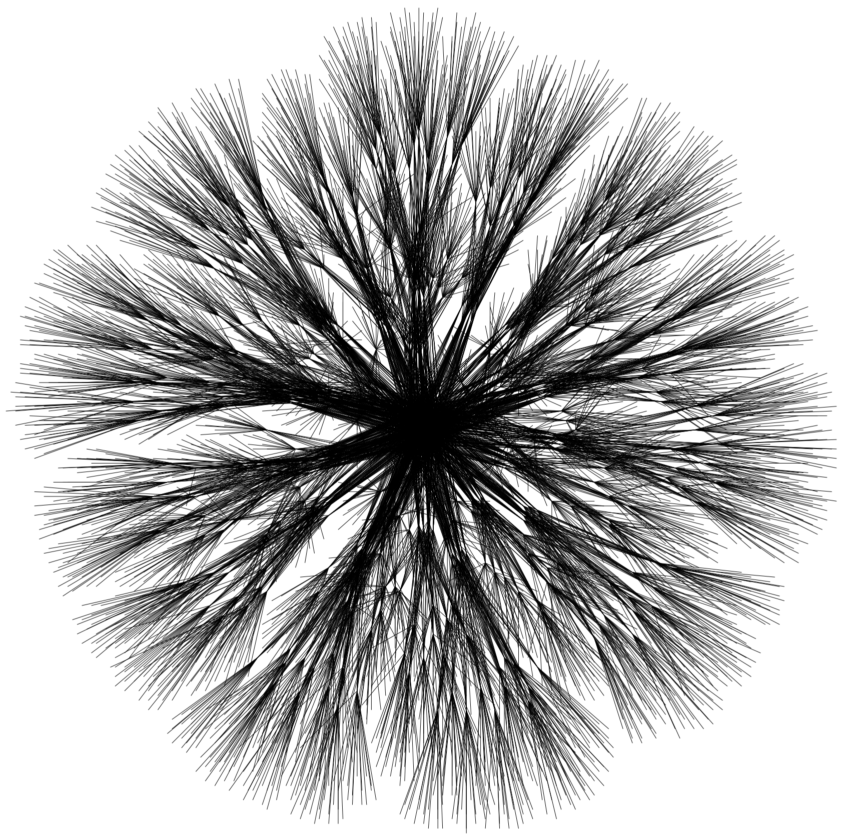
PRINCIPLES FOR COORDINATED OPTIMIZATION OF COMPUTATION AND COMMUNICATION IN LARGE-SCALE PARALLEL SYSTEMS

Torsten Hoefler



Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University
October 2008

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Andrew Lumsdaine, Ph.D. (Indiana University)

Randall Bramley, Ph.D. (Indiana University)

Jack Dongarra, Ph.D. (University of Tennessee)

Richard Graham, Ph.D. (Oak Ridge National Laboratory)

Minaxi Gupta, Ph.D. (Indiana University)

October 6th, 2008

... gewidmet meinen Eltern und моей любимой Наталии ...

Acknowledgments

First I want to thank my advisor, Prof. Andrew Lumsdaine, who is clearly one of the brightest and nicest people I know. His strong scientific and also personal support made it possible for me to complete my greatest academic challenge and finish this dissertation. Throughout my work with him, I was (and am) constantly impressed by his ability to analyze, respond to, present and understand a very broad range of scientific problems. I was also astonished about his interesting and very successful style of working.

Second, I would like to thank Prof. W. Rehm who helped me to start my academic career and guided me into the right directions after I graduated. I also want to thank him for the energy and the support he provided in the fight for funding and against the German administration and finally, that he understood my reasons to transfer to Bloomington.

Third, I want to thank all the students who chose to work with me on different problems. The students and friends Timo Schneider, Christian Siebert, and Andre Lichei deserve special naming due to their outstanding contributions to my work. I also want to thank my colleagues and friends in Chemnitz, Torsten Mehlan and Frank Mietke, for being the best conceivable office mates. I miss my old group in Chemnitz. Some members of the Open Systems Lab, especially Dr. Peter Gottschling, Prabhanjan 'Anju' Kambadur, Dr. Douglas Gregor, Dr. Timothy Mattox and Dr. Jeff Squyres (who originally invited me to IU) also provided an excellent collaborative environment.

Fourth, I would like to thank my committee members, Prof. Randall Bramley, Prof. Jack Dongarra, Dr. Richard Graham, and Prof. Minaxi Gupta for valuable suggestions and for taking the time to participate in this process. Many presentations by Prof. Jack Dongarra guided me towards my current career; I am very happy to have him in my committee. I also enjoyed the many runs accompanied by technical discussions with Dr. Rich Graham.

This work would not have been possible without the support of the following grants and institutions (chronologically by the time of support): AMD Saxony, the Saxon Ministry of Science and the Fine Arts, the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift of the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative.

I also want to thank the people who agreed to review this dissertation (additionally to the doctoral committee, alphabetically): Наталия Березнёва, Dr. Douglas Gregor, Dr. Laura Hopkins, Dr. Timothy Mattox and Christian Siebert.

At least as important as (if not more than) the technical support was the personal support during the time of my research and dissertation-writing for me. Most outstanding are my parents Rüdiger and Annemarie Höfler, who supported me continuously and made me to what I am today.

Many of my newer and older non-technical friends in Indiana and around the world accompanied me in hard and joyful moments and made my life worth living. I would like to thank Martin Calov, Benjamin Kammerer, Peiwei Li, Monique Simon, Christina Stouder, Mario Trams, Jungjin Yi, and Miho Yamagishi for being the great people around me. I hope to see you all again at different places! I also want to thank Siobhan Carroll for providing me with the energy to venture the big step into a different country, culture and language.

Finally, but most importantly, I want express my gratitude for the unconditional love and support of my girlfriend Наталия that helped me to manage all the difficulties of the last months and the writing process. I hope with all my heart that our love will defy the huge distance and will last forevermore.

Torsten Höfler
October, 2008, Bloomington, IN

Torsten Hoefler

Principles for coordinated Optimization of Computation and Communication in large-scale Parallel Systems

Solving complex scientific problems with massively parallel distributed computer systems remains a complex task. Data-dependencies in parallel applications enforce communication in addition to the computation of the solution. We aim to optimize communication and computation to ideally play together and minimize all overheads. Our work is based on the hypothesis that global knowledge about communication patterns and techniques to overlap communication and computation can be combined to decrease communication overhead and thus increase the scalability of parallel algorithms. We therefore analyze the communication patterns of several real-world applications and assess the detailed parameters of a particular interconnection network. Based on our findings and application analyses, we propose two new classes of collective operations: non-blocking collectives and topological (sparse) collectives. Theoretical analyses supported by an optimized implementation support our hypothesis. We propose different techniques to apply the newly defined operations to different application kernels, a parallel programming scheme and finally two real-world applications. Practical experiments show that the new operations can be used to reduce the communication overhead of the application kernels by up to 92%. The proposed pipelining techniques can be used as an abstract mechanism to achieve higher performance for many algorithms. We show reasonable performance improvements for the two analyzed real-world applications. We have also been able to simplify the interface in case of sparse communication patterns. However, we conclude that more fundamental changes to the algorithms and programming models are necessary to take full advantage of the new semantics. We expect that our results will influence programming models and parallel algorithm design. We conjecture that programmers and algorithm designers will need to combine intelligent layout of communication patterns and overlapping of computation and communication in order to achieve high performance and scalability.

Contents

About this Document

This document is structured into five chapters. Each of the chapters covers a different main topic, has a short introduction, and the most important findings are discussed in the conclusions of the chapter. The reader might skim the introduction and conclusion to decide whether to read a chapter. Although the dissertation is designed to be read from the beginning to the end, chapters are mainly self-contained and can be skipped. Cross-references and dependencies are clearly mentioned in the text and are relatively rare. A good overview about the whole work can be gained by reading introductions and conclusions of all chapters.

Chapter I introduces the problem and defines the "environment" for this thesis. Chapter II discusses the influence of the interconnection network to application communication patterns and shows several ways to optimize network architecture and communication patterns in order to improve the performance. Chapter III proposes a new class of complex operations that enable overlapping of computation and communication in today's high performance networks. Chapter IV shows an optimized implementation of those operations for InfiniBand^{TM}, analyzes several issues and tradeoffs and shows that the assumptions about overlap are true. Chapter V discusses the application of overlapping high-level communication routines to application kernels and real-world applications. Chapter VI summarizes the dissertation and research performed.

Theses

This dissertation states and proves different theses (chapters addressing/showing the thesis are in brackets):

- i There is untapped potential for performance improvement through coordinated optimization of computation and communication (II,III,IV)
- ii Features of modern high-performance systems and networks provide particular opportunities for coordinated optimization (asynchronous message processing) (II,IV)
- iii Non-blocking collective operations (NBC) are well suited to implement coordinated optimization of computation and communication in parallel algorithms (III,V)
- iv Even in coordinated optimization, individual pieces need to be well optimized: There is further improvement than can be made to (blocking) collective algorithms (II,IV)
- v One has to take into account numerous inter-related issues in providing coordinated optimization capabilities through NBC (modeling becomes very important) (II,IV,V)
- vi Applying NBC to parallel algorithms and codes involves combining communication and computation into an optimized (coordinated) schedule (V)
- vii Several important algorithms can overlap almost all communication latency with computation, resulting in significant performance improvements (V)

*) The picture on the front page represents the network graph of Thunderbird's InfiniBand network. One of the largest existing InfiniBand installations. The network layout was extracted and analyzed as part of this dissertation.

Chapter I

Introduction

"It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could be relegated to anyone else if machines were used." – Gottfried Wilhelm von Leibnitz, (1646-1716) German Philosopher and Mathematician

Today's High Performance Computing (HPC) is dominated by parallel computing. HPC mainly aims at providing solutions to large computational problems. HPC comprises different branches for different purposes. Some computations do not demand high computing power, but many independent calculations have to be performed to solve a single problem. This is common in scientific parameter studies. To enable such computations efficiently, parallel systems have to be tuned for high job throughput. This is a scheduling problem and good heuristics exist to solve it. Other computations need as much computing power as possible to solve a single algorithm with data-dependencies. Systems where such computations are run have to be optimized for parallel application performance. Those systems are often called capability computing systems. In this work, we primarily focus on capability computing to enable the solution of large-scale computational problems.

The most demanding computational problems, called *Grand Challenges* [159] are defined by a US agency as:

> *"A Grand Challenge is a long-term science, engineering, or social advance, whose realization requires innovative breakthroughs in information technology research and development (IT R&D) and which will help address our country's priorities"*

In other words, those problems can not be solved with today's computing systems. It is anticipated that solving one or more of those problems will have a significant impact on human life in general. However, solving such problems will require larger massively parallel computing systems than the ones that exist today. Grand Challenge problems include huge physical, biological, cognitive and other demanding tasks that yet have to be solved. Many problems require the efficient solution of some basic equations, for example the Navier-Stokes Equation, the Maxwell Equations, large-scale Poisson Equations and the Schrödinger Equation. Solutions of those equations can usually be found for smaller problems but problem-sizes needed in reality require long runs on large-scale systems. The questions "How would the ideal computer for the solution of those problems look like?" and "How do current computers compare to this ideal?" arise.

Large-scale capability computing, also called Supercomputing, is a fast-changing field of scientific research. It is mainly driven by the pragmatic needs of computational scientists to run large codes on current machines. A huge variety of different programming models and languages, essentially one language per architecture and vendor, was available in the early years of supercomputing. Thus, the portability and programmability of parallel codes was limited. Programming models can be categorized roughly into distributed and shared memory models. The shared memory model, whose best known implementation today is OpenMP [161], has traditionally been used for small-scale computations on single nodes. Approaches to use it in large-scale, shared memory machines and even distributed memory machines have been made but did not find wide adoption. Thus, we focus on the the distributed memory model. Two portable programming models, the Parallel Virtual Machine (PVM) [194] and the Message Passing Interface (MPI) [151], were formed in the 90's to standardize programming of distributed memory machines. MPI is today's most used programming model in large-scale computations and is also available on nearly all parallel architectures. MPI is often regarded the "assembler language" of parallel computing because it can be and is used as a basic building block for higher level systems. Programming models, such as UPC [206] or the HPCS languages X10 [212], Chapel [54] and Fortress [69] could use MPI as a compilation target. Thus, all concepts that affect MPI will affect languages based on similar principles as well.

Basing on the experience with Supercomputing and the nearly 15 years of MPI, we see the following shortcomings in today's HPC systems and programming principles:

1. Communication overhead, i.e., the time spent to communicate data, often limits scalability of parallel programs on large-scale machines due to Amdahl's law [32].
2. Applications often use point-to-point communication where collective operations could provide more information and thus give more optimization potential to the communication middleware.
3. The MPI standard is inconsistent because some, but not all, communication operations offer non-blocking interfaces.
4. The MPI standard does not support a high level of abstraction for sparse collective operations such as nearest neighbor exchange[1].
5. Using overlapping and latency-hiding techniques to improve application performance is difficult in the current setting.

Most of those problems result from the changed environment in HPC systems. Supercomputers grow in scale at different levels. Interconnection networks, CPU counts in network endpoints, and even the number of processing elements on a single die increase. Those new systems offer a hierarchical parallelism in different domains (shared and distributed memory) to the user. How-

[1]point-to-point implementations are possible but neither provide high abstraction to the user nor many optimization possibilities to the middleware

ever, it is clear that already existing memory bottlenecks, where the latency is limited by the speed of light and the bandwidth by space constraints, will prevent the use of all processing elements for computation. Moore's law predicts a doubling of transistors on die approximately every 18 months. Typically, those transistors have been used to implement additional parallelism in the CPU by exploiting the relative independence of single instructions. Tomasulo's algorithm [201] helped to exploit this Instruction Level Parallelism (ILP) transparently for application developers. Other ways to leverage the parallelism in user-codes are vector processors or vector extensions such as Intel's Streaming SIMD Extensions (SSE). However, the ILP of typical applications is limited to 4-5 instructions. The last possibility without leveraging explicit parallelism is to increase the clock speed of today's CPUs. This is also fundamentally limited by physical constraints such as leakage current in the transistors. However, compiler research focused many years on optimizations that enable as much implicit parallelism as possible.

Similarly to computer architecture, network architecture also evolved because Moore's law also applies to networking hardware. However, similar limitations as for CPUs apply. Exploiting parallelism leads to a higher packet-rate and bandwith in the network. Some networks even come with controllers that enable packet processing independently of the main CPU. The biggest limitation in networking is latency, i.e., the transmission time of a single small message, which is ultimately limited by the speed of light. The options to leverage this architecture are clear: the application must transmit much data in parallel. Also, more intelligent network hardware could be used to transmit messages while the CPU proceeds with the computation. However, many programming interfaces do not allow to post multiple outstanding network operations easily which effectively limits the exploitation of network parallelism. Thus, small changes to the networking side of supercomputing could offer new optimization potential.

We conclude that we will face massive parallelism at multiple levels (host and network level) and ubiquitous parallel programming in the near future. This observation leaves several open questions:

1. How much CPU interaction is needed to communicate with current networking hardware? or: How much potential for overlapping computation and communication do current networks offer?
2. Can overlapping techniques be efficiently combined with current network optimization techniques such as collective communication?
3. How much effort is needed to support non-blocking collective communication?
4. Can applications benefit from collective overlapping techniques?

This dissertation focuses on finding possible solutions to the identified problems and on answering the questions. We decided to investigate MPI in this work due to its basic nature and because it is the most adopted and a highly portable parallel programming model. It is also well analyzed and optimized and many applications build on MPI as a messaging layer. By using MPI, we ensure that our results can be compared to the optimized implementations for current supercomputers and that current applications can directly benefit from our findings.

In the next Chapter of this dissertation, we show ways to model modern interconnection networks and to improve the parameter measurement for a particular network model. Furthermore, we discuss a particular network in detail and show optimization potential for several collective communication patterns. We also discuss the influence of large-scale networking tolologies and routing to collective communication patterns and propose a way to optimize collective communication by using this information. In Chapter III, we propose new classes of collective operations that promise to solve many of the mentioned problems. Non-blocking collective operations could be used to hide latency by leveraging bandwidth and overlapping techniques. Sparse collective operations enable the handling of scalable sparse communication patterns in the middleware. We use our findings from Chapter II to derive models for the new collective operations and to gather first hints for optimization principles. In Chapter IV, we show simple techniques that enable an optimized implementation of our proposed operations. We also discuss a new benchmark scheme for those operations. We conclude the chapter with the description of an optimized implementation for a particular network and the detailed discussion of message-progression issues. We show different methods to use the new operations for the implementation of several application kernels in the first part of Chapter V. In the second part, we discuss optimization possibilities for two real-world applications: a quantum mechanical solver and a medical image reconstruction. Chapter VI summarizes and conludes the results of our work.

The following sections of the introduction define the environment of our work and discuss some common approaches used in HPC. We begin with a discussion of point-to-point communication and collective communication.

1 Point-to-point vs. Collective Communication

"Make everything as simple as possible, but not simpler." – Albert Einstein (1879-1955), German physicist, Nobel Prize 1921

The message passing interface uses processes to denote a basic unit of execution. An MPI job usually consists of multiple processes and each process runs on a single processing element (PE). A PE can either be a CPU or a CPU core (depends on the system configuration). Processes can be combined into process groups that can form a communication context (known as a Communicator in MPI). Each process has a unique rank in all its communication contexts from 0 to $P-1$ where P is the number of processes in the Communicator.

A central concept in MPI is, as the name suggests, message-passing. There are generally two options to pass messages between a source process, called the sender, and a destination process, called the receiver. The first option, two-sided communication, involves the receiver such that it has to post a receive request to the MPI library in order to receive a message. Each message is identified by the tuple (sender, receiver, tag, communicator) and matched in order of sending at the receiver. The tag can be used to distinguish logical communication channels between the same pair of processes (cf. TCP port numbers). The second option, called one-sided communication, aims to provide communication that is initiated by the sender only (with or without minimal receiver interaction). Point-to-point communications are often used for irregular communication patterns or communications in limited neighborhoods (communication peers per process).

Point-to-point messages can be used to build any complex communication pattern. However, the performance of those communication patterns heavily depends on network parameters, such as, for example, the physical topology. Even if all network parameters are known to the user, it is not trivial to design optimal algorithms. Even if optimal algorithms could be found, they would differ from system to system. To simplify the optimization process and overcome the related portability problems, MPI defines a fixed set of collective communication patterns that match most common application patterns. Those collective communication operations ("Collectives") involve all processes in a Communicator to perform a specific operation, such as a data broadcast.

Collective operations are beneficial in most scenarios. They take the burden of implementing an optimized communication pattern from the application developer. They also enable performance portability, i.e., the same application runs reasonably well on a wide variety of different architectures and network topologies because the collective implementation varies. Furthermore, collective operations are easy to use for common communication operations and also avoid errors made by application developers in the implementation of those recurring patterns. Thus, application developers should use the predefined collective operations whenever possible.

In a wider scope, one could argue that all point-to-point communications should be replaced by collective communications in order to produce well-structured parallel programs. Gorlatch [83] discusses the following five benefits:

1. *Simplicity*: Collective operations substantially simplify the implementation of programming patterns by supporting most common programming models like BSP [208] efficiently.
2. *Programmability*: Gorlatch defines several high-level program transformations that can be used in systematic program design. For example, a Fast Fourier Transform can be expressed as a series of (Map; All-to-all) operations.
3. *Expressiveness*: A broad class of communication patterns used in today's algorithms is covered by collective communication. Furthermore, flexible operations like the vector variants of all-to-all can be used to express arbitrary communication relations.
4. *Performance*: Even though many collective operations are implemented over point-to-point, the effort put into finding optimal algorithms and the use of special hardware support outweighs any point-to-point mechanism. Several collective operations can also be supported in hardware efficiently. We will demonstrate techniques for hardware-specific optimization of collective operations in Chapter II.
5. *Predictability*: It has been shown that collective operation performance can be predicted with roughly the same accuracy as point-to-point operations if the implementation is known. Furthermore, collective operation models are significantly simpler to handle than point-to-point models because they only model a single operation instead of $O(P)$ point-to-point operations.

We conclude this discussion by citing Sergei Gorlatch:

"The various kinds and modes of send-receive used in the MPI standard, buffered, synchronous, ready, (non-)blocking, etc., are just too primitive; they are too much an invitation to make a mess of one's parallel program."

2 Optimizing Scalable Parallel Applications

"When there were no computers programming was no problem. When we had a few weak computers, it became a mild problem. Now that we have gigantic computers, programming is a gigantic problem." – Edsger Dijkstra (1930-2002), Dutch computer scientist, Turing Award 1972

Optimization of parallel programs can be subdivided into two parts, serial (often compiler-based) optimization and communication optimization, that are partially orthogonal. Serial optimization deals with mapping the algorithm to the CPU architecture as efficiently as possible. This is often done with optimizing compilers, restructuring the code to enable better cache utilization, or by using special libraries that are optimized for the underlying architecture. Those methods have been studied for decades and are rather mature.

> *"Now, that everybody has a parallel computer - and everything is a parallel computer - it is everyone's problem."*

We conjecture that communication optimization becomes more relevant because the stagnation in single processor speed will force developers to exploit explicit parallelism in order to achieve higher performance. Running an algorithm in parallel usually requires communication between the processing elements. Those communications can be expressed in terms of element-to-element or point-to-point communications. However, this might lead to a limited view of the global communication. Thus, higher-level approaches look at the more abstract notion of communication patterns, for example broadcast. Those patterns can also be defined in terms of point-to-point operations but they often allow optimized communication algorithms (e.g., a tree for small-message broadcasting).

In order to achieve highest communication performance, it is necessary to match the communication pattern to the underlying communication network. Communication networks have several important parameters (e.g., latency, bandwidth and topology) that are needed to find optimized communication algorithms. For example, an optimized broadcast operation on a torus-topology is different from a broadcast algorithm optimized for a star-topology. Network models are often used in order to help designing and to prove algorithms for this mapping. Thus, network models need to reflect the essential parts of the network. The following Chapter discusses the detailed influence of communication networks to parallel applications.

Chapter II

Influence of the Communication Network

"An expert is a man who has made all the mistakes which can be made in a very narrow field." – Niels Bohr, (1885-1962) Danish physicist, Nobel Prize 1922.

As discussed in Chapter I, the performance of single processing elements does not increase significantly and parallelism becomes more important for future computing systems. Large-scale programs require large-scale parallelism and large-scale networks. Those communication networks are the crucial factor in the performance and success of large-scale applications and computer systems. Communication usually does not advance the computation, but it is necessary in order to achieve correct results. Thus, communication times are often seen as overhead that has to be reduced. The simplest way to reduce this overhead is to employ low-latency and high-bandwidth networks. Using those networks represents today's state of the art and we use this as a base for further development.

In order to produce useful results and make our work practically relevant, we have to choose a particular communication network. We implement our ideas with InfiniBand^{TM} [199], mostly due to its wide availability and large-scale deployment. Many newly designed cluster-based supercomputers use InfiniBand^{TM} as their interconnection network. The InfiniBand^{TM} network is able to offer latencies as low as $1.3\mu s$ [65] and a bandwidth of up to 60Gb/s [197].

This chapter summarizes and extends results from the articles "Assessing Single-Message and Multi-Node Communication Performance of InfiniBand" [20], "LogfP - A Model for small Messages in InfiniBand" [24], "A Communication Model for Small Messages with InfiniBand" [13], "Fast Barrier Synchronization for InfiniBand" [23], "A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast" [17], and "Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks" [16].

In order to understand all the network details, their implications and finally parametrize a network model, we need to analyze the network and all its options in detail. We developed a microbenchmark tool named Netgauge [12] that implements all the different transport options to measure all important parameters. The following section provides a detailed analysis of the InfiniBand^{TM} network. Then, we discuss different established models for parallel computation and how accurately they model our needs. We select a particular model which reflects the InfiniBand^{TM} architecture well. Section 2.1.6 describes an extension to this model for increased accuracy and Section 2.4 shows a novel benchmarking technique for the model's parameters. Section 3 describes optimization techniques for collective operations over InfiniBand^{TM}. Section 4 and 5 analyze the behavior of InfiniBand^{TM} networks at large scale and with collective communication patterns, and finally, Section 6 points out a path towards topology-aware collective operations.

1 A practical Analysis of InfiniBand

"There's no sense in being precise when you don't even know what you're talking about." – John von Neumann, (1903-1957) Hungarian Mathematician

The analyses done in this section contribute to the understanding of the InfiniBand^{TM} hardware in terms of collective multi-node communication. We analyze the performance of 1:n and n:1 communications over InfiniBand^{TM} and provide detailed benchmark results for several cluster systems. This section contains and extends results from the article "Assessing Single-Message and Multi-Node Communication Performance of InfiniBand" [20].

The terms (MPI) process and node are used throughout this section. We consider a "process" similarly to MPI as an activity that may join a multicast group and a "node" as a physical machine. Although multiple processes are allowed on a single physical machine, we used only one process per node for all benchmark results in this chapter.

Related Work Only a few microbenchmarks are available for InfiniBand^{TM}. The most common practice is to test the performance atop MPI with MPI-level benchmarks (e.g., the Pallas Microbenchmarks [163]). The available MIBA Microbenchmark Suite [53] is able to measure point-to-point communication performance for InfiniBand^{TM}. Additional studies [141] used this benchmark suite to measure communication and application performance on different systems. However, it does not analyze the effects of node to multi-node communication patterns. We implemented a new benchmark in Netgauge in order to evaluate the collective behavior with multi-node configurations.

1.1 InfiniBand Transport Types

The InfiniBand^{TM} standard offers different transport services like stream or datagram to the user. Each transport service has special features and shows different communication times and overheads. Additionally, different transport functions can be performed with each transport service (see Table 36 at Page 245 in the InfiniBand^{TM} specification [199]). We call a combination of transport function and transport service *transport type*. All different transport types have to be analyzed in order to draw an accurate conclusion for the implementation of a parallel algorithm or a collective operation. The different transport types are briefly explained in the following.

1.1.1 Transport Services

This section briefly explains the different transport services that are defined in the InfiniBand^{TM} specification.

Unreliable Connection The Unreliable Connection (UC) offers point-to-point transmission based on unreliable connections without flow control. Packets may be silently discarded during transmission. A queue pair (QP) has to be created at each host and connected to each other in order to transmit any data. The queue pair acts as a connection and endpoint identifier.

Reliable Connection The Reliable Connection (RC) is basically identical to UC despite the fact that the correct in-order transmission of packets is guaranteed by the InfiniBand^{TM} hardware. An automatic retransmission and flow control mechanism is used to ensure correct delivery even if slight network errors occur. The role of the QPs remains the same as for UC.

Unreliable Datagram The Unreliable Datagram (UD) transport type offers connectionless data delivery. The reception or in-order delivery is not guaranteed in this case. QPs are only used to send or receive packets and each packet can have a different destination (they do not denote a virtual channel as for UC/RC). A single datagram must not exceed the MTU of the network (typically $2KiB$).

Reliable Datagram The Reliable Datagram (RD) transport type adds a guaranteed reliable in order delivery to UD. QPs can be used to reach any target as in the UD case. RD is currently not supported in our test system.

RAW The RAW Transport type can be used to encapsulate other transmission protocols as Ethernet or IPv6. It has nearly the same characteristics as UD and will not be discussed in the following.

1.1.2 Transport Functions

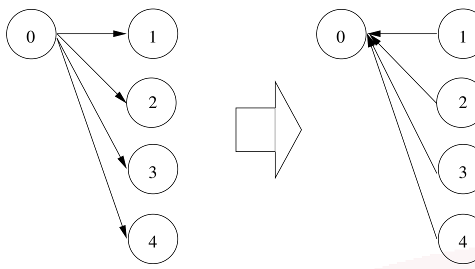
In this section, the possible transport functions are discussed and bound to a specific transport type for the benchmark.

Send The simple send function is available for all IBA transport services and will be measured for all of them.

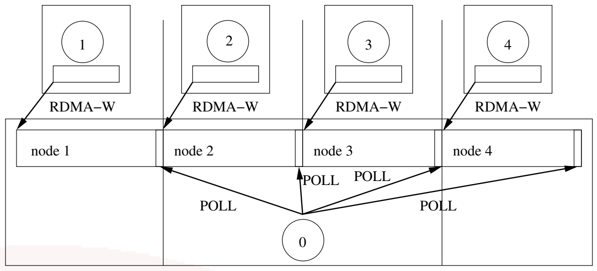
RDMA Write The Remote Direct Memory Access Write (RDMA-W) can write to explicitly registered memory at a target without the need to interrupt the target's CPU. RDMA-W can be used atop RC, UC and RD. We present results for RDMA-W atop RC because RD is currently not supported.

RDMA Read The Remote Direct Memory Access Read (RDMA-R) can read from registered memory at a target system without influencing the remote CPU. RDMA-R can be used atop RC and RD. We present results for RDMA-R atop RC because RD is currently not supported.

Atomic Operations Atomic Operations (AO) can be useful to support interprocess synchronization calls. Our current system does not support the optional AOs and we cannot present any measurement values.



(a) 1:n n:1 Benchmark Communication Pattern



(b) RDMAW Receive Buffer Layout at Node 0

Figure II.1: Illustrations of the used Benchmarking Schemes for 5 Hosts

Multicast Multicast (MC) is available for UD only and could be used to enhance collective patterns like broadcast. Several studies [139, 121] have been conducted to leverage the multicast features for different collective operations. Our measurements will provide a tool to give a theoretical proof of their efficiency.

1.2 Benchmark Principle

We use four different benchmark scenarios to test the four different InfiniBand$^{\text{TM}}$ transport functions. We use notified receives in the Send transport function such that each received packet creates a Completion Queue (CQ) entry which can be consumed via the poll CQ Verbs API (VAPI) call. The simple send-receive is tested with the following 1:n n:1 principle between P nodes numbered from $0..P-1$:

1. Node 0 posts a request to each node $1..P-1$ (ping).
2. Node 0 polls its CQ until all nodes answered.
3. Nodes $1..P-1$ wait for the reception of a message (poll CQ).
4. Nodes $1..P-1$ send the message back to node 0 (pong).

The resulting communication pattern for $P=5$ is depicted in Figure II.1(a).

The RDMA-W benchmark uses basically the same principle with unnotified receives (step 2 and 3) for performance reasons (i.e., no event is generated at the receiver). Polling a counter is the only way to detect whether a message arrived. RDMA-W operation has been finished when the last byte of the receive buffer is changed (in-order delivery is guaranteed in InfiniBand$^{\text{TM}}$). The receive buffer layout in node 0 is depicted in Figure II.1(b). Node 0 polls n bytes to check the reception from n peers. Polling introduces some memory congestion overhead on node 0 which has to be accepted because there is no faster method of testing whether a message has been received or not (interrupts or CQ elements are too slow in this context).

The RDMA-R time for n nodes to read from a single node is measured in a different way:

1. Node 0 sends via RDMA-W to nodes $1..P-1$ (ping).
2. Nodes $1..P-1$ wait via polling for the RDMA-W from node 0.
3. Nodes $1..P-1$ read via RDMA-R from node 0 (pong).
4. Nodes $1..P-1$ take the used time for their RDMA-R operation.

The multicast benchmark is performed by sending a single UD Multicast packet from node 0 to all other nodes. All other nodes wait for the CQ entry via polling the CQ. Each node returns a unicast UD packet to node 0.

All round-trip-times (RTT) are measured at node 0 between the first send and the last receive (except in the second stage of RDMA-R). The 1:n n:1 benchmark principle also tests the performance of the InfiniBand$^{\text{TM}}$ implementation (the switch as well as the HCA and the driver stack) under heavy load and maximum congestion (n nodes send/receive to/from a single node).

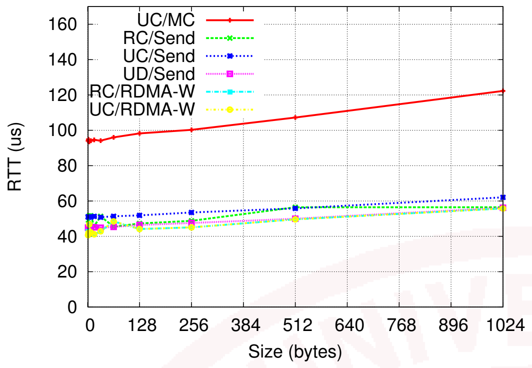
To enable statistical analysis as well as the assessment of minimal and maximal transmission times we use a different measurement scheme than most other benchmarks. Common benchmarks (e.g., [53, 163]) often perform a defined number of repetitions s (e.g., $s = 1000$) in a loop and divide the measured time of all tests by s. This prohibits a fine-grained statistical analysis (to find outliers as well as determine absolute hardware limited minima). Our approach measures each packet separately and stores the results. This makes it possible to find minimum and maximum values and enables the calculation of the average and median. This measurement scheme has also some impact on the measured values themselves. We measure each packet, which means that the whole pipeline startup (in the network cards itself and in the network) is measured each time. This results in poor performance compared to the usual fully pipelineable benchmarks (many repetitions). However, we argue that parallel applications do not communicate s messages between two hosts (usually this is done in a single bigger message) and that our scheme represents the reality better. All displayed curves are normalized to the number of addressed hosts. Thus, the RTT costs or o costs per message, called RTT/n and o/n are plotted for different numbers of hosts n.

1.3 Benchmark Results

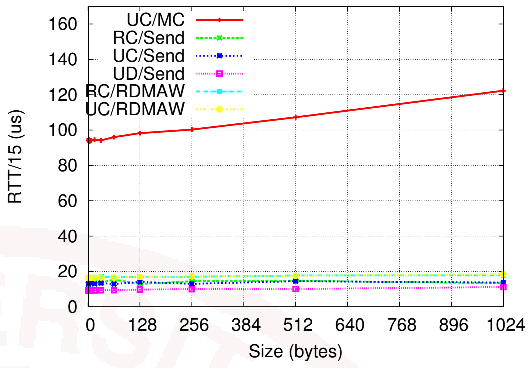
Round trip times ($RTTs$) for different message sizes and different numbers of participating nodes of a specific InfiniBand$^{\text{TM}}$ cluster are presented in the following section. The cluster system consists of 64 nodes with 3 GHz dual Xeon CPUs, $4GiB$ main memory, Mellanox "Cougar" (MTPB 23108) HCAs running Red Hat Linux release 9 (Shrike) with kernel 2.4.27 SMP.

1.3.1 Scaling with Message Size

First, we analyze the RTT scaling with the message size for 1:1 communication (normal ping-pong benchmark). The RTT scaling for increasing small message sizes up to $1KiB$ is shown in Figure II.2(a). The MTU has been adjusted to $2KiB$ so that every message fits into a single packet. The communication latency ($RTT/2$) scales linearly with the message size as predicted by well-known network models. A LogGP modeling of this scenario would predict accurate communication times because G in the LogGP model predicts linear scaling with the message size (the detailed LogGP model will be discussed later in this chapter). The multicast latency is about twice as high as normal point-to-point measurements. This is due to old hardware; newer systems have similar latencies for multicast as for UD/Send. A second analysis, shown in Figure II.2(b), determines the latency scaling up to $1KiB$ with the message size for a 1:15 communication. The measurement results show the unexpected result that the single packet latencies to address 16 hosts are lower than the according latencies in the 1:1 case. This leads us to the conclusion that the interface design ben-

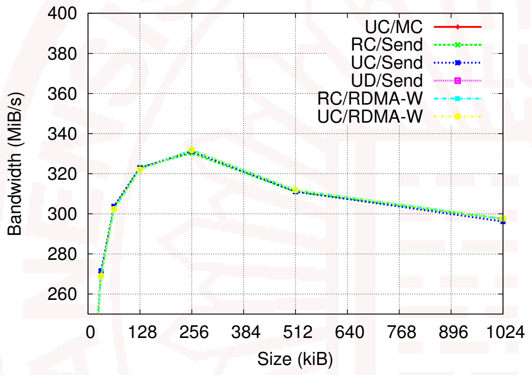


(a) 1:1 RTT with varying message sizes

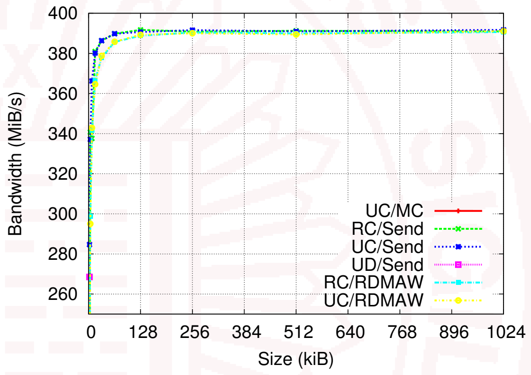


(b) 1:15 RTT with varying message sizes

Figure II.2: Latency scaling with different communication options



(a) 1:1 Bandwidth with varying message sizes

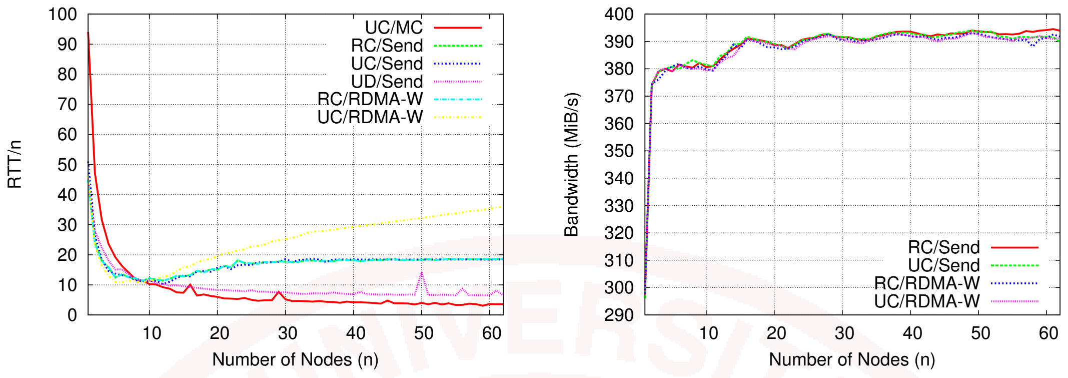


(b) 1:15 Bandwidth with varying message sizes

Figure II.3: Bandwidth scaling with different communication options

efits multiple successive packets. The only exception is the multicast result, which uses a single multicast in the 1:15 direction and 15 UD in the 15:1 direction.

