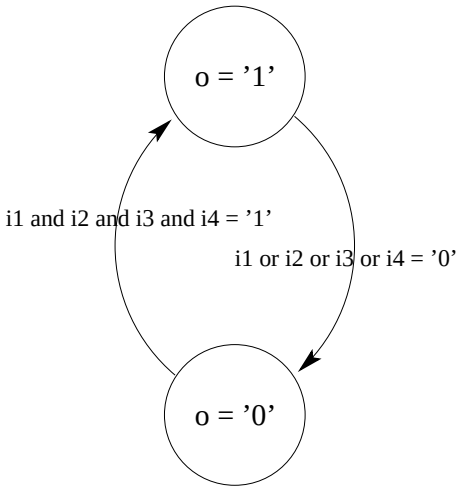

Figure 3.6: Two-state machine to implement the Barrier

3.2.2 Runtime and Scalability

This approach can be modeled with the following parameters:

- o_w CPU overhead to write to the parallel port
- o_r CPU overhead to read from the parallel port
- $o_p(P)$ processing overhead of a state change
- P number of participating processors

If a barrier completion is indicated ba a state transition and the application saves the state, the minimal running time of the barrier can be described by:

$$t_b = o_w + o_p(P) + o_r$$

One write is performed to indicate that the barrier is reached and one read (minimum) is performed to test if all nodes reached their barrier. The expected state (0 or 1) has to be saved by the application and toggled for every barrier entry.

3.2.2.1 Parameter Benchmark

A benchmark of the single parameters resulted in the following values on our testcluster ($1 \leq P \leq 4$):

$$\begin{aligned} o_w &= 1.2\mu s \\ o_r &= 1.2\mu s \\ o_p(P) &= P \cdot 10ns \end{aligned}$$

Thus, the running time can be predicted for our cluster with 4 nodes as:

$$\begin{aligned} t_b &= 2 \cdot 1.2\mu s + 4 \cdot 0.01\mu s \\ &= 2.44\mu s \end{aligned}$$

This mechanism is extremely scalable because the overall running time is nearly not changed even if the o_p parameter increases linearly.

3.2.3 Further Ideas

Some further ideas to enhance the barrier functionality are mentioned in the following. They are not analyzed in depth because the main focus of this paper is targeted at optimizing the barrier operation for InfiniBand™ .

A packet-like interaction with the barrier hardware could be used to add more flexibility. 12 bits can be send to and 5 bits can be received from the hardware through the parallel port. A possible mechanism to leverage this could be to write an 11 bit value and a status bit to the barrier hardware and to read one bit, to check if all participating hosts reached the given barrier number or not. 11 bit can be used to address 2^{11} different barriers, but several problems arise (e.g. if not all hosts are participating in a specific barrier). To tackle all these problems, the mechanism could be enhanced with the ideas proposed in section 3.1 to support a big number of very flexible barrier operations in parallel.

Another idea may be the implementation of the whole mechanism into the operating system to prevent access problems (the current approach is only suitable for privileged processes) by creating several new devices (e.g. `/dev/barrier0` ... `/dev/barrier<n>`) which block a reading process until all nodes reached the barrier. A non blocking approach, to support non-blocking barrier operation could also be provided (e.g. `/dev/nbarrier0` ... `/dev/nbarrier<n>`) where a read returns the actual status of the barrier.

3.3 Summary

This chapter described the possibilities to achieve a constant time barrier either with barrier support implemented inside the InfiniBand™ network or within a separate barrier network. The approach of a separate network is cost optimized and uses only commodity components and a cheap control hardware. This barrier was implemented as a proof of concept and has shown constant latency times about $2.5\mu s$. The next chapter evaluates all implementations based on this thesis and draws a conclusion for future work.

Chapter 4

Practical Results and Conclusion

4.1 Implementation

The different barrier techniques have been implemented within the Open MPI framework as described in section 1.4. The three different collective modules are tested in the following. The first implementation is the most portable one and does not require any special network, it is layered completely on the top of normal MPI point to point messages and is called swbarr which emphasizes the fact that no hardware support is needed. The second implementation is InfiniBand™ specific and leverages the perceptions from the LoP model to implement a low-latency InfiniBand™ specific barrier and is called ibbarr. The last module, called hwbarr requires our proprietary parallel port hardware to run and therewith is the least portable implementation.