The bandwidth scaling ($bandwidth = messagesize/(RTT/2)$) for larger message sizes up to $1MiB$ in the 1:1 case is shown in Figure II.3(a) and the 1:15 case is shown in Figure II.3(b). Again, this benchmark shows clearly that the full capacity of the HCA cannot be reached by sending single packets and that a 1:n communication should be preferred over a 1:1 communication. This shows that the accuracy of the LogGP model is limited for this type of communication because its predicted communication times are directly proportional to the number of addressed hosts (a single g is accounted for each host). These results are quite interesting for the optimization of collective communication because they show that the sending of multiple packets in a single communication round can increase the throughput significantly. Today's collective algorithms [149] usually use only a single communication partner per round. The bandwidth is much lower than the expected $1000MiB/s$. This is due to the single message sends. We changed the benchmark to send multiple messages successively and were able to achieve higher throughput ($\approx 950MiB/s$).



(a) Single byte Transmission Latency (b) $1MiB$ Message Transmission Latency

Figure II.4: Scaling with the Number of Hosts

1.3.2 Scaling with the Number of Hosts

To further investigate this phenomenon, we change the number of hosts. Figure II.4(a) shows the measured latency in relation to the host-number for the transmission of a single byte message. We see that the latency is decreasing with a growing the number of addressed hosts for MC and UD. Other transport services seem to have a local minimum around $n = 10$ and the latency is increasing for higher processor counts. This may result from local congestion (polling the memory or creating and/or polling CQ entries). This result shows clearly that one should send up to 10 messages in parallel to achieve best results.

All measured parameters are highly system dependent. We have conducted these measurements on different InfiniBand^{TM} systems and saw similar results (consult [24] for details). This makes it possible to use our benchmark suite to evaluate and compare the performance of different InfiniBand^{TM} implementations and to find bottlenecks during InfiniBand^{TM} development. The benchmark suite is also able to measure parameters like send and receive overhead and latencies to poll the completion queue under different circumstances.

2 Modelling the InfiniBand Architecture

"All models are wrong; some models are useful." – George Box, (1919) English Statistician

Communication models play an important role in designing and optimizing parallel algorithms or applications. A model assists the programmer in understanding all important aspects of the underlying architecture without knowing unnecessary details. This abstraction is useful to simplify the algorithm design and to enable mathematical proofs and runtime assessments of algorithms. Models also help non-computer scientists to understand everything they need for programming and are useful for computer architects to give running time estimations for different architectures. These models have to be as accurate as possible and should reflect all important underlying hardware properties. But an architectural or communication model must also be feasible for programmers. This means that the number of parameters and the model functions must be relatively small.

The programmer has to understand the model and all its implications. It is easy to see that the accuracy and the ease of use are conflicting and the designer of a network model has to find the golden mean. This section presents results published in the articles "A Communication Model for Small Messages with InfiniBand" [13] and "LogfP - A Model for small Messages in InfiniBand" [24].

Related Work Many different models have been developed in the past. There are models for specific network architectures [131, 39] or for the shared memory paradigm such as CICO [128, 80]. Other models like the Parallel Random Access Machine (PRAM) [72, 115], BSP [208], C^3 [91] or LogP [61] aim to be architecture independent and to give a general estimation of programming parallel systems. These are quite inaccurate due to their high level of abstraction. Several comparative studies [144, 90, 37] and [22] are available for assessing the accuracy of subsets of these models. Our comparative study [22] and the prediction of the MPI_BARRIER latency [21] with LogP shows that the LogP model is quite accurate for small messages. Some groups are working on evaluating different models for different hardware architectures. For example Estefanel et al. has shown in [70] that the LogP model is quite accurate for Fast Ethernet. Many efforts [29, 153, 107, 119] have been made to enhance the LogP model in its accuracy for different network architectures and large messages.

2.1 Comparison of Established Models

This section discusses several established network models. To simplify the analysis, we choose InfiniBand™ as an example network. Each mentioned model is described by its main characteristics. A reference to the original publication is given if the reader is interested in further details (e.g., detailed information about execution time estimation). Each model is analyzed with regards to its advantages and disadvantages for modeling the InfiniBand™ architecture. A conclusion for further usage in the design process of a new model is drawn. Different enhancements by third authors that have only a small impact on the accuracy of the model are omitted.

2.1.1 The Parallel Random Access Machine Model

The PRAM model was proposed by Fortune et al. in 1978 [72]. It is the simplest parallel programming model. It was mainly derived from the RAM model, which bases itself on the "Von Neumannn" model. It is characterized by P processors sharing a common global memory. Thus it can be seen as a MIMD[1] 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 measure for the ideal parallel complexity of an algorithm.

The main advantage is the ease of applicability, but to achieve this simplicity, several disadvantages have to be accepted. Namely all processors are assumed to work synchronously, the interprocessor communication is nearly free (zero latency and infinite bandwidth lead to excessive fine-grained algorithms), and it neglects the contention when different cells in one memory module are accessed. Thus, mainly because interprocessor communication is not free but orders of magnitude slower than the CPU clock, this model is not suitable for modeling InfiniBand™.

[1]Multiple Instruction Multiple Data

2.1.2 The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model was proposed by Valiant in 1990 [208]. The BSP model divides an algorithm into several consecutive supersteps. Each superstep consists of a computation and a communication phase. All processors start synchronously at the beginning of each superstep. In the computation phase, the processor can only perform calculations on data inside its local memory (if this is data from remote nodes, it has been received in one of the previous supersteps). 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 an h-relation. A cost of $g \cdot h$ (g is a bandwidth parameter) is charged for the communication.

Latency and (limited) bandwidth are modeled as well as asynchronous progress per processor. Each superstep must be long enough to send and receive the h messages (the maximal h among all nodes!). This may lead to idle-time in some of the nodes if the communication load is unbalanced. Another problem is that messages received in a superstep cannot be used in the same superstep even if the latency is smaller than the remaining superstep length. Even though the BSP model is an excellent programming model for parallel algorithms, it is not suitable for our purposes because communication algorithms have to be as asynchronous as possible in order to achieve the highest performance.

2.1.3 The C^3 Model

The C^3 model, proposed by Hambrusch et al. in 1994 [91], 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 Section 2.1.2).

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. 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 (e.g., Intel Touchstone Delta). Therefore, this model cannot be used to model InfiniBand$^{\text{TM}}$ networks and is not investigated further.

2.1.4 The Hockney Model

The Hockney model [101] simply models the network transmission time as the latency α and the message-size s multiplied by the reciprocal of the bandwidth β.

It is reasonably accurate on today's networks but the simple modeling approach ignores architectural details of offloading-based networks like InfiniBand$^{\text{TM}}$. It also does not distinguish between the CPU and network parts of the network transmission and is thus oblivious of the hardware parallelism.

2.1.5 The LogP Model Family

The LogP Model [61] was proposed by Culler et al. in 1993. It was developed as an addition to the PRAM model (see Section 2.1.1) to consider 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 (e.g., the Intel Delta or Paragon, Thinking Machines CM-5). It is based on four main parameters:

- L - communication delay (**upper** bound 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, can be refined to o_s for send overhead and o_r for receive overhead).
- g - gap (indirect communication bandwidth, minimum interval between consecutive messages, $bandwidth \sim \frac{1}{g}$).
- P - number of processors.

Several additions to the original LogP model exist and are discussed later.

The LogP model has several advantages over other models. It is designed for distributed memory processors and the fact that network speed is far smaller than CPU speed. It is easily applicable for a flat network model (central switch based, diameter = 1) and it encourages careful scheduling of computation and overlapping communication as well as balanced network operations (no single processor is "flooded"). This is profitable for accuracy of determining the run time of many applications. It is easy to understand that developing and programming in the PRAM model is easier than in the LogP model, but the higher accuracy of this model justifies the additional effort. Some 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^{TM}) are able to perform collective operations (e.g., Multicast) ideally in $O(1)$.

The different LogP model extensions aim at improving the prediction accuracy of the standard LogP model by taking different network effects into consideration. The LogGP model by Alexandrov et al. [29] models large messages with the new G parameter that indicates bulk-transfer rates. The LoGPC model by Moritz et al. [153] discusses the effects of network contention in the LogGP model. Synchronization overhead during the sending of large messages in high-level communication libraries such as MPI is modeled in the LogGPS model. To address the hardware parallelism (e.g., pipelining, super-scalar principles) in current high-performance networks like InfiniBand^{TM}, the LogfP model [24] adds the new parameter f, which represents the number of consecutive small messages that can be sent for "free". All those different models can be combined to predict the performance of a specific network. We decided to use the LogGP model for this work because it has been proved accurate and it is general enough to ensure the applicability of our measurement method for many networks.

The models of the LogP family have been used by different research groups to derive new algorithms for parallel computing, predict the performance of existing algorithms, or prove an algorithm's optimality [36, 60, 104, 116, 149]. While the derivation of new algorithms and the proof of optimality can be done without the real parameter values, accurate measurement methods for the single parameters are necessary to predict performance of algorithms or message transmission processes.

2.1.6 Choosing a Model

As described in 2.1.5, the LogP model family is the most accurate model for our specific scenario. Thus, we use it for all running time estimations.

Several simplifying architectural assumptions can be made without lowering the asymptotic accuracy of the model. Based on the fact that most clusters operate a central switch hierarchy which connects all nodes, the properties of this interconnect can be assumed as follows:

- Full bisection bandwidth (will be discussed later in this chapter).
- 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 overhead (o) is constant for single messages (for simplicity: $o_s = o_r = o$).

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 one on the sender side (o_s). The according visualization of the different parameters for a LogP compliant network (e.g., Ethernet) can be seen in Figure II.5.

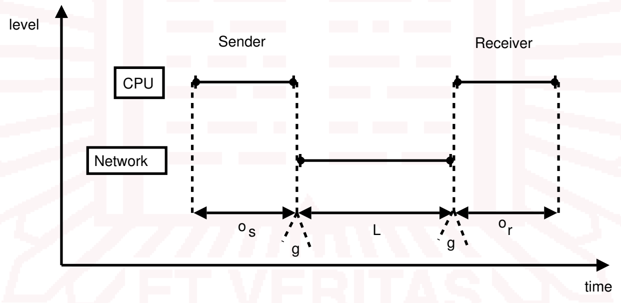


Figure II.5: Visualization of the LogP parameters in a simple point-to-point message transmission from a “Sender” process to a “Receiver” process. The ordinate (level) shows the shows CPU and Network.

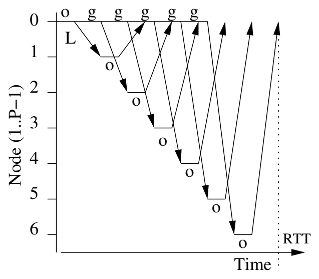
The following constraints also apply to the model:

- $\left\lceil \frac{L}{g} \right\rceil$ - count of messages that can be in transmission on the network from one to any other processor in parallel (network capacity).
- L, o and g are measured as multiples of the processor cycle.

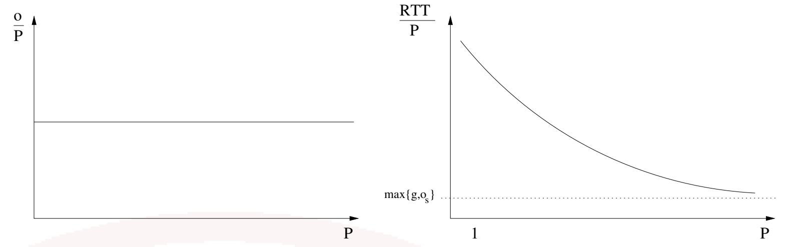
An additional study [62] describes options of assessing the network parameters for real-life supercomputers.

2.2 Verifying the LogGP Model on InfiniBand

In order to verify the LogGP model for InfiniBand$^{\text{TM}}$, we compare the 1:n n:1 InfiniBand$^{\text{TM}}$ benchmark results from the previous section with the LogGP prediction. Our benchmark measures trans-



(a) LogP Communication Structure ($P = 7$)



(b) LogGP Predictions

Figure II.6: LogP Graph and Prediction for Ping-Pong Benchmark Scheme

mission latencies and CPU send overheads for posting the send request. We examine only RDMA-W because the previous section and other studies [142, 138] showed that it is the fastest way to transmit data over InfiniBand^{TM}, and it causes no CPU overhead on the receiver side. The LogP model predicts a constant o and

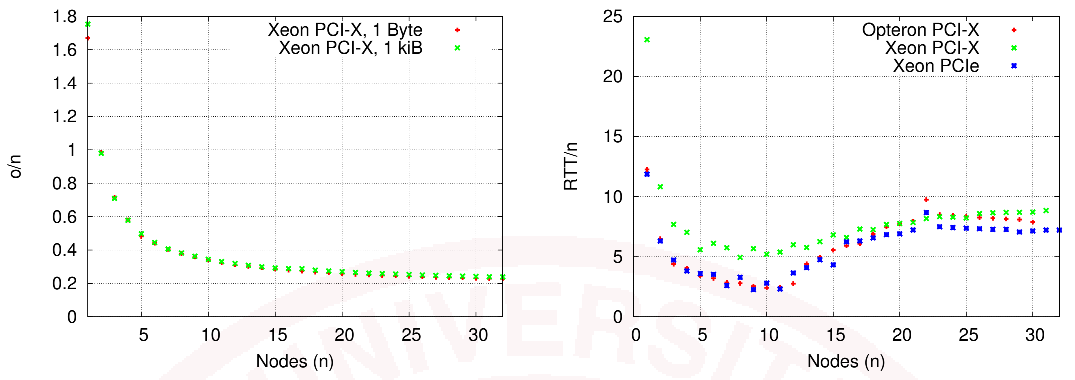
$$\begin{aligned} RTT &= o + (n-1) \cdot max\{o, g\} + L + o + L \\ &= 2L + 2o + (n-1) \cdot max\{o, g\} \end{aligned} \tag{II.1}$$

needed for transmitting a single packet to n hosts and back. The LogP communication diagram is shown in Figure II.6(a). The expected graph signature of the RTT/n (cf. Equation (II.1)) normalized by n is shown in Figure II.6(b).

Our InfiniBand^{TM} 1:n n:1 overhead (o) benchmark results are shown in Figure II.7(a) for 1 byte and $1KiB$. It can be assumed that there is no difference, and the overhead does not depend on the size of the posted message (ignoring InfiniBand^{TM}'s memory registration costs). The LogP prediction for o, shown in Figure II.6(b), is constant and cannot express the benchmarked function signature properly. The RTT/n latency benchmark results for 1 byte messages on different cluster systems are shown in Figure II.7(b). We show results for an AMD Opteron Cluster where the HCA is connected with PCI-X and two Xeon clusters with PCI-express- and PCI-X-connected HCAs to evaluate different architectures. The small message function has a totally different signature than the LogP prediction (the time per message has a global minimum at $n \approx 10$). This special behavior of InfiniBand^{TM} has been analyzed and modeled in "A Communication Model for Small Messages with InfiniBand" [13]. Large messages are modeled well with LogGP. The next section will discuss a slight modification to the LogGP model for small messages that takes the hardware parallelism into account.

2.3 Modeling small Messages over InfiniBand

In order to increase the prediction accuracy of the LogGP model, we parametrized a detailed model, called *LoP*. However, this model has multiple non-linear parameters and is thus very hard to apply. In this section, we introduce the slighly less accurate LogfGP model which is much easier to use. The LogfGP model is a simplified version of the original *LoP model* which is defined in [13] and is



(a) CPU overhead measurement results (b) Latency measurement results

Figure II.7: o and 1:n n:1 Benchmark Results

derived from the original LogGP model. The main characteristics and the ease of use are retained in the new design. The f parameter indicates the number of messages where no g has to be charged, which are essentially for $f$ree. Since only g is affected, we explain the model based on the LogP model. However, the gap per byte parameter G can be added to this model without any changes.

2.3.1 Overhead Model

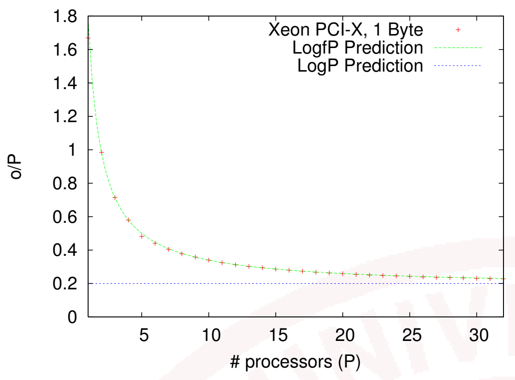
The overhead can be modeled with a simple pipeline startup function. We assume that this is due to cache effects in the HCA and CPU (instruction) cache effects. The parameters are chosen to be more mnemonic:

$$o(P) = o_{min} + \frac{o_{max}}{P} \qquad \text{(II.2)}$$

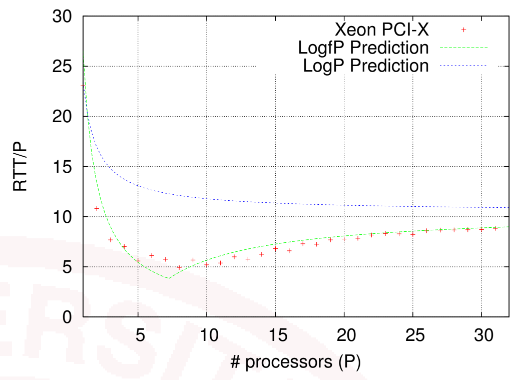
Where o_{min} is the lowest achievable (fully "warmed up") $o(P)$ for $P \rightarrow \infty$ which is $0.18\mu s$ on our system. The maximal value for $o(P)$, o_{max} is exactly $o(1)$ and $1.6\mu s$ in our example. Both parameters are easy to derive if you have just two measurement results, $o(1)$ and $o(\inf)(\approx o(x)$ for a sufficiently large x). This model is relatively accurate and easy to use. The model's prediction and the measured values are shown in Figure II.8(a). The $o(P)$ parameter is important to assess the CPU load for each send operation. It does not play a big role for the send process itself because the L parameter is usually 10 to 100 times bigger for InfiniBand^{TM}. Thus, the $o(P)$ could be replaced with the scalar o from the LogP model for network transmissions (this introduces a slight inaccuracy but reduces the number of parameters).

2.3.2 A Model for the Round Trip Time

Our RTT model is similar to the LogP model. The difference is that the g parameter is not charged for every message. We assume that multiple small messages can be processed in parallel by the network hardware (as discussed in the previous section) and sent nearly simultaneously to the network. Thus, g is only paid for every message after f messages have been sent, which means that the first f small messages are essentially for $f$ree in our model. It is obvious that this cannot



(a) Overhead o



(b) Round Trip Time (RTT)

Figure II.8: $1 : P - P : 1$ Benchmark Results with Predictions

hold for large messages, due to the limited bandwidth. Thus, the $RTT(P)$ can be modeled as:

$$\forall(P \leq f)\ RTT(P) = 2L + P \cdot o_s(P) + o_s(1) \tag{II.3}$$

$$\forall(P > f)\ RTT(P) = 2L + o(P) + o_s(1) + max\{(P-1) \cdot o(P), (P-f) \cdot g\} \tag{II.4}$$

The benchmark results show that our simple modification of introducing the f parameter enhances the accuracy of the model significantly. The LogfP model is quite accurate for the prediction of small messages while the LogP model overestimates all $RTTs$. The introduction of the $o_{min,max}$ parameters enhances the o modeling of the LogP model. LogP underestimates the needed CPU time to send a message because it models it as constant.

2.3.3 LogfP Parameter Assessment

All LogfP parameters can be gathered from the 1:n n:1 benchmark described in the previous section. They are explained in the following:

- o_{min} - equals $o(\infty)/P$ of the $o(P)$ benchmark
- o_{max} - equals $o(1)/P$ of the $o(P)$ benchmark
- L - equals $\frac{RTT(1)-2o_{min}-2o_{max}}{2P}$ of the $RTT(P)$ benchmark
- g - equals $RTT(\infty)/P$ of the $RTT(P)$ benchmark
- f - is the global minimum of the $RTT(P)/P$ curve
- P - number of processors

These parameters can easily be measured and used for modeling the running time of parallel algorithms which use small messages (e.g., the MPI_Barrier algorithm).

As already stated earlier in this chapter, the LogfGP model only introduces f and changes o slightly and all other parameters remain identical. We discussed this change with the easier LogfP model, but G can simply be added to derive the LogfGP model. After we defined the assessment of the f parameters, we will discuss accurate schemes to measure the other parameters below.

2.4 Accurately Parametrizing the LogGP Model

Network models can also be used to modify algorithms in changing environments adaptively. This is important for wide-area networks as they are typically used in grid computing. Another application field is multi-interface message scheduling, i.e., schedule messages across multiple, probably homogeneous, network interfaces to minimize the cumulative transmission time. Those methods need to assess the model parameters during the running time of the application. This makes a low-overhead measurement method for the LogGP parameters necessary (a method that avoids network flooding or saturation).

Another pitfall for the parameter measurement is the fact that most modern communication systems use message-size dependent protocols to optimize communication (e.g., [76, 85, 142]). Small messages are often copied to prepared (i.e., pre-registered in case of InfiniBand$^{\text{TM}}$ cf. [152]) local send or remote receive buffers to speed up the communication. This method is commonly named "eager protocol". Larger messages can not be copied to a buffer on the receiver side (because there may not be enough space), and a local copy would introduce too much overhead. Those messages force a synchronization, and the protocol type is often called "rendezvous protocol". More protocol types can be introduced by the developer of the communication subsystem as needed. The switch between those protocol types is usually transparent to the user, i.e., he does not realize it explicitly. Our method is able to recognize protocol switches automatically since changes in the message transmission times can be detected. We compute different parameter sets for all identified protocol ranges. This section contains and extends results from the article "Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks" [5].

The following section describes related work to assess LogGP parameters and discusses positive and negative effects of the proposed methods.

2.4.1 Existing Approaches and Related Work

Previous works used different strategies and changes of the original model to assess the single parameters as accurately as possible. We discuss the well-known approaches below.

The first measurement method for the LogP model was proposed by Culler et al. [62]. They differentiate between o_s on the sender side and o_r on the receiver side - which complicates the model slightly. To assess o_s, he measured the time to issue a small number (n) of send operations and divided it by n. This could be problematic on modern architectures, because they tend to copy the message to a temporary buffer and send it later (e.g., TCP). This would make the measurement of o_s depend on n and only realistic for large n when all buffers are filled. But this is not possible because a large number of n would effectively measure g. Culler et al. use a delay between messages that is larger than a single round trip time (RTT) to assess o_r, based on the measurement of o_s, which makes the result dependent on the accuracy of o_s and introduces second-order errors. The g parameter is simply benchmarked by flooding the network with many small messages and dividing the time by the number of messages. Finally, L can be computed from the other parameters $L = RTT/2 - o_r - o_s$.

A second, similar approach was used by Ianello et al. in [105]. He uses similar techniques to assess the LogP parameters for Myrinet.

Kielmann et al. used heavy model changes and proposed a solution to assess parameters for his pLogP (parametrized LogP) model in [119]. He uses the time for a single send operation to assess o_s. This could be influenced by caching effects similar to the original idea in [62]. He defines o_r as the time to copy the message from the receive buffer. This clearly neglects the time in which the system is busy to copy the message from temporary buffer (cf. TCP). The g assessment sends n messages to a peer and the peer sends a single message back after it receives n. The time between the first send and the reception of the final answer divided by n is used as g. Those n messages and a single reply message need $(n \cdot g + L) + (L + g)$ in pLogP. An error of $(2L + g)/n$ is made if one simply divides this sum by n. The impact of this error can be reduced if n is large enough so that $(2L + g)/n << g$. If we try to reach 1% accuracy, we need $n > (2L + g)/(g \cdot 0.01)$, which is 19640 for the LogGP parameters gained for TCP (see Section 2.4.3). L is computed from the RTT of a zero-byte message $L = (RTT(0) - 2g(0))/2$. The fact that every parameter depends on the message size slightly complicates the model and the predictions.

The latest work, the only one that assesses all LogGP parameters besides L, was proposed by Bell et al. in [34]. The parameter o_s is measured with a delay between message sends. This delay d is adjusted until $d + o$ fits $g + (s - 1)G$ for a specific message size s exactly. This requires multiple measurement steps to adjust the correct d. Now, o_s is computed via $g + (s - 1)G - d$, which relies on the correctness of g and G. The method to assess o_r is similar to the method used by Culler in [62], but he delays the transmission of the answer on the receiver side. He uses a similar technique as Kielmann to measure g. This method suffers from the same problem that it has to send a huge number of packets (n) to get an accurate measurement - the network is effectively flooded. L can not be measured because modern networks tend to start the message transmission before the CPU is done (L is started before o ends). Bell et al. introduce the end-to-end latency (EEL) which denotes the RTT for a small packet.

All proposed schemes use messages with a fixed size to derive all parameters, which could be inaccurate for some networks that show anomalies at specific message sizes. The second problem with some methods is that the accuracy depends on the number of sent messages, which makes network flooding necessary to achieve good predictions. However, flooding causes unnecessary network contention and should not be used during application runs. We propose a new measurement scheme that avoids flooding as much as possible and delivers accurate parameters. The following section describes the working principle of our new measurement method.

2.4.2 Low Overhead Parameter Measurement

We describe a new low-overhead LogGP parameter assessment method in the following. We implemented our approach as a new "communication pattern" in the extensible open source Netgauge tool [12]. Netgauge is a modular network benchmarking tool that uses high-precision timers to benchmark network times. The difference from other tools like NetPipe [203], coNCePTuaL [162] or the Pallas Micro Benchmarks (PMB) [163] is that the the framework offers the possibility of using MPI as infrastructure to distribute needed protocol or connection information for other low-level APIs (e.g., Sockets, InfiniBand^{TM}, SCI, Myrinet/GM) or to benchmark MPI_Send/MPI_Recv itself. This is important to compare low-level performance with MPI performance and enables the user to assess the quality and overheads of a specific MPI implementation. Our LogGP communication pattern enables a detailed analysis of the introduced software overhead.

Transmission modules for MPI, TCP, UDP, InfiniBand^{TM}, and several other networks are included in Netgauge and enable us to compare the performance of MPI with the underlying low-level API's performance. More low-level modules (e.g., SCI) are under development. We followed the useful hints provided by Gropp et al. [86] to achieve reproducible and accurate benchmark results.

2.4.2.1 General Definitions

Many parallel systems do not have an accurately synchronized clock with a resolution that is high enough to measure network transmissions (in the order of microseconds). This shortcoming forces developers of network benchmarks to do all time measurement on only one machine and measure a round trip time (RTT). Many benchmarks (e.g., NetPipe, PMB) use the so called ping-pong scheme. This scheme uses two hosts, the client that initiates the communication and measures needed RTTs and the server that mirrors all received packets back to the client. This common scheme is depicted in the left part of Figure II.9. Other schemes, like our simplified "ping-ping" (originally mentioned in [163]), depicted in the right part of Figure II.9, can be used to get the performance of multiple consecutive message sends. However, one has to be aware that a ping-ping with many packets is

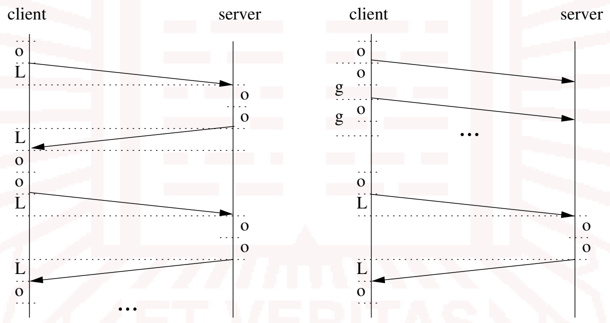


Figure II.9: Left: ping-pong micro-benchmark scheme for 1 byte messages in the LogGP model; Right: ping-ping micro-benchmark scheme for 1 byte messages in the LogGP model.

able to saturate the network and introduce contention easily. An additional possibility to influence the benchmark is a ping-ping scheme with an artificial delay between each message send. The delay can easily be achieved with calculation on the CPU.

We combine all those possibilities and use them to assess all LogGP parameters as unintrusively as possible. We introduce the notion of the parametrized round trip time (from now on *PRTT*) to define a specific parameter combination for the RTT. The possible parameters are the number of ping-ping packets (*n*), the delay between each packet (*d*) and the message size (*s*). A measurement result of a specific combination of *n*, *d* and *s* is denoted as *PRTT(n,d,s)*.

A pseudocode for server and client to measure a single *PRTT(n,d,s)* is given in Listing II.1. The following subsections show that the notion of *PRTT(n,d,s)* is sufficient to assess all LogGP parameters accurately without network flooding or unnecessary contention. The parametrized round trip

```
void server(int n, int s) {
  for(int i=0; i<n; i++)
    /* receive s bytes from client */
    recv(client, s);
  /* send s bytes to client */
  send(client, s);
}

double client(int n, int d, int s) {
  t = -time(); /* get time */
  /* send s bytes to server */
  send(server, s);
  for(int i=0; i<n-1; i++) {
    wait(d); /* wait d microseconds */
    /* send s bytes to server */
    send(server, s);
  }
  /* receive s bytes from server */
  recv(server, s);
  t += time(); /* get time */
  return t;
}
```

Listing II.1: Pseudocode to measure *PRTT(n,d,s)*

time for a single ping-ping message without delay can be expressed in terms of the LogGP model as follows:

$$PRTT(1,0,s) \quad = \quad 2 \cdot (L + 2o + (s-1)G). \qquad \text{(II.5)}$$

If we define the cumulative hardware gap G_{all} as

$$G_{all} \quad = \quad g + (s-1)G\,, \qquad \text{(II.6)}$$

n ping-ping messages can be modeled as (remember that the LogGP model defines $o < G_{all}$)

$$\begin{aligned} PRTT(n,0,s) \quad &= \quad 2 \cdot (L + 2o + (s-1) \cdot G) + \\ &\quad\quad (n-1) \cdot G_{all}\,. \end{aligned} \qquad \text{(II.7)}$$

With (II.5), we get

$$\begin{aligned} PRTT(n,0,s) \quad &= \quad PRTT(1,0,s) + \\ &\quad\quad (n-1) \cdot G_{all}\,. \end{aligned} \qquad \text{(II.8)}$$

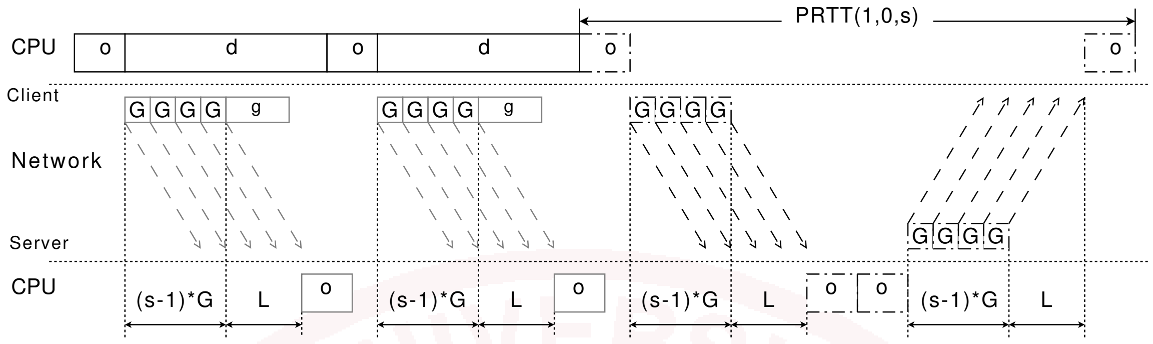


Figure II.10: Measurement Method for o for $n = 3$. This figure shows the components of $PRTT(n, d, s)$.

This equation can easily be extended to the general case with a variable delay d as

$$\begin{aligned} PRTT(n,d,s) &= PRTT(1,0,s) + \\ &\quad (n-1) \cdot max\{o+d, G_{all}\} . \end{aligned} \tag{II.9}$$

The following section uses the PRTT to assess the LogGP parameters of different network interfaces.

2.4.2.2 Assessment of the Overhead o

If we rewrite Equation II.9 to

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

and choose d, such that $d > G_{all}$, we get

$$\frac{PRTT(n,d,s) - PRTT(1,0,s)}{n-1} = o + d . \tag{II.10}$$

This enables us to compute o from the measured $PRTT(n, d, s)$ and $PRTT(1, 0, s)$. We chose $PRTT(1, 0, s)$ for d to ensure that $d > G_{all}$. This assumption has been proved to be valid for all tested networks. However, if a network with a very low latency L and a very high gap g exists, one can fall back to $d = PRTT(2, 0, s)$ to guarantee $d > G_{all}$. We chose $PRTT(1, 0, s)$ to avoid unnecessarily long benchmark times.

The measurement of o for $n = 3$ is illustrated in Figure II.10. The whole figure represents a LogGP model for $PRTT(3, d, s)$, it is easy to see that the last part is a simple $PRTT(1, 0, s)$. If we subtract $PRTT(1, 0, s)$ from $PRTT(3, d, s)$, we get $2d+2o$ which equals to $(n-1)(d+o)$ (remember that $n = 3$ in our example) as shown in Equation (II.10).

This measurement method enables us to get an accurate value of o for each message size s. It needs only a small number of messages (we used $n = 16$ in our tests) that does not saturate or flood the network unnecessarily to measure o. Furthermore, we are able to compute o directly from a single measurement and without inter-dependencies to other LogGP parameters (that are computed themselves and contain already an error of first order). We do also not need to adjust d stepwise to fit other values.

2.4.2.3 Assessment of the Gap Parameters g,G

Using Equation (II.8), we get a linear function in the form $f(s) = G \cdot s + g$:

$$G(s-1) + g = \frac{PRTT(n,0,s) - PRTT(1,0,s)}{n-1} \tag{II.11}$$

One could simply measure $PRTT(n,0,s)$ and $PRTT(1,0,s)$ for two different s and solve the resulting system of linear equations directly. However, several networks have anomalies or a huge deviation between different data sizes. Another problem is that this method would not allow us to detect protocol changes in the lower levels that influence the LogGP parameters.

We chose to measure $PRTT(n,0,s)$ and $PRTT(1,0,s)$ for many different s and fit a linear function to these values. The function value for $s = 1$ is our g and the slope of this function represents our G.

We use the least squares method [38], which can be solved directly for the needed two degrees of freedom (g and G), to perform the fit. This method gives us an accurate tool to assess g and G with multiple different message sizes and to detect protocol/parameter changes in the underlying transport layers (see Section 2.4.2.5). By using this scheme, we also avoid mispredictions and detect anomalies at specific message sizes.

We use only a small n ($n = 16$ for our tests) to benchmark every single message size. Thus, we do not need to flood or overload the network and our results are not influenced by anomalies for specific message sizes (as we experienced with TCP). Furthermore, we are able to use our method to detect changes in the underlying communication protocol, as described in Section 2.4.2.5.

A graphical representation of our method with Open MPI over InfiniBand™ is shown in Figure II.11.

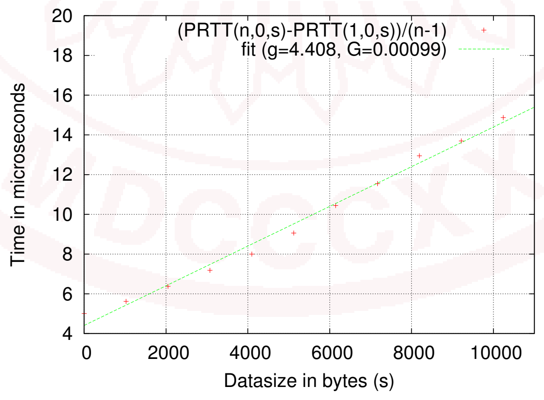


Figure II.11: Parameter Benchmark and Fit for Open MPI over InfiniBand™.

2.4.2.4 Assessment of the Latency Parameter L

Bell et al. discussed the interesting phenomenon that the occurrence of L and o is not ordered. It happens on modern interconnect networks that o and a part of L overlap (some message processing is done after the sending of the message is started). This is due to the fact that the network developers want to minimize the round trip time and try to move all the bookkeeping after the message send. This effect does not allow us to measure a useful L (L may even be negative in certain situations). We take a similar approach as [34] and report half of the round trip time (EEL in [34]) of a small message as latency. We use $PRTT(1, 0, 1)/2$ for this purpose.

2.4.2.5 Detection of Protocol Changes

Modern network APIs are complex systems and try to deliver the highest performance to the user. This requires the use of different transport protocols for different message sizes. It is obvious that each transport protocol has its own unique set of LogGP parameters. The problem is that the network APIs aim to be transparent to the user and often do not indicate protocol switches directly. These facts can make LogGP benchmarks inaccurate if one does not differentiate between the transport modes used. Our approach is to detect those protocol changes automatically and provide a different set of LogGP parameters for each transport type to the user.

We define the mean least squares deviation from measurement point k to l and the fit-function $f(s) = G \cdot s + g$ as

$$lsq(k, l) \quad = \quad \frac{\sum_{i=k}^{l} (G \cdot size(i) + g - val(i))^2}{l - k - 2}\,, \tag{II.12}$$

where $val(i)$ is the measured value at point i and $size(i)$ is the message-size at point i. The subtraction of 2 in the denominator is because we have 2 degrees of freedom for the solution of the least squares problem.

We take an x point look-ahead method and compare the mean least squares deviation of the intervals $[lastchange : current]$ with the deviation of the interval $[lastchange : current + 1]$, $[lastchange : current{+}2]$, ..., $[lastchange : current{+}x]$. We define $lastchange$ as the first point of the actual protocol (the point after the last protocol change, initially 0) and $current$ as our current point to test for a protocol change. If $current$ is the last measured value of a protocol, and a new protocol begins at $current + 1$, the mean least squares deviation rises from this point on. We consider the next x (typically 3-5) points to reduce the effect of single outliers. If $lsq(lastchange, current + j)$ $\forall 1 \leq j \leq x$ is larger than $lsq(lastchange, current) \cdot \mathrm{pfact}$, we assume that a protocol change happened at $current$. The factor pfact determines the sensitivity of this method. Empirical studies unveiled that $\mathrm{pfact} = 2.0$ was a reasonable value for our experiments. However, this factor is highly network dependent and further network-specific tuning may be necessary to detect all protocol changes accurately.