4.1.1 Software Barrier

The Software Barrier (Swbarr) component offers implementations of different barrier algorithms based on point-to-point operations. It offers an extensible framework which is open to further additions (e.g. new algorithms). The configuration is done via mca-parameters in the `~/.openmpi/mca-params.conf` file. The following keywords are recognized:

- `coll_swbarr_priority` - priority of the swbarr
- `coll_swbarr_selection` - selected algorithm
- `coll_swbarr_dissn` - n-parameter (n-way Dissemination)
- `coll_swbarr_combn` - n-parameter (Combining Tree)

The following algorithms are available for selection (the `coll_swbarr_selection` number is given in brackets):

- (1) Central Counter (original Open MPI implementation)
- (2) Combining Tree Barrier
- (3) Tournament Barrier
- (4) Dissemination Barrier
- (5) Binomial Tree Barrier (original Open MPI implementation)
- (6) n-way Dissemination Barrier

If `coll_swbarr_selection` is set to 0 all available barrier algorithms are benchmarked during the init-phase and the fastest is chosen for later usage. This is the recommended type of operation because it adds only a small overhead during the creation of new communicators, but can enhance the barrier-performance significantly.

4.1.2 Hardware Barrier

The Hardware Barrier (Hwbarr) is implemented as described in section 3.2 and utilizes external synchronization hardware based on an Altera UP1 FPGA Board (see figure 4.1). The parallel-port of each node is connected to the external interface of the FPGA with a self-made connector cable (connecting ERR as input and DATA0 as output - compare figure 3.5). One limitation is that the current layout supports only `MPI_COMM_WORLD` and all nodes have to be connected to the Hwbarr. The MPI-Program has to run

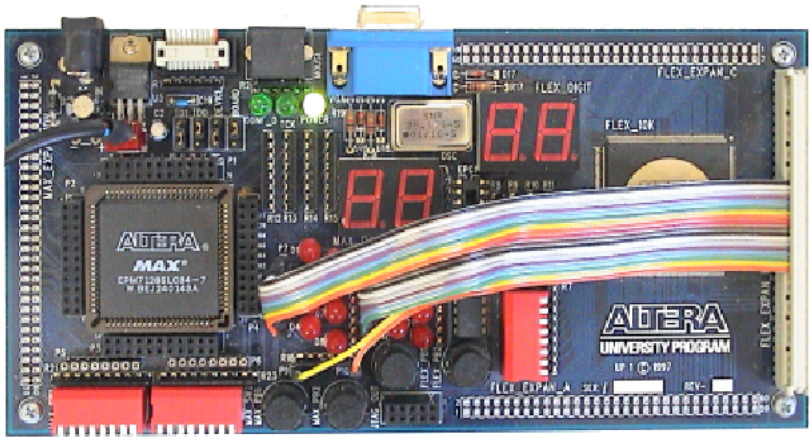

Figure 4.1: Altera Development Board

with superuser-privileges because of the direct parallel-port access. This implementation is only a proof of concept design and has to be enhanced for production usage.

4.1.3 InfiniBand™ Barrier

The InfiniBand™ Barrier (Ibbarr) Open MPI module offers an InfiniBand™ barrier and is implemented with the n-way Dissemination Algorithm (variable n). The current implementation uses RDMA Write with inline send to avoid race conditions in the send buffer during the send. The whole calculation of send peer and receive peer (compare listing 2.8) is done before the actual run (cached) to speed up the critical path.

Normal work requests are unsignaled, but the current implementation of the Mellanox HCA enforces the use of a signaled send request every n sends to complete all previous unsignaled ones. The maximum count of unsignaled send requests (n) can be specified with the constant `MAX_UNSIGNALED_WR_REQS` in the file `coll_ibbarr.h`.

4.2 Benchmark Environment

Three different clusters have been utilized to test the performance of the implementations due to the different hardware requirements.

4.2.1 Mozart

The Mozart Cluster, located at the University of Stuttgart, is populated with 64 nodes and the biggest InfiniBand™ cluster which was used to verify the results of this paper. A single node offers the following configuration:

- Processor: 2x3GHz Xeon
- Memory: 4GB

- OS: Red Hat Linux release 9 (Shrike)
- Kernel: 2.4.27 SMP
- HCA: Mellanox "Cougar" (MTPB 23108)

The nodes are interconnected with a 64 port Mellanox InfiniBand™ MTS 9600 switch and Gigabit Ethernet.

4.2.2 CLiC

The Chemnitzer Linux Cluster was used to verify all non InfiniBand™ related results (Swbarr). It consists of 528 nodes interconnected by 2 Fast Ethernet Interfaces (a single management and a single communication connection). The single nodes are configured as follows:

- Processor: 1x0.8Ghz Pentium III (Coppermine)
- Memory: 0.5GB
- OS: Red Hat Linux release 7.3 (Valhalla)
- Kernel: 2.4.18

The service network is connected by a hierarchy of 48 Port Cisco 3548 XL Switches and the communication network utilizes a single Extreme Black Diamond 6x96 Port Switch.