2.4.3 Applying the Method

This section discusses first results of the application of our measurement methods with the tool Netgauge. We analyzed different interconnect technologies and parallel systems to evaluate their performance in the LogGP model. The used test-systems are described in Table II.1. All systems

Transport	**CPU**	**Additional Information**
MPICH2	Opteron 246, 2GHz	MPICH2 1.0.2, BCM5704 GigE
NMPI/SCI	Xeon 2.4GHz	NMPI-1.2, SCI-Adapter PSB66 D33x
Open MPI/OpenIB	Opteron 244	Open MPI 1.1.2, OFED-1.0, MT25208
Open MPI/gm	Athlon MP 1.4GHz	Open MPI 1.1.2, GM 2.0.23, Myrinet 2000
Open MPI/10GigE	Xeon 5160 3.0GHz	Open MPI 1.2.6, Chelsio T3 iWARP, cxgb3_0
Open MPI/ConnectX	Xeon 5160 3.0GHz	Open MPI 1.2.6, ConnectX, mlx4_0

Table II.1: Details about the test systems.

ran the Linux 2.6.9 operating system.

We benchmarked TCP over Gigabit Ethernet and MPICH2 1.0.3, SCI with NMPI 1.2, InfiniBand^{TM} with Open MPI/OpenIB and Myrinet with Open MPI/GM. The graphs for $G \cdot (s-1) + g$ (cf. Equation (II.11)) for InfiniBand^{TM} and Myrinet are shown in Figure II.12.

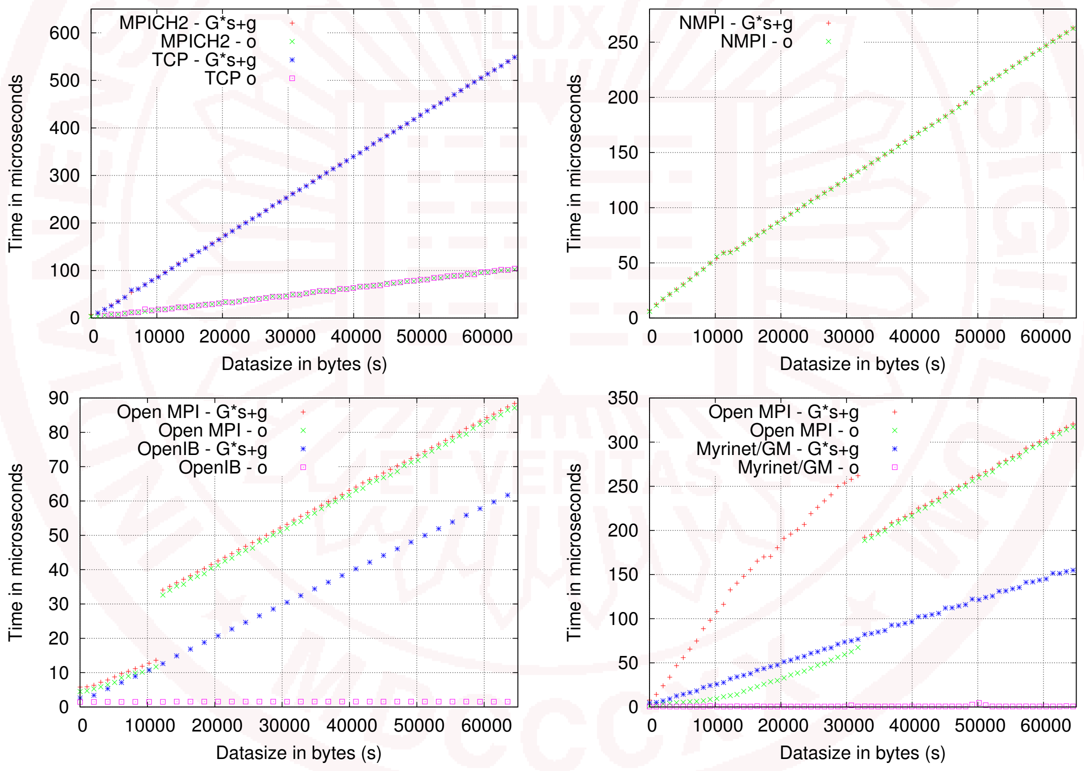


Figure II.12: Measurement results for GigE/TCP, SCI, InfiniBand^{TM} and Myrinet/GM (in this order). The graphs show $f(s) = G \cdot (s-1) + g$, such that the slope indicates G and $f(1) = g$. The parameters of the fitted functions for g and G can be found in Table II.2.

Table II.2 shows the numerical results for the LogGP measurements on the different systems. We used blocking MPI_Send and MPI_Recv to measure those values. The TCP results show that o, g and G are nearly identical for MPICH2 and RAW TCP (they are not distinguishable in the diagram

Transport	**Protocol Interval (bytes)**	**L (μs)**	**o(1) (μs)**	**g (μs)**	**G ($\mu s/byte$)**
MPICH2/TCP	$1 \leq s$	45.74	3.46	0.915	0.00849
NMPI/SCI	$1 \leq s < 12289$	5.48	6.10	7.78	0.0045
	$12289 \leq s$	5.48	6.10	13.34	0.0037
Open MPI/openib	$1 \leq s < 12289$	5.96	4.72	5.14	0.00073
	$12289 \leq s$	5.96	4.72	21.39	0.00103
Open MPI/gm	$1 \leq s < 32769$	10.53	1.27	9.44	0.0092
	$32769 \leq s$	10.53	1.27	52.01	0.0042
Open MPI/ConnectX	$1 \leq s < 12289$	2.98	2.01	2.88	0.0031
	$12289 \leq s$	2.98	2.01	11.60	0.00101
Open MPI/10GigE	$1 \leq s < 12289$	10.97	5.05	5.00	0.0023
	$12289 \leq s$	10.97	5.05	42.00	0.00101

Table II.2: LogGP Parameters for different Transport Protocols

because they lie practically on the same line). We also see that o is not constant as assumed in the LogGP model, but rather has a linear slope. We encounter no protocol change for TCP in the interval $[1, 65536]$ bytes. The SCI results indicate that the implementation uses polling to send and receive messages because $o \approx G_{all}$.

InfiniBand^{TM} also shows an interesting behavior. The Open MPI OpenIB component uses polling to send or receive messages. A protocol change at approximately $12KiB$ leads to a large increase of g. This is due to the rendezvous protocol which introduces an additional RTT of a small status message, which costs $\approx 2L + 4o$ in LogGP, before the actual transmission begins. The blocking MPI_Send charges this to g because it has to wait until a message is sent before it sends the next one. G is nearly identical across all message-sizes. The low-level OpenIB API has a small g and G which exhibits no protocol change. The low-level overhead to post a send request is independent of the message size. Open MPI introduces an additional overhead which is due to the local copy (for eager send) or InfiniBand^{TM} memory registration (cf. [152], for rendezvous).

Myrinet appears to be using no polling for small messages ($o < G_{all}$) and polling for messages larger than $32kiB$ ($o \approx G_{all}$). The protocol change is again clearly visible in the graph and is correctly recognized by our method. The low-level API delivers a slightly lower G and a similar g in the measured interval. The overhead o of the GM API is constant as for InfiniBand^{TM}. We achieve similar results as Pjesivac-Grbovic in [171] (Table 4.13 on Page 60). However, our latencies are higher because we do not subtract o (as discussed by Bell et al. in [34]).

We used the same method to benchmark Mellanox ConnectX (MT_04A0120002) and Chelsio 10 Gigabit Ethernet adapters supporting iWARP [111] (Chelsio T3 iWARP). The iWARP protocol uses the Open Fabrics Verbs interface to access the hardware. This makes it similar to InfiniBand^{TM} and enables the same optimizations. The measured parameters are shown in Table II.2

3 Improving Collective Communication Performance

"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?" – Seymore Cray, (1925-1996) US Engineer

We can draw three main conclusions from the previous sections:

1. Sending multiple messages can benefit the small-message latency substantially.
2. Multicast operations are beneficial when the number of addressed hosts is huge (at least for small messages).
3. A huge potential lies in overlapping computation and communication because the CPU parts in the LogGP parameters are small in comparison to the network portions.

This section uses the first two findings and derives new ways to leverage them for collective operations. Even though there has been a bulk of work that deals with optimizing collective operations for InfiniBand^{TM} [88, 138, 121, 139, 142, 140, 146, 147], we are still able to find two new principles that improve the performance of MPI-standardized collective operations further. Since we derive general optimization principles for networks with similar properties than InfiniBand^{TM} and not InfiniBand^{TM}-specific "hacks", we concentrate on two MPI operations MPI_Barrier and MPI_Bcast. We are also convinced that even though optimized collective operations can benefit applications substantially, the fundamental limits (e.g., logarithmic scaling) are still a limiting factor that leads us to the use of overlapping techniques later in this work. However, the following two optimization principles apply to standard collective operations without any changes to the application.

3.1 A Fast Barrier Implementation for InfiniBand

Our goal is to decrease the latency of the MPI_Barrier operation over the InfiniBand^{TM} network by using the findings from the previous sections. We leverage the implicit parallelism of today's InfiniBand^{TM} adapters to improve the performance of the barrier operation.

3.1.1 Related Work

Several barrier implementations and algorithms have been developed and could be used to perform the MPI_Barrier operation over the InfiniBand^{TM} network. These include the Central Counter approach [73, 82], several tree-based barriers as the Combining Tree Barrier [216], the MCS Barrier [184], the Tournament Barrier [93], the BST Barrier [205] and butterfly or dissemination based barriers [46, 93]. We compared all these different approaches in [22] with the result that the dissemination algorithm is the most promising algorithm for cluster networks. Several studies have also been made to find special barrier solutions, either with additional hardware [190] or with programmable Network Interface Cards [218]. Our approach uses only the InfiniBand^{TM} network which does not offer programmable NIC support or special hardware barrier features. A similar study has been done by Kini et al. in [121] and implemented in MVAPICH. Our study shows that the use of implicit parallelism of the InfiniBand^{TM} network and maybe also other offloading based networks can lower the barrier latency significantly.

3.1.2 The n-way Dissemination Principle

The Dissemination Barrier, proposed by Hengsen et al. in 1988 [93] was proved to be the best barrier solution for single-port LogP compliant systems [21]. The n-way dissemination algorithm is a generalization of the dissemination principle for multi-port networks and can be proved to be optimal for this task. The main change is the additional parameter n which defines the number of communication partners in each round to leverage the hardware parallelism. The n-parameter should refer to the number of messages which can be sent in parallel (cf. LogfP model Section 2.3). The InfiniBand^{TM} network, and probably more offloading based networks do not offer this parallelism explicitly, but because of hardware design, an implicit parallelism is implemented. This means that the n-way dissemination barrier can use this parallelism to speed the barrier operation up. The original algorithm typifies the 1-way Dissemination Barrier ($n = 1$) in this context.

The algorithm is described in the following: Every node p sends n packets to notify n other nodes that it reached its barrier function in each round and waits for the notification of n other nodes. Subsequently, at the beginning of a new round r, node p calculates all its peer nodes (the sendpeer - $speer_i$ and the receive peer - $rpeer_i$, $\{i \in \mathbb{N}; 0 < i \leq n\}$) as follows:

$$speer_i = (p + i \cdot (n+1)^r) \; mod \; P \tag{II.13}$$

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

$$rpeer_i = (p - i \cdot (n+1)^r) \; mod \; P \tag{II.14}$$

For the original algorithm ($n = 1$), the peer calculation gives the same rules as stated in the original paper [93].

$$speer = (p + 2^r) \; mod \; P \tag{II.15}$$

$$rpeer = (p - 2^r) \; mod \; P \tag{II.16}$$

An example for $n = 2$ and $P = 9$ is given in Figure II.13.

A possible pseudo-code for a RDMA based implementation (e.g.,InfiniBand^{TM}) is given in Listing II.2.

3.1.3 Implementation Details

The n-way dissemination barrier is implemented as an Open MPI collective component. It uses the Mellanox Verbs API to communicate directly with the InfiniBand^{TM} hardware. The general Open MPI framework and the component framework are introduced in [76, 193]. The collective framework offers space to implement MPI collective routines. The *ibbarr* component is an optimized MPI_Barrier implementation for the InfiniBand^{TM} architecture using RDMA-W . A caching strategy precomputes the communication partners for each round in advance (during the communicator initialization) to reduce the amount of calculation during the critical MPI_Barrier path.

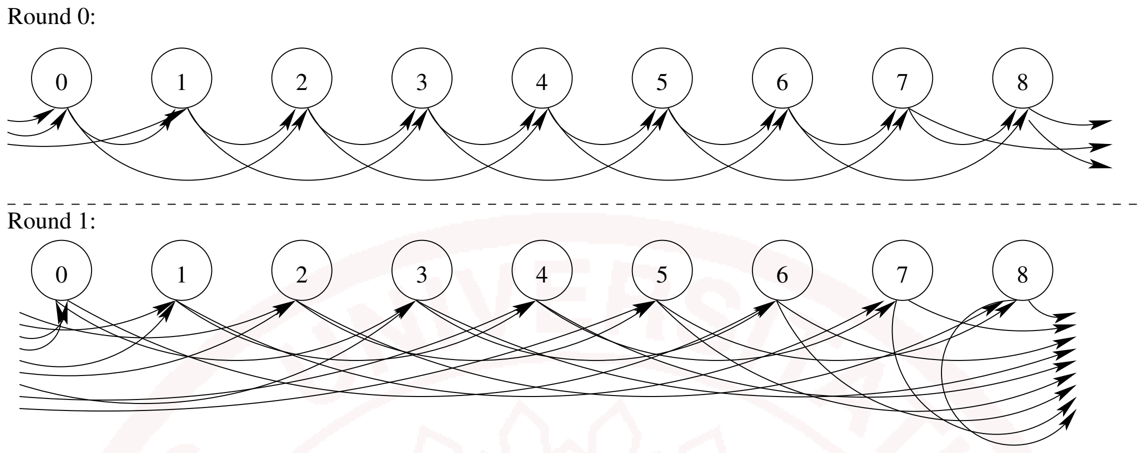
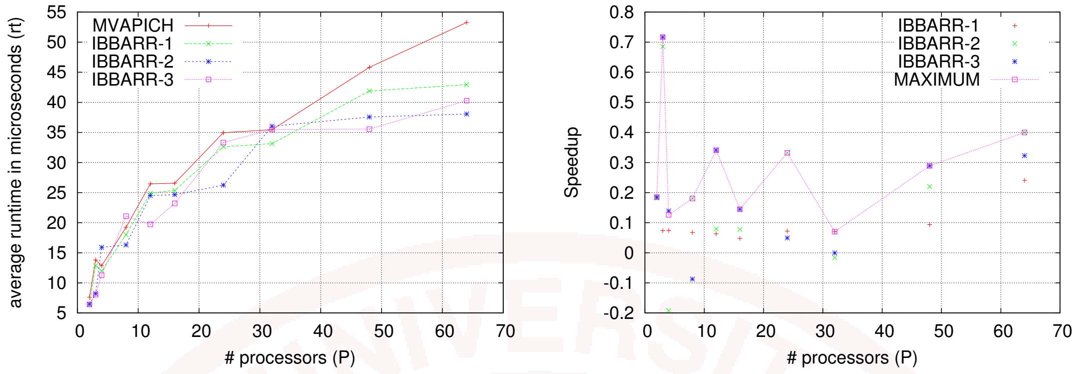


Figure II.13: Example of the 2-way Dissemination Barrier

```
// parameters (given by environment)
set n = 2 // parameter
set P = number of participating processors
set rank = my local id
// phase 1 – initialization (only once)
// the barrier counter – avoid race conditions
set x = 0
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 array[rank] in node sendpeer to x
  forend
  for i in 1..n do
    set recvpeer = (rank – i*(n+1)^round) mod P
    wait until array[recvpeer] >= x
  forend
until round = ceil(log(P)/log(n))
```

Listing II.2: Pseudocode for the n-way Dissemination Barrier



(a) Different MPI_Barrier Latencies (b) Relative Speedup with regards to MVAPICH

Figure II.14: Comparison of different MPI_Barrier Algorithms

3.1.4 Benchmark Results

Different MPI implementations for InfiniBand^{TM} have been taken and compared with regards to their MPI_Barrier latency to evaluate the new approach. The results show that the Open MPI *ibbarr* component is faster than all open-source MPI implementations, even faster than the current leader MVAPICH [138] for which much research has been done to enhance the barrier performance of InfiniBand^{TM} [88, 121]. The 1-way dissemination barrier is already faster than the dissemination barrier of MVAPICH. This can be explained with the precomputed communication partners and the lower latency of the Open MPI framework. The results are shown in Figure II.14 and show that the optimized n-way dissemination algorithm can be up to 40% better than the fastest algorithm, implemented in MVAPICH. The performance gain from the $n > 1$ parameter can be seen between the IBBARR-1 and the IBBARR-n graphs. The gain increases with the node count. However, if the n parameter is too big, the memory congestion effects on the receiver side increase the latency. Thus, the task to deduce the optimal n-parameter for InfiniBand^{TM} is not easy and will be done in a self-tuned fashion (cp. [207]), where different n parameters are benchmarked during communicator initialization and the best one is chosen for the MPI_Barrier.

3.2 A Practically Constant-Time Broadcast for InfiniBand

A key property of many interconnects used in cluster systems is the ability to perform a hardware-supported multicast operation. Ni discusses the advantages of hardware multicast for cluster systems and concludes that it is important for cluster networks [156]. This feature is common for Ethernet-based systems and is supported by the TCP/IP protocol suite. Other widely used high-performance networks like Myrinet or Quadrics use similar approaches to perform multicast operations [78, 210, 219, 217]. The new emerging InfiniBand^{TM} [199] network technology offers such a hardware-supported multicast operation too. Multicast is commonly based on an unreliable datagram transport that broadcasts data to a predefined group of processes in almost constant time, i.e., independent of the number of physical hosts in this group. Multicast groups are typically addressed by network-wide unique multicast addresses in a special address range. The multicast operation can be utilized to implement the MPI_Bcast function, however, there are four main problems:

1. The transport is usually unreliable.
2. There is no guarantee for in-order delivery.
3. The datagram size is limited to the Maximum Transmission Tnit (MTU).
4. Each multicast group has to be network-wide unique (i.e., even for different MPI jobs!).

We examine and resolve all those problems in this section and introduce a fast scheme that ensures reliability and makes the implementation of MPI_Bcast over InfiniBand^{TM} viable.

Node-locally, the underlying communication library is responsible for delivering the received hardware multicast datagrams efficiently to all registered processes. Furthermore, this work addresses mainly cluster systems with flat unrouted InfiniBand^{TM} networks. We assume, and we show with our benchmarks, that the multicast operation finishes in an almost constant time on such networks. However, the ideas are also applicable to huge routed networks, but the hardware multicast might lose its constant-time property on such systems. Anyhow, it is still reasonable to assume that the hardware multicast operation is, even on routed InfiniBand^{TM} networks, faster than equivalent software-initiated point-to-point communication.

3.2.1 Related Work

Some of the already mentioned issues have been addressed by other authors. However, all schemes use some kind of acknowledgment (positive or negative) to ensure reliability. Positive acknowledgments (ACK) lead to "ACK implosion" [139] on large systems. Liu et al. proposed a co-root scheme that aims at reducing the ACK traffic at a single process. This scheme lowers the impact of ACK implosion but does not solve the problem in general (the co-roots act as roots for smaller subgroups). The necessary reliable broadcast to the co-roots introduces a logarithmic running time. This scheme could be used for large messages where the ACK latency is not significant. Other schemes, that use a tree-based ACK, also introduce a logarithmic waiting time at the root process. Negative acknowledgment (NACK) based schemes usually not have this problem because they contact the root process only in case of an error. However, this means that the root has to wait, or at least store the data, until it is guaranteed that all processes have received the data correctly. This waiting time is not easy to determine and usually introduces unnecessary process skew at the root process. Sliding window schemes can help to mitigate the negative influence of the acknowledgment-based algorithms, but they do not solve the related problems.

Multicast group management schemes have been proposed by Mamidala et al. [147] and Yuan et al. [220]. Both approaches do not consider having multiple MPI jobs running concurrently in the same subnet. Different jobs that use the same multicast group receive mismatched packets from each other. Although errors can be prevented by using additional header fields, a negative performance impact is usually inevitable.

Multicast group management should be done with the standardized MADCAP protocol [92]. However, the lack of available implementations induced us to search for a more convenient scheme.

The multicast operation has also been applied to implement other collective operations like

MPI_Barrier, MPI_Allreduce or MPI_Scatter [55, 121, 146]. We use the scheme proposed in [189] for MPI_Bcast and adapt it for the use with the InfiniBand^{TM} multicast technology.

3.2.2 The Multicast-Based Broadcast Algorithm

Several multicast-based broadcast algorithms have been proposed. The most time-critical problem, especially for smaller broadcast messages, is the re-establishment of the reliability which is needed by MPI_Bcast but usually not supported by hardware multicast. We propose a two-stage broadcast algorithm as illustrated in Figure II.15. The unreliable multicast feature of the underlying network technology is used in a first phase to deliver the message to as many MPI processes as possible. The second phase of the algorithm ensures that all MPI processes finally receive the broadcast message in a reliable way, even if the first stage fails partially or completely.

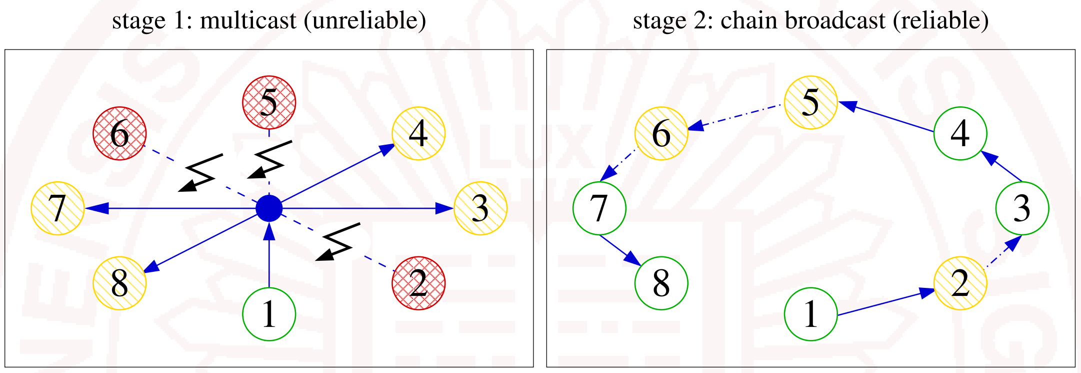

stage 1: multicast (unreliable)
stage 2: chain broadcast (reliable)
1
2
3
4
5
6
7
8

Figure II.15: The two-stage broadcast algorithm

3.2.2.1 Stage 1: Unreliable Broadcast

Multicast datagrams usually get lost when either the corresponding recipient is not ready to receive them or when there is network congestion. Therefore, a common approach is to use a synchronizing operation (similar to MPI_Barrier) that waits until all P processes are prepared to receive the datagrams. If such an operation is built on top of reliable point-to-point communication this synchronization will need $\Omega(log\ P)$ communication rounds to complete. Instead of targeting at a 100% rate of ready-to-receive processes, it is more than sufficient if only a subset of all MPI processes is already prepared, provided that a customized second stage is used for the broadcast algorithm. A further disadvantage of such a complete synchronization operation is the fact that real-world applications are usually subject to process skew which can lead to a further increment of the operation's time consumption.

We conjecture that a wide variety of applications works perfectly without any synchronization operation during this stage. However, if the root process was the first process that calls MPI_Bcast, all non-root processes are not be ready to receive the multicast message and therefore an immediately executed multicast operation might become totally useless. This remaining fraction of applications, with such a worst-case broadcast usage pattern, can be handled by explicitly delaying the root process. A user-controlled delay variable (e.g., MCA parameter for Open MPI) is not only the simplest solution for implementers of this algorithm, but also effective because an optimal value for a given application can be determined using a small number of test runs. Adaptive delay parameter adjustments at run-time, e.g., based on heuristic functions, might be feasible too. A randomized single-process synchronization (instead of a complete MPI_Barrier synchronization) is a third solution to this problem: a seed value is distributed at communicator creation time to all MPI processes. Within each MPI_Bcast operation, a certain non-root process is chosen globally with the help of a pseudo-random number generator and the current seed. The root process than waits until this single non-root process joins this collective operation. On average, such a procedure prevents the worst broadcast scenarios and is thereby independent of the application type. However, the first solution (without any delay) offers naturally the highest performance for applications where the root process rarely arrives too soon.

The first phase of the new broadcast algorithm starts with this optional root-delay and uses multicast to transmit the complete message (fragmenting it if necessary) from the root process to all recipients. A process-local status bitmap can be utilized to keep track of correctly received data fragments.

3.2.2.2 Stage 2: Reliable Broadcast Completion

Even without any preceding synchronization, it is not unusual for a large proportion (typically about 50%) of all MPI processes to have correctly received the broadcast message during the unreliable broadcast stage. The third synchronization method ensures this 50% proportion in the average case even if the application processes always arrive in the worst-case broadcast pattern. This second stage of our new algorithm guarantees that those MPI processes which have not yet received the data (whether partially or completely) will accomplish this eventually. The common approach is to use some kind of acknowledgment scheme to detect which processes have failed and to retransmit the message to these recipients. Unfortunately, existing ACK schemes (positive or negative ones) are quite expensive because of the introduced performance bottleneck at the root process and the necessary time-out values.

Instead of using this kind of "feedback" channel, which can be efficient for large messages where those overheads are negligible, it is more efficient for smaller messages to send the message a second time using a fragmented chain broadcast algorithm. This means that every MPI process has a predefined predecessor and successor in a virtual ring topology. The root process does not need to receive the message because it is the original source of this broadcast. Therefore, the connection with its predecessor (e.g., $8 \rightarrow 1$ in Figure II.15) is redundant and can be omitted. As soon as a process owns a correct fragment of the broadcast message, it sends it in a reliable way to its direct successor. Whether a fragment has been received via multicast or via reliable send is not important - the second receive request can be canceled or ignored.

Using this technique, each MPI process that gets the message via multicast serves as a new "root" within the virtual ring topology. After forwarding this message to its single successor, a process can immediately finalize its broadcast participation. Only those processes that have failed to receive the multicast datagram(s) need to wait until they get the message in the second stage. If its predecessor received the message via multicast then only a single further message transfer operation, called "penalty round" in the following, is necessary. But the predecessors might have failed too in the first stage and the number of "penalty rounds" would increase further. For a given failure probability ϵ of a single message transmission, the chance that P processes fail in a row is ϵ^P. Therefore, the average number of "penalty rounds" is reasonably small (given that $\epsilon = 50\%$, the number of penalty rounds is just 1.0). Nevertheless, the worst-case (i.e., all processes failed to receive the multicast message) leads to a number of "penalty rounds" (and therewith time) that scales linearly with the communicator size. However, real-world applications that call MPI_Bcast multiple times are mainly affected by the average case time and only rarely by this worst-case time.

A different kind of virtual distribution topology (e.g., a tree-based topology) for the second stage could help to reduce this worst-case running time. However, with the knowledge about the applications broadcast usage-pattern or a proper synchronization method, this worst-case scenario will rarely occur. While a process in the virtual ring topology needs to forward the message only to a single successor, a process in a virtual tree-based topology would need to serve several successors (e.g., two in a binary tree) which usually increases the time for the second stage by this fan-out factor. In addition, the broadcast duration per process would not be as balanced as in the proposed chain broadcast. When a single MPI process enters the collective operation late, it can not delay more than one other process in the ring topology but it will delay all its direct successors in a tree-based topology.

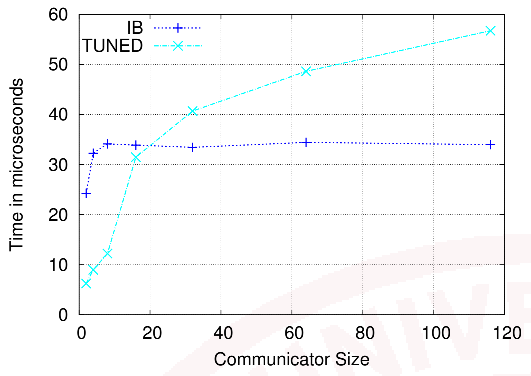
3.2.3 Performance Evaluation

We evaluated our implementation with the Intel Microbenchmark suite version 3.0 (formerly known and introduced as Pallas Microbenchmark [163]) and a second, more realistic, microbenchmark that uses the principles mentioned in [157]. The following results show a comparison of our collective component called "IB" with the existing "TUNED" component in Open MPI 1.2b3. We focus on small messages because their performance is extremely important for the parallel speedup of strong scaling problems (a constant problem size with an increasing number of processes causes small messages) and the new broadcast is especially suited for this use case.

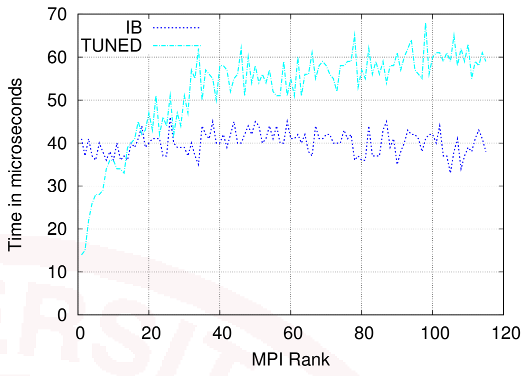
A broadcast along a binomial tree has two main disadvantages. First, the communication time is clearly unbalanced when the communicator size is not a proper power of two. And even if it is, the root process might return immediately from a call to MPI_BCAST when the outgoing messages are cached by the underlying communication system (e.g., in eager buffers), while the last process (e.g., rank #4 in the example Figure III.6(c) in Chapter III) needs $\lceil log_2 P \rceil$ communication rounds to complete the operation (cf. [171]). This introduces an artificial process skew regardless of the initial process skew (the MPI processes might have been completely synchronized). Second, the overall duration of the broadcast operation increases logarithmically with the number of participating MPI processes. On the contrary, our new broadcast algorithm is able to overcome both disadvantages.

3.2.3.1 Benchmark Environment

Our measurements were executed on the *Odin* cluster which is located at Indiana University. This system consists of 128 compute nodes, each equipped with dual 2.0 GHz Opterons 246 and 4 GB



(a) (IMB) MPI_Bcast latency in relation to the communicator size



(b) MPI_Bcast latency for each MPI rank with a communicator size of 116

Figure II.16: MPI_Bcast latencies

RAM. The single Mellanox MT23108 HCA on each node connects to a central switch fabric and is accessed through the MVAPI interface. We used one MPI process per node for all presented runs. Our implementation has also been tested successfully on different smaller systems using the MVAPI and OpenFabrics interface.

3.2.3.2 Results

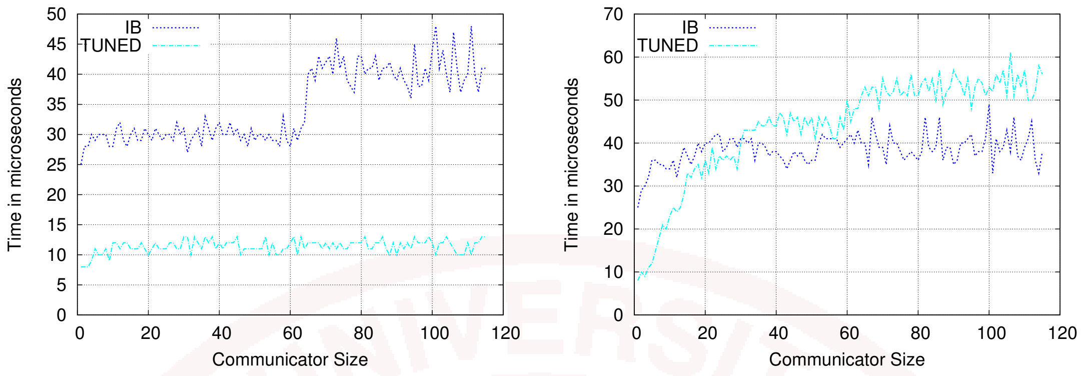
The results gathered with the IMB and a 2 *byte* message are shown in Figure II.16(a). The small-message latency is, as expected, independent of the communicator size[2]. Our implementation outperforms the "TUNED" Open MPI broadcast for communicators larger than 20 processes with the IMB microbenchmark.

For this reason, our collective module calls the "TUNED" component if the communicator contains less than 20 MPI processes (this value is system-dependent and therefore adjustable by the user with an Open MPI MCA parameter to tune for the maximum performance).

Our own (more comprehensive) broadcast benchmark gives a detailed insight into the performance of the new implementation. We measured the time that every single process needs to perform the MPI_Bcast operation with a 2 *byte* message. The result for a fixed communicator size of 116 is shown in Figure II.16(b). It can be seen that the "TUNED" broadcast introduces a significant process skew (rank #1 finishes 79.4% earlier than rank #98), which can have a disastrous impact on applications that rely on synchronicity or make use of different collective operations (that cause different skew patterns). On the contrary, our multicast-based implementation delivers the data to all processes in almost the same time (only a 14% deviation from the median), minimizing the skew between parallel processes. Several (e.g., round-based) applications derive benefit from this characteristic that reduces waiting time in consecutive communication operations.

Figure II.17(a) shows the MPI_Bcast latency of rank 1 for different communicator sizes (the sudden change at 64 nodes has to be attributed to the fact that we had to take the measurements for $1-64$ processes and $64-116$ processes separately due to technical problems). Figure II.17(b)

[2]the IB outlier with two processes exists because the virtual ring topology is split up before the root process (this optimization saves a single send operation at the last process in the chain)



(a) Broadcast latency of rank #1 of a 2 byte message broadcasted from rank #0 for varying communicator sizes

(b) Broadcast latency of rank #N-1 of a 2 byte message broadcasted from rank #0 for varying communicator sizes N

Figure II.17: MPI_Bcast latencies

shows the latency of the MPI_Bcast operation at the last node in the communicator. The increasing running time can be easily seen. With the "TUNED" component, rank #1 leaves the operation after receiving the message from the root process - much earlier than it finishes in our implementation. However, process 1 is the only exception for this component that achieves a constant behavior like in our implementation. Apart from that, the latency to the last rank (like to all other processes) steadily increases with the size of the communicator. Whereas our "IB" component reveals a similar latency for each process, without any noticeable influence of the communicator size.

4 Network Topologies and the Impact of Static Routing

"We can only see a short distance ahead, but we can see plenty there that needs to be done" – Alan Turing, (1912-1954) English Mathematician

Commodity-based clusters with central-switch-based networks have become an established architecture for high-performance computing. The use of a central switch significantly simplifies the communication model for such systems. Compared to other interconnection network topologies, such as tori or rings, the central-switch-based structure has the advantage of being able to efficiently embed structured communication patterns and to support unstructured patterns as well. A network with a true crossbar as its central switch has almost ideal network properties: constant latency between all pairs of endpoints as well as full bisection bandwidth (any half of the endpoints can simultaneously communicate with the other half at full line rate).

However, although they are often treated as if they were true crossbar switches, practical central switches are generally implemented as multistage interconnection network (MINs). As a result, MINs are able to approximate, but not truly provide, the latency and bisection bandwidth characteristics of crossbars. The point-to-point latency in a MIN is not constant for all port combinations (although it is usually the case that the variance is relatively low). Less obvious, but more important to application performance, is the effect of MIN architecture on network bisection bandwidth, particularly as it is seen by applications.

As with other networks, MINs can be characterized by their bisection bandwidth, which, following [95] we define as the total bandwidth between the two halves of the worst-case segmentations of a network. However, this definition of bisection bandwidth only considers the capacity provided by the hardware. It does not consider how the usage of that capacity may be affected by routing policies. That is, unlike with a true crossbar, connections must be routed in a MIN and different communication patterns may require different routing in order to achieve the rated bisection bandwidth for a MIN. If proper routing is not established in the MIN for a given communication pattern, that pattern may not be able to achieve satisfactory performance.

This issue is particularly problematic in networks, such as InfiniBand, that employ static routing schemes because of the potential for mismatch between the static routes and the communication requirements of running applications. Applications can be oblivious to network parameters if the network can provide its rated bisection bandwidth for all communication patterns. Indeed, most applications and communication libraries today are written using this assumption. However, different applications have different communication patterns (and communication patterns may even vary significantly within a single application).

Given the prevalence of MINs in high-performance computing, a more thorough understanding of their characteristics is an important step towards more effective utilization of these resources.

Contributions Therefore, in this section we assess the impact of static routing on the expected bisection bandwidth for arbitrary patterns and for real applications on large-scale production clusters. We address several myths about the definition of bisection bandwidth (and full bisection bandwidth) and introduce the new application-driven definition of *effective bisection bandwidth*. Furthermore, we provide a methodology and tool for cluster and network designers to characterize the theoretical performance of applications running on InfiniBand (and potentially other) MIN networks. More generally, we argue that the scalability of MINs is limited due to practical constraints (routing, hot spots).

4.1 Background

4.1.1 Network Topologies

Different network topologies with different properties have been proposed to be used in parallel computers: trees [47, 133], Benes networks [35], Clos networks [58] and many more — consult [131] for a complete overview. Crossbar switches are often used as basic building blocks today. Crossbar circuits usually implement a symmetric crossbar, i.e., the number of input ports is equal to the number of output ports. Available HPC networks, such as Myrinet, InfiniBand and Quadrics implement crossbars of sizes 32, 24 and 8 respectively. A fully connected network is not scalable because the number of necessary links grows with $O(P^2)$ for P endpoints.

The Clos network The Clos network was designed by Clos in order to build telecommunication networks with switch elements of limited size [58]. It is used today in many InfiniBand and Myrinet switches. The typical Clos network consists of three stages and is described by three parameters. Those parameters n and m and r describe the input and output port count and number of switches in the first layer respectively. The m switches in the middle stage are used to connect input to output ports. Clos networks are "strictly non-blocking" if $m \geq +2n - 1$ and rearrangably non-blocking if $m \geq n$. Most of todays networks are built with $n = m$ which makes those Clos topologies "rearrangably non-blocking" (see [58] for details). They can have the full bisection bandwidth but only for certain combinations of routing and traffic patterns. An easy routing algorithm (up*/down* [183]) can be used to ensure deadlock-free routing and multiple paths between any pair of nodes exist (failover possible).

The (fat) tree network The tree network has a fixed tree topology and has been used to connect the TMC CM-5 and Meiko CS-2 machines and many InfiniBand-based cluster systems, such as the world's largest unclassified InfiniBand system (Thunderbird) at Sandia National Laboratories. The nodes are connected at the leaves and routing is simple (go up until the target node is reachable on a down route). However, this introduces congestion near the root and limits the bisection bandwidth. Leiserson simply increased the link width at every level to avoid this congestion [133]. The resulting "Fat Tree" network can be designed to offer full bisection bandwidth and can be efficiently implemented in an on-chip network. However, it is not possible to construct it easily from fixed-size crossbar elements with fixed link bandwidths. A tree-like network topology that can be constructed with equally sized crossbar switches has been proposed in [132]. Similar to Clos networks, the k-ary n-tree networks [166] retain the theoretical full bisection bandwidth in a rearrangable way.

Practical installations Most modern networks, such as Myrinet, Quadrics and InfiniBand allow arbitrary network layouts but usually use Clos networks or Fat Trees. The main difference lies in the size of the crossbar elements and the routing strategy. Myrinet employs 16 or 32 port crossbar switches. InfiniBand and Quadrics build on 24 and 8 port switch elements respectively. The routing strategy is also important. Myrinet uses source-based routing where every packet can define a different route. InfiniBand uses a static switch based routing scheme and Quadrics uses a non-disclosed adaptive scheme [164].

4.1.2 The InfiniBand Network Architecture

We focus on a practical analysis of deployed InfiniBand networks in this section. However, our results also apply to other statically routed networks with similar topologies. The InfiniBand standard does not define a particular network topology, i.e., switches can be connected in an arbitrary way to each other. The switches have a simple static routing table which is programmed by a central entity, called the subnet manager (SM). The SM uses special packets to explore the network topology in the initialization phase. Then it computes the routing table for every switch. The routing algorithm can be freely chosen by the SM. This initialization usually requires network downtime and is not performed often. Thus, the routing can be seen as static.

4.1.2.1 Hot-spot problems in the InfiniBand Network

Most large-scale InfiniBand networks use the k-ary n-tree topology to connect high node-counts. This topology is able to provide full bisection bandwidth, but is limited to a fixed routing table. This means that, for a given balanced routing, there is at least one pattern that delivers full bi-

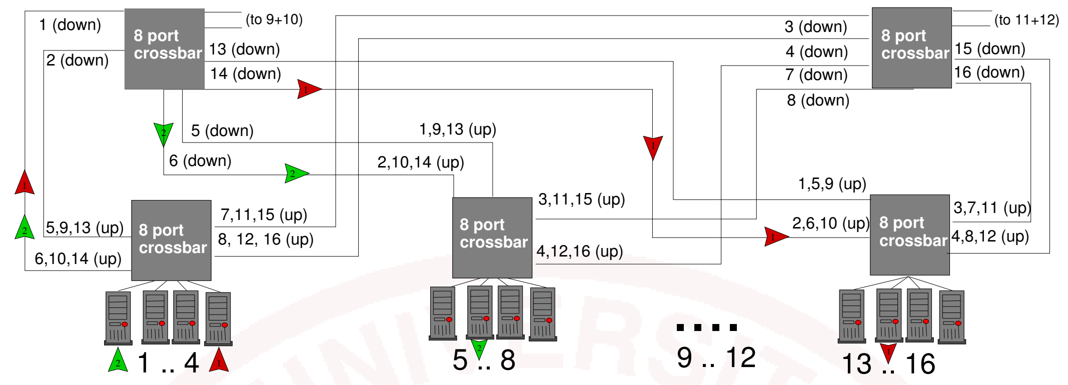
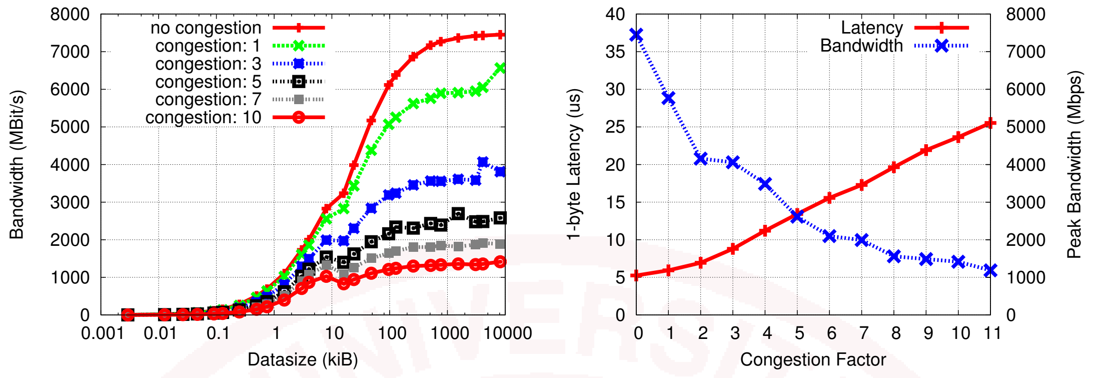


Figure II.18: Statically routed Fat Tree example with 16 nodes and 8 port crossbars

section bandwidth, but all other communication patterns might perform significantly worse. The problem is not the number of available physical links, which is sufficient for any communication pattern, it is the routing which might oversubscribe physical links even though other links remain idle. This problem has been discussed as a source for performance loss in [221, 167]. Zahavi [221] uses specialized routing tables in the switches to avoid hot-spots in the network for a certain communication pattern. This method only works if the processes are carefully placed in the network, the application pattern is known well in advance, and the system can be taken down to re-program the switches before the application run. This scenario is unlikely; applications have to run with the pre-installed routing table that might not support their specific traffic pattern. Most routing algorithms try to distribute routes evenly over the available links, leading to a so called "balanced routing" scheme. We will illustrate the congestion with a small 16 node example network built from 8 port crossbar elements using ideally balanced routing. Figure II.18 shows the network and the fixed routing tables as attributes on the physical connections (up and down routes). Figure II.18 also shows a congestion case where node 4 sends packet 1 to node 14 and node 1 sends packet 2 to node 6. Even though those two node pairs are distinct and the down-route is contention-free, the static routes to the upper-left switch send both packets over a single link which causes congestion.

We call a set of $n/2$ communication partners a communication pattern. Our example network allows us to create some patterns that have full bisection bandwidth, such as (1,5), (2,6), (3,7), (4,8), (9,13), (10,14), (11,15), (12,16). But other patterns, for example (1,5), (2,9), (3,13), (4,6), (7,8), (10,11), (12,14), (15,16) show different characteristics such that every connection has a different bandwidth/oversubscription. The connections (1,5), (2,9) and (3,13) have an oversubscription of 3 and thus only one third of the bandwidth available. Other connections, such as (7,8), (10,11) and (15,16) have the full bandwidth available. The congestion points in the network (in our example the up-link between the lower and upper left switches) are called hot spots [168]. Some techniques have been introduced to avoid those hot spots, such as adaptive source-based routing or multi-path routing [136]. However, those techniques often use simple round-robin schemes to disperse the network traffic, which is limited. Other techniques require a global view of the network [67] or significant changes to the InfiniBand network [148, 179].



(a) Bandwidth (b) Latency

Figure II.19: InfiniBand bandwidth and Latency in different Congestion Situations

4.1.2.2 Measuring the Effects of Fabric Congestion

To analyze the impact of the hot-spot and congestion, we performed several benchmarks on the CHiC cluster, a 528 node InfiniBand system offering full bisection bandwidth. The network topology of this system is a Fat Tree network built from 44 24 port leaf switches (crossbar) and two 288 port top switches (internal Clos network). We queried the routing tables and topologies of the switches with the tools `ibnetdiscover` and `ibdiagnet` and chose pairs of communicating peers that cause congestion in the Fat Tree such that every node is in exactly one pair (no endpoint congestion). We benchmarked the point-to-point latency and bandwidth between a pair of nodes while adding more congestion (we did this by adding sending 8MB ping-pong MPI messages between the other node pairs). To achieve comparable results, we used the well-known Netpipe [204] MPI benchmark with Open MPI 1.2.5 and OFED 1.3 to perform those measurements.

The results in Figure II.19(a) show the transmission curves for different numbers of pairs causing congestion even though all communications have been done among independent pairs of nodes. This shows clearly that congestion and routing-based hot-spots in the fabric can have a significant impact on the communication performance. However, from a user perspective it is not trivial to know if congestion occurs because the fabric looks homogeneous.

Figure II.19(b) shows the latency for all possible hot spot congestions for the CHiC network. The congestion might vary from 0 (no congestion) to 11 (maximum congestion) because every crossbar has 12 down- and 12 up-links. Thus, a maximum of 12 nodes can be connected to a single crossbar. We see a nearly linear increase in 0-byte transmission latency and a significant reduction of the available link bandwidth.

4.1.2.3 InfiniBand's Lid Mask Control

InfiniBand's Lid Mask Control (LMC) mechanism has been discussed as a solution to the hot-spot problem. It assigns multiple LIDs (InfiniBand's endpoint identifiers) to hosts and thus enables multiple routes for every pair of peers. However, the ordering constraints of the InfiniBand network prevent dispersion at the connection (queue pair) level. Thus only whole messages can be scheduled to different routes. Simple round-robin schemes have been shown to improve the worst-case network performance slightly in [213]. This "blind" way of dispersing routes does not provide near-optimal performance but finds some average performance and has not been analyzed well. It is thus unclear how much the difference to optimal routing is. Another problem with multi-path routing is that the switches need to store and evaluate the potentially high number of routes for every pair of endpoints. The worst problem however, is the growing number of endpoints (queue pairs) per process (especially on SMP or multicore systems) which is not scalable to higher node counts. Much work has been invested to limit the number of queues [185, 196] or even use alternative communication methods such as Unreliable Datagram [74, 125]. Thus, LMC-based mechanisms can be seen as counter-productive at large scale.

We focus on the analysis of single-path routed InfiniBand systems to analyze the impact of the static routing.

4.2 Hot Spot Analysis

While we have demonstrated that hot-spot problems exist in current InfiniBand networks, it is still not known how big the negative influence on real applications is. Our first step is to check the assumption that every communication pattern can be embedded in the network efficiently. Thus, we have to assume the hardest class of parallel applications with randomly changing arbitrary communication patterns .

To examine bisection bandwidth under this assumption, we have to split P nodes into two equally sized partitions A and B. There are $\binom{P}{\frac{P}{2}}$ possibilities to select sets of size $\frac{P}{2}$ as A. For each selection of A, we sort the remaining $P/2$ nodes into B. Since the sets A and B can be exchanged and still represent the same partitioning, half of the possibilities are identical (can be derived by exchanging A and B). Thus, we have $\frac{1}{2} \cdot \binom{P}{\frac{P}{2}}$ possibilities to partition P nodes into two equally sized partitions. Furthermore we have to decide which node of partition A communicates to which node from partition B. The first node in partition A has $\frac{P}{2}$ possible partners in partition B, the second has $\frac{P}{2} - 1$ left and so on. This yields to a total of $\frac{P}{2}!$ pairing schemes.

If we combine all different possibilities to split the P nodes into two groups and all different pairings between the two groups, we get

$$\binom{P}{\frac{P}{2}} \cdot \frac{(P/2)!}{2} = \frac{P! \cdot (P/2)!}{2 \cdot (P/2)!^2} = \frac{P!}{2 \cdot (P/2)!} = \frac{1}{2} \prod_{i=\frac{P}{2}}^{P} i$$

possible communication patterns; which is already $259,459,200$ in our simple 16 port network. Only a small fraction of those patterns has full bisection bandwidth. Due to the huge number of possible patterns, there is no possibility of benchmarking reasonably-sized networks exhaustively. Our approach is to simulate the network contention of a huge (statistically significant) number of communication patterns because this is significantly faster and easier than benchmarking.

4.2.1 Choosing a Network Model

Our congestion model is based on the measurement results of Section 4.1.2.2. We model the network as a topology of fully connected, contention-free crossbar switches with limited port count interconnected by physical wires with bandwidth/capacity γ. Each crossbar switch has a static routing table that defines an output port to every node in the network. If we assume a specific communication pattern, we assign a usage count π to every physical connection that indicates how many connections are routed over this link. This usage count models the congestion factor in 4.1.2.2. The throughput per cable is thus defined as $\frac{\gamma}{\pi}$. In the following, we only discuss relative bandwidths and thus set $\gamma = 1$.

This simple model can be used to derive parameters for more complex models. The LogGP [29] model for example can be parametrized for a point-to-point connection while G is multiplied with π. The latency can be modeled as a linear function $L(\pi)$. The parameters o and g are independent of the congestion.

4.2.2 The Network Simulator

The design goal of our simulator is to simulate real existing InfiniBand systems (topologies + routing tables) to investigate contention effects on applications running on those systems. The simulator thus accepts a network structure (queried from the running system with `ibnetdiscover` and `ibdiagnet`, represented as a directed acyclic graph) and a specific communication pattern (represented by $P/2$ communication pairs) as inputs. The output is the maximum usage count σ along each of the $P/2$ routes (each route might use multiple physical cables but the maximum congested cable mandates the transmission speed and thus the route's overall congestion). For example, the pattern (1,5), (2,9), (3,13), (4,6), (7,8), (10,11), (12,14), (15,16) would lead to the congestions 3, 3, 3, 1, 1, 1, 1, 1 respectively. The simulator assumes full duplex communication, i.e., messages using the same link in different directions do not cause congestion.

4.2.2.1 Related Work

Several publications, such as [41, 67, 136, 148, 166, 167], rely on network simulations. The ability to read arbitrary communication patterns and real-world topologies (not limited to Fat Tree or Clos) distinguishes our work from all earlier network simulations that used predefined patterns. Those predefined patterns do often not reflect the communication patterns used in high performance applications. Patterns are:

- "uniform traffic" [67, 136] which might cause congestion at the endpoints (destinations are chosen uniformly, i.e., two or more nodes might send to the same destination).
- "complement traffic" [166] where node i sends to the node with the number that equals the bit-wise (one-) complement of i. This pattern reflects a possible bisection of the network.
- "bit reversal" and "transpose" [166] are patterns that reflect all-to-all like communications which are used for some problems.
- "hot-spot traffic" [136, 167] is traffic where some percentage of the traffic is targeted at single nodes, so called hot-spots. Hot-spots should be avoided in high performance computing, thus, this pattern does not reflect typical applications.
- "localized traffic" [41] might reflect nearest neighbor communication schemes in real-world applications but the definition is vague.

Another problem with former simulations is that for each pattern, usually only the average packet latency and bandwidth are reported. However, the average values might have little meaning. Some applications can accept a wide variety of bandwidths in the communications, others can not. Fine-grained applications running tightly synchronized with collective communication (lock-step) are usually only as fast as the slowest link. In application modeling, the average values might mislead to assume uniform bandwidth on every link and thus misinterpret those simulations. A more detailed application model is presented in Section 4.3.

4.2.2.2 Simulated Cluster Systems

Throughout this section, we use four of the biggest InfiniBand cluster installations available to perform our simulations. The "CHiC" at the University of Technology Chemnitz has 528 nodes connected to 44 24-port leaf switches which are connected to two 288 port switches in the second level. The CHiC network is able to deliver full bisection bandwidth. The second larger input system is the "Atlas" system located at the Lawrence Livermore National Lab has 1142 nodes and a Fat Tree network with full bisection bandwidth. The "Ranger" system at the TACC uses two Sun "Magnum" 3456 port switches to connect 3936 nodes with full bisection bandwidth. The largest simulated system, the "Thunderbird" (short: Tbird) cluster, is also the largest InfiniBand installation (with the biggest number of endpoints) and its network has 4391 InfiniBand endpoints arranged in a Fat Tree network with $1/2$ bisection bandwidth.

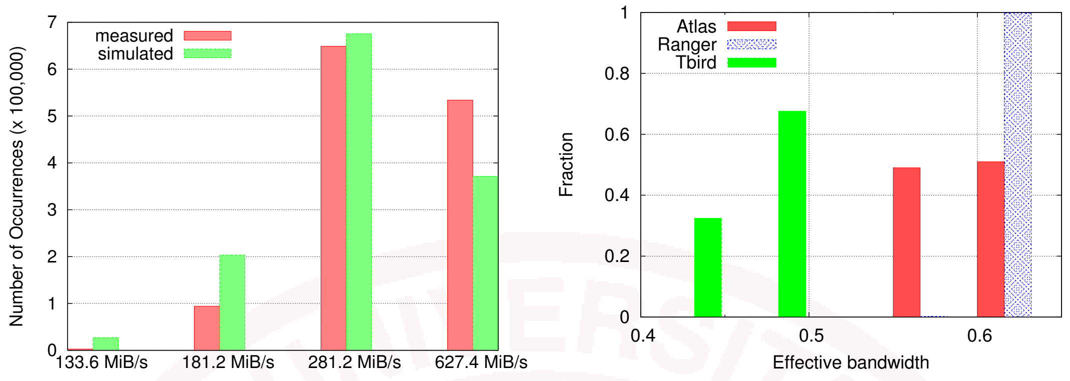
4.2.2.3 Simulator Verification

To verify our simulation results, we implemented a benchmark that measures randomly changing "bisect" communication patterns and records the achieved bandwidths on every connection into several bandwidth classes. A "bisect" communication pattern is created as follows:

- split the network of size P into two equally sized groups A and B
- create $P/2$ pairs such that every pair consists of a node from A and B
- guarantee that no node is in more than a single pair (has more than one partner in the other group)

Our benchmark generates a random "bisect" pattern at process 0, scatters it to all P processes and synchronizes the time on all nodes with a tree-based algorithm as described in [15]. Process 0 broadcasts a starting time to all processes. All processes start simultaneously and benchmark the time needed to send 100 fixed-size packets between the pairs. Process 0 gathers all $P/2$ timing results and scatters a new random "bisect" pattern. This procedure is repeated for 5000 random bisect patterns. We use the Mersenne Twister [150] algorithm to generate the random patterns. The root node records all $5000 \cdot P/2$ point-to-point bandwidth results and sorts them into 50 equally-sized bins.

The benchmark results of the full CHiC system are shown in Figure II.20(a). This benchmark, using the full system, showed clean results with only 4 of the 50 possible bins filled. The measured link bandwidth with MPI between two nodes in the CHiC system is $\gamma = 630MiB/s$ for $1MiB$ messages. Our simple model $\frac{\gamma}{\pi}$ would predict $315MiB/s$, $210MiB/s$, $157.5MiB/s$ for a congestion π of 2, 3 and 4 respectively. The benchmark results are slightly lower than that but still reflect our expectations. Runs with fewer nodes also supported our thesis; however, the results were scattered across many bins due to background communication during those jobs.



(a) Simulation and Benchmark Results for a 512 node bisect Pattern with 1MiB messages in the CHiC system (b) Bisection pattern bandwidth results for all simulated systems

Figure II.20: "bisect" Simulation Results for different InfiniBand Systems

These experiments show that our simulator design accurately reflects a real-world environment. It is usually not easily possible to perform measurements at full scale, thus we will show the effects of the network contention to applications by simulating the application traffic patterns. This will give us some estimation of how real applications are influenced by the network topology.

4.2.2.4 Simulating the Effective Bisection Bandwidth

To get an estimation of how much bandwidth can be expected from a random "bisect" pattern, we ran N ($N = 10^6$ for our simulation) simulations with different patterns for our three systems. Each pattern simulation resulted in $P/2$ oversubscription factors π (one per channel).

Many applications are round-based and every round exhibits a potentially different communication pattern. To model this behavior, we chose to evaluate every pattern as an entity by plotting the pattern-specific average bandwidths in a histogram. Thus, we compute the pattern-specific average oversubscriptions as $\left(\sum_{i=1}^{P/2} \pi_i\right) \cdot \frac{2}{P}$ and sorted these N results into bins. The normalized height of any histogram bin shows the fraction of mappings which showed the specific effective bandwidth. Figure II.20(b) shows the histograms of the bandwidth-distribution. The achieved average bandwidths are interestingly stable (nearly all patterns exhibit a similar average bandwidth) and only two bins are (nearly equally) filled for all simulated systems.

Based on the observation that all *bisect* patterns seem to have a pretty stable average bandwidth, we define the application-oriented network parameter *effective bisection bandwidth* as the average bandwidth for an arbitrary communication pattern. This most generic definition reflects applications with non-predictable and irregular communication patterns such as parallel graph algorithms as a single number. This parameter is unfortunately not easy to measure for a given network. It might be assessed by an exhaustive search or in combination with statistical methods.

The simulated *effective bisection bandwidths* are 57.6%, 55.6% and 40.6% of the full bandwidth for Ranger, Atlas and Tbird, respectively. An interesting observation is that the large Tbird system with half bisection bandwidth does not deliver a significantly worse *effective bisection bandwidth* than the Atlas system with full bisection bandwidth.

However, this analysis still does not reflect many real-world applications well. The next section explains an analysis of 5 real world application codes and simulates their communication patterns.

4.3 Parallel Applications Communication

To understand how the network affects applications, we analyze four large open-source applications for the main sources of communication overhead. Later, we will use those results to derive communication patterns as input for the simulation.

Most parallel high-performance applications are written with the Message Passing Interface standard (MPI). Thus, we used the MPI profiling interface to record application communications and measure their running time as a share of the application's running time. We analyzed point-to-point collective communication by representing the neighborhood relations in a directed graph. Collective communication calls can not be recorded that easily because the used pattern depends on the implementation. Thus, we just recorded the communicator size on which the collective operations were run. All runs were done on 64 processes with InfiniBand as the interconnection network.

Massively Parallel Quantum Chemistry Program The Massively Parallel Quantum Chemistry (MPQC) Program [110] is an open-source implementation that solves the Schrödinger equation to compute several properties of atoms and molecules. The MPQC run took 852 seconds and had 9.2% communication overhead. We were able to identify three collective routines that caused nearly all communication overhead: MPI_Reduce (67.4%), MPI_Bcast (19.6%) and MPI_Allreduce (11.9%). All routines used the full set of processes (64) as communication group.

MIMD Lattice Computation The MIMD Lattice Computation (MILC) code is used to study quantum chromodynamics, the theory of the strong interactions of subatomic physics as used in high energy and nuclear physics. We benchmarked a 9.4% communication overhead running the MILC code for 10884 seconds. More then 86% of the overhead was caused by point-to-point communication (MPI_Isend/MPI_recv/MPI_Wait) and 3.2% by MPI_Allreduce in the complete process group. We analyzed the send/receive pattern and found that every process communicates with exactly 6 other processes (neighbors).

Parallel Ocean Program The Parallel Ocean Program (POP) is an ocean circulation model. It is the ocean component of the Community Climate System Model and has been used extensively in ocean-only mode for eddy-resolving simulations of the global ocean. We measured 32.6% communication overhead for a 2294-second run. About 84% of this overhead are due to point-to-point communications and 14.1% are caused by a global MPI_Allreduce. Every process uses the point-to-point operations to communicate with 4, 5 or 6 neighbors and rank 0.

Octopus The last analyzed application, Octopus, is a part of the Time-dependent density functional theory (TDDFT) package which solved the time-dependent Schrödinger equation in real-space. The application ran on 64 nodes for 258 seconds and a communication overhead of 10.5% was measured. Most of this time was spent in MPI_Allreduce (61.9%) and MPI_Alltoallv (21.9%) on all processors.

Application Conclusions and Summary The five analyzed applications spend most of their communication time in regular neighbor or collective communications. Used collective communications are reductions, broadcasts, reductions-to-all and all-to-all and are usually performed with all processes in the communication context (communicator). We also identified point-to-point patterns with 4 to 6 neighbors. Thus, we conclude that we can express the network patterns of many real-world applications that exist today by simulating collective communication and nearest neighbor point-to-point patterns. The following section describes common patterns for the implementation of collective communications based on point-to-point messages.

4.4 Application Communication Simulation

A common way to implement collective operations is to use algorithms based on point-to-point communication. Multiple algorithms exist and are used in different scenarios. A general rule for algorithm selection is that small-message all-to-one or one-to-all operations (e.g., broadcast or reductions) use tree-like communication schemes and large versions of those operations use double-trees or pipelined (ring) communication schemes. All-to-all communication schemes (e.g., reduce-to-all or personalized all-to-all) usually implement the dissemination algorithm [93] or also a pipelined ring scheme. A more detailed analysis of algorithm selection can be found in [171].

4.4.1 Simulating Collective Patterns

To examine the effect of fabric congestion on applications we simulate different collective traffic patterns and record the oversubscription π per route. Most optimized collective communication patterns consist of multiple communication stages r (a.k.a. "rounds", e.g., the dissemination algorithm uses $\lceil log_2 P \rceil$ rounds). Figure III.8(a) in Chapter III shows the communication pattern of the dissemination and tree algorithm for 7 processes as an example. Every of those rounds reflects a different communication pattern that is performed on the network.

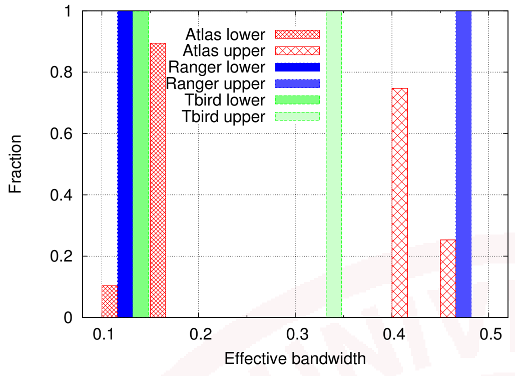
We extended the pattern generator to also generate collective communication patterns such as dissemination, pairwise exchange, tree, pipeline, ring, scatter and gather patterns. Those patterns usually have multiple communication rounds with a specific pattern for each round.

Our simulator accepts a pattern and generates a random mapping from each rank in the input pattern to an endpoint[3] in the network. Then it simulates all communication rounds for this mapping. However, the merging of the $r \cdot P/2$ results for each multi-round pattern is not trivial because each link might have a different oversubscription factor π. Thus, we apply two strategies that determine an upper and lower bound to the communication bandwidth:
1) we use the maximum congestion factor π for every round and sum it up to the final result.

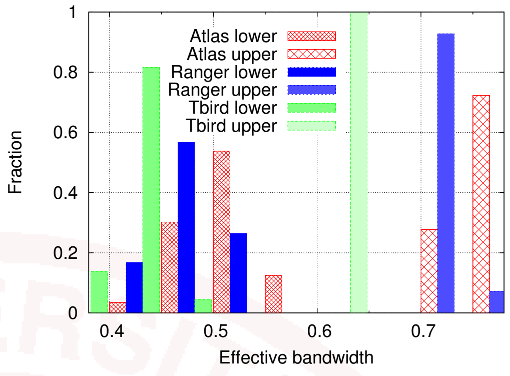
$$\pi_{sum\_max} = \frac{1}{r} \cdot \sum_{i=1}^{r} \max_k(\pi_{i,k})$$

This represents the most pessimistic case where all processes have to synchronize (wait for the slowest link) at the end of each round.

[3]we focus on networking effects in this chapter and thus we only simulate the single process per node case



(a) Dissemination



(b) Tree

Figure II.21: Collective Pattern Simulation Results for different Cluster Systems

2) we use the average congestion factor of each round and sum them up to the final result.

$$\pi_{sum\_avg} = \frac{1}{r} \cdot \sum_{i=1}^{r} \frac{2}{P} \cdot \sum_{k=1}^{P/2} \pi_{i,k}$$

This represents, similar to our definition of the *effective bisection bandwidth*, the optimistic case where no synchronization overhead occurs and every process can proceed to the next round without waiting for other processes.

The results of N different rank-to-node mappings are then combined into equally sized bins and plotted as histograms (we used $N = 10^5$ in our simulations). The normalized height of any histogram bin shows the fraction of mappings which showed the specific effective bandwidth.

The dissemination, ring and recursive doubling simulation results are rather similar and we present only the dissemination pattern in Figure II.21(a). The dissemination pattern uses $\lceil log_2 P \rceil$ communication rounds to perform the operation as shown in Figure III.8(a). The lower bound in the histogram shows that the average of the minimum bandwidths of all rounds (strategy 1) is as small as 10-15%. The upper bound shows the bandwidths with around 40-50% in the optimistic estimation (strategy 2). The “effective bandwidth” (the average of the “upper” simulations) for random rank-to-node mappings of the dissemination pattern is 41.9%, 40.2% and 27.4% for the Ranger, Atlas and Tbird systems respectively.

The tree pattern, depicted in Figure II.21(b), shows the best results because it does not leverage the network fully (each one of the $\lceil log_2 P \rceil$ rounds doubles the number of communicating peers beginning from 1 while all peers communicate from round 1 in most other patterns, cf. Figure III.8(a) in Page 76). The “effective bandwidths” of the tree pattern for the CHiC, Atlas and Tbird system were 69.9%, 71.3% and 57.4% respectively.

The nearest-neighbor communication simulation results — assuming 6 simultaneous neighbor communications — are shown in Figure II.22. This limits the bandwidth due to congestion at the endpoints to $1/6$. Thus, we scaled the results by a factor of 6 to isolate the effect of fabric congestion. We see a huge gap between the tightly synchronized communication (lower) and the optimistic

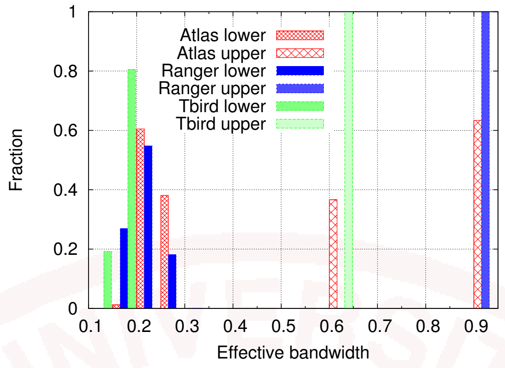


Figure II.22: Nearest neighbor communication with 6 neighbors

measure (upper). The "effective bandwidths" are about 62.4%, 60.7% and 37.4% for the Ranger, Atlas and Tbird system respectively.

4.4.2 Influence on Applications

With our simulation results and with several simplifying assumptions, we can make some approximate statements about the performance penalty due to the routing effects of the four analyzed applications. We assume that the applications are written in a scalable way such that a weak scaling problem keeps the communication overhead approximately constant at large scale and that collective algorithms like reductions, broadcast and alltoall operations are implemented in a tree-based scheme to favor small messages. We use our simulation results to extrapolate the measured communication overhead to the full effective bisection bandwidth case. Then, we calculate the difference in running time of each application. For example POP spent 27.4% of its execution time in neighbor communications with up to 6 neighbors and 4.6% in an allreduce operation. The average bandwidths for those communication patterns on the largest simulated system (Tbird) are about 37.4% and 57.4%. This means that this communication for those patterns would be 2.6 and 1.74 times faster with real full bisection bandwidth. This would decrease a hypothetical application running time of 100s to only 80.89s, i.e., would reduce the application running time to 80.89% of the original time. The following table shows an estimation of the possible application improvements on the three simulated systems (using the whole network with one process per node) assuming they could leverage full *effective bisection bandwidth*:

Application	Overhead	Ranger	Atlas	Tbird
MPQC	9.2%	97.23%	97.36%	96.08%
MIMD	9.4%	96.87%	97.73%	94.81%
POP	32.6%	88.32%	87.91%	80.89%
Octopus	10.5%	97.18%	97.23%	95.79%

The rated bandwidth of the Tbird system is only $1/2$ bisection bandwidth. However, we would still argue that the nearest neighbor and tree-based communication can be efficiently embedded into the network (assuming a good node-to-host mapping) such that the application effectively has (for this pattern/mapping) full bandwidth. Thus, we compare in the results in the table to full bandwidth. However, if we assume an arbitrary mapping (i.e., that the rated bandwidth is a strict limit), then the gain in application performance for all application on the Tbird system halves.

We can conclude that the effect on application performance is significant and that the usage of hot-spot avoiding routing schemes would be beneficial for real-world applications.

In the next section, we extend this work by analysing the theoretical design-space of Fat Tree networks.

5 Searching for the Ideal Fat Tree Network Design

"The whole is more than the sum of the parts" – Aristotle, (384-322 BC) Greek Philosopher

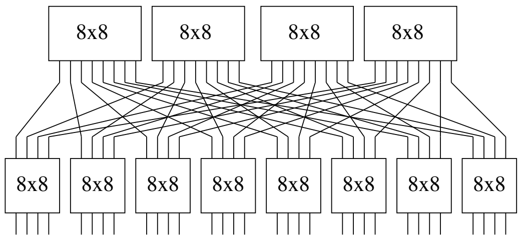
We have seen in the previous section that rearrangable non-blocking Clos networks, as they are used today in most HPC systems, are not able to deliver full bandwidth for all communication patterns. We introduced the *effective bisection bandwidth* as a more accurate measure than bisection bandwidth of real-world applications. Simulation results for the effective bisection bandwidth show that even though one network has half of the bisection bandwidth of another, the *effective bisection bandwidth* was significantly more than a half. That leads us to the conclusion that cost-effective networks with relatively low bisection bandwidths could achieve a similar effective bandwidth as more expensive topologies.

To explore this new network design space, we begin with the design and simulation of different rearrangable non-blocking Clos networks and fat trees. We chose this setup, rather than a fully non-blocking Clos networks or other topologies, because it represents common practice used to design and build many HPC networks today. Thus, we develop a network creation tool that generates recursive Clos networks from an abstract definition (such as recursion depth and crossbar-size). This tool consists of two steps. Step one generates a physical topology description and step two adds the distributed routing tables to the physical topology by analyzing the topology. The result is an abstract network definition (a directed graph in the dot-language) that can be used as an input for the network simulator described in Section 4.

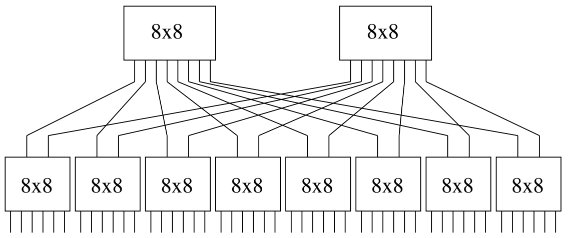
5.1 Generating Recursive Fat Tree Networks

In this section, we discuss the generation of recursive Fat Tree networks with different bisection bandwidth characteristics. The bisection bandwidth is defined as the minimal number of cables between any two equally-sized partitions of a network multiplied by the capacity of a link. One can define an oversubscription factor for each network that indicates what share of the full bandwidth is achieved. An oversubscription of 1:1 means full bisection bandwidth, 1:2 half, 1:3 one third and 5:11 means 0.455 of the bisection bandwidth respectively. Figure II.23(a) shows a Fat Tree network with 32 ports built from 8 port crossbar switches with full bisection bandwidth (1:1). Figure II.23(b) shows a 48 port network with similar base-switches with $\frac{1}{3}$ bisection bandwidth (1:3).

We can define a cost-assessment with the number of switches τ and the number of backplane-connections ν per port. The number of switches per port is $\tau_{1:1} = \frac{12}{32} = 0.375$ and $\tau_{1:3} = \frac{10}{48} = 0.208$ in the 1:1 and 1:3 examples respectively. The number of connections per port is $\nu_{1:1} = \frac{32}{32} = 1$ and



(a) 32 Port 1:1 Fat Tree (Clos) Network



(b) 48 Port 1:3 Fat Tree Network

Figure II.23: Different Fat Tree Network Layouts with 8 port switches

$\nu_{1:3} = \frac{16}{48} = \frac{1}{3}$. It can be shown that ν equals the oversubscription-factor for non-recursive Fat Tree networks.

Now, we can begin to construct different Fat Tree networks with this strategy, but we'll face the major limitation that the number of ports is defined by the oversubscription factor. For example, one can not build a 1:1 Clos network with 16 ports from 8 port crossbars. This was addressed by Leiserson in [134] who generalized Clos networks to Fat Tree networks that enable more flexibility with port-counts. For example, a 16-port 1:1 Fat Tree can be constructed from six 8-port crossbars (2 in the top, 4 leaf switches), i.e., by "halving" the network shown in Figure II.23(a). However, this works only to decrease the port count for a given crossbar-size and oversubscription factor. A new strategy has to be found in order to increase the port count.

Recursive, or multi-stage Fat Tree networks are a natural extension to increase the port count with a given crossbar size. For example, to build a 1:1 512-port network from 8-port crossbars, one can use 48 1:1 32-port Clos networks (16 in the top, 32 leaf switches - cf. Figure II.23(a)). To build a 1:3 528-port Fat Tree from 8-port crossbars, one can use 33 1:3 48-port Fat Tree networks (11 in the top, 22 leaf switches, combined in a 1:1 way). One could also use 32 port crossbars and wire them up respectively. In this dissertation, we use only 1:x base switches wired up in an 1:1 manner to build 1:x Fat Tree networks. The huge design-space of other topologies will be explored in future works.

We analyze different network sizes with different oversubscription factors and crossbar-sizes. We generate fat-tree networks with approximately 500, 3000, 7000 and 10000 ports from crossbars (CB) of size 8, 16, 24 and 32. We analyze oversubscription of 1:1, 1:2 and 1:3. However, the port-counts and oversubscription factors are often not exactly 500, 3000, 7000 or 10000 or 1:2, 1:3 because of the restriction discussed before. The exact simulated networks are shown in Table II.24.

5.2 Generating Distributed Routing Tables

The next step in the generation of a functional network model is to populate the switch routing tables in the network. This task is similar to the function of the subnet manager (SM) in InfiniBand$^{\text{TM}}$. However, we assume that we have much more time to find an ideal solution than the SM has under real-world conditions.