4.2.3 Oscar

The Oscar Cluster is our local test system, it consists of 4 InfiniBand™ capable nodes interconnected by a Mellanox MTS 2400 24 port switch. The single node configuration is given in the following:

- Processor: 2x2.4GHz Xeon
- Memory: 2GB
- OS: Fedora Core release 1 (Yarrow)
- Kernel: 2.4.22
- HCA: Mellanox "Cougar" (MTPB 23108)

Additional networks, like a Fast Ethernet service network and Gigabit Ethernet communication network are also available but not used for the following benchmarks.

4.3 Benchmark Applications

4.3.1 Expected Results

As shown in a long-term study at the HLRS[1] [Rab00], the barrier operation stands at the fifth position of the most time-consuming collective operations. It is responsible for about 6% of the CPU time consumed by the MPI library and respective 0.81% of the whole application running time of all profiled applications (in the average case). This is partially caused by the barrier latency itself to synchronize the processors (ideally if all processors reach the barrier synchronously) and the synchronization overhead which occurs on one processor while it has to wait for the others to reach the barrier (unbalanced application). The barrier latency is compared to the whole barrier waiting time very small, thus it can be assumed that the lion's share of the CPU time for barrier is caused by the the synchronization overhead in unbalanced applications. Thus, speeding up the barrier operation should only have a very small influence on the overall application running time, but can be very useful for benchmarks and other operations which require much synchronization (e.g. gang scheduling).

[1] HLRS [http://www.hlrs.de]

```
set reps = number of barriers (cnt - parameter)

MPI_Barrier(MPI_COMM_WORLD)

take_time(t0)
for i in 1..reps do
   MPI_Barrier(MPI_COMM_WORLD)
forend
take_time(t1)
set res = (t1-t0)/reps

MPI_Reduce(res, max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD)
if rank == 0 then
   print(max)
ifend
MPI_Finalize()
```

Listing 4.1: Pseudocode for the Microbenchmark

4.3.2 The Microbenchmark

The microbenchmark was developed especially for this task, because well known benchmarks like the Pallas MPI benchmark do not fulfill the demands. The Pallas benchmark is very inflexible and performs a big number of barrier operations and measures the mean time for this operation. The number of nodes can be given by a starting value and a multiplier, which is not suitable for bigger node counts.

The newly developed benchmark utilizes the RDTSC time measurement mechanism as used in section 2.3.5 to measure the exact time also for a single barrier operation. The pseudocode is given in listing 4.1. The first barrier ensures that all nodes enter the benchmark loop simultaneously.

4.3.3 The Application Abinit

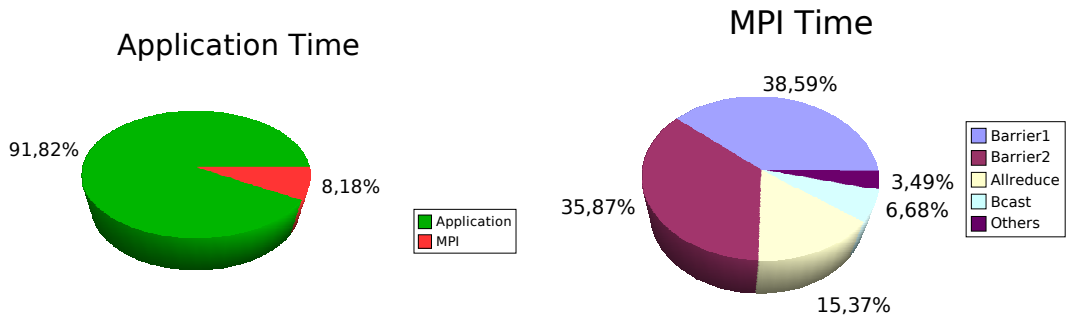

Figure 4.2: Abinit execution Time Allocation

Abinit was chosen to represent an application which wastes a reasonable amount of time for barrier synchronization. All tests are black-box tests, which means that the same input file was used to measure all times and the inner structure of the application has not been investigated. As shown in figure 4.2, Abinit spends about 8% of its execution time inside the MPI library and about 65% of this time for barrier synchronization. To determine if the time consumed by the barrier synchronization is caused by the implementation or by an unbalanced application, the disposition of the time to the single nodes (8 in this case) is shown in figure 4.3. One can see that the application is extremely unbalanced in Barrier2, where all nodes wait inside the barrier for node00 to reach its call. Barrier1 is slightly unbalanced, node01 and node06 seem to reach the barrier call usually as last nodes and the others waste CPU time while waiting for them. It is assumed that the execution time of Barrier1 can be enhanced. These facts show