We define the ideal routing table according to *effective bisection bandwidth* metric that we discussed in Section 4. That means that an ideal set of routing tables T leads to minimal congestion in

factor	ports/d	τ	ν	E	σ	min	max	eff bisect bw
1:1	512/2	1.125	5.00	541.3	94.2	194	858	0.599
3:5	520/2	0.825	3.80	667.5	132.2	207	932	0.562
1:3	528/2	0.625	3.00	749.7	125.2	411	1062	0.536
1:1	3072/3	3.375	14.0	2954	623.0	417	4395	0.632
3:5	3200/3	2.475	10.4	3013	523.6	590	4576	0.598
1:3	3456/3	1.875	8.00	3825	852.5	839	6158	0.555
1:1	7168/3	3.375	14.0	7365	1115	1133	10208	0.594
3:5	7200/3	2.475	10.4	-7969	1569	1945	12090	0.570
1:3	6912/3	1.875	8.00	8448	1329	1582	12581	0.503
1:1	10240/3	3.375	14.0	10627	1444	1556	14900	0.569
3:5	10400/3	2.475	10.4	11471	2009	3092	19487	0.557
1:3	10368/3	1.875	8.00	12240	1905	3125	18774	0.502

(a) 8 port crossbar

factor	ports/d	τ	ν	E	σ	min	max	eff bisect bw
1:1	512/2	0.562	5.00	467.4	110.9	156	732	0.717
5:11	528/2	0.375	3.36	498.7	89.63	280	771	0.685
1:3	576/2	0.312	3.00	611.5	96.17	408	1068	0.635
1:1	3072/2	0.562	5.00	3241	403.5	1004	4956	0.596
5:11	2992/2	0.375	3.36	3266	398.5	1530	4767	0.584
1:3	3072/2	0.312	3.00	4491	549.3	2604	6222	0.493
1:1	7040/2	0.562	5.00	7595	982.9	2000	11520	0.555
5:11	7040/2	0.375	3.36	9969	1280	5589	17035	0.486
1:3	6912/2	0.312	3.00	9569	1188	7032	15006	0.472
1:1	8192/2	0.562	5.00	9193	1406	2692	14244	0.542
5:11	10032/2	0.375	3.36	18984	3839	6903	27743	0.442
1:3	9984/2	0.312	3.00	25375	5155	7320	37770	0.391

(b) 16 port crossbar

factor	ports/d	τ	ν	E	σ	min	max	eff bisect bw
1:1	576/2	0.375	5.00	431.9	127.9	144	690	0.777
1:2	384/1	0.083	1.50	736.0	0.00	736	736	0.581
1:3	432/1	0.069	1.33	1242	0.00	1242	1242	0.455
1:1	3168/2	0.375	5.00	3165	490.3	654	4764	0.668
1:2	3072/2	0.250	3.50	3269	415.4	2288	4992	0.601
1:3	3024/2	0.208	3.00	3357	555.4	2313	5805	0.597
1:1	6912/2	0.375	5.00	7565	1230	2460	11142	0.574
1:2	6912/2	0.250	3.50	7298	739.2	4520	10088	0.584
1:3	6912/2	0.208	3.00	9307	1375	5337	14121	0.531
1:1	10080/2	0.375	5.00	10964	2589	2406	20994	0.579
1:2	9984/2	0.250	3.50	10819	1159	5616	16408	0.568
1:3	9936/2	0.208	3.00	11278	1365	7083	17244	0.548

(c) 24 port crossbar

factor	ports/d	τ	ν	E	σ	min	max	eff bisect bw
1:1	512/1	0.093	2.00	496.0	0.00	496	496	0.812
11:21	672/1	0.063	1.52	1282	63.04	1071	1491	0.583
1:3	768/1	0.052	1.333	7803	996.0	5340	13260	0.443
1:1	3072/2	0.281	5.00	2881	572.1	728	4600	0.684
11:21	3360/2	0.191	3.57	3329	592.0	1617	5467	0.650
1:3	3072/2	0.156	3.00	3622	497.6	2208	5400	0.582
1:1	7168/2	0.281	5.00	7139	996.0	1616	10600	0.658
11:21	7392/2	0.191	3.57	7762	1277	3364	12841	0.628
1:3	6912/2	0.156	3.00	7803	1208	5340	13260	0.584
1:1	10240/2	0.281	5.00	10323	1169	2888	14248	0.638
11:21	10080/2	0.191	3.57	10480	1376	4608	17047	0.619
1:3	9984/2	0.156	3.00	11422	1731	7464	18684	0.578

(d) 32 port crossbar

Figure II.24: Constructed Fat Tree Network Layouts and their Parameters

a sufficiently large number of random *bisect* communication patterns. For *regular* networks like Fat Tree networks, we can assess the route quality by measuring the expected value E and standard deviation σ in the maximum number of routes of all links connecting all P^2 possible communication pairs.

Chung et al. define the forwarding index ξ which describes the number of routes that lead through a given edge in [56]. It is clear that a reduction of this forwarding index will lead to an increased *effective bisection bandwidth* of the network. Several researchers analyzed the problem of minimizing the routing index [98, 97]. However, Saad showed that the problem is NP-complete for arbitrary graphs [176]. We will focus on the edge forwarding index π, introduced by Heydemann et al. in [98]. However, we argue that the expected value E is more meaningful than the maximum value π in the real-world settings we analyze.

Since solving this problem optimally is NP-complete, we propose the following abstract greedy algorithm to find a reasonable approximation:

1. For all P^2 pairs of nodes as S (source) and T (target).
2. Find all circle-free paths p from S to P.
3. Choose the set of shortest paths p_s from p.
4. Delete all paths in p_s that conflict with the current set of distributed routes.
5. Find the maximum number of routes (R) of all paths that use elements of this path.
6. Find the path p_{ST} that has the lowest R.
7. Update all routing tables t along path p_{ST}.

This abstract greedy algorithm not feasible due to it's time complexity of at least $O(P^4)$ (all P^2 paths have to be traversed twice). Thus, we introduce another simplification by employing Dijkstra's single-source shortest-paths algorithm P times:

1. For all P nodes as S find shortest paths to all $P-1$ other nodes p_s.
2. Delete all paths in p_s that conflict with the current set of distributed routes.
3. Update all routing tables t along paths p_{ST}, $\forall T \in \{P \backslash S\}$.

Using Dijkstra's algorithm might introduce more congestion than the original algorithm because paths near the source node will likely be shared. However, our experiments showed that this averages out in mid-scale networks. Step (3) in the previous algorithm can be very costly. In our case of an undirected graph, we can invert the shortest paths by changing s and t after finding the routes and so avoid conflicts and save the checking costs. This leads to the final algorithm:

1. For all P nodes as S find shortest paths to all $P-1$ other nodes p_s.
2. Update all routing tables t along inverted paths p_{TS}, $\forall T \in \{P \backslash S\}$.

We assessed quality of the routing tables of our generated and queried networks. In order to do this, we needed to determine the forwarding index for every edge by simulating all $P(P-1)$ connections in the network. After we generated the forwarding index for each edge, we ran $P(P-1)$ simulations in order to assess the maximum forwarding index along each route. The results for the average edge forwarding index E, it's standard deviation σ and its minimum and maximum along all routes are shown in Table II.24. The results for some real-world systems are shown in the following table:

Name	# Nodes	τ	ν	E	σ	min	max	eff bisect bw
Odin	128	0.141	2.109	138.67	34.79	40	262	0.746
CHiC	566	0.222	3.078	645.93	151.62	58	1743	0.606
Atlas	1142	0.210	3.013	1806.7	670.13	1012	4211	0.556
Ranger	4081	0.339	4.212	7653.43	11140.17	184	90435	0.568
Tbird	4391	0.129	2.036	10868.8	2878.47	7658	25169	0.406

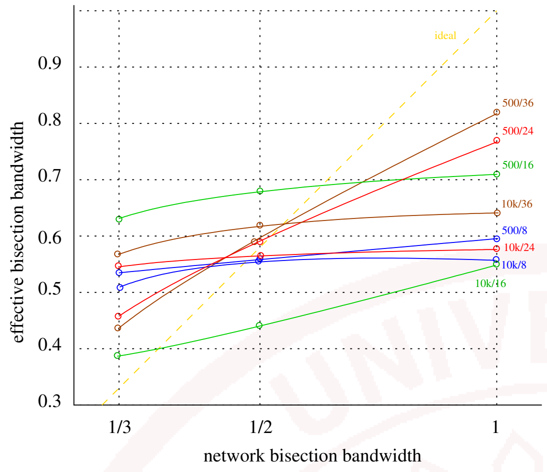
We conclude that the routing tables in our generated networks are at least as balanced as the routing tables in systems that are currently operated. Thus, we use those network layouts for all further analyses. Designing better routing algorithms will be a topic for future work.

5.3 Simulation Results

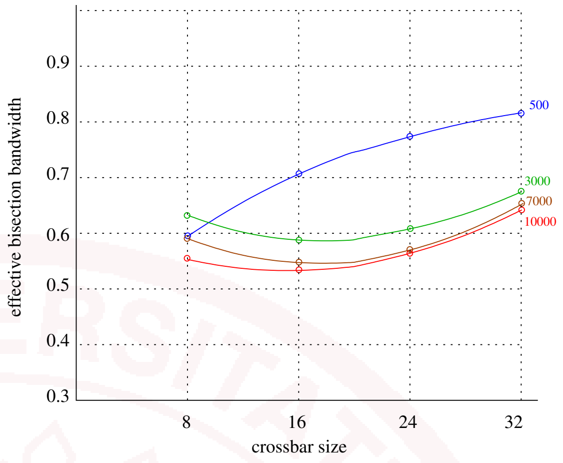
We simulated all generated Fat Tree networks with the simulator discussed in Section 4. We used the *effective bisection bandwidth*, which is defined as the average of all bandwidths in a bisect-communication pattern (cf. Section 4.2.2.3), as a metric to assess the quality of a network topology. All simulation results are shown in Figure II.24. They can be compared to the results shown at the end of the previous section that were gathered by simulating real-world systems.

We present the data of Figure II.24 in two different diagrams. Diagram II.25(a) shows the relationship between effective bisection bandwidth and bisection bandwidth as stated for the network. We clearly see that the static routing influences the results significantly. The *effective bisection bandwidth* is higher than the ideal (which seems like a theoretical maximum) because we investigate random communication patterns and not the worst-case like in the original definition. Networks, where the ratio between port-count and crossbar-size is small, are behaving closer to the ideal network because our Fat Tree networks are structured symmetrically. Generally, it can be concluded that bigger crossbars perform better for most configurations and that the effects of oversubscription are less important than our model assumes. Diagram II.25(b) shows the relationship between the crossbar-size and *effective bisection bandwidth* for different network sizes. The crossbar-size is an important parameter for the cost-effective production of large-scale networks. This diagram shows that larger crossbars deliver better bandwidths. There seems to be an anomaly for 8-port crossbars and network sizes bigger than 3000 nodes. However, this is due to the fact that Fat Trees used to build those networks had a recursion depth of three (see Figure II.24) and use thus significantly more resources (switches and cables) than networks based on 16-or-higher-port crossbars.

We conclude that bigger crossbars are generally beneficial and that statically routed and not fully non-blocking networks can not achieve full bisection bandwidth. In our metric of *effective bisection bandwidth*, it seems that up to 1:3 oversubscribed networks can still deliver reasonable performance. However, a specific network design has to be selected based on price/performance with the particular parameters of crossbar- and cable-costs on a case-by-base basis. We show a way to evaluate different network designs and enable such a selection. Our simulations can also be extended to different communication patterns such as typical data-center workloads.



(a) Different network sizes, oversubscription factors and crossbar sizes



(b) Different crossbar sizes and full bisection bandwidth network layouts

Figure II.25: *Effective Bisection Bandwidth* for different Network Layouts

6 Topology-Aware Collective Communication

"We are all agreed that your theory is crazy. The question which divides us is whether it is crazy enough to have a chance of being correct." – Niels Bohr, (1885-1962) Danish physicist, Nobel Prize 1922.

We have gained some experience with the detrimental effects of congestion in large-scale InfiniBand networks, and we know ways to avoid congestion proactively. The problem is related to the problem that MagPIe [120] addresses. The first obvious way is to change the node-assignment in the batch-system to minimize the probability of congestion situations. However, in order to do this, the batch system would need detailed information about the communication patterns of the parallel applications. Otherwise, it can just fall back to the simple heuristic to map the application on the smallest number of neighboring (in the Fat Tree) leaf-switches. This heuristic is usually simply done by naming the nodes accordingly and using linear node-mapping strategies. Another possibility would be to collect information about the application communication characteristics between runs. But this method has several fundamental problems (e.g., applications often implement different methods with communication patterns) and is thus not further investigated.

The first and simple node mapping strategy can be efficient for a huge number of small-scale jobs on a large-scale system. However, large-scale jobs that use the whole system can not benefit from this strategy. As shown in Section 4.4.2, many applications use regular collective communication patterns. Using this information, one can use different techniques to mitigate congestion. One way is to change the routing tables in order to support a particular pattern. This is not suitable because clusters are usually used by many applications with different requirements at the same time and re-assigning the routing tables is a complex task. Our approach does not change the routing tables but tries to match the communication pattern as closely as possible to the existing routing tables.

6.1 A Rank-Remapping Strategy for Collective Algorithms

In order to map the collective operations better to the underlying communication network, we use a given collective algorithm (e.g., Bruck) and a fixed rank-to-node mapping (given by the batch system) and just change the numbering of the nodes, i.e., the order of communication.

For example, we use the network shown in Fig. II.18, a bruck pattern (i.e., rank n sends to rank $n + 2^r$, $\forall\, 0 < r < log_2 P$) with 16 nodes and a linear rank-to-node mapping (R1→N1, R2→N2, . . . , R16→N16).

round	communications
1	N1→N2, N2→N3, N3→N4, N4→N5, . . . , N15→N16, N16→N1
2	N1→N3, N2→N4, N3→N5, N4→N6, . . . , N15→N1, N16→N2
3	N1→N5, N2→N6, N3→N7, N4→N8, . . . , N15→N3, N16→N4
4	N1→N9, N2→N10, N3→N11, N4→N12, . . . , N15→N7, N16→N8

We can now change the rank-to-node mapping of the collective algorithm without changing the communication patterns (only commutative and associative reductions can use the bruck pattern), for example the reverse mapping (R1→N16, R2→N15, . . . , R16→N1) would still represent the collective communication pattern correctly. The big advantage of this remapping is that it is completely transparent to the user, i.e., it can be done in the collective implementation and different mappings can be chosen for different collective algorithms.

It is not obvious that something can be gained by remapping the rank-to-node assignment in the communication pattern. If we apply the three metrics (1) sum of the maximum congestions per round, (2) sum of all congestions per round and (3) sum of all congestions per cable each round, we get congestions of 7, 104 and 216 respectively. This can be reproduced by manually routing the connections through the network in Figure II.18. This linear mapping seems to be the best mapping because the communication that spans switch borders is pretty low (especially in round 1 and 2). So there seems to be no optimization potential for this simple mapping. We will nevertheless try to achieve some optimization.

Our goal is to minimize the congestion. However, we have seen that assessing the congestions for the tiny 16 node case is hard to do manually; optimizing the mapping is even harder. Thus, we propose to use a common optimization scheme, genetic algorithms [122] (GA), to find a better mapping. We implemented the optimization with the PGAPACK [135] framework which also allows parallel optimization. A gene in our case is a set of integers of length P in the range $0 \ldots P-1$. The position and the integer represents the mapping, i.e., the gene $2, 0, 1$ represents the mapping R0→N2, R1→N0, R2→N1. The strings are initialized randomly. Mutation is performed by flipping two neighbor-elements. Crossover is done by combining the first part of gene 1 with the second part of gene 2. The evaluation function for the GA is the simulation described before. The simulation returns infinite for invalid strings (invalid strings may be a result of mutation). The algorithm has been adjusted to stop if no change happens after 500 iterations.

Using the genetic algorithm with the simple 16 node example results in the following optimized gene (mapping) for metric (1) (15 2 12 5 16 6 11 9 14 3 10 8 13 4 1 7) and the sum of the maximum congestions for this gene is 6 instead of 7 for the linear mapping. The GA was able to find a better solution even though the simple mapping seems rather good at first glance. Using metric (2) returns the gene (16 6 3 11 1 9 2 8 14 7 13 10 15 12 4 5) with a congestion of 92 instead of 104. Metric (3) returns the same gene as for metric (1) and a congestion of 204 instead of 216. This means that the bruck communication pattern for gene 1 changes to:

round	communications
1	N15→N2, N2→N12, N12→N5, N5→N16, . . . , N1→N7, N7→N15
2	N15→N12, N2→N5, N12→N16, N5→N6, . . . , N1→N15, N7→N2
3	N15→N16, N2→N6, N12→N11, N5→N9, . . . , N1→N12, N7→N5
4	N15→N14, N2→N3, N12→N10, N5→N8, . . . , N1→N11, N7→N9

The communication pattern is still the same, just the peers that communicate are different.

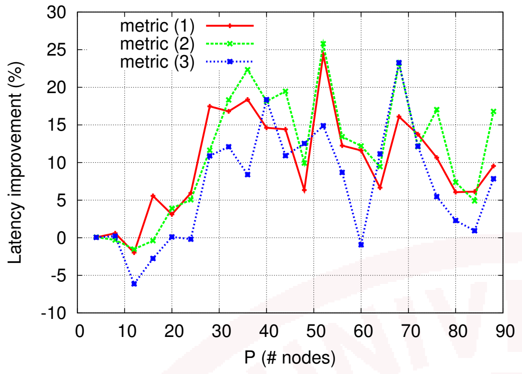
6.2 Benchmarking the Benefits

We implemented a benchmark that uses the rank-remapping technique to optimize collective algorithms on real-world systems. The benchmark works as follows:

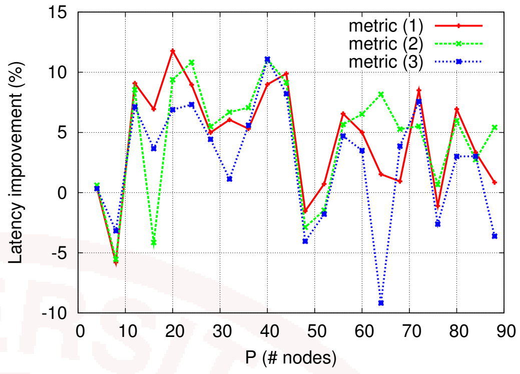
1. Extract the network topology of the system (can be done in advance; explained in Section 4).
2. Get the position of each rank in the network (extracting InfiniBand^{TM}'s GUID).
3. Generate a collective communication pattern.
4. Run a benchmark using the generated communication pattern without remapping.
5. Run the genetic algorithm described above to remap ranks in the pattern (the GA is run in parallel on all nodes in the allocation).
6. Run a benchmark using the optimized mapping.

The results for the *Odin* system using different node-numbers and the bruck-algorithm using $10MiB$ messages are shown in Figure II.26(a). The results for the tree-pattern, shown in Figure II.26(b) show a smaller gain but are still positive. We assume that the used metrics are not accurate enough for this pattern (and the congestion is less, i.e., the optimization potential is lower). The GA fails to optimize the nearest neighbor pattern because the simple linear mapping is extremely good in this case (the GA often converges on a pattern (local minimum) that is worse than the simple linear mapping).

We also see decreased performance of some cases; this might be due to two things. First, the metrics we use oversimplify the problem and do not really have any time-dependency. This means they model the algorithm insufficiently. A full LogGP modeling would probably help but is, due to its complexity, outside of the scope of this work. A second problem lies in the optimization approach. The GA can converge in a local minimum and not find the best solution. The solution found might even be worse than the simple linear mapping because the genes are initialized randomly, i.e., they might all be worse than the linear mapping. This can not be addressed easily because the search space for an optimal mapping is huge (the number of valid mappings (genes), i.e., all permutations of P integers is $P!$, which is already $\approx 2 \cdot 10^{13}$ in the small example $P = 16$). Covering a huge part of the search space or an exhaustive search is not possible. The structure of



(a) Bruck Pattern



(b) Tree Pattern

Figure II.26: Algorithm latency improvement on the *Odin* system

the search space has to be analyzed in detail to make any statements about the occurrence, number and distribution of local minima. However, metric (2) was able to deliver reasonable performance improvements for the bruck and tree pattern.

7 Conclusions

"Science may be described as the art of systematic oversimplification." – Karl Popper (1902-1994) British Philosopher

We proposed a new 1:n n:1 benchmarking principle to assess the performance of the InfiniBand^{TM} network for all different transport types. Our proposed solution uses time measurement for each packet to enable detailed statistical analysis. We analyzed all different transport types and concluded that RDMA-W enables the fastest data transmission. Especially the discovery that single message sends are quite slow compared to multi message sends is important for collective algorithms (as most of them use mainly single message sends). We also show that the unreliable multicast operations offer a scalable interface to perform one-to-many send operations.

Based on these results, we showed that simple modifications can enhance the accuracy of the LogGP model significantly. The new LogfGP model is easy enough to be used by developers to optimize parallel algorithms because it encourages the programmer to leverage the hardware parallelism for the transmission of small messages. We think that our model is more accurate than the LogP model for other offloading-based networks where most packet processing is done in hardware (e.g., Quadrics, Myrinet).

In order to allow accurate algorithm modeling, we compared well known Log(G)P measurement methods and derived a new accurate LogGP parameter measurement scheme. Our method is able to detect protocol changes in the underlying communication subsystem. An open source implementation within the Netgauge framework is available for public use. This implementation has been tested extensively with different modern MPI implementations and low-level networking APIs.

We used our findings to improve two MPI collective algorithms. The first collective, MPI_Barrier, uses the LogfP model and leverages the inherent hardware parallelism of the InfiniBand^{TM} network. The new n-way dissemination principle could be used to improve other collective algorithms (e.g., MPI_Allreduce). We use automated tuning to choose the n-parameter. However, the barrier operation could be enhanced up to 40% on a 64 nodes cluster in comparison to the well tuned implementation of MVAPICH, and the gap is expected to widen for larger node numbers.

The second algorithm uses InfiniBand^{TM}'s multicast feature to improve MPI_Bcast. Contrary to all other known approaches, we are able to avoid all scalability/hot-spot problems that occur with currently known schemes. The new multicast-based broadcast implementation accomplishes a practically constant-time behavior in a double meaning: it scales independently of the communicator size and all MPI processes within a given communicator need the same time to complete the broadcast operation. Well-known microbenchmarks substantiate these theoretical conclusions with practical results. We proved it for up to 116 cluster nodes and there is no reason to assume scalability problems with our approach.

We showed that the original definition of full bisection bandwidth [95] does not take the network routing into account and might thus not be meaningful for practical applications. Thus, we defined a more application performance oriented measure, the *effective bisection bandwidth* which takes routing into account. We demonstrated a benchmark to perform this measurement. We also propose a simulation methodology and simulate three of the largest existing InfiniBand clusters. Our results show that none of those systems achieves more then 61% of the theoretical bisection bandwidth. We also analyzed different real-world applications for their communication patterns and simulated those patterns for the three analyzed systems. A rough estimation of the communication behavior showed that the communication time of those applications could nearly be halved and the time to solution could be optimized by more than 12% with a network offering full *effective bisection bandwidth*. Our results also show that an optimized process-to-node layout as offered by topological MPI communicators might result in a significant performance increase.

Furthermore, we discussed and simulated different options for network topologies and routing tables for static distributed routing. With the simulation of different Fat Tree network configuration we were able to show that the problem occurs for all configurations that are not non-blocking Clos networks. The simulation leads us to the interesting conclusion that full bisection bandwidth networks are not delivering a significant higher performance than oversubscribed networks in the average case.

A genetic algorithm strategy to remap collective ranks in order to increase the bandwidth for a given topology was reasonably successful for small scale networks, however, the search space in large-scale networks ($P!$) seems too large to explore in a reasonable way.

Chapter III

New Collective Operations

"The nice thing about standards is that there are so many to choose from. And if you really don't like all the standards you just have to wait another year until the one arises you are looking for." – Andrew Tanenbaum, (1944) Dutch Computer Scientist, in "Introduction to Computer Networks"

Scalable parallel systems are specialized machines that differ dramatically in their architecture. Those machines are often hard to understand and to program. The current MPI standard defines collective operations that reflect high-level communication patterns to simplify the programmer's task to implement scalable parallel applications. This addition of abstraction does not only benefit programmability, it also simplifies performance portability and reduces common errors in parallel programs.

The defined collective operations in the standard cover a wide range of dense one-to-many, many-to-one and many-to-many communication patterns that have been shown to be useful. Vector variants of most collectives enable contributions of different sizes from different processes and also sparse communication patterns. However, the interface is still dense and makes the implementation of large-scale sparse communication patterns suboptimal. In Chapter II, we discovered that overlapping communication and computation has the potential to reduce the communication overhead by several orders of magnitude for large point-to-point messages. However, the use of overlapping techniques in high-level communication patterns has not been analyzed yet.

In this chapter, we discuss sparse collective operations, a new class of collectives that efficiently enables sparse communication patterns at large scale. We demonstrate a possible interface to those new sparse collectives and provide an example application that uses this interface to implement an irregular communication pattern on a graph. Furthermore, we will investigate the possibility of combining overlapping techniques with high-level communication patterns. For this, we summarize and extend results from the article "A Case for Non-Blocking Collective Operations" [19]. The next section introduces sparse collective operations followed by a discussion and analysis of non-blocking collective operations. Section 3 models the CPU overhead and latency of non-blocking collective operation and defines new algorithm selection criteria based on overhead models.

1 Sparse Collective Communication

"The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them." – William Lawrence Bragg, (1890-1971) Australian Physicist, Nobel Prize 1915

MPI topologies currently provide the MPI implementation with information about the typical communication behavior of the processes in a new communicator, illustrating the structure of communication via a Cartesian grid or a general graph. Topologies contain important application- and data-specific information that can be used for optimized collective implementations and improved mapping of processes to hardware. However, from the application perspective, topologies provide little more than a convenient naming mechanism for processes.

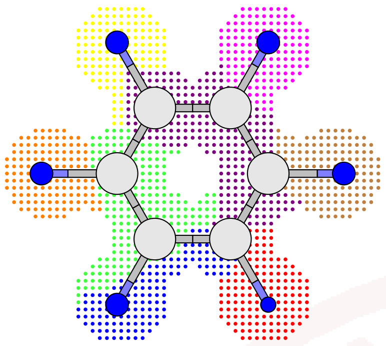
We propose new collective operations that operate on communicators with process topologies. These topological collectives express common communication patterns for applications that use process topologies, such as nearest-neighbor data exchange and shifted Cartesian data exchange. These collectives are usually implemented by the application programmer. However, a high-performance implementation of those operations is not trivial and programmers frequently face problems with deadlocks.

Nearest neighbor exchanges are performed in many parallel codes such as in real space electronic structure codes like Octopus [51]. The mesh's points are distributed among the nodes to increase computational throughput. Figure III.1(a) shows the geometry of a benzene molecule on top of a real space grid. The grid points are divided into eight partitions. Communication between adjacent partitions is necessary to calculate derivatives by finite-difference formulas, or, more generally, any non-local operator. Figure III.1(b) shows the situation for a third order discretization: to calculate the derivative at point i, the values at points $i-1, \ldots, i-3$ have to be communicated from partition 2 to partition 1. The communication structure can be represented as a graph with each vertex representing one computational node and edges representing neighboring partitions (cf. Figure III.1(c)). Using this abstraction, the data exchange prior to the calculation of a derivative can be mapped onto a general nearest neighbor communication pattern.

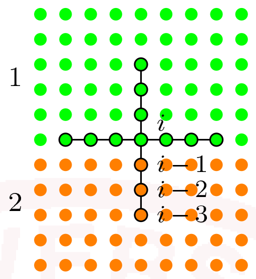
We propose a new class of collective operations defined in terms of topology communicators. The new collective operations describe a neighbor exchange and other sparse collectives where the communication relations are defined by the process (graph) topology.

1.1 Programmability

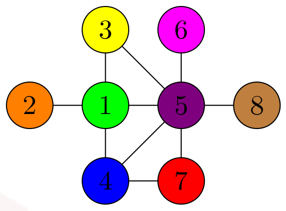
Listing III.1 shows the NBC_Ialltoall implementation which uses four different arrays to store the adjacency information. The programmer is fully responsible for administering those arrays.



(a) The benzene ring distributed on eight nodes indicated by different colors.



(b) A third order stencil leaking into a neighboring partition.



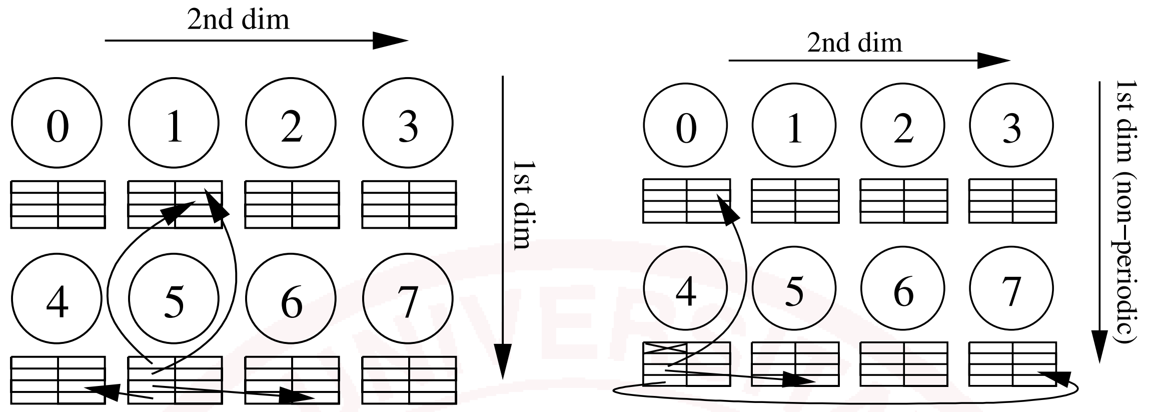
(c) The communication pattern for the benzene ring.

Figure III.1: Process topologies in real space electronic structure codes.

```
rdpls = calloc(p,sizeof(int)); sdpls = calloc(p,sizeof(int));
rcnts = calloc(p,sizeof(int)); scnts = calloc(p,sizeof(int));
for(i=0; i<len(list); i++) if(list[i][0] == rank)
    scnts[list[i][1]] = count; rcnts[list[i][1]] = count;
sdispls[0] = rdispls[0] = 0;
for(i=1; i<p; i++) {
   sdpls[i] = sdpls[i-1] + scnts[i];
   rdpls[i] = rdpls[i-1] + rcnts[i]; }
MPI_Alltoallv(sbuf, scnts, sdpls, dt, rcnts, rdpls, dt, comm, req);
```

Listing III.1: Neighbor Exchange with Alltoallv.

It seems more natural to the programmer to map the output of a graph partitioner (e. g., an adjacency list that represents topological neighbors) to the creation of a graph communicator and simply perform collective communication on this communicator rather than performing the Alltoallv communication. To emphasize this, we demonstrate a pseudocodes that perform a similar communication operation to all graph neighbors indicated in an undirected graph (`list[i][0]` represents the source and `list[i][1]` the destination vertex of edge `i` and is sorted by source node). The Alltoallv implementation has already been shown in Listing III.1. Listing III.2 shows the implementation with our newly proposed operations that acquire the same information from the MPI library (topology communicator layout). The processes mapping in the created graph communicator might be rearranged by the MPI library to place tightly coupled processes on close processors (e. g. on the same SMP system). The collective neighbor exchange operation allows other optimizations (e. g. starting off-node communication first to overlap local memory copies of on-node communication).



(a) MPI_Neighbor_xchg, illustrating the communication operations originating at rank 5 in a 2-dimensional cartesian communicator. The left side of the buffer represents the send memory and the right side the receive memory.

(b) MPI_Neighbor_xchg, ilustrating the communication originating at node 4 for a non-periodic and a periodic dimension. The crossed buffer will be ignored by the collective (but has to be allocated)

Figure III.2: Sparse Communication Patterns in Cartesian Communicators

```
last = list[0][0]; counter = 0; // list is sorted by source
for(i=0; i<len(list); i++) {
 if(list[i][0] != last) index[list[i][0]] = counter;
 edges[counter++] = list[i][1];
}
MPI_Graph_create(comm, nnodes, index, edges, 1, topocomm);
MPIX_Neighbor_xchg(sbuf, count, dt, rbuf, count, dt, topocomm, req);
```

Listing III.2: Implementation using MPIX_Neighbor_xchg.

1.2 Nearest Neighbor Exchange

We propose to add a new collective function named MPI_Neighbor_xchg (and its vector variant) that performs nearest-neighbor communication on all processes in a communicator with a topology. Each process transmits data to and receives data from each of its neighbors. The neighbors of a process in the communicator can be determined by MPI_Comm_neighbors.

The outcome is as if each process executed a send to each of its neighbors with a call to, MPI_Send(sendbuf + i*sendcount*extent(sendtype), sendcount, sendtype, neighbor[i], ...) and a receive from every neighbor with a call to, MPI_Recv(recvbuf + i*recvcount*extent(recvtype), recvcount, recvtype, neighbor[i], ...). Figure III.2(a) shows an example with a Cartesian communicator.

1.3 Neighbor Query

We propose two new functions that allow one to determine the neighbors of any communicator with a topology. These functions are modeled after MPI_Graph_neighbors_count and MPI_Graph_neighbors, but they work for both communicators with graph topology and for communicators with Cartesian topology. When passing a graph communicator to

MPI_Comm_neighbors_count, the user will receive the number of neighbors in the graph (i.e., the same result that MPI_Graph_neighbors_count provides). For a Cartesian communicator, the user will receive the number of processes whose distance from the calling process is 1. If provided with a communicator that does not have a topology, the call returns MPI_ERR_TOPOLOGY. When passing a graph communicator to MPI_Comm_neighbors, the user receives the ranks of each of the neighbors of the calling process (i.e., the same result that MPI_Graph_neighbors provides). The order of the ranks returned defines the layout of the send/receive arrays. This order is unspecified, but successive calls to MPI_Comm_neighbors for the same communicator will return ranks in the same order. When passing a Cartesian communicator to MPI_Comm_neighbors, the user receives the ranks of each of the neighbors of the calling process. The order of the ranks is first along the first dimension in displacement +1 and -1 (either of which may wrap around, if the topology is periodic in that dimension, or will be omitted, if not periodic in that dimension), then along the second, third and higher dimensions if applicable. Figure III.2(b) shows an example communication with periodic dimensions.

1.4 Other Sparse Collective Functions

The proposed interface can be extended to all collective functions. For example, reduction operations defined on neighbor groups can be very helpful for parallel computational fluid dynamics (CFD) programs. Those new collective operations will follow similar principles and will be investigated further. We are also actively participating in the MPI Forum and these operations have been proposed for consideration in the upcoming MPI-3 standard.

2 Non-Blocking Collective Communication

"The more physics you have the less engineering you need." – Ernest Rutherford, (1871-1937) English Physicist, Nobel Prize 1908

A non-blocking interface to collective operations enables the programmer to start the execution of a collective operation and then perform other computations. Later, the program can wait or test for the end of the operation separately. Non-blocking collective operations and their possible benefits have already been discussed at meetings of the MPI standardization committee. The final decision to not include them into the MPI-2 standard fell at March 6, 1997[1]. However, the fact that the decision was extremely marginal (11 yes / 12 no / 2 abstain) suggests that the role of non-blocking collective operations is still debatable. Our contention is that non-blocking collective operations are a natural extension to the MPI standard. We show that non-blocking collective operations can be beneficial for a class of applications to utilize the available CPU time more efficiently and decrease the time to solution of these applications significantly. Further, we discuss two main problems of blocking collective communication which limit the scalability of applications.

First, blocking collective operations have a more or less synchronizing effect on applications which leads to unnecessary wait time. Even though the MPI standard does not define blocking collective operations other than MPI_Barrier to be strictly synchronizing, most algorithms force

[1]see: http://www.mpi-forum.org/archives/votes/mpi2.html

many processes to wait for other processes due to data dependencies. In this way, synchronization with a single process is enforced for some operations (e.g., a MPI_BCAST can not be finished until the root process called it) and the synchronizing behavior of other operations highly depends on the implemented collective algorithm. In either case, pseudo-synchronizing behavior often leads to many lost CPU cycles, multiplication of the effects of process skew (e.g., due to daemon processes which "steal" the CPU occasionally and introduce a pseudo-random skew between processes [214, 198]), and a high sensitivity to imbalanced programming (e.g., some processes do slightly more computation than others in each round).

Second, most blocking collective operations can not take much advantage of modern interconnects which enable communication offload to support efficient overlapping of communication and computation. Abstractly seen, each supercomputer or cluster consists of two entities, the CPU which processes data streams and the network which transports data streams. In many networks, both entities can act mostly independently of each other, but the programmer has no chance to use this parallelism efficiently if blocking communication (point-to-point or collective) is used.

Another rationale to offer non-blocking semantics for collective communication is an analogy between many modern operating systems and the MPI standard. Most modern operating systems offer possibilities to overlap computation on the host CPU with actions of other entities (for example hard disks or the network). Asynchronous I/O and non-blocking TCP/IP sockets are today's standard features for communication. The MPI standard offers non-blocking point-to-point communication which can be used to overlap communication and computation. It would be a natural extension to offer also a non-blocking interface to the collective operations.

Recent work has shown that non-blocking operations can be beneficial, both in terms of performance and abstraction. Non-blocking operations allow communication and computation to be overlapped and thus to leverage hardware parallelism for the asynchronous (and/or network-offloaded) message transmission. Several studies have shown that the performance of parallel applications can be significantly enhanced with overlapping techniques (e.g., cf. [44, 63]). Similarly, collective operations offer a high-level interface to the user, insulating the user from implementation details and giving MPI implementers the freedom to optimize their implementations for specific architectures.

It has long been suggested that non-blocking collective functionality is not needed explicitly as part of MPI because a threaded MPI library could be used with collective communication taking place in a separate thread. However, there are several drawbacks to this approach. First, it requires language and operating system support for spawning and managing threads, which is not possible on some operating systems—in particular on operating systems such as Catamount designed for HPC systems. Second, programmers must then explicitly manage thread and synchronization issues for purposes of communication even though these issues could and should be hidden from them (e.g., handled by an MPI library). Third, the required presence of threads and the corresponding synchronization mechanisms imposes the higher cost of thread-safety on all communication operations, whether overlap is obtained or not (cf. [87]). Finally, this approach provides an asymmetric treatment of collective communications with respect to point-to-point communications (which do support asynchronous communications).

Non-blocking collective operations provide some performance benefits that may only be seen at scale. The scalability of large scientific application codes is often dependent on the scalability of the collective operations used. At large scale, system noise affects the performance of collective communications more than it affects the performance of point-to-point operations because of the collective dependencies which are often introduced by collective communication algorithms. To continue to scale the size of HPC systems to Peta-scale and beyond, we need communication paradigms that will admit effective use of the hardware resources available on modern HPC systems. Implementing collective operations so that they do not depend on the the main CPU is one important means of reducing the effects of system noise on application scalability.

2.1 Related Work

The obvious benefits of overlapping communication with computation and leveraging the hardware parallelism efficiently with the usage of non-blocking communication is well documented. Analyses [103, 106, 129] try to give an assessment of the capabilities of MPI implementations to perform overlapping for point-to-point communications. Many groups analyze the possible performance benefits for real applications. Liu et al. [137] showed possible speedups up to 1.9 for several parallel programs. Brightwell et al. [45] classifies the source of performance advantage for overlap and Dimitrov [66] uses overlapping as fundamental approach to optimize parallel applications for cluster systems. Other studies, as [49, 33, 63, 25] apply several transformations to parallel codes to enable overlapping. Several studies of the use of overlapping techniques to optimize three-dimensional FFTs have been done [68, 26, 50, 81]. The results of applying these non-blocking communication algorithms (replacing MPI All-To-All communications) were inconclusive. In some cases the non-blocking collectives improved performance, and in others performance degraded a bit. Danalis et al. [63] obtained performance improvement by replacing calls to MPI blocking collectives with calls to non-blocking MPI point-to-point operations. However, little research has been done in the field of non-blocking collectives. Studies like [68, 44] mention that non-blocking collective operations would be beneficial but do not provide a measure for it. Kale et al. [113] analyzed the applicability of a non-blocking personalized exchange to a small set of applications in practice. However, many studies show that non-blocking communication and non-blocking collectives *may* be beneficial. Due to its channel semantics, MPI/RT [114] defines all operations, including collective operations, in a non-blocking manner. IBM extended the standard MPI interface to include non-blocking collectives in their parallel environment (PE), but have dropped support for this non-standard functionality in the latest release of this PE. Our work contributes to the field because we actually assess the potential performance benefits of a non-blocking collective implementation.

2.2 Possible Performance Benefits

The most obvious benefits of non-blocking collective operations are the avoidance of explicit pseudo synchronization and the ability to leverage the hardware parallelism stated in Chapter II. The pseudo-synchronizing behavior of most algorithms cannot be avoided, but non-blocking collective operations process the operation in the background, which enables the user to ignore most synchronization effects. Common sources for de-synchronization, process skew and load imbal-

```
1 for( proc=1; proc<nproc; proc=proc*2) {
   create_communicator(nproc, comm);
   for( size=1; size<maxsize; proc=proc*2) {
    gettimeofday(t1); getrusage(r1);
5   for( i=0; i<max_iters; i++)
      MPI_Coll(comm, size, MPI_BYTE, ...)
    getrusage(r2); gettimeofday(t2);
  }}
```

Listing III.3: Benchmark Methodology (pseudocode)

ance are not easily measurable. However, results can increase the application running time dramatically, as shown in [165]. Theoretical [27] and practical analyses [112, 165] show that operating system noise and resulting process skew is definitely influencing the performance of parallel applications using blocking collective operations. Non-blocking collective operations avoid explicit synchronization unless it is necessary (if the programmer wants to wait for the operation to finish). This enables the programmer to develop applications which are more tolerant of process skew and load imbalance.

Another benefit is to use the parallelism of the network and computation layers. Non-blocking communication (point-to-point and collective) allows the user to issue a communication request to the hardware, perform some useful computation, and ask later if it has been completed. Modern interconnect networks can perform the message transfer mostly independently of the user process. The resulting effect is that, for several algorithms/applications, the user can overlap the communication latency with useful computation and ignore the communication latency up to a certain extent (or totally). This has been well analyzed for point-to-point communication. Non-blocking collective operations allow the programmer to combine the benefits of collective communication [83] with all benefits of non-blocking communication. The following subsections analyze the communication behavior of current blocking collective algorithms and implementations, and show that only a fraction of the CPU time is involved into communication related computation. In relation to previous studies we show, theoretically and practically, that a similar percentage, in many cases even more, idle CPU time as with non-blocking point-to-point communication can be gained. We assume that the biggest share of the remaining (idle) CPU time can be leveraged by the user if overlap of communication and computation together with non-blocking collective communication can be applied.

2.3 Assessing the Benefits

We implemented a benchmark which measures the CPU utilization for different MPI collective operations. The benchmark uses the standard `gettimeofday()` and `getrusage()` functionality of modern operating systems to measure the idle time. It issues collective calls with different message sizes and communicator sizes. The benchmark methodology is described as pseudocode in Listing III.3. The `getrusage()` call returns system time and user time used by the running process separately. We chose a high number of iterations (10000) in the inner loop (`max_iters`, Line 5) to

Implementation	**Networks**
LAM/MPI 7.1.2	InfiniBand, TCP/IP
MPICH2 1.0.3	TCP/IP
Open MPI 1.1a3	InfiniBand and TCP/IP
OSU MVAPICH 0.9.4.	InfiniBand

Table III.1: Tested MPI Implementations

be able to neglect the overhead and relative impreciseness of the system functions. We conducted the benchmark for different MPI implementations shown in Table III.1.

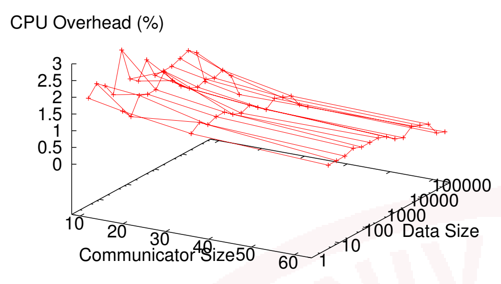
Many MPI libraries are implemented in a non-blocking manner which means that the CPU overhead is, due to polling, 100% regardless of other factors. Only LAM/MPI with TCP/IP and MPICH2 with TCP/IP used blocking communication to perform the collective operations. However, it is correct to use polling to perform blocking MPI collective operations because, at least for single threaded MPI applications, the CPU is unusable anyways and polling has usually slightly lower overhead than interrupt-based (blocking) methods.

We investigated all collective operations for LAM/MPI and MPICH2 and want to discuss the frequently used MPI_Allreduce (cf. [172]) in detail. Both MPI_Allreduce implementations exhibit a similar behavior and use only a fraction of the available CPU power for communicators with more than 8 nodes. MPICH2 causes in the average of all measurement points less than 30% CPU load while LAM/MPI consumes less then 10%. We see also that the data size plays an important role because there may be switching points in the collective implementation where the collective algorithms or underlying point-to-point operations are changed (e.g., 128kb for MPICH2). However, this overhead includes the TCP/IP packet processing time spent in the kernel to transmit the messages which is measured with the `getrusage()` function as system time. User-level, kernel-bypass, and offloading communication hardware like InfiniBand, Quadrics or Myrinet does not use the host CPU to process packets and does not enter the kernel during message transmission. Figure III.3 shows the user-level CPU usage (without TCP/IP processing) for both examples from above.

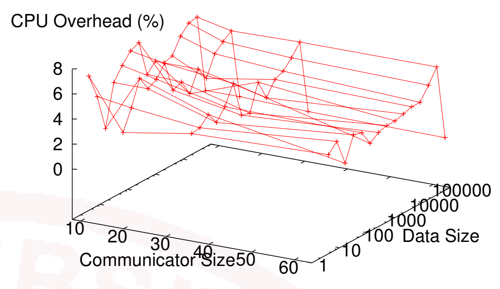
It shows that the CPU overhead for MPI_Allreduce, which implies a user-level reduction operation in our case, is below 10% in the average for MPICH2 and below 3% for LAM/MPI. These figures show also that the share of CPU idle time grows with communicator and data size. Other collective operations exhibit a similar behavior.

However, generally speaking, the time to perform a collective operation grows also with communicator and data size. This means that the overall (multiplicative) CPU waste is even higher. Figure III.4 shows the absolute CPU idle time of both implementations, several collective operations, and a fixed communicated data size with varying communicator sizes. The effect of growing CPU waste during blocking collectives is clearly visible. Especially the MPI_Alltoall operation, which usually scales worst, shows high CPU idle times with a growing number of participating processes.

Figure III.5 shows the absolute CPU idle time of both implementations, for a fixed communi-

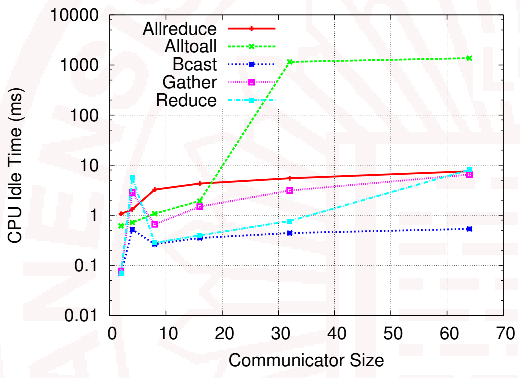


(a) LAM/MPI

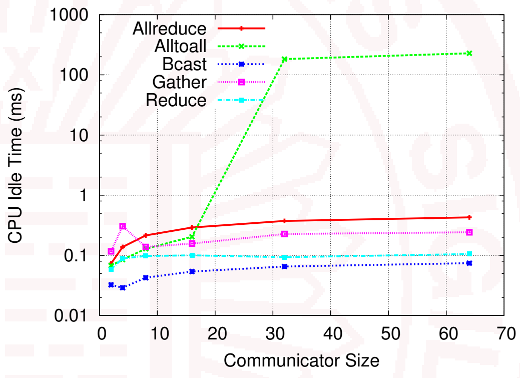


(b) MPICH-2

Figure III.3: MPI_Allreduce (user time) overheads for different MPI libraries



(a) LAM/MPI



(b) MPICH-2

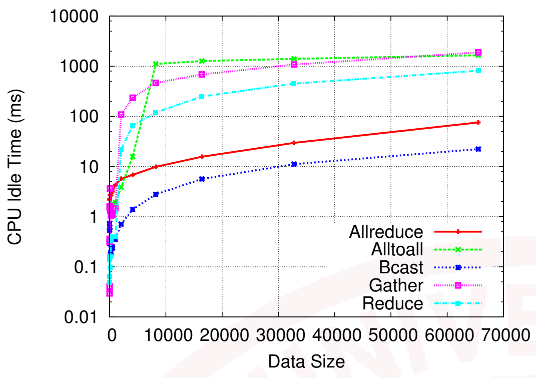
Figure III.4: CPU idle times with varying Communicator Sizes for a constant Data Size of 1kiB

cator size, and varying data sizes. The CPU waste is even higher and scales worse than for the varying communicator size, nearly linearly with the data size (the figures are plotted with a logarithmic scale).

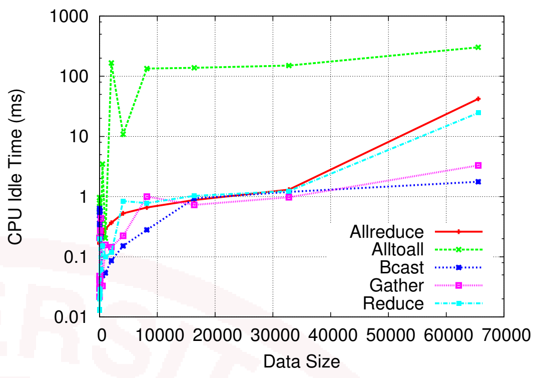
The results show clearly that, using TCP/IP, more than 70% of the CPU time is wasted in average during blocking collective operations. We assume that the gap is more than 90% for offloading-based networks such as InfiniBand, Quadrics or Myrinet which do not process messages on the host CPU. Absolute measurements show the wasted time per collective which can easily be converted into wasted CPU cycles. These considerations lead to possible optimizations using non-blocking collective operations.

2.4 Application Programming Interface

We propose an API for the non-blocking collectives that is similar to that of the blocking collectives and the former proprietary IBM extension. We use a naming scheme similar to the one used for the non-blocking point-to-point API (e.g., MPI_Ibarrier instead of MPI_Barrier). In addition, request objects (MPI_Request) are used for a completion handle. The proposed interfaces to all collective



(a) LAM/MPI



(b) MPICH-2

Figure III.5: CPU idle times with varying Data Size for 16 Processes

operations are defined in detail in the Appendix. For example, a non-blocking barrier would look like:

```
MPI_Ibarrier(comm, request);
...
/* computation, other MPI communications */
...
MPI_Wait(request, status);
```

Our interface relaxes the strict MPI convention that only one collective operation can be active on any given communicator. We extend this so that we can have a huge number (system specific, indicated by MPI_ICOLL_MAX_OUTSTANDING, cf. [18]) of parallel non-blocking collectives and a single blocking collective outstanding at any given communicator. Our interface does not introduce collective tags to stay close to the traditional syntax of collective operations. The order of issuing a given collective operation defines how the collective-communications traffic matches up across communicators. Similar to point-to-point communications, progress for these non-blocking collective operations depends on both underlying system hardware and software capabilities to support asynchronous communications, as well implementation of these collectives by the MPI library. In some cases MPI_Test or MPI_Wait may need to be called to progress these non-blocking collective operations. This may be particularly true for collective operations that transform user data, such as MPI_Allreduce.

3 Modelling Non-Blocking Collective Communication

"To show this diagram properly, I would really need a four dimensional screen. However, because of government cuts, we could manage to provide only a two dimensional screen" – Steven Hawking, (1942) British Physicist

Modelling of collective operations is an important tool for their implementation and usage. Implementers can develop and prove the optimality of new collective algorithms and users can assess the overhead and scaling behavior of collective algorithms in their applications. We used modeling techniques to develop new algorithms in the previous chapter. This chapter uses modeling to assess the potential benefits of overlapping communication and computation in collective algorithms.

Several previous works discussed different models for collective operations. Pjesivac-Grbovic discusses in [170, 171] methods to model MPI collective operations in order to automate optimizations and select optimized algorithms based on certain parameters. In our work [21], we model and compare several different MPI_Barrier algorithms and also conclude that models are sufficient to select switching points for algorithms.

In order to assess the potential performance benefits of using non-blocking collective communication, we use the previously defined models for optimized collective algorithms. We model different collective operations that reflect the areas of one-to-many, many-to-one and many-to-many communication. We model MPI_Bcast, MPI_Gather, MPI_Alltoall and MPI_Allreduce. As shown in [211, 42, 172], these operations are frequently used in real applications. However, our results can also be applied to all other collective operations.

3.1 Collective Algorithm Models

The complex interaction between communication and computation in real-world applications requires models to understand and optimize parallel codes. Serial computation models, such as Modeling Assertions [28], have to be combined with network models such as the LogP [61]. Unfortunately, today's systems are too complex to be described entirely by an execution or communication model. It is necessary to assess the model parameters for every real-world system, in the serial execution case in [28] as well as in the communication case in [62], [5, 24]. Pjesivac-Grbovic et al. showed in [171] that the latency of collective operations that are implemented on top of point-to-point messages can be modeled with the LogP model family and we showed in an earlier work [3] that it is also possible to predict application performance by modeling the communication and computation separately. Thus, it is crucial to the modeling of parallel applications to have accurate models for the latency of collective operations and the overhead of non-blocking collective operations.

Precise models for collective operations have been proposed in [171] and for barrier synchronization in [21]. Both studies show that the LogP [61] or LogGP [29] model is able to predict the communication time sufficiently accurately if the processes enter the collective operation simultaneously.

We use a combination of LogGP and LogfP which we call LogfGP. This new model simply adds the f parameter to the well known LogGP model. The LogfP model and the f parameter have been discussed in Section 2.1.6 in Chapter II. We also chose not to model all collective operations separately because we found that most algorithms are fundamental and can be used to implement many collective operations. We chose broadcast as an example for a one-to-all operation and allreduce as an example for an all-to-all operation. We model different algorithms for those two collective operations and show how the theoretical analysis can lead to new insight regarding the selection of algorithms either by latency or host overhead. We focus on non-blocking collective operations and select thus by host-overhead. Please refer to [171] for latency-based selection maps.

3.1.1 Algorithms for MPI_Bcast

As discussed in [171], there are mainly five different algorithms to implement the MPI_Bcast operation. The algorithms are described in detail in [171] and we refer to them as linear, pipeline, binomial, binary and split-binary in the following. However, the models provided in [171] are based on several simplifying assumptions that lead to significant simplifications in the blocking case, e.g., $g > o$ leads to an omission of o if o and g are to be accounted in parallel. In the general non-blocking case, those assumptions are not valid because both entities (CPU and network) are used by the application at the same time. This makes the models for non-blocking collective communication significantly more complex than the blocking models listed in [171].

We develop models for the linear, segmented pipeline (s indicates the number of segments), binomial and n-ary binomial algorithms. The other algorithm models show similar trends. In order to model the LogfGP model easily, we introduce the new operator ζ that converts all negative numbers to zero, for example $\zeta(4.5) = 4.5$, $\zeta(-3.25) = 0$ and $\zeta(-100.2) = 0$. The models for s segments of size m are

$$t_{bcast}^{lin} = max\{(P-2)o, \zeta(P-f-2)g\} + (m-1)G + L + 2o \tag{III.1}$$

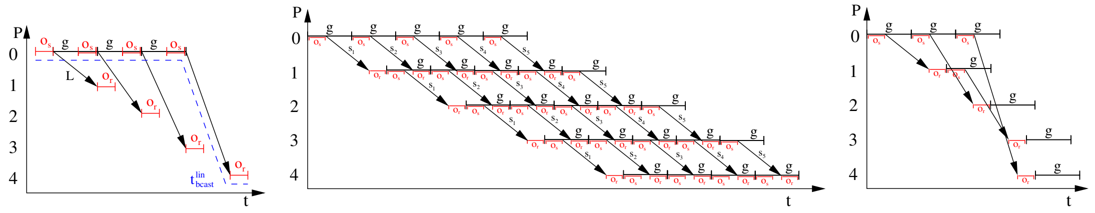
$$t_{bcast}^{pipe} = o + max\{(s-1)o, \zeta(s-f-1)g\} + (P-1)(2o+L+G(m-1)) - o \tag{III.2}$$

$$t_{bcast}^{bin} \approx \lceil log_2 P \rceil \cdot (2o + L + (m-1)G) \tag{III.3}$$

$$t_{bcast}^{nbin} \approx \lceil log_n P \rceil \cdot (\zeta(n-f-1)g + (n+1)o + L + (m-1)nG) \tag{III.4}$$

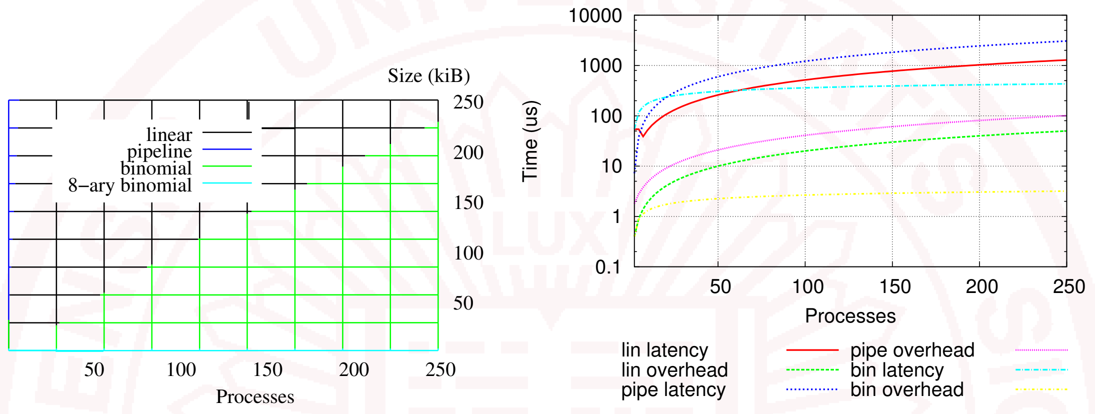
The LogP communication graphs are shown in Figure III.6. We omitted the G parameter in order to keep the figure legible.

Figure III.7(a) shows the map of the best performing algorithms (of our selection) for the broadcast operation. This map selects the algorithm by their communication latency. The map shows that the pipelined algorithm performs well on small communicators and large messages, the binomial algorithm should be chosen for larger communicators and smaller messages and the 8-way binomial algorithm for very small messages. The remaining cases seem well covered by the linear algorithm. The authors are aware that a segmented pipelining algorithm could perform better than the linear algorithm, but generating algorithm-selection maps is not focus of this work. Figure III.7(b) shows the communication latency and the CPU overhead of a $64kiB$ broadcast. This shows that the latency-wise selection would be the pipelined algorithm for small communicators,



(a) Linear (b) Pipelined with 5 Segments (c) Binomial Tree

Figure III.6: LogGP Model Graphs for different Broadcast Algorithms



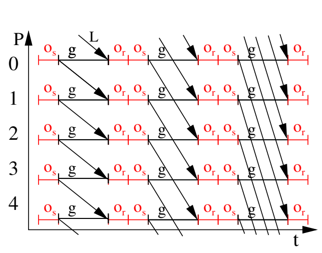
(a) Best performing algorithms depending on communicator and message-size (b) Latency and Overheads for a $64kiB$ broadcast

Figure III.7: Latency and Overlap for different broadcast Algorithms

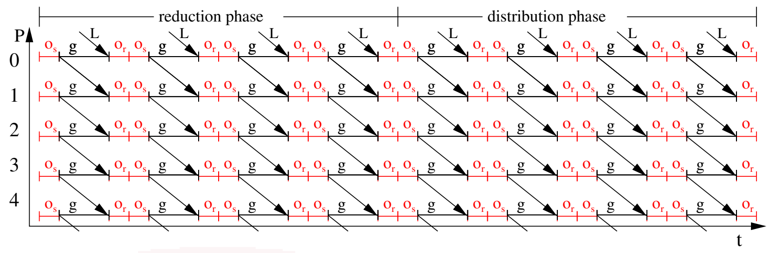
the linear algorithm for mid-sized communicators and the binomial algorithm for large communicators. However, the decision is much easier if we base it on the CPU overhead. The binomial algorithm has the smallest number of message-sends on the critical path and is thus optimal in most cases (a decision map like in Figure III.7(a) based on CPU overhead would show the binomial algorithm for nearly all combinations and is thus omitted).

3.1.2 Algorithms for MPI_Allreduce

Pjesivac-Grbovic mainly discusses three different MPI_Gather: recursive doubling, segmented ring and Rabenseifner's algorithm. We chose to model the dissemination, n-way dissemination [23] and segmented ring algorithm for our analysis. Reduction operations have additional CPU overhead, the computation that is performed in the data. We assume a linear cost model depending on the data size: $t = \gamma \cdot m$. The system-specific constant γ is the measured memory bandwidth (we assume that the memory access time, not the computation itself, is the limiting factor). The LogGP models



(a) Dissemination



(b) Ring with 5 Segments

Figure III.8: LogGP Model Graphs for different Allreduce Algorithms

for the three algorithms are:

$$t_{allreduce}^{diss} = \lceil log_2 P \rceil \cdot (2o + L + (m-1)G + m\gamma) + \zeta((\lceil log_2 P \rceil - f - 1))g \quad \text{(III.5)}$$

$$t_{allreduce}^{ndiss} = \lceil log_n P \rceil \cdot ((n+1)o + L + (m-1)nG + nm\gamma) + \zeta((\lceil log_n P \rceil - f - 1))g \quad \text{(III.6)}$$

$$t_{allreduce}^{ring} = (2P-2) \cdot (2o + L + (\frac{m}{P} - 1)G) + \zeta(2P - 2 - f)g + (P-1) \cdot \frac{m}{P}\gamma. \quad \text{(III.7)}$$

The LogP communication graphs are shown in Figure III.8. We omitted the G and γ parameters in order to keep the figure legible. The 8-way dissemination algorithm degenerates to a linear algorithm for 5 processes; the figure is not shown.

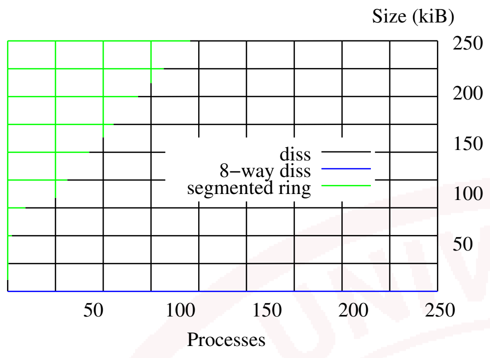
Figure III.9(a) shows the algorithms with the lowest latency for the allreduce operation. The segmented ring algorithm is preferable for small communicators and relatively large messages while the dissemination algorithm covers the rest nicely. The 8-way dissemination algorithm should be used for small messages only. However, a selection based on CPU overhead would prefer the segmented ring algorithm over dissemination for mid-sized and large messages because of the better balance of the computation. The dissemination performs better for small messages (that can not be segmented efficiently). Figure III.9(b) shows the latency and overhead for a reduction if $64kiB$ data. It shows that overhead of the ring algorithm is much lower than for the other choices.

4 Conclusions

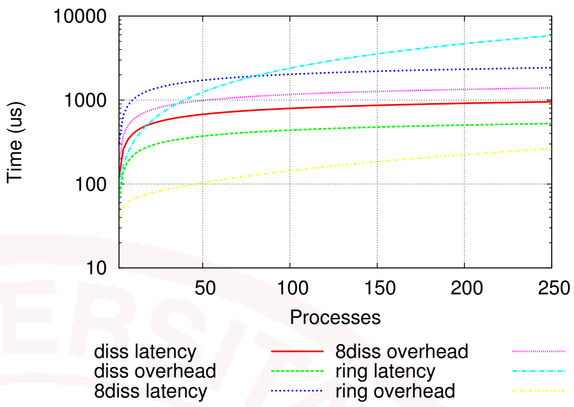
"Mathematics seems to endow one with something like a new sense." – Charles Darwin, (1809-1882) English Scientist, Royal Medal 1853

We introduced new sparse collective operations to support arbitrary communication patterns. This class of collective operations is especially useful for nearest neighbor communication and other sparse but relatively persistent communication techniques. This improves scalability in comparison to the current Alltoallv approach and also simplifies programming significantly.

A new set of non-blocking collective operations is proposed to enable overlapping of computation and communication also for high-level communication patterns. All collective operations can



(a) Best performing algorithms depending on communicator and message-size



(b) Latency and Overheads for a $64kiB$ allreduction

Figure III.9: Latency and Overlap for different Allreduce Algorithms

be supported in a non-blocking manner and we demonstrated the potential performance benefit with MPI benchmarks that showed that the CPU is up to 97% idle during the collective operations. This CPU time can theoretically be gained back and used for useful computation.

We argue that it is more useful to optimize first for low CPU overhead and later for lower latency if collective operations can be overlapped. We model several collective algorithms in the LogfGP model in order to understand the CPU overheads better. We design new selection criteria and show selection maps based on CPU overhead for the two collective operations broadcast and allreduce. Our model reveals new insight into the distribution and total duration of CPU overheads and interruptions. We also see that the CPU must be activated frequently in order to ensure progression of the collective algorithm.

Chapter IV

Optimized Reference Implementation

"One test result is worth one thousand expert opinions." – Wernher von Braun, (1912-1977) German Physicist, National Medal of Science 1975

Concurrency of computation and communication is an important feature of non-blocking collective operations. This means that the communication should ideally progress independently of the computation. However, InfiniBand^{TM} and many other networks do not allow a full independence because communication schedules and protocols have to be executed on the main CPU, "stealing" cycles from the application. There are two different options to implement support for non-blocking collective operations and progression. The first way is to process the blocking collective operation in a separate thread and the second way is to implement it on top of non-blocking point-to-point operations. We will evaluate both implementations in the following. We use a library approach, e.g., both variants are implemented in a library with a standardized interface which is defined in [18]. This enables us to run identical applications and benchmarks with both versions.

This chapter summarizes and extends results from the articles "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI" [11], "Accurately Measuring Collective Operations at Massive Scale" [15], "Optimizing non-blocking Collective Operations for InfiniBand" [9] and "Message Progression in Parallel Computing - To Thread or not to Thread?" [8]. The next section describes the portable reference implementation of all collective operations. Section 2 defines a simple benchmarking scheme for overhead measurements and Section 3 describes a low-overhead implementation for the InfiniBand^{TM} network. Finally, (threaded) progression issues, an improved microbenchmarking scheme and several operating system effects are discussed in Section 4.

1 A collective Communication Library: LibNBC

"Simplicity is prerequisite for reliability." – Edsger Dijkstra (1930-2002), Dutch computer scientist, Turing Award 1972

The NBC (Non-Blocking Collective) library (short: LibNBC) is a portable implementation of the non-blocking collective operations proposed as an addition to MPI [18]. The main advantage of non-blocking collective operations is that they offer a high flexibility by enabling the programmer to use non-blocking semantics to overlap computation and communication, and to avoid pseudo-synchronization of the user application.

LibNBC is based on MPI and written in ANSI C to enable high portability to many different parallel systems. Some MPI implementations offer the possibility to overlap communication and computation with non-blocking point-to-point communication (e.g., Open MPI [76]). Using LibNBC with such a library enables overlap for collective communications. However, even if the overlap potential is low, most MPI implementations can be used to avoid pseudo-synchronization [106].

The main goal of LibNBC is to provide a portable and stable reference implementation of non-blocking collective operations on top of MPI. LibNBC supports all collective operations defined in MPI and is easily extensible with new operations. The overhead added by the library is minimal so that a blocking execution (i.e., NBC_Ibcast immediately followed by a NBC_Wait) has similar performance as the blocking collective operation (MPI_Bcast). However, this is not generally true because other factors such as overhead and progression are more important for the execution of non-blocking collective operations. The potential to overlap communication and computation is as high as the lower communication layers support and support for special hardware operations can be added easily.

1.1 Implementation

The most basic version of LibNBC is written in ANSI C (C90) to compile with `gcc -W -Wall -ansi -pedantic` without warnings or errors. The library uses collective schedules to save the necessary operations to complete an MPI collective communication. These schedules are built with helper functions and executed by the scheduler to perform the operation.

1.1.1 The Collective Schedule

A collective schedule is a process-specific "execution plan" for a collective operation. It consists of all necessary information to perform the operation. Collective schedules are designed to support any collective communication scheme with the usual data dependencies. A schedule can consist of multiple rounds to model the data dependencies. Operations in round r may depend on operations in rounds $i \leq r$ and are not executed by the scheduler before all operations in rounds $j < r$ have been finished. Each collective operation can be represented as a series of point-to-point operations with one operation per round. Some operations are independent of each other (e.g., the data sent in a simple linear broadcast), and some depend on previous ones (e.g., a non-root, and non-leaf process in a tree implementation of MPI_Bcast has to wait for the data before it can send to its children). Independent operations can be in the same round and dependent operations have to be

```
schedule ::= size {round rdelim} round rend | size rend
round ::= num {type targs}

size ::= size of the schedule (int)
num ::= number of operations in round (int)
type ::= operation type (enum NBC_Fn_type)
targs ::= operation specific arguments (struct NBC_Args_<type>)
rdelim ::= '1'; (char), indicates next round
rend ::= '0'; (char) indicates next round
```

Listing IV.1: EBNF-like syntax description of the Schedule Array

in the right order in different rounds.

Definition IV.1: *A collective schedule is a process specific plan to execute a collective operation. It consists of $r \geq 1$ rounds and $o \geq 0$ operations.*

Definition IV.2: *A round is a building block of a collective schedule which may consist of $o \geq 0$ operations. Rounds are interdependent, round r will only be started if all operations in round $r - 1$ are finished. Round 0 can be started immediately.*

Definition IV.3: *An operation is the basic building block of a collective schedule. It is used to progress collective operations. Operations are grouped in rounds and executed by the scheduler. Possible data dependencies between operations are expressed in their grouping into rounds.*

This schedule design enables coarse-grained dependencies which allow some degree of parallelism and an optimized implementation of the schedule. It is possible to implement automatic and transparent segmentation and pipelining.

1.1.2 Memory Layout of a Schedule

The easy round-based design of a schedule allows the schedule to be stored contiguously in memory. This design is cache-friendly; the fetch of the first operation of the round will most likely also load the following operations. This special design has been used previously to optimize MPI_Barrier synchronization for InfiniBand^{TM} where the calculation of the communication peers during the execution was too costly and had been done in advance [23].

The first element of each schedule is the size in bytes, stored as integer value. At least one round follows the size element. A round consists of a number of operations ($= o$) and o operation argument structures. The operation argument structures consist of all operation-specific arguments and vary in size. A delimiter follows the last operation of each round. A delimiter may be followed by another round or indicates the end of the schedule. An EBNF-like syntax description of the schedule array is given in Listing IV.1.

1.1.3 Identifying a Running Collective Operation

A handle (NBC_Handle), which is similar to an MPI_Request, is used to identify running collective operations, which are called instances in the following.

Definition IV.4: *An instance of a collective operation is a currently running operation which has not been completed by Test or Wait.*

The handle identifies the current state, the schedule, and all necessary information to progress the collective operation. Each handle is linked to a user communicator. Each user communicator is duplicated at the first use into a so called shadow communicator. All communication done by LibNBC is performed on the shadow communicator to prevent tag collisions on the user communicator. A tag, specific to each communicator-instance pair, is also saved at the handle to distinguish between different ongoing collective operations on the shadow communicator. The handle holds all information related to outstanding requests (in the current implementation MPI_Requests, but other (e.g., hardware specific) request types are also possible) and a void pointer to store arbitrary information needed by the collective routines (e.g., temporary buffers for reductions).

The current state of an instance is determined by the pending requests and the current round in the schedule. The open requests are attached to the handle and the current round is saved as an integer offset (bytes) in the schedule array. This means that a single schedule could be used by many instances at the same time (i.e., can be attached to multiple handles) and may remain cached at the communicator for future usage (this is not implemented yet).

1.1.4 The Non-Blocking Schedule Execution

The execution of a schedule is implemented in the internal function NBC_Start. It checks if a shadow communicator exists for the passed user communicator and creates one if necessary. MPI_Attributes attached to the user communicator are used to store communicator specific data (shadow comm and tag). Tags are reset to 1 if they reach 32767 (the minimum tag-size in MPI) or if a new shadow communicator is constructed. The NBC_Start function sets the schedule offset at the handle to the first round and starts the round with a call to NBC_Start_round.

The function NBC_Start_round issues all operations for the next round (indicated by the handle's offset). All requests returned by non-blocking operations (like Isend, Irecv) are attached to the handle and all blocking (local) operations (like Copy, Operation) are executed at this point. The progress function is called to check if the round can already be finished (e.g., if it only had local operations).

1.1.5 Progressing a Non-Blocking Instance

We differentiate two kinds of progress. The first kind of progress is done inside the library (e.g., to send the next fragments). The MPI progress may be asynchronous (e.g., separate thread), or synchronous (user has to call MPI_Test to achieve progress) depending on the MPI implementation. The second progress is the transition from one round to another. We only analyze operations/algorithms that have a single round in this section. Algorithms with multiple rounds are discussed in Section 4.

NBC_Test tests all outstanding point-to-point requests for completion. A successful completion of all requests means that the active round is finished and the instance can be moved to the next round. The whole operation is finished if the current round is the last round and NBC_OK is returned. If a next round exists, NBC_Start_round is called after adjusting the handles offset to the new round. The function returns NBC_CONTINUE if there are still outstanding requests in this round.

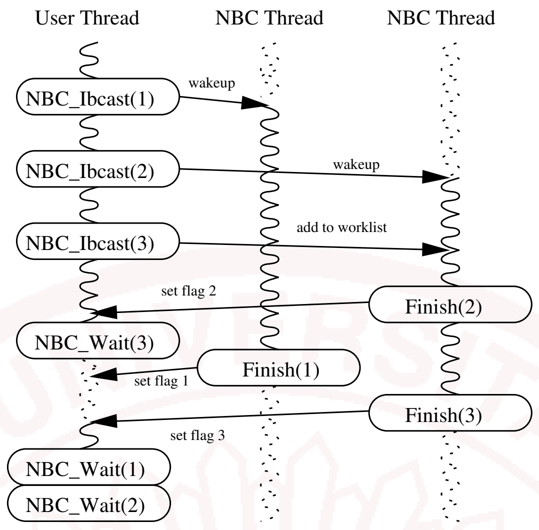


Figure IV.1: Execution of NBC calls in separate threads

The complete internal and external APIs to use and extend LibNBC are described in Appendix C

1.2 Evaluating MPI Collectives and Threads

The threaded implementation, based on the pthread interface, is able to spawn a user-defined number of communication threads to perform blocking collective operations. It operates using a task-queue model, where every thread has its own task queue. Whenever a non-blocking collective function is called, a work packet (containing the function number and all arguments) is placed into the work queue of the next thread in a round robin scheme.

The worker threads could either poll their work queue or use condition signals to be notified. Condition signals may introduce additional latency while constant polling increases the CPU overhead. We will analyze only the condition wait method in the next section because short experiments with the polling method showed that it is worse in all regards. Since there must be at least one worker thread per MPI job, at most half of the processing cores is available to compute unless the system is oversubscribed.

Whenever a worker thread finds a work packet in its queue (either during busy waiting or after being signaled), the thread starts the corresponding collective MPI operation and sets a flag after its completion. All asynchronous operations have to be started on separate communicators (mandated by the MPI standard). Thus, every communicator is duplicated on its first use with any non-blocking collective and cached for later calls. Communicator duplication is a blocking collective operation in itself and causes matching problems when it's run with threads (cf. Gropp et al. in [87]). The communicator duplication has to be done in the user thread to avoid race conditions, which makes the first call to a non-blocking collective operation with every communicator block. All subsequent calls are executed truly non-blocking.