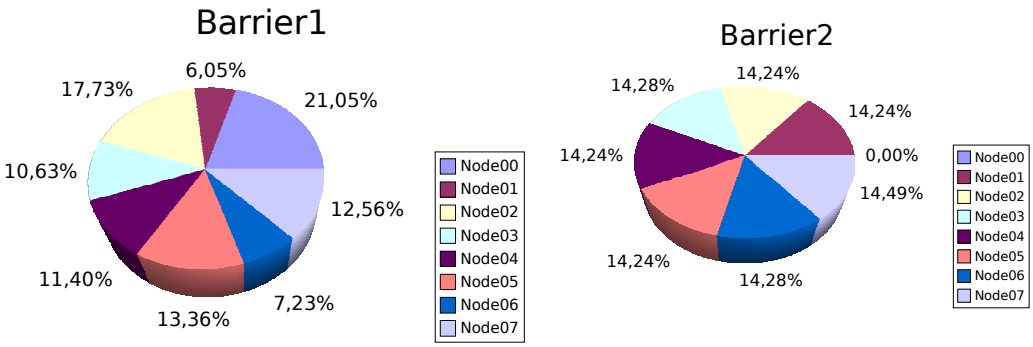

Figure 4.3: Abinit Barrier dispersion

that the overall application performance of Abinit can not be enhanced significantly with a better barrier implementation. The benchmarks should only show a very small improvement.

4.4 Microbenchmark Results

4.4.1 Software Barrier

The benchmark results of the n-way dissemination algorithm on the CLiC are depicted in the left side of figure 4.4. It shows that the 1-way dissemination barrier seems to be the best solution for this system. This is due to the fact that Open MPI does not schedule small messages across several network links (all are sent across the single low-latency connection). The right picture in figure 4.4 shows a comparison between the original Open MPI barrier algorithm and the new implemented Swbarr.

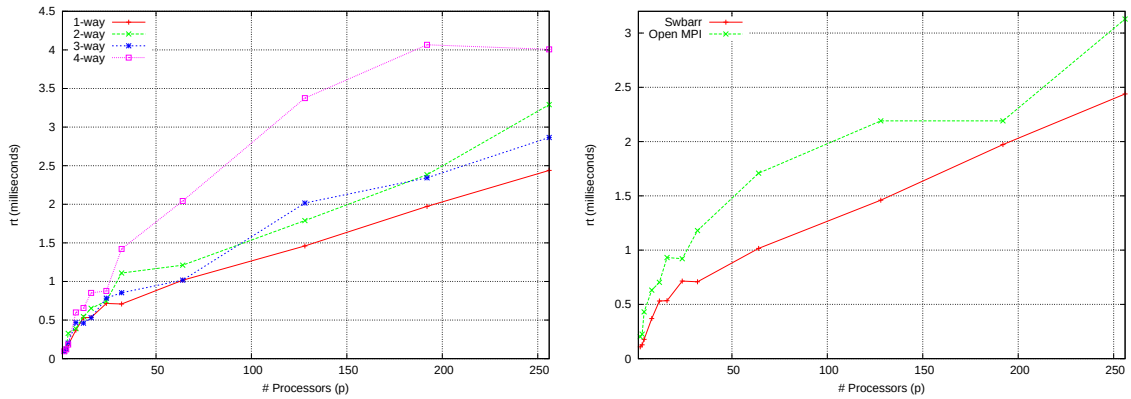

Figure 4.4: Swbarr Microbenchmark Results

4.4.2 Hardware Barrier

The implemented Hwbarr connects four of the Oscar nodes and works only for `MPI_COMM_WORLD`. The achieved results are constantly about $2.57\mu s$. This is around five times faster than the best known InfiniBand™ implementation and is assumed to remain constant also for bigger node-counts.