When the user calls NBC_Test, the completion flag is simply checked and the appropriate return

```
MPI_Gather(...); /* warmup */
MPI_Barrier(...); /* synchronization */
t0 = MPI_Wtime(); /* take time */
for (i=0; i<reps; i++) {
  MPI_Gather(...); /* execute benchmark */
}
t1 = MPI_Wtime(); /* take time */
MPI_Barrier(...);
time = t1-t0;
```

Listing IV.2: MPPTEST Benchmark Scheme

code generated. A call to NBC_Wait waits on a condition variable. The next section describes a scheme to benchmark the CPU overhead and the latency of non-blocking collective operations.

2 A Microbenchmarking Scheme for Non-Blocking Collective Communication

"Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest." – Isaac Asimov, (1920-1992) Russian Chemist

Several benchmarking studies of collective operations exist for different systems [40, 170, 178]. In this section, we discuss and analyze different established measurement methods for collective operations on parallel computing systems and point out common systematic errors. Our measurement method is derived from the SKaMPI benchmarks [215] and is universal and not limited to point-to-point based methods or specific algorithms. The following section discusses other established benchmark methods.

2.1 Related Work

Different benchmark schemes have been proposed. Currently known methods can be divided into three groups. The first group synchronizes the processes explicitly with the use of synchronization routines (i.e., MPI_Barrier). The second scheme, proposed by Worsch et al. in [215], establishes the notion of a global time and the processes start the operation synchronously. The third scheme assesses the quality of a collective implementation by comparison to point-to-point operations [188] and is thus limited to algorithms using point-to-point messages. We investigate several publicly available benchmarks in the following and characterize them in the three groups.

MPPTEST implements ideas on reproducible MPI performance measurements from [86]. As described in the article, the operation to measure is executed in a warm-up round before the actual benchmark is run. The nodes are synchronized with a single MPI_Barrier operation before the operation is run N times in a loop. A pseudo-code is shown in Listing IV.2. Only the time measurements at rank 0 are reported to the user.

```
t0 = TIMER(); /* take time */
for (i=0; i<reps; i++) {
    /* execute benchmark */
  MPI_Alltoall(...); }
t1 = TIMER(); /* take time */
MPI_Barrier(...);
time = t1-t0;
```

Listing IV.3: MPBench Benchmark Scheme

```
for(i=0; i<numbarr; i++)
    MPI_Barrier(...);
t0 = MPI_Wtime(); /* take time */
for (i=0; i<reps; i++) {
    /* execute benchmark */
  MPI_Alltoall(...);
}
t1 = MPI_Wtime(); /* take time */
time = (t1-t0)/reps;
```

Listing IV.4: Intel MPI Benchmark Scheme

MPBench was developed by Mucci et al. [154]. MPBench does not synchronize at all before the benchmarks. Rank 0 takes the start time, runs N times the collective operation to benchmark and takes the end time. A pseudo-code is shown in Listing IV.3.

The timer can use the RDTSC CPU instruction or gettimeofday(). Time measurement is only performed and printed on rank 0.

Intel MPI Benchmarks (formerly Pallas MPI benchmarks [163]), measure a wide variety of MPI calls including many collective functions. The code issues a definable number of MPI_Barrier operations before every benchmark and measures the collective operation in a loop afterwards. The time needed to execute the loop is taken as a measurement point. The scheme is shown in Listing IV.4. The benchmark prints minimum, maximum and average time over all processes.

SKaMPI The SKaMPI benchmark uses a time-window based approach, described in [215], that ensures that all processes start the operation at the same time. No explicit synchronization is used and the times are either reported per process or cumulative.

2.2 Systematic Errors in Common Measurement Methods

Benchmarking collective operations is a controversial field. It is impossible to find a single correct scheme to measure collective operations because the variety of real-world applications is high. Every benchmark may have its justification and is not erroneous in this case. However, microbenchmarks are often used to compare implementations and to model the influence of the communication to several applications. Thus, a benchmark should represent the *average* or at least the majority of applications. Our model application for this work is a well balanced application that issues at least two different collective operations in a computational loop (cf. [172]). This model application would benefit from well balanced collective operations that do not introduce process skew. The following paragraphs describe common systematic errors done in the measurement of collective operations. This section is concluded with the selection of a benchmark method. We begin with a simplification of the LogP model to describe common mistakes.

A simplified LogP model, derived from Culler's original LogP model [61], is used to model the network transmissions and effects in collective communication. The LogP model uses four parameters to describe a parallel system. The parameter **L** models the network latency and **g** is the time that has to be waited between two packets. The CPU overhead **o** does not influence the network transmission and is thus omitted in our simplified model. The number of participating processes **P** is constantly four in our examples.

2.2.1 Implementation Assumptions

The implementation of collective communication operations is usually not standardized to provide as much optimization space as possible to the implementer. Thus, any point-to-point algorithm or hardware-supported operation, that offers the functionality defined in the interface, is a valid implementation. Some research groups used elaborate techniques (e.g., hardware optimization and/or specialized algorithms) to optimize collective communication on different systems (cf. [23], [207, 218]). A portable benchmark that uses the collective interface, for example MPI, can not make any assumptions about the internal implementation.

2.2.2 Results on multiple processes

A second problem is that benchmarks are usually providing a single number to the user, while all processes benchmark their own execution time. Some benchmarks just return the time measured on a single node (e.g., the rank 0), some use the average of all times and some the maximum time of all processes. The decision which time to use for the evaluation is not trivial it is even desirable to include the times of all ranks in the evaluation of the implementation. Worsch et al. define three schemes to reduce the times to a single number: (1) the time needed at a designated process, (2) maximum time on all processes and (3) the time between the start of the first process and the finish of the last. This list can be extended further, e.g., (4) the average time of all processes or (5) the minimum time might play a role and is returned by certain benchmarks.

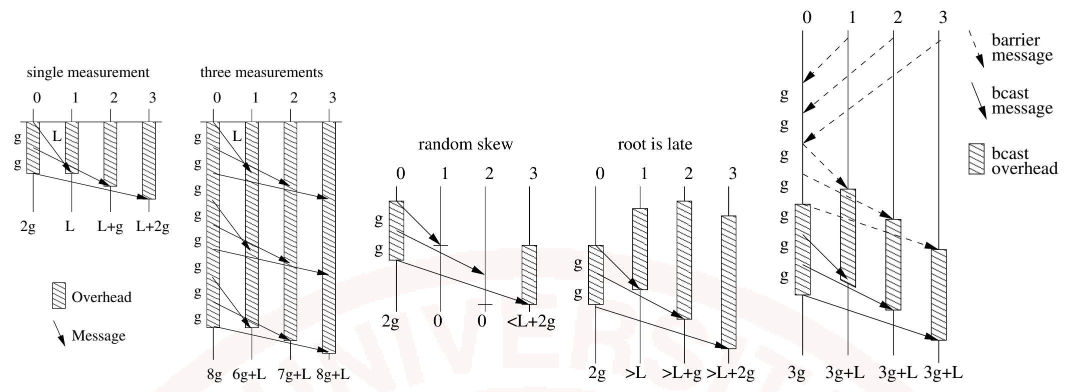
2.2.3 Pipelined Measurements

Another source for systematic errors are pipelining effects that occur when many operations are executed in a row. A common scheme is to execute N operations in a loop, measure the time and divide this time by N. This scheme was introduced to avoid the relative high inaccuracy of timers when short intervals are measured. We show in Section 2.3 that this is not necessary for high precision timers. An example LogP modeling for MPI_Bcast with root 0, implemented with a linear scheme, is shown in Figure IV.2(a).

A single execution is much more likely to model the behavior of real applications (multiple successive collective operations should be merged into a single one with more data). Both schemes result in different execution times, e.g., the worst-case (maximum among all nodes) returned latency for a single execution is $L + 2g$ for a single operation and $(8g + L)/3$ for three successive operations. The pipelined measurement tends to underestimate the latency in this example.

2.2.4 Process Skew

The LogP models in the previous paragraph assumed that the first operation started at exactly the same (global) time. This is hardly possible in real parallel systems. The processes often arrive at the benchmark in random order and at undefined times. Process skew is influenced by operating system noise [27, 109] or other collective operations (cf. Figure IV.2(a) where rank 0 leaves the



(a) Pipeline Effects for a linear Broadcast Implementation (b) Influence of Process Skew on a Broadcast Benchmark (c) Possible Effect of Barrier Synchronization on Measurements

Figure IV.2: Common Errors in measuring collective operations

operation after $2g$ and rank 3 finishes only after $L + 2g$). A LogP example for the influence of process skew is shown in Figure IV.2(b). The left side shows a random skew pattern where rank 1 and 2 arrive relatively late and do not have to wait for their message while the right side shows a situation where the root (rank 0 in this example) arrives late and all ranks have to wait much longer than usual. Process skew can not be avoided and is usually introduced during the runtime of the parallel program. This effect is well known and several benchmarks use an MPI_Barrier before the measurement to correct skew.

2.2.5 Synchronization Perturbation and Congestion

The MPI_Barrier operation has two problems, the first one is that this operation may be implemented with any algorithm because it only guarantees that all processes arrived before the first leaves the call but not that the processes leave at the same time (i.e., the barrier operation may introduce process skew). The second one is that it may use the same network as other collective operations, which may influence the messages of the investigated collective operation. An example with a linear barrier is shown in Figure IV.2(c).

2.2.6 Network Congestion

Network congestion can occur if multiple operations are started successively or synchronization messages interfere with the measurement. This influences the measured latencies.

2.2.7 Selecting a Benchmark Scheme

The SKaMPI benchmarks avoid most of the systematic errors with the window-based mechanism [215]. This mechanism relies heavily on the assumption, that the difference between the local clocks does not change (drift) or changes in a predictable way (can be corrected). We analyze the clock drift in the following sections. Another problem might be the variation in the latency of point-to-point messages. This variation is also analyzed and a new fast point-to-point synchronization scheme is presented. Furthermore, we propose and implement a new scalable group synchronization algorithm which scales logarithmically instead of the linear SKaMPI approach.

2.3 Measurements in Parallel Systems

Several restrictions apply to measurements in parallel systems. One of the biggest problems is the missing time synchronization. Especially in cluster systems, where every node is a complete and independent system with its own local time source, one has to assume that potentially all processes of a parallel job run at (slightly) different clock speeds. However, to perform the necessary measurements, we need to synchronize all clocks or have at least the time offsets of all processes to a global time. It can also not be assumed that the processes start in any synchronous state. To ensure portability, MPI mechanisms have to be used to synchronize. However, collective operations semantics do not guarantee any timing, thus, we need to synchronize the processes with point-to-point operations. Those operations do not guarantee timing either but are less complex than collective operations (no communication patterns). We analyze local time sources and their accuracy in the following. This is followed by an analysis of the clock skew in parallel systems and the distribution of latencies. This analyzes are used to derive a novel and precise synchronization scheme in the next section.

2.3.1 Local Time Measurement

All time sources in computing systems work in a similar way: They use a crystal that oscillates with a fixed frequency and a register that counts the number of oscillations since a certain point in time (for example the system startup). However, the way to access this information can be different. Some timing devices can be configured to issue an interrupt when the register reaches a certain value or overflows and others just enable the programmer to read the register.

An important timer in a modern PC is the Real Time Clock (RTC) which is powered by a battery or capacitor so that it continues to tick even when the PC is turned off. It is used to get the initial time of the day at system startup, but since it is often inaccurate (it is optimized for low power consumption, not for accuracy), it should not be used to measure short time differences, like they might occur in benchmarking scenarios.

Another time-source is the Programmable Interval Timer (PIT). It can be configured to issue interrupts at a certain frequency (Linux 2.6 uses 1000.15 Hz). These interrupts are used to update the system time and perform several operating system functions. When using the system time (for example via `gettimeofday()`) one must be aware that the returned value might be influenced by `ntp` or other time-synchronization mechanisms and that there is a whole software stack behind this simple system call which might add additional perturbation (i.e., interrupts, scheduling, etc.).

The resolution of the discussed time sources is not high and not accurate enough for the benchmarking of fast events like single message transmissions. Thus, many instruction set architectures (ISA) offer calls to read the CPU's clock register which is usually incremented at every tick. For example the x86 and x86-64 ISAs offer the atomic instruction `RDTSC` call [108] to read a 64 bit CPU tick register. In fact most modern ISAs support similar features. The resolution of those timers is usually high (e.g., $0.5ns$ on a 2GHz system). It has to be noted that this mechanism introduces several problems on modern CPUs. The first issue is caused by techniques that dynamically change the CPU frequency (e.g., to save energy) such as "Intel-SpeedStep" or "AMD-PowerNow". Changing the CPU clock results in invalid time measurements and many other problems. Thus, we recommend to disable those mechanisms in cluster systems. A second problem, called “process hopping”, may occur on multi-processor systems where the process is re-scheduled between multiple CPUs. The counters on the CPUs are not necessarily identical. This might also influence the measurement. This problem is also minor because most modern operating systems (e.g., Linux 2.6) offer interfaces to bind a process to a specific CPU (e.g., CPU affinity), and in fact, most operating systems attempt to avoid “process hopping” by default.

2.3.2 Clock Skew and Network Latencies

The crystals used for hardware timers are not ideal with respect to their frequency. They may be a little bit slower or faster than their nominal rate. This drift is also temperature dependent and has been analyzed in [124] and [155]. It was shown that this effect is significant enough to distinguish / identify single computers and sometimes even the timezone which they are located in. Due to other effects, such as the NTP daemon that synchronizes every 11 minutes (local time!) by default, the clock difference between two nodes may behave unpredictable, which will lead to erroneous results. Therefore the usage of a software independent clock like the TSC is generally a viable alternative. Of course those effects could have a negative influence on collective benchmarks which rely on time synchronization, especially if the synchronization is done only once before a long series of benchmarks. In our experiments it turns out that two clocks (even on identical hardware) always run at slightly different speeds. Therefore we analyzed the clock skew between various nodes in a cluster system over a long period of time.

Similar to the clocks, that do not run totally synchronously, we do also expect a variance in the network transmission parameters for different messages. The most important parameter for benchmarks and synchronization is the network latency (or round-trip-time (RTT) in ping-pong benchmarks). Thus, we have to analyze the variance of RTT for different networks.

We used a simple ping-pong scheme to determine the RTT: Rank 1 sends its local time t_1 as an eight byte message with a blocking send to rank 2. As soon as rank 2 has completed the corresponding recv, it sends its local time t_2 back to rank 1. After rank 1 is finished receiving that timestamp it checks his local time t_3. To use a portable high-precision timing interface and to support many network interconnects, the benchmark scheme was implemented in the Netgauge performance measurement framework [12]. A pseudo-code is shown in Listing IV.5. The difference between the first and the second timestamp obtained by node 1 is the *roundtrip time* ($t_{rtt} \leftarrow t_3 - t_1$). On our x86 systems, we used `RDTSC` in the `take_time()` macro because this gives us a high resolution and accuracy. However, we double-checked our findings with MPI_Wtime (which uses `gettimeofday()` or similar functions) to avoid common pitfalls described in Section 2.3.1.

This measurement was repeated 50,000 times, once every second over a period of 14 hours. We gather data to get information about the RTT distribution and also the clock skew between two nodes. Thus, we define the difference between t_1 and t_2 as *clock-difference* ($t_{diff} \leftarrow |t_1 - t_2|$) and collect statistical data for the clock differences too.

The benchmark results, a histogram of 50,000 RTT measurements in 200 uniform bins, for the latency (RTT/2) distributions of InfiniBand, Myrinet, and Gigabit Ethernet are shown in Figure 2.3.2.

```
if (rank == 0) {
  t1 = take_time();
  module->send(1, &t1, 8);
  module->recv(1, &t2, 8);
  t3 = take_time();
} else {
  module->recv(0, &t1, 8);
  t2 = take_time();
  module->send(0, &t2, 8);
}
```

Listing IV.5: The RTT Benchmark

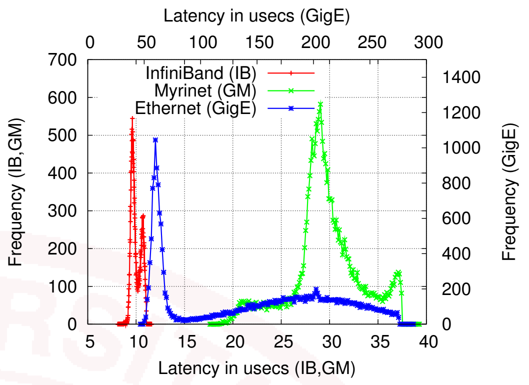


Figure IV.3: Distribution of RTT

The clock skew results of 6 different node pairs on our test systems are shown in Figure IV.4(a). We see that the clock difference behaves relatively linear and can thus be corrected by a linear error function. That way synchronization error below 1 ppm (parts per million) can be achieved. Without linear error correction the error can be up to 300 ppm. The detailed methodology that is used to determine the clock difference is described in the next section.

2.4 Time Synchronization

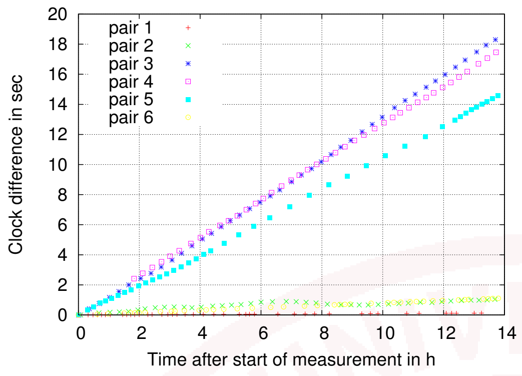
We describe a new time synchronization schemes that bases on our analysis of clock skew and latency variation. We begin by defining a method to synchronize two processes and derive a scalable scheme to synchronize large process groups. We use this schemes to synchronize the processes in our collective benchmark NBCBench [11] that uses a window-based benchmark scheme.

2.4.1 Synchronizing two Processes

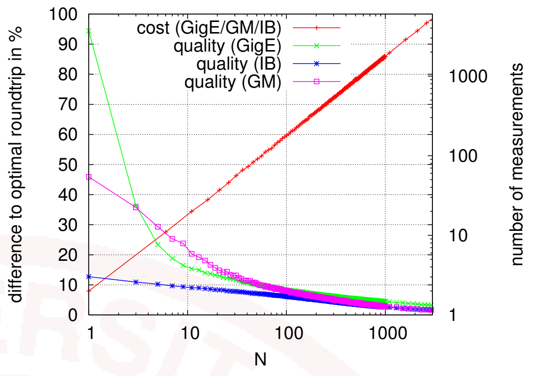
A clock synchronization between two peers is often accomplished with a ping-pong scheme similar to the one described above: Two nodes calculate their clock difference so that the client node knows his clock offset relative to the server node. This offset can be subtracted from the client's local time when clock synchronization is required. However this procedure has certain pitfalls one has to be aware of.

Many implementations of the scheme described above, for example the one found in the SKaMPI code, use MPI_Wtime to acquire timestamps. This is of course the most portable solution and works for homogeneous nodes as well as heterogeneous ones. But you can not be sure which timing source is used by MPI_Wtime, for example the usage of `gettimeofday()` is, due to its portability, quite likely. But the clock tick rate of this clock can vary (cf. Section 2.3.1). We showed in Section 2.3 that a linear correction can be used for nodes running at different clock speeds if software errors (e.g., NTP) are avoided.

We also showed in Section 2.3 that measured network latencies are varying with an unpredictable distribution. The effect of this pseudo-random variation of the latency to the clock synchronization has to be minimized. Using the average or median of the clock difference, like many



(a) Clock Drift for Different Pairs



(b) Influence of N on Quality

Figure IV.4: Different time-synchronization parameters

codes do today, is not a viable option because, as you can see in the histograms, they vary a lot. A better approach is the measurement of roundtrip time and clock difference at the same time and only use the clock differences obtained in measurements which showed a latency below a certain threshold.

The distribution of roundtrip times can not be known in advance. That implies that the threshold can not be selected easily. We chose a different approach to ensure accurate measurements. For the measurements in Section 2.3, we used only the 25% of the results that had the smallest roundtrip time. However, this requires a huge number of measurements to be conducted every time which makes this scheme unusable for online measurements. For this purpose, we developed another approach. We conduct as many measurements as we needed so that the minimal observed roundtrip time does not become smaller for N consecutive measurements.

While this scheme is guaranteed to converge, it is not possible to predict how many measurements have to be conducted for a certain N. It is also not possible to select an N mathematically that ensures a certain quality of the observed "minimal" roundtrip time. Thus, we performed a simulation to find suitable values for N for different interconnection networks.

The simulation takes a random roundtrip time from our list [1] and checks if it was bigger than the smallest one observed in this run. If this condition is met N consecutive times the run is completed and we compute the difference between the smallest RTT in our whole dataset and the one observed in the current run. The averaged values for several networks are graphed in Figure IV.4(b). It shows that the quality of the measurement and the measurement costs (i.e., the number of measurements) grow relatively independent of the network with N (the costs for the different networks look nearly identical on the double-logarithmic plot, thus we plotted it only once). However, the quality of the results grows pretty fast for small N and seems to "saturate" around 5%. To support every network, we chose $N = 100$, which has an error of less than 10% for our tested networks and converges after approximately 180 measurements.

[1] we used the 50,000 RTTs gathered as described in Section 2.3

2.4.2 Scalable Group Time Synchronization

Collective time synchronization means that all nodes know the time difference to a single node so that every node can compute a global time locally. Rank 0 is conveniently chosen as global time source, i.e., after the time synchronization, every rank knows the difference between its own clock and rank 0's clock. This enables globally synchronous events. For example, rank 0 can broadcast a message that some function is to be executed at its local time x. Every rank can now calculate its local time when this operation has to be executed.

A common scheme to synchronize all ranks is to start the point-to-point synchronization procedure between rank 0 and every other rank. This disadvantage of this scheme is that it takes $P-1$ synchronization steps to synchronize P processes. We propose a new and scalable time synchronization algorithm for our scalable benchmark. Our algorithm uses $\lceil log_2 P \rceil$ communication steps to synchronize P processes.

The algorithm divides all ranks in two groups. The first group consists of the maximum power of two ranks, $t = 2^k \; \forall \, k \in N \, , t < P$ beginning from rank 0 to rank $t-1$. The second group includes the remaining ranks t to $P-1$.

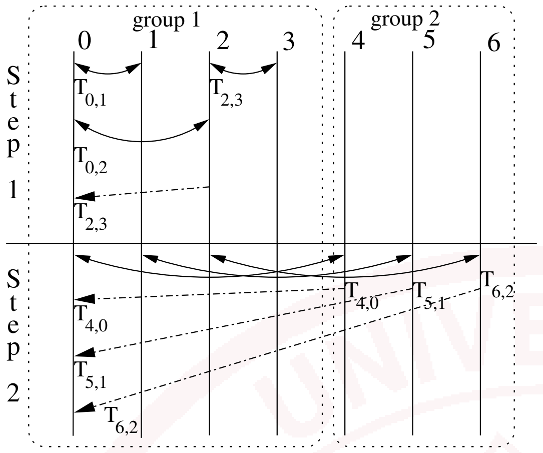
The algorithm works in two steps, the first group synchronizes with a tree-based scheme in $log_2 t$ synchronization rounds. The point-to-point scheme, described in Section 2.4.1 is used to synchronize client and server. Every rank r in round r acts as a client if $r \; mod \; 2^r = 0$ and as a server if $r \; mod \; 2^r = 2^{(r-1)}$. All clients r use rank $r + 2^{(r-1)}$ as server and all servers rank $r - 2^{(r-1)}$ as client. All client-server groups do the point-to-point synchronization scheme in parallel. Every server gathers some time difference data in every round. This gathered data has to be communicated at the end of every round. To be precise, a server communicates $2^{(r-1)} - 1$ time differences to its client at the end of every round. The clients receive the data and update their local differences to pass them on in the next step. After $log_2 t$ rounds, rank 0 knows the time differences to all processes in group 1.

In the second step, all processes in group 2 choose peer $r - t$ in group 0 to synchronize with. All nodes in group 2 synchronize in a single step with their peers and send the result to rank 0 which in turn calculates all the time offsets for all nodes and scatters them accordingly. After this step, all nodes are time synchronized, i.e., know their time difference to the global clock of rank 0. The whole algorithm for an example with $P = 7$ is shown in Figure IV.5(a).

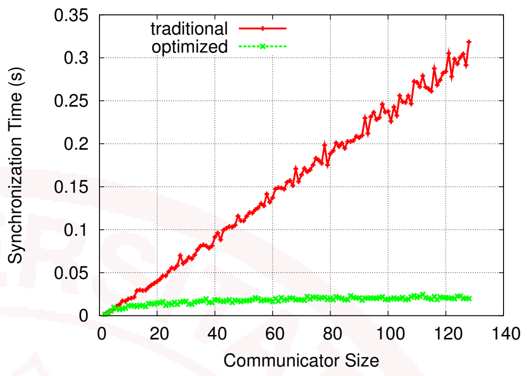
Figure IV.5(b) shows the difference in synchronization time between a linear scheme and the proposed tree-like algorithm. The benchmark, which measures the synchronization time at rank 0, was run on the *Odin* cluster at Indiana University, which consists of 128 Dual Opteron dual-core nodes. To simulate a real application, we used all available cores of the machine. The synchronization time is greatly reduced (up to a factor of more than 16 for 128 processes) with the new scheme. The gain of the new method is even higher for Gigabit Ethernet.

2.5 Measuring Non-Blocking Collectives

We developed a micro-benchmark to assess the performance and overlap potential of the MPI threaded and point-to-point implementation of non-blocking collective operations. This benchmark uses the interface described in Section 1. For a given collective operation, it measures the



(a) Synchronization Method



(b) Synchronization Time over IB

Figure IV.5: New Time Synchronization Method

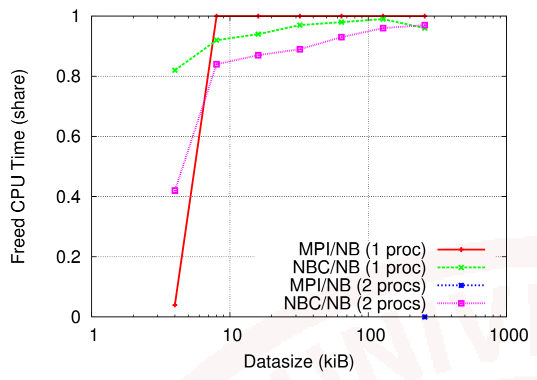
time to perform the blocking MPI collective, the non-blocking collective in a blocking way (without overlap) and the non-blocking collective interleaved with busy loops to measure the potential computation and communication overlap. In addition, the benchmark measures the communication overhead of blocking and non-blocking collective operations. Overhead is defined as the time the calling thread spends in communications related routines, i.e., the time the thread can't spend doing other work. The communication overhead of blocking operations is the latency of the collective operation, because the collective call does not complete until the collective operation has completed locally.

Using both implementations and the benchmark results, four different times are measured:

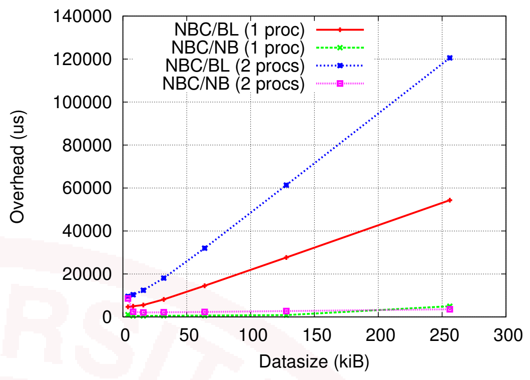
- Blocking MPI collective in the user thread (MPI/BL)
- Blocking MPI collective in a separate communications thread to emulate non-blocking behavior (MPI/NB)
- Non-blocking NBC operation without overlap, i.e., the initiation is directly followed by a wait (NBC/BL)
- Non-blocking NBC operation with maximum overlap, i.e., computing at least as long as an NBC/BL operation takes (NBC/NB)

We benchmarked both implementations with Open MPI 1.2.1 [76] on the Coyote cluster system at Los Alamos National Labs, a 1290 node AMD Opteron cluster with an SDR InfiniBand network. Each node has two single core 2.6 GHz AMD Opteron processors, 8 GBytes of RAM and a single SDR InfiniBand HCA. The cluster is segmented into 4 separate scalable units of 258 nodes. The largest job size that can run on this cluster is therefore 516 processors.

Figure IV.6 shows the results of the microbenchmark for different CPU configurations of 128 processes running on Coyote. The threaded MPI implementation allows nearly full overlap (frees nearly 100% CPU) as long as the system is not oversubscribed, i.e., every communication thread runs on a separate core. However, this implementation fails to achieve any overlap (it shows even



(a) Share of the freed CPU time with the non-blocking MPI and NBC alltoall operation with regards to the blocking MPI implementation.



(b) Blocking and non-blocking NBC_Ialltoall overhead for different CPU configurations.

Figure IV.6: Overhead of MPI Collectives in a Thread ("MPI/NB") with LibNBC's point-to-point based collectives ("NBC/NB") on a system with two CPUs per node. Running with 128 processes on 64 ("2 procs") or 128 ("1 proc") nodes.

negative impact) if all cores are used for computation. LibNBC, based on non-blocking MPI point-to-point allows decent overlap in all cases, even if all cores are used for computation. Those results and the lack of control over the MPI collective implementation leads us to the conclusion that a separate implementation of non-blocking collective operations is necessary to achieve highest performance. We will discuss threaded progression issues with LibNBC later in Section 4. We decided to optimize LibNBC for InfiniBand^{TM} to achieve maximum performance in the threaded and non-threaded case. The next section describes our InfiniBand^{TM}-optimized implementation in the non-threaded case.

3 Case-Study: Optimized Implementation for InfiniBand

"For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled." – Richard Feynman, (1918-1988) American Physicist

The portable version of LibNBC is built on top of non-blocking MPI point-to-point operations. As a result, the overhead and the achievable overlap are directly dependent on the overhead and (independent) progress of MPI_Isend and MPI_Irecv in the underlying MPI library. Several benchmarking studies about overlap [103, 106, 129] show controversial results for different MPI libraries. However, the studies do mainly agree that independent progress is limited in current open-source MPI libraries. Especially the analysis of applying overlapping techniques to parallel Fast Fourier Transform led to controversial results [26, 50, 68, 81], which is most likely due to the different abilities of the underlying communication system to perform asynchronous message progression. As we will discuss below, our own experience was similar: the overlap potential of current open-source MPI libraries is limited. Thus, we decided to implement our own InfiniBand^{TM} transport layer to leverage the full hardware features discussed in Chapter II. First, we define functions needed by LibNBC as a subset of the functionality that MPI offers and then, we describe an InfiniBand^{TM}-optimized implementation of this subset.

3.1 LibNBC Requirements

The architecture of LibNBC has been discussed in Section 1. However, the overlap potential is limited by this design, mainly by two factors [11]. First, the user has to call NBC_Test periodically to advance the internal state of the scheduler (start new communication rounds) and second, the overlap is limited by the underlying MPI library.

The first problem could be solved if the scheduler was executed in a separate progress thread. However, this would require a thread-safe (MPI_THREAD_MULTIPLE) MPI library and thus limit portability significantly. The second problem, the underlying communication system, causes a higher performance loss. This could be overcome if another transport layer would be used to send and receive messages asynchronously. It would be beneficial if this transport layer ensures asynchronous progress. However, the MPI standard does not define a clear progress rule and the support for asynchronous progress is limited.

The first step in optimizing LibNBC for a particular network like InfiniBand would be to use the low-level network interface in a way that enables highest overlap, full asynchronous progress and is optimized for the needs of LibNBC. Not all features of MPI are needed by LibNBC, for example, the MPI library needs to implement a rather complicated protocol to support MPI_ANY_SOURCE which is not needed by LibNBC. The requirements of LibNBC are listed in the following.

- **non-blocking send** is used to start a send operation and should return immediately (low overhead)
- **non-blocking receive** is used to post a receive and should return immediately (low overhead)
- **request objects** are needed to identify outstanding communications. All request objects have to be relocatable!
- **communication contexts aka communicators** are used as a communication universe to represent MPI communicators passed by the user
- **message tags** are used to differentiate between multiple different outstanding collective operations on a single communicator
- **message ordering** message with the same tag must match in the receiver side in the order they were issued on the sender side
- **test for completion** this test should be non-blocking and specific to a request object. It might be used to progress the communication. However, fully asynchronous progress is preferable.
- **wait for completion** is optional (can be a busy test), but might be used for different optimizations (e.g., lower power consumption by using blocking OS calls)

Having defined the requirements of LibNBC, we will briefly describe MPI libraries for the InfiniBand^{TM} network in the next section.

3.2 MPI Implementations for InfiniBand

Two popular open-source MPI implementations for InfiniBand, Open MPI and MVAPICH, exhibit similar performance characteristics. Neither implementation offers asynchronous progress (a progress thread) of outstanding messages. The two-sided semantics of MPI force the implementer to implement a protocol where the sender has to wait until the receiver posted the receive request because the message size is not limited (this prevents the usage of pre-posted receive buffers). This protocol is commonly called "rendezvous protocol". Another MPI feature, the MPI_ANY_SOURCE semantics, force the implementation to perform at least three message exchanges for every large message [186].

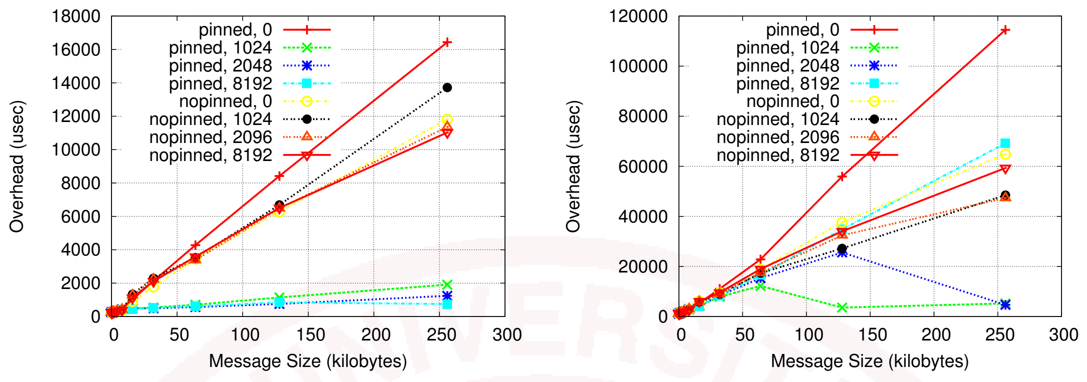
All benchmarks are conducted on the *Odin* cluster at Indiana University. Since the investigated MPI libraries don't have (fully) asynchronous progress for large messages, the user-program has to progress the requests manually. The only way to progress in a fully portable way is to test every outstanding request for completion because the MPI standard mandates that repeated calls of MPI_Test must complete a request eventually. However, calling MPI_Test during the computation is not only a software-technological nightmare (passing the requests down to the computation kernels) but is also a source of two kinds of significant overhead. The first source of overhead is simply the time spent in MPI_Test itself. The second overhead source is less obvious but more influential. Calls to libraries (e.g., BLAS [130]) must be split up into smaller portions which, first, destroys code structure, and, second, might lower efficiency (i.e., cache efficiency, the cache is also polluted by the calls to MPI_Test). Thus, calling MPI_Test is not a feasible option. However, not calling any test with the MPI implementations results simply in no overlap at all (see analyses in the following section).

Thus, the user of the current LibNBC is forced to perform test calls in the application. Accepting this, the user faces another problem because the decision of when and how often MPI_Test should be called is non-trivial. Too many calls cause unnecessary overhead and not enough calls will not progress the library and causes unnecessary waiting. It is easy to show that the optimal "test-patterns" depend on the protocol used by the MPI library and therewith on the library itself. Given this complexity that an application programmer faces today, he usually just applies a simple heuristic of calling test when it is convenient or not at all. However, overlap performance in this case is clearly suboptimal.

We will analyze different test strategies in the following section with the goal of deriving better heuristics.

3.2.1 Open MPI Message Progression for LibNBC

We use LibNBC to analyze the progression strategies. LibNBC's scheduler calls MPI_Testall on all outstanding requests related to the NBC_Handle that NBC_Test is called with. Thus, the test behavior is transparent. We extended our benchmark NBCBench which was first introduced in Section 2 to support different test strategies. NBCBench follows the principles for collective benchmarking described in [86, 215], [15] to ensure highly accurate results. The benchmark is run twice for every combination of message size and communicator size. The first run determines the time that the (blocking) execution takes and the second run executes a computation of the length of the first run



(a) Ialltoall on 16 Nodes (b) Ialltoall on 64 Nodes

Figure IV.7: CPU overhead of NBC_Ialltoall with Open MPI

between init and wait of the collective communication. The implemented test strategy issues N tests in equidistant times during the simulated computation. N is a function of the message size and therewith indirectly of the transfer time, it is computed as

$$N = \left\lfloor \frac{size}{interval} \right\rfloor + 1$$

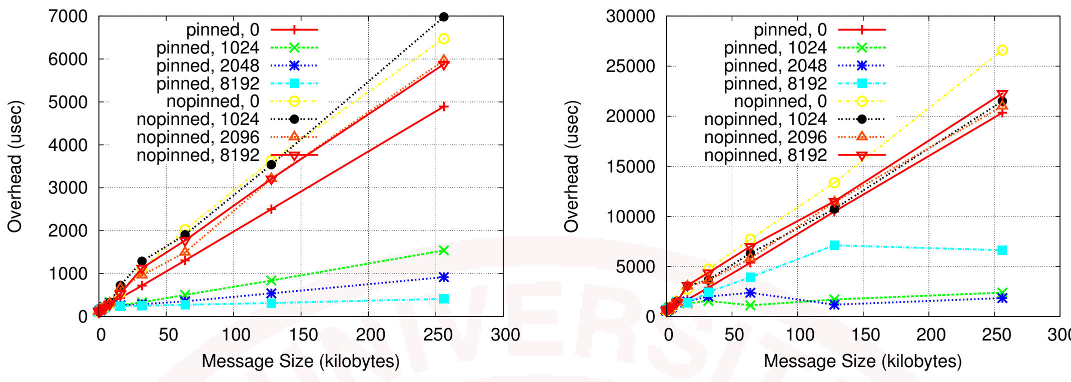
For example, if the datasize is 4096 bytes and the interval is 2048 bytes, the benchmark issues one test at the beginning, one after 50% of the computation and one at the end. The test-interval is chosen by the user.