4.4.3 InfiniBand™ Barrier

Three different public domain MPI implementations which support InfiniBand™ have been compared for their `MPI_Barrier()` latency with the n-way Dissemination Barrier. The three implementations are namely:

- LAM-MPI 7.1.1
- MPICH2 0.9.6p2 (SHM+IBA device)
- MVAPICH 0.9.4

The results of the average barrier latency for 5000 consecutive runs on the Mozart cluster are shown in figure 4.5. The left side shows all measured implementations and the right only the four best. The n-way dissemination barrier is for $n < 3$ faster than today's fastest published barrier algorithm in MVAPICH (which also used RDMA Write). But the running time seems quite unpredictable, figure 4.6 shows in the left side a comparison to the predicted values. The LoP model represents a lower border. Additionally, it has to be mentioned that the n-way dissemination barrier falls back to a p-way dissemination barrier if $n > p$! The prediction varies quite a lot for $n = 1$, but the deviation becomes smaller with increasing n,

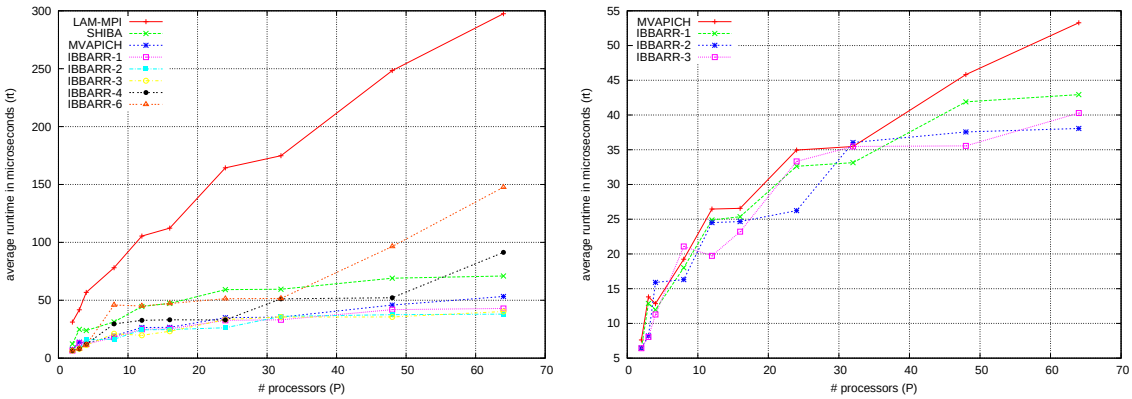

Figure 4.5: Microbenchmark Results

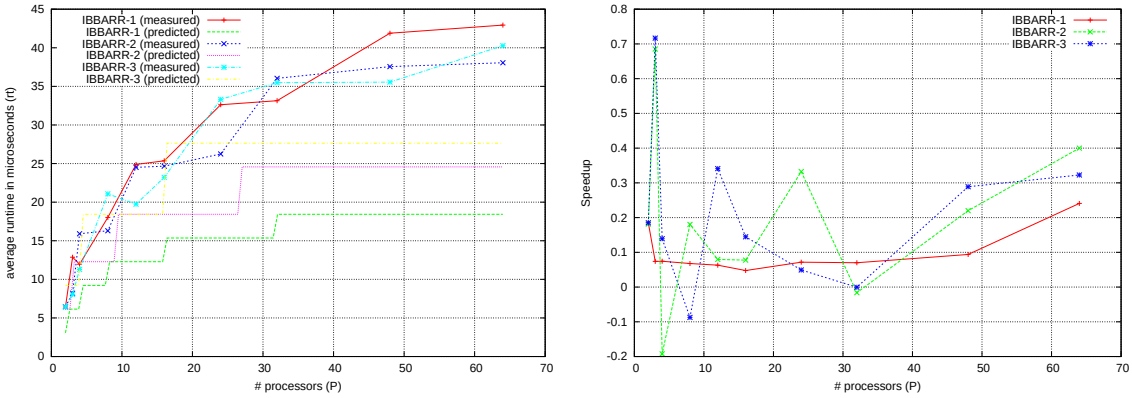

Figure 4.6: n-way dissemination measured and predicted running time

this can be explained with contention and synchronization effects which have been already described in section 2.3.7.1.5. The difference between the RDMA barrier of MVAPICH [LMP03] and the 1 or 2-way dissemination barriers increases with growing node numbers. The performance gain up to 40% for many nodes relative to the fastest published implementation is shown in the right of figure 4.6. It is assumed that the performance gain increases for more nodes. Further predictions cannot be made because the memory contention effects of RDMA Write are not predictable (compare figure 2.33).

Table 4.1: Abinit Benchmark Results

Barrier Type	Execution Time	Speedup	MPI Speedup
Open MPI	4:34 min	0%	0%
Ibbarr	4:30 min	1.45%	17.77%
Hwbarr	4:27 min	2.55%	31.17%

4.5 Application Results

The application results with abinit on four nodes are shown in table 4.1. The results are as expected, the overall application running time is only slightly enhanced, but the MPI Time can be drastically reduced with the usage of a constant time barrier. But this is not generalizable, because it is not recommended to use barrier operations in high-performance applications. Thus most of these applications use no barrier at all, or only once at startup or end.

4.6 Conclusion and Future Work

This thesis proposed different techniques to enhance the barrier operation for InfiniBand™ with or without a specialized supporting network.

It was shown that a cheap constant time barrier, leveraging commodity components (parallel ports) reaches a constant barrier latency (relative to the number of participated nodes) of approximately $2.5\mu s$ and can be realized quite easy. Ideas for a high-performance implementation and the management of barrier operations for InfiniBand™ switches have been developed and theoretically analyzed. Ideas to improve the proof-of-concept barrier hardware are described in section 3.2.3 and could be implemented to enhance the barrier latency for MPI applications.

A new InfiniBand™ barrier without additional hardware requirements has been implemented inside the Open MPI framework leveraging the newly developed n-way dissemination algorithm. The average barrier latency is up to 40% lower than the latency of the best known InfiniBand™ barrier implementation. It can be assumed that this gap grows for a bigger number of nodes.

Additionally, a new communication model (called LoP) for 1:n and n:1 communications in offloading based networks, based on the LogP model has been developed and parametrized for InfiniBand™ . A new InfiniBand™ benchmark was developed to set parameters for the LoP model. It was proven that this model provides at least lower bounds to all benchmarked values (the parametrization was done with ideal values). The behavior of the model for the average case has to be analyzed further to achieve more accurate predictions. The LogP model has been verified in its accuracy to model barrier algorithms for underlying TCP/IP based Ethernet networks.

The current Open MPI barrier operation has been enhanced up to 22% with a fully parametrizeable adaptive Coll component which benchmarks 6 different algorithms during the initialization of a new communicator and chooses the fastest one. The newly developed n-way dissemination barrier could be used more efficiently inside the Open MPI framework by providing a new PML component which schedules also small messages to different network interfaces.

The author realized during this work that the need for collective communication is emergent and the implementation of these so called "collectives" can be enhanced on different levels. The future work, based on this study will investigate the different collective operations in order to minimize their running time and the system overhead.

4.7 Acknowledgments

I want to thank my girlfriend and my family for providing the necessary social support for this thesis. I was technically supported by my advisers Frank Mietke, Torsten Mehlan and Prof. Rehm. I also want to thank Lavinio Cerquetti and Christian Siebert for many good and encouraging ideas. At last but not least, I received the necessary mathematical support to assess the equations for the LoP model from Prof. Junghanns, Prof. Spellucci and Dr. Sven Beuchler.

Appendix

A.1 List of Links

A.2 List of Figures

A.3 List of Listings

A.4 List of Pseudocode-Listings

A.5 List of Tables

A.6 Glossary

ACK *Acknowledgement* 6
API *Application Programming Interface* 7
BSP *Bulk Synchronous Protocol* 13
CPU *Central Processing Unit* 14
CQ *Completion Queue* 4
DMA *Direct Memory Access* 6
EVAPI *Extended Verbs API* 7
GID *Global Identification (Number)* 5
HCA *Host Channel Adapter* 3
HPC *High Performance Computing* 1
HTX *HypterTransport™Expansion* 3
IBA *InfiniBand Architecture* 3
IBTA *InfiniBand Trade Association* 3
IPv6 *Internet Protocol Version 6* 5
LID *Local Identification (Number)* 5
MIMD *Multiple Instruction Multiple Data* 12
MPI *Message Passing Interface* 1
NIC *Network Interface Card* 14
PCI-Express *Periphal Component Interconnect Express* 3
PCI-X *Periphal Component Interconnect Extended* 3
PD *Protection Domain* 7
PRAM *Parallel Random Access Machine* 12

PVM *Parallel Virtual Machine* 2
QoS *Quality of Service* 4
QP *Queue Pair* 4
RAM *Random Access Machine* 12
RC *Relieable Connection* 6
RD *Relieable Datagram* 6
RDMA *Remote Direct Memory Access* 4
RQ *Receive Queue* 4
RQE *Receive Queue Entry* 5
RR *Receive Request* 4
RST *Reset* QP Reset state 7
RTR *Ready to Receive* QP Ready to Receive state 7
RTS *Ready to Send* QP Ready to Send state 7
SF-IBAL *Sourceforge InfiniBand Access Layer* 7
SLOC *Source Lines of Code* 47
SQ *Send Queue* 4
SQE *Send Queue Entry* 6
SR *Send Request* 4
TCA *Target Channel Adapter* 3
TCP/IP *Transmission Control Protocol/Internet Protocol* 5
UC *Unrelieable Connection* 6
UD *Unrelieable Datagram* 6
VAPI *Verbs API* 7
WQ *Work Queue* 4
WR *Work Request* 4