We chose two collectives that are not influenced by the missing asynchronous progress of LibNBC itself, but represent a common subset. The first operation, NBC_Igather, represents a many-to-one operation while the second operation, NBC_Ialltoall represents the group of many-to-many collectives.

We benchmarked different test-intervals (0 for no tests, 1024, 2048, 4096 and 8192) for NBC_Igather and NBC_Ialltoall for 16 and 64 nodes. We analyzed two different memory registration modes of Open MPI, leave pinned (where the memory is cached in a registration cache) and no leave pinned (the memory is registered in a pipelined way to overlap registration and communication) [186].

Figure IV.7(a) and IV.7(b) show the results of the overhead benchmark with Open MPI 1.2.4/openib on 16 and 64 nodes respectively.

The overhead is defined as the time that is spent for communication. This is **not** the latency, but the sum of the time spent in initialization (e.g., the call to NBC_Igather call), testing (NBC_Test) and the waiting time at the end (NBC_Wait). Thus, the benchmark models the ideal overlap if all communication can be overlapped. The cases where no tests were performed were equal to the blocking execution of MPI_Alltoall in this scenario (no overlap at all). The optimal test intervals differ between 16 and 64 nodes. While, on 64 nodes, testing every 1024 bytes seems most beneficial, a test-interval of 8192 bytes performs better on 16 nodes.



(a) Igather on 16 Nodes

(b) Igather 64 Nodes

Figure IV.8: CPU overhead of NBC_Igather with Open MPI

The results for NBC_Igather on 16 and 64 nodes, shown in Figure IV.8(a) and Figure IV.8(b), show also different optimal test-intervals. It is even more complicated because a test-interval of 2048 bytes seems better for several message sizes. Thus, we can conclude that the optimal test strategy does not only depend on the MPI implementation, but also on the message size and communicator size. A simple heuristic does not seem feasible. However, testing itself is suboptimal for several reasons, thus we should focus on different strategies.

This section has shown that even if the application programmer is willing to progress the MPI manually, the selection of an optimal strategy is highly non-trivial (and not likely to be performed by a developer). Thus, we decided to implement a library (LibOF) that offers all necessary features to LibNBC directly on top of InfiniBand^{TM} (the OFED verbs interface). The design and implementation of this library is described in the next section.

3.3 Implementation of LibOF

Many MPI implementations are tuned for microbenchmark performance of the blocking Send/Recv operations. Thus, things like minimal CPU overhead are often neglected to achieve higher blocking performance. Only some implementations, like MVAPICH [195], optimize for overlap and often need to be enabled explicitly. In the design of LibOF, we did not focus on microbenchmark performance but we tried to minimize the CPU overhead (and thus maximal overlap). Thus, our blocking microbenchmark results are expected to be slightly worse than Open MPI's. Measurements with blocking communication showed that our microbenchmark latency and bandwidth lie within a 5% margin of Open MPI's. Plots of those results can just be omitted.

In addition to the low overhead of the calls themselves, we also have to ensure that there is no need to call them often. To achieve this, we have to design protocols that enable the maximum asynchronity between the calling program and the InfiniBand^{TM} network which offers asynchronous progress.

The library uses MPI communicators as communication context. It attaches it's data as an attribute to every communicator. This communicator-specific structure stores all peer-specific information for this communicator and is initialized at the first use of this communicator. Due to the InfiniBand^{TM} connection establishment mechanisms, the first call with every communicator is blocking and needs to be performed by all ranks in the communicator to avoid deadlocks.

Orienting on current MPI implementations, we decided to implement an Eager and Rendezvous protocol in order to achieve the blocking performance. Those protocols will be described in the following.

3.3.1 Eager Protocol

Our Eager protocol is designed to proceed completely asynchronously of the calling program. Every peer has a number of buffers that can store the eager message, its size the tag and some protocol information. Those peer-specific buffers are registered during communicator initialization and the necessary data (`r_key, address`) is exchanged. When a new send operation is initiated with `OF_Isend`, all necessary data is attached to the request, which is set to the status `EAGER_SEND_INIT`, and the function returns to the user. The first test call with this request copies the data into a pre-registered send-buffer (if available) and posts a signaled `RDMA_WRITE` send request to the peer's SQ. The send-buffer is a linear array in memory and a tag of -1 marks an entry as unused. To find an unused buffer, the array is scanned for a tag equal to -1 and the buffer-index is attached to the request. The WR id is set to the address of the request so that the buffer can be freed (tag set to -1) when the WR completes on the send side. `OF_Irecv` attaches the arguments to the request and sets the request's status to `RECV_WAITING_EAGER`. Every test on a request with this status scans the eager array for the tag. It copies the data in the receive buffer if the tag is found and notifies the sender that the receive buffer can be re-used. The notifications (`EAGER_ACKs`) are piggybacked (in the protocol information) to other eager messages or sent explicitly if more than a certain number of eager buffers are used.

3.3.2 Rendezvous Protocol

Our rendezvous protocol differs from the protocol used in any MPI implementation because LibNBC does not require a receive from any source. Thus, we can drive a receiver-based protocol where the receiver initiates the communication and the sender is passive until it is triggered. The receiver attaches all necessary information to the request and sets it to `RNDV_RECV_INIT` during `OF_Irecv` function. The first test on this request registers the receive buffer, packs `tag`, `r_key` and `address` into a pre-registered RTR message buffer. This buffer is then sent with `RDMA_WRITE` to a pre-registered location at the sender and the request is set to `RECV_SENDING_RTR`. The RTR send buffer is freed with a similar mechanism as the eager send buffer. `OF_Isend` sets the request's status to `SEND_WAITING_RTR` after attaching the arguments to the request and registering the send memory. A test on the sender-side scans the RTR array for the request's tag. If the tag is found, it posts the `RDMA_WRITE_WITH_IMM` send request to its local SQ and sets the request status to `SEND_SENDING_DATA`. A receive request is finished when the receiver received the data.

3.3.3 Optimizing for Overlap

Optimizing for overlap means minimizing the CPU overhead and maximizing the asynchronous InfiniBand^{TM} progress. We minimize the CPU overhead by using our optimized protocols that only use a minimal number of operations to send and receive messages. For example, we use only a single CQ for send and receive requests because polling a CQ is relatively expensive [20].

Achieving the maximum asynchronity is easy in the case of the eager protocol and tricky in the case of rendezvous. A first simple optimization, called test-on-init in the following, is to call the first test in the send of the eager protocol and the receive in the rendezvous protocol during the `OF_Isend` and `OF_Irecv` functions respectively. This hands the (ready) message immediately to the InfiniBand^{TM} network and does not introduce unnecessary waiting until the first test is called by the user. However, it obviously increases the CPU overhead in those functions.

The test-on-init optimization makes the progress in the eager protocol completely asynchronous (no test is necessary to "push" messages). However, the rendezvous protocol does still need a test on the sender-side to send the message after the RTR arrived. Thus, no progress will happen before the test. The optimal time between the `OF_Isend` and the first test is also not trivially determinable (it would be a single latency if receive and send were started at the same global time). A simple approach would be to poll test until the RTR message has arrived, but this might introduce deadlocks because `OF_Isend` would depend on the receiver. We decided to implement a timeout-based mechanism that polls only a limited time to avoid deadlocks and will refer to it later as "wait-on-send". However, this mechanism increases the CPU overhead of the rendezvous send drastically. We will discuss techniques to mitigate this after we analyzed and compared the influence in the next section.

3.4 Performance Results

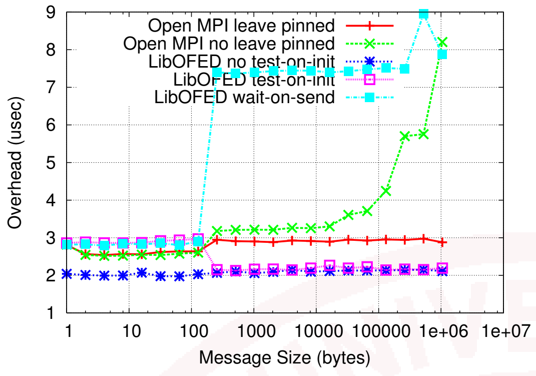
We compare the overhead and overlap of our implementation to Open MPI's overhead with different techniques. We used two different microbenchmarking tools, Netgauge and NBCBench to assess raw performance.

3.4.1 Netgauge

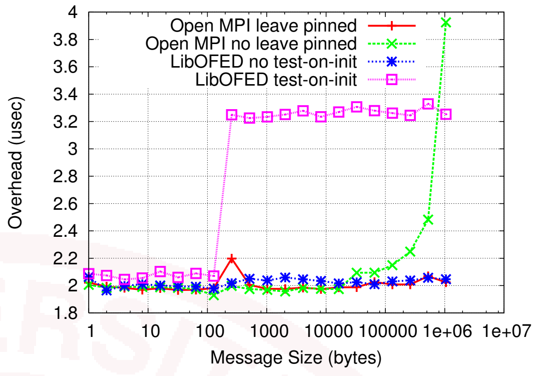
We extended Netgauge with a module to use LibOF as communication channel and added a new communication pattern which assesses the overheads of non-blocking communication. The module's implementation is trivial and just maps Netgauge's (blocking and non-blocking) send/recv and test functions to `OF_Isend`, `OF_Irecv` and `OF_Test`. The communication pattern "nbov" does a simple ping-pong and measures the times to issue the non-blocking send or receive calls and loops on test until the operation succeeds. Additionally, it takes the time for all calls to test and divides them by the number of issued tests to get a rough estimation for the average time for the test operation. We use the Netgauge's high-precision timers (RDTSC [108]) to benchmark single messages and repeat the ping-pong procedure multiple times (1000) and average the results afterwards.

We ran our new pattern with Open MPI and LibOF to determine the overheads. The Isend overhead is shown in Figure IV.9(a). We set the eager protocol limit to 255 bytes for LibOF and Open MPI. LibOF without test-on-init performs best because it does not start any operation during the Isend. The wait-on-send adds a huge overhead to every send operation as expected.

The Irecv overheads are shown in Figure IV.9(b). The wait-on-send and test-on-init show the



(a) Isend Overhead



(b) Irecv Overhead

Figure IV.9: Isend and Irecv Overheads

same performance because wait-on-send implies test-on-init and the receiver side is not different. The overhead of registering the memory and sending the RTR message can be seen when the protocol is switched to rendezvous. It is still not clear if the minimization of the send/recv overhead is more important than the minimization of the test overhead. The test overheads lie in a range from 0.1 (rendezvous) to 2 (eager) microseconds.

3.4.2 Optimizing Wait-On-Send

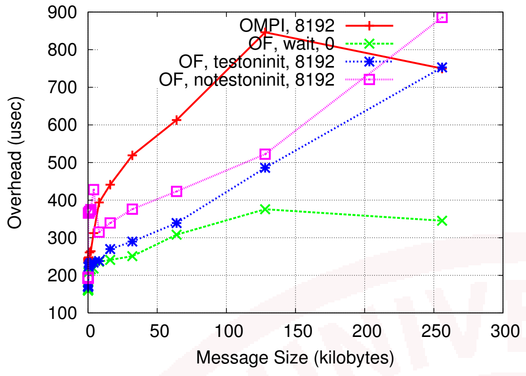
We saw in the previous Section that wait-on-send adds a huge constant CPU overhead per message. LibNBC usually issues many messages at the same time (dependent on communicator size) so that this overhead adds up per message. To mitigate this effect and since we have transparent access to our implementation, we implemented a hook `OF_Startall` in LibOF that progresses multiple send requests until they leave the status `SEND_WAITING_RTR`. Thus, the overhead (which is basically the waiting time for the RTR to be transmitted) is only paid once for multiple messages. The `OF_Startall` is called by LibNBC directly after a new round is started and has also a timeout mechanism to prevent deadlocks.

3.4.3 NBCBench

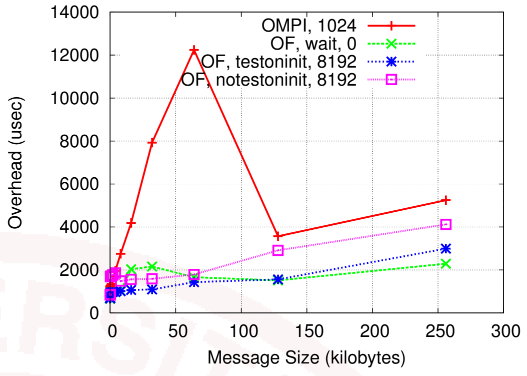
We use NBCBench again to compare our implementation with the best results (with the "optimal" test interval) achieved with Open MPI (cf. Section 3.2.1). We ran the same test-intervals as previously used with and without test-on-init. We ran the wait-on-send implementation without tests because it is designed to run asynchronously and test would only add overhead.

The results for NBC_Ialltoall on 16 and 64 nodes are shown in Figure IV.10(a) and IV.10(b) respectively. The results indicate that our wait-on-send implementation (the optimized version) performs best in nearly all cases. The 64 node case where test-on-init, with tests every 8192 bytes, performs better lies in the small message range where blocking collective operations are faster (the overhead of generating the schedule is significant for small messages, cf. [11]).

Similar results can be found in Figures IV.11(a) and IV.11(b) which shown the comparison for NBC_Igather on 16 and 64 nodes respectively.

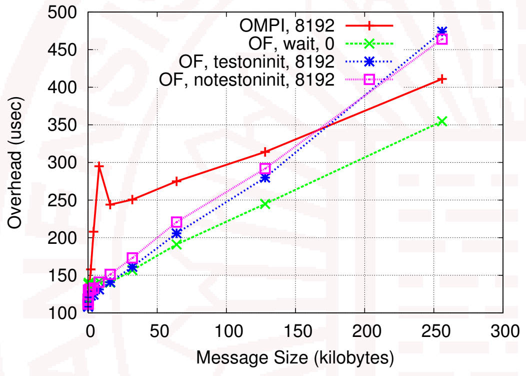


(a) Ialltoall overheads on 16 nodes

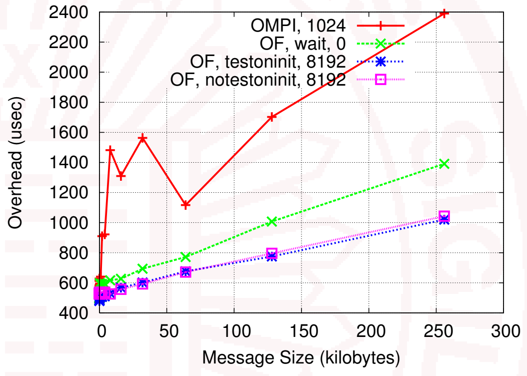


(b) Ialltoall overheads on 64 nodes

Figure IV.10: NBC_Ialltoall overheads with LibOF



(a) Igather overheads on 16 nodes



(b) Igather overheads on 64 nodes

Figure IV.11: NBC_Igather overheads with LibOF

All the benchmarks and results in this section have been gathered with LibNBC's non-blocking collectives. Figure IV.12 compares the Performance of NBC_Ialltoall (A2A) and NBC_Igather (GAT) to the highly optimized blocking MPI implementations [193].

3.5 Applicability to 10 Gigabit Ethernet

Many high-performance 10 Gigabit Ethernet networks support iWARP in hardware. The communication with the iWARP stack is implemented with the OpenFabrics Verbs interface. Thus, the same strategies used to optimize LibNBC for InfiniBand^{TM} can be used to optimize for 10 Gigabit Ethernet and should result in similar performance. The LogGP parameters allow a quantification of the remaining CPU overhead. Measurements for an adapter that supports iWARP in hardware are presented in Section 2.4 in Chapter II. However we do not have access to a reasonable number of 10 Gigabit Ethernet cards to perform measurements with LibNBC.

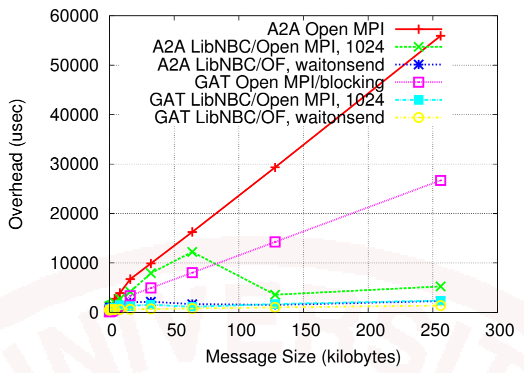
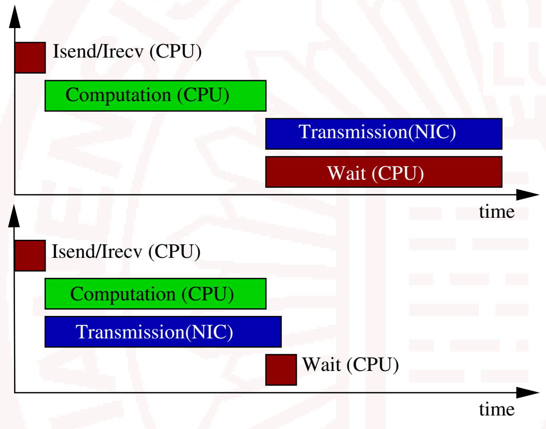
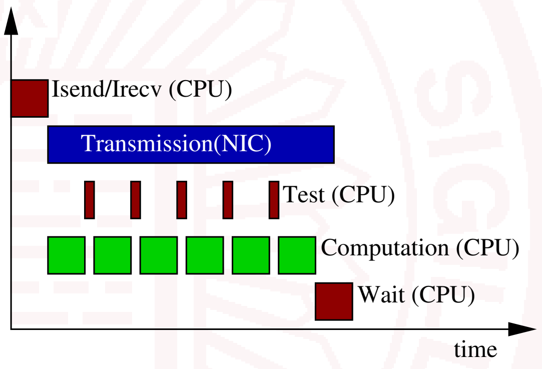


Figure IV.12: Comparison of the Alltoall (A2A) and Gather (GAT) overheads between (non-blocking) LibNBC and (blocking) Open MPI on 64 nodes



(a) Fully asynchronous versus wait-based message transmission for overlapped communication



(b) Black-box test based manual progression strategy

Figure IV.13: Different Progression Strategies

4 Communication Progression Issues

"Technology presumes there's just one right way to do things and there never is." – Robert Pirsig, (1928) American philosopher

After we discussed and optimized point-to-point progression for InfiniBand$^{\text{TM}}$ in the previous section, we analyze different fully asynchronous progression schemes in this section. To achieve full asynchronity, we leverage threads to progress the state of LibNBC. This would ideally result in fully-asynchronous progression as shown in Figure IV.13(a).

A common assumption is that the progress just happens in the background without user intervention. While this might be true for some communication libraries or systems, there are cases where the user needs to progress the messaging subsystem manually. An easy way to do this for MPI is to call MPI_Test as described in the previous section. A high quality implementation should ensure fully asynchronous progress whenever possible. In this section, we analyze and evaluate different options for message progression at the point-to-point and collective level. First, we discuss well-known strategies to implement point-to-point messages and to interact with the communication hardware.

4.1 Messaging Strategies in Communication Middleware

To discuss message progression schemes efficiently, we introduce common messaging protocols and implementation options. Most MPI libraries implement two different protocols to transmit messages. Depending on the message size, either an eager or a rendezvous protocol is selected to implement the message transmission. Eager transmissions send the message without synchronization to the receiver where it is buffered until the application process receives it. The rendezvous protocol delays the message transmission until the receive process has posted the receive operation to the library. Another option is to use pipelined message transmission in the rendezvous protocol [186].

These protocols build on single message sends. The eager protocol sends only one message from the sender to the receiver. However, the rendezvous protocol requires at least two synchronization messages and the actual data transmission, which might be pipelined and thus consist of many send operations. Different strategies to send those messages are based on the two simple operating system (OS) concepts, polling and interrupt. Most middleware systems only implement polling mode (i.e., the program spins on the main CPU while querying the hardware) for user-level messaging to enable OS bypass. The other option, interrupts (i.e., the process enters the operating system and frees the CPU until a message arrives asynchronously) requires interaction with the OS. It is assumed that the necessary syscall (privilege change) to enter the operating system code is rather expensive and thus, many modern messaging systems focus on OS-bypass schemes where all communication is performed in userspace. Thus, the polling based approach delivers, due to OS bypass, a slightly lower point-to-point latency and is therefore used in common high-performance MPI libraries for InfiniBand such as Open MPI and MVAPICH.

A more complex issue is the development of non-blocking high-level communication routines that involve interactions between multiple processes. Similar overlapping principles than in the point-to-point case can be used with those operations. However, the optimization for overlap is much more complicated because the communication protocols and algorithms are becoming significantly more complex than in the point-to-point case. Thus, we focus on the more complicated case to analyze the overlap of non-blocking collective operations in our work which of course also covers point-to-point progression.

4.2 Message Progression Strategies

Three fundamentally different messaging strategies can be found in parallel systems. A common strategy is to enforce manual progression by the user. This is often perceived as no progression because the programmers do not progress or can not progress the library manually. A second strategy is the hardware-based approach where the message handling is done in the network interface card. The third approach, using threads for progression, is often discussed as the "silver bullet" but it has not found widespread adoption yet.

4.2.1 Manual progression

This scheme is the simplest to implement from the MPI implementer's perspective because there is no asynchronous progress. Every time, the user calls MPI_Test with a request, the library checks if it can make any progress on this request. Thus, the complete control and responsibility is given to the user in this case. There are several problems with this approach. The biggest problem is the opaqueness of the MPI library, i.e., the user does not know about the protocol and the status of a specific operation. Thus, for portable programs, he has to assume the worst case for asynchronous progress, the pipeline protocol, where he has to call MPI_Test to progress every fragment in order to achieve good overlap. However, the missing status information forces the user to adopt a black box strategy.

In the previous section, we proposed a black box testing scheme that issues N tests during a message transmission; the scheme is shown in Figure IV.13(b).

4.2.2 Hardware-based progression

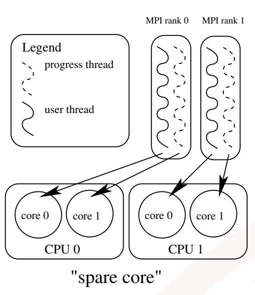
A possible solution to ensure full asynchronous progress is to do the protocol processing in the communication hardware. The Myrinet interconnection network offers a programmable network interface card (NIC) and several schemes have been proposed to offload protocol processing and message matching on this external CPU [118]. A similar offload scheme was proposed for Ethernet in [187]. Some proposals, such as [219], also implement NIC-based message broadcast schemes. The relatively simple barrier operation has also been implemented with hardware support [218]. Other schemes [214, 48] support collective operation offload for some operations but impose some limitations. However, none of those implementations allow overlap because they only offer a blocking interface.

4.2.3 Threads for Message Progression

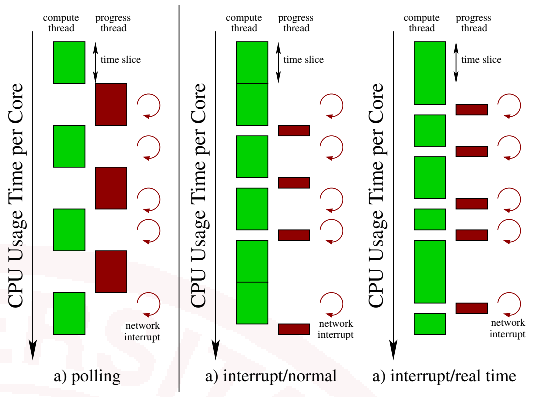
Asynchronous progression threads have often been stated as a silver bullet in future work, but they are not widely used. Adoption might have happened because the threaded programming model puts a huge burden on the system software implementer because the whole driver infrastructure and all libraries must be implemented reentrant (thread safe). However, some libraries, for example Open MPI, begin to explore the possibility of threaded progress. Other libraries like MPI/pro or HP MPI offer asynchronous progression threads but have not been analyzed in detail.

Threads have usually been used in high performance computing to implement thread-level parallelism (OpenMP, also in combination with MPI as a hybrid programming model [173]) or for other tasks that are not critical for communication, such as checkpoint/restart functionality.

Threads are a promising model for asynchronous progression. One of the biggest problems with manual progression strategies is that it is very unlikely that MPI_Test hits the ideal time. It comes either too early and there is nothing to progress or too late and overlap potential is wasted. A threaded implementation would be either polling and thus get all messages immediately or the thread could be woken up to progress the communication layer at exactly the right times. A progress thread also enables fully asynchronous progression, i.e., without any user interaction. We will focus on a threaded progression of non-blocking collective operations in the following sections.



(a) Different subscription schemes for a dual-CPU dual-core configuration



(b) CPU Scheduling Strategies for the fully subscribed case, comparing a polling progress thread with an interrupt based normal and real time progress thread

Figure IV.14: Scheduling Issues

However, before introducing our implementation, we discuss several operating system effects that influence the execution of threads.

4.2.3.1 Operating System Effects

There was great effort to circumvent the operating system in the past with so called OS-bypass methods. User-level networking without operating system support was implemented for most modern network interconnects to enable lower latencies and avoid system calls. However, the operating system plays an important role in the administration of user and progress threads. We have to distinguish the different methods to access the network hardware (polling, interrupt) and the subscription factor of the cores (all cores run user threads vs. spare cores can run progression threads). The subscription scheme is shown for a rather common dual-CPU dual-core combination in Figure IV.14(a).

Combining both schemes leads us to the following 4 combinations:

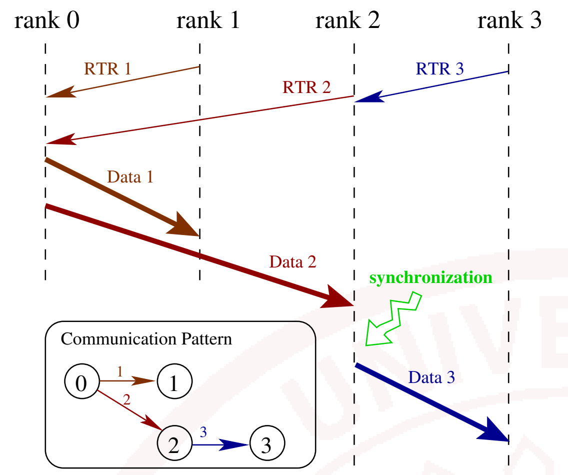
access method	core subscription
polling	fully subscribed
polling	spare cores available
interrupt	fully subscribed
interrupt	spare cores available

Operating System Scheduling The operating system scheduler usually arranges the threads (or processes) in two or more queues, a runnable queue and a waiting queue. Threads (or processes) in the runnable queue are waiting for the CPU and share this resource among each other, threads on the waiting queue wait for some other hardware event (i.e., a packet from the wire). To ensure fairness of the CPU sharing, each process has a time-slice to run on the CPU. If this time-slice is over, the scheduler schedules other runnable threads. The scheduler bases its decision on the thread priorities (which depends on the particular operating system). Typical length of time-slices are between 4 and 10 milliseconds. The Linux 2.6 default scheduler (called the O(1) scheduler) implements such a time-slice based mechanism.

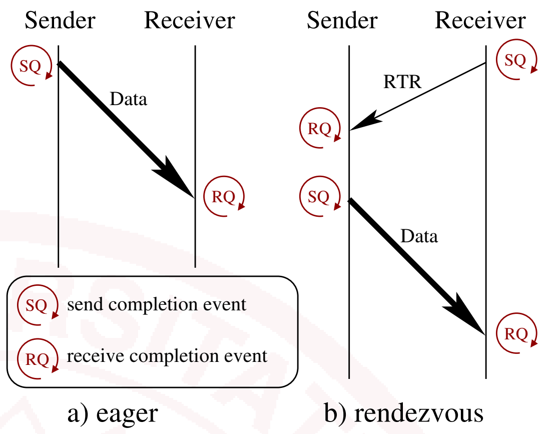
The analysis is easy if a spare core is available to each MPI process to run the progression thread. The difference between the polling (common implementation) and the interrupt based approach is that the polling might reduce transmission latencies slightly (due to OS bypass). The interrupt based approach might be more power efficient (since the hardware is idle during the message transmission) but might also have some effect to the performance of the operating system (and thus to other threads). However, this highly depends on the implementation of the OS (e.g., How is the locking implemented? Does the scheduling overhead depend on the number of threads?).

The analysis is much more complicated if there are no idle cores available. We would argue that this is the common case in today's systems, i.e., if a user has 4 cores per machine, he usually launches 4 (MPI) processes on each machine to achieve highest performance. In this scenario, the progression thread has to share the CPU with the computation thread. In the polling approach, the computation thread and the progression thread are both runnable all the time which leads to heavy contention in the fully subscribed case. This effectively halves the CPU availability for the computation thread and thus also halves the overall performance. Those effects can be limited by calling `sched_yield()` in the progression thread after some poll operations which leads to a re-scheduling. However, the progression thread is still runnable and will be re-scheduled depending on priority. The interrupt approach seems much more useful in this scenario because the progression thread goes to sleep (enters the wait queue) when no work is to be done and is woken up (enters the run queue) when specific network events (e.g., a packet is received) occur. This schedules the thread at exactly the right time (work is to be done) and is thus significantly different from the manual progression and the polling approach.

The interrupt-based mechanism raises two concerns. First, it seems unclear how big the interrupt latency and overheads are on modern systems. Second, the scheduler has to schedule the progression thread immediately after the interrupt arrives to achieve asynchronous progress. It is not sufficient if the thread is just put on the run queue and the computation thread is re-scheduled to finish its time-slice. Waiting until the time-slice of the active thread is finished increases the interrupt-to-run latencies by a time-slice/2 on average which effectively disables asynchronous progress because the time slices are one to two order of magnitudes higher than the transmission latency of modern networks. Unfortunately, this mechanism is common practice to ensure fairness, i.e., avoid processes that get many interrupts to preempt other compute-bound processes all the time. The Linux scheduler favors I/O bound processes but it might still not be sufficient to achieve highest overlap. To overcome this problem, one can increase the relative priority of the progression thread. We experiment with the highest possible priority, real time threads in Linux. If a real time (RT) thread is runnable, then it preempts every other thread by default. RT threads might decrease the interrupt-to-run latency significantly. However, they might also increase the interrupt and context switching overhead significantly because the thread is scheduled every time an interrupt occurs and goes to sleep shortly after. All options are illustrated in Figure IV.14(b). The next section discusses our implementation of non-blocking high-level communication operations and our extensions for threaded asynchronous progression.



(a) Receiver initiated point-to-point messages for a tree broadcast pattern among four processes



(b) Completion events generated by the hardware for both point-to-point protocols

Figure IV.15: Synchronization in Communication Protocols

4.3 Threaded Implementation of LibNBC

LibOF, described in the previous section, implements the standard non-blocking transmission functions (send/recv) in a way that enables the most asynchronous progress without user intervention. In this work, we extend and analyze LibOF to use a progression thread to ensure fully asynchronous progression.

4.3.1 Partially Asynchronous Collective Communication

The structure of LibNBC reflects the two levels of communication (point-to-point and collective). The user issues a non-blocking collective operation which returns a handle. This handle has a (potentially multi-round) schedule of the execution (cf. [11]) and a list of point-to-point operations of the current round attached. Progress is thus defined on the two levels, point-to-point progress and collective progress.

The two levels of progression for a binomial tree broadcast on four processes are displayed in Figure IV.15(a). Even though process 1 received the RTR message from process 3 early, it can only send the data after it received it from process 0. Those data-dependencies incurred by the collective algorithms add a new complexity to the progression.

The InfiniBand optimized LibOF supported nearly asynchronous point-to-point progress at the messaging level for the rendezvous protocol. The implemented wait-on-send strategy (the sender polls for a while to receive the ready-to-receive (RTR) message from the receiver) progresses if the nodes post the send and receive operations at similar times. However, if the RTR message arrives late, there will be no progress (unless the user progresses manually). The eager point-to-point protocol is fully asynchronous if free slots are available at the receiver-side. The implementation uses unsignaled RDMA-Write and polls the memory to detect memory completion. Consult [9] for a detailed analysis of the different progression schemes in comparison to manual progression.

However, this protocol does not support progress on the collective messaging layer. If the collective algorithm consists of multiple communication rounds (e.g., broadcast is implemented as a tree or Allreduce in a pipelined way), only the first round is progressed automatically and the user has to progress the following rounds manually. This is the big limitation in the current implementation that our new threaded implementation seeks to overcome.

4.3.2 Fully Asynchronous Collective Communication

This section describes the design of the fully threaded version to deal with multiple communication contexts (MPI communicators) and multiple point-to-point InfiniBand connections (queue pairs (QP)). Our design focuses on an interrupt-based implementation because polling has been well analyzed in previous work (e.g., [4]). Each queue pair represents a channel between two hosts and is associated with a completion queue where events are posted. Those events could be the reception of a new message or the notification of a message transmission. Each completion queue is associated with a so called completion channel to use interrupt driven message progression. Each completion channel offers a file descriptor that can be used in system calls (e.g., `select()` or `poll()`) to wait for events.

To get a notification for every packet, the implementation is changed to use RDMA-Write with immediate (signaled RDMA-Write) for every message transmission. This makes sure that the receiver receives an interrupt for incoming messages.

In our design, the progress thread handles all collective and point-to-point progressions. After the user thread posts a new collective operation, the new handle is added to the thread's worklist. This is one of two synchronization/locking points between the two threads. The second synchronization is when the user wants to wait on the communication where it waits until a (shared memory) semaphore attached to the handle becomes available (is activated by the progress thread). The test call just checks if the semaphore would block and returns true if not.

The progress thread itself generates a list of point-to-point requests from its collective worklist (every collective handle in the list has a list of point-to-point requests attached) and calls OF_Waitany() with the full list. When OF_Waitany() returns, a point-to-point request finished and the progress thread calls LibNBC's internal scheduler with the associated collective handle to see if any progress can be made on the collective layer. Then, it compiles a new list of point-to-point requests (since a handle might have been changed) and enters OF_Waitany() again.

The implementation of OF_Waitany() uses the list of point-to-point requests to assemble a list of file pointers (by checking the associated completion channels). The function then blocks (with the `poll()` system call) until one of the file pointers gets available (i.e., a completion channel event occurred). Then it polls the associated completion queue, progresses the message for which the event occurred and returns if a message transmission finished. If no transmission finished, the function just enters the `poll()` call again. The number of completion events depends on the point-to-point protocol used. Figure IV.15(b) illustrates the protocol-dependent completions.

The program must also ensure that if the user issues a new collective operation, this is picked up by the thread (to avoid deadlocks and achieve best progress). In order to do so, a pipe read file descriptor is added to the list of file pointers to `poll()`. Whenever a new request is added to the threads worklist, it is woken up from the wait queue by writing to this pipe.

This scheme enables fully asynchronous progress and the collective operations are finished in the background without any user interaction. The following sections discuss benchmarks and performance results for several different configurations.

4.3.3 Point-to-point Overhead

We benchmark our implementation on the *Odin* cluster. We implemented a new communication pattern benchmark for Netgauge [12] that measures the overlap potential and communication overhead for point-to-point messages and different progression strategies. The benchmark works as follows:

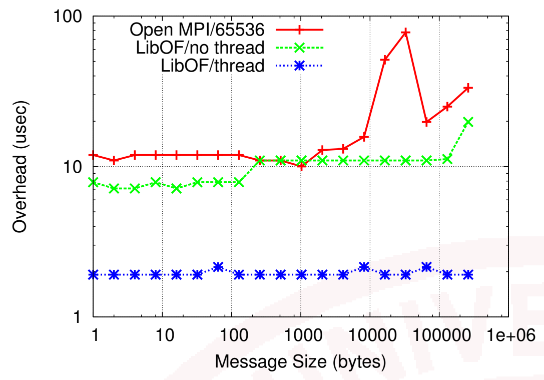
1. Benchmark the time t_b for a blocking communication.
   (a) start timer t_b
   (b) start communication
   (c) wait for communication
   (d) stop timer t_b
2. Start the communication.
3. Compute for time t_b.
   (a) endtime = current time + t_b
   (b) while(current time $<$ endtime) do computation
4. Wait for communication to finish.

The measurement is done as a ping pong with pre-posted receives on the client side, i.e., "start communication" posts a non-blocking receive and a non-blocking send and "wait for communication" waits until both operations finished. The server side simply returns the packets to the sender. The overhead t_o is the sum of the times spent to start the communication, progress the communication (test) and wait for the communication to complete.

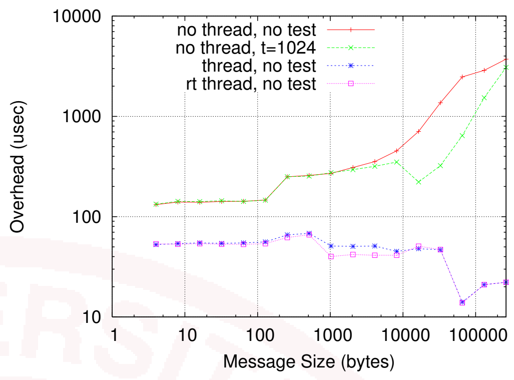
We compare the Open MPI implementation which needs manual message progression (cf. Section 4.2) with LibOF. The first experiment is a parameter study which aims to find the best parameters for the manual progression of Open MPI. We conducted benchmarks for tests every $2^n, \forall n = 10..18$ bytes. The test every 65536 bytes performed best for most message-sizes.

The results between two *Odin* nodes are shown in Figure IV.16(a). It compares Open MPI (using the best test configuration for every 65536 bytes) with our threaded and non-threaded Mini-MPI implementation (LibOF). The results show that the overlap optimized Mini-MPI has a generally lower CPU overhead than the Open MPI implementation. Adding a progression thread to this implementation