A.7 References

[ABP92] John B. Andrew, Carl J. Beckmann, and David K. Poulsen. Notification and multicast networks for synchronization and coherence. *J. Parallel Distrib. Comput.*, 15(4):332–350, 1992.

[ACS89] A. Aggarwal, A. Chandra, and M. Snir. On Communications Latency in PRAM Computations. In *Proceedings of the 1st Symp. on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[ACS90] A. Aggarwal, A. Chandra, and M. Snir. Communications Complexity of PRAMs. *Theoretical Computer Science, 71:3-28*, pages 3–28, 1990.

[Aga91] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, 1991.

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1995.

[Amd00] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 79–81, 2000.

[BHP96] G. Bilardi, K. T. Herley, and A. Pietracaprina. BSP vs LogP. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 25–32. ACM Press, 1996.

[Ble87] G. Blelloch. Scans as Primitive Operations. In *Proc. of the International Conference on Parallel Processing*, pages 355–362, August 1987.

[BP91] Carl J. Beckmann and Constantine D. Polychronopoulos. Broadcast networks for fast synchronization. In *ICPP (1)*, pages 220–224, 1991.

[Bro86] Eugene D. Brooks. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[CDS94] William E. Cohen, Henry G. Dietz, and J. B. Sponaugle. Dynamic barrier architecture for multi-mode fine grain parallelism using conventional processors. In *ICPP*, pages 93–96, 1994.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[CLMY96] David Culler, Lok Tin Liu, Richard P. Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.

[CZ89] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 169–178. ACM Press, 1989.

[DCMM95] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad. Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL PAPERS Design. *Technical Report, Purdue University School of Electrical Engineering*, 1995.

[DHM96] Henry G. Dietz, Raymond Hoare, and Timothy Mattox. A fine-grain parallel architecture based on barrier synchronization. In *ICPP, Vol. 1*, pages 247–250, 1996.

[EM04] L. A. Estefanel and G. Mounie. Fast Tuning of Intra-Cluster Collective Communications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users Group Meeting Budapest, Hungary, September 19 - 22, 2004. Proceedings*, 2004.

[FG91] Eric Freudenthal and Allan Gottlieb. Process Coordination with Fetch-and-Increment. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 260–268. ACM Press, 1991.

[For95] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *www.mpi-forum.org*, 1995.

[FW78a] S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM Press, 1978.

[FW78b] 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. ACM Press, 1978.

[GFB+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*, Budapest, Hungary, September 2004.

[GGK+98] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer: designing a mimd, shared-memory parallel machine. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 239–254. ACM Press, 1998.

[Gib89] P. Gibbons. A More Practical PRAM Model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM Press, 1989.

[GMR97] P. G. Gibbons, Y. Matias, and V. Ramachandran. Can a shared memory model serve as a bridging model for parallel computation? In *ACM Symposium on Parallel Algorithms and Architectures*, pages 72–83, 1997.

[GTNP02] Rinka Gupta, Vinod Tipparaju, Jare Nieplocha, and Dhabaleswar Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, page 83. IEEE Computer Society, 2002.

[GV94] Dirk Grunwald and Suvas Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 604–608. IEEE Computer Society, 1994.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, 1989.

[Ham96] Susanne E. Hambrusch. Models for parallel computation. In *ICPP Workshop*, pages 92–95, 1996.

[HFM88] Debra Hengsen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[HK94] Susanne E. Hambrusch and Asfaq A. Khokhar. An architecture-independent model for coarse grained parallel machines. In *Proceedings of the 6-th IEEE Symposium on Parallel and Distributed Processing*, 1994.

[HMMR04] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A Survey of Barrier Algorithms in the Context of the LogP Model and Proof of Optimality. *Chemnitzer Informatik Berichte - CSR-04-03*, 2004.

[IBA] *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. ACM Press, 2001.

[KHB+99] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems.

In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press, 1999.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J.van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, pages 869–941. Elsevier, Amsterdam, 1990.

[KS93] R. E. Kessler and J. L. Scwarzmeier. Cray t3d: A new dimension in cray research. In *COMPCON*, pages 176–182, 1993.

[LAD+96] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[LCW93] James R. Larus, Satish Chandra, and David A. Wood. CICO: A Practical Shared-Memory Programming Performance Model. In Ferrante and Hey, editors, *Workshop on Portability and Performance for Parallel Processing*, Southampton University, England, July 13 – 15, 1993. John Wiley & Sons.

[Lei92] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., 1992.

[LJW+04] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Int'l Parallel and Distributed Processing Symposium, Proceedings*, 2004.

[LM88] C. Leiserson and B. Maggs. Communication-Efficient Parallel Algorithms for Distributed Random-Access Machines. *Algorithmica*, 3:53–77, 1988.

[LMP03] J. Liu, A. Mamidala, and D. Panda. Fast and scalable mpi-level broadcast using infiniband's hardware multicast support, 2003.

[LRWW98] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the nelder–mead simplex method in low dimensions. *SIAM J. on Optimization*, 9(1):112–147, 1998.

[LWP04] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *Int'l Journal of Parallel Programming, 2004*, 2004.

[LZ95] Welf Löwe and Wolf Zimmermann. Upper Time Bounds for Executing PRAM-Programs on the LogP-Machine. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 41–50. ACM Press, 1995.

[MCS91a] John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[MCS91b] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. *SIGARCH Comput. Archit. News*, 19(2):269–278, 1991.

[MF01] C. A. Moritz and M. I. Frank. LoGPC: Modelling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404, 2001.

[MMT95] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of Parallel Computation: A Survey and Synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61–70, 1995.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memory. *Acta Inf.*, 21(4):339–374, 1984.

[OD89] M. O'Keefe and H. Dietz. Performance analysis of hardware barrier synchronization. *Tech. Rep.*, 89(51), 1989.

[Pal00] Pallas GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, Pallas GmbH, 2000.

[Pan] Dhabaleswar K. Panda. Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms. pages 200–209.

[PN85] Gregory F. Pfister and V. Alan Norton. "hot spot" contention and combining in multistage interconnection networks. In *ICPP*, pages 790–797, 1985.

[Rab00] Rolf Rabenseifner. Automatic mpi counter profiling. In *42nd CUG Conference, CUG Summit 2000*, 2000.

[Sco96] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[SL04] Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, St. Malo, France, July 2004.

[SMC94] Michael L. Scott and John M. Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *Int. J. Parallel Program.*, 22(4):449–481, 1994.

[SSP97] Rajeev Sivaram, Craig B. Stunkel, and Dhabaleswar K. Panda. A reliable hardware barrier synchronization scheme. In *11th International Parallel Processing Symposium (IPPS '97), 1-5 April 1997, Geneva, Switzerland, Proceedings*, pages 274–280. IEEE Computer Society, 1997.

[TK97] Nian-Feng Tzeng and Angkul Kongmunvattana. Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 148–155. ACM Press, 1997.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[vN45] J. von Neumann. First draft of a report on the edvac. Technical report, University of Pennsylvania, 1945. *The report that got von Neumann's name associated with the serial, stored-program, general purpose, digital architecture upon which 99.99% of all computers today are based.*

[YBGP04] Weikuan Yu, Darius Buntinas, Rich L. Graham, and Dhabaleswar K. Panda. Efficient and scalable barrier over quadrics and myrinet with a new nic-based collective message passing protocol. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, 2004.

[YTL87] P.C. Yew, N.F. Tzeng, and D.H. Lawrie. Distributing Hot Spot Addressing in Large Scale Multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, 1987.

A.8 Theses

I Open MPI offers an extensible framework which is very suitable for implementing new MPI collective algorithms.

II Currently published MPI_Barrier implementations do not consider the architectural specialties of InfiniBand™ and can be enhanced further.

III The LogP model is only accurate for a big number of messages within the InfiniBand™ network

IV The LoP model is more accurate than the LogP model and can be simplified without loosing this advantage.

V Abstract programming models like the PRAM, BSP, C^3 and their modifications are not suitable for the derivation of optimal MPI collective algorithms.

VI The newly developed n-way Dissemination algorithm is the only published barrier algorithm which performs optimal on networks offering hardware parallelism.

VII The LoP model can be used to enhance other MPI collective algorithms which uses small messages for communication.

VIII A constant time barrier inside the InfiniBand™ network is achievable with hardware support integrated into the switches.

IX A constant time barrier can be built by leveraging commodity components and a cheap external network.

X Most application run times (e.g. abinit) suffer much more from unbalanced implementations than from barrier latencies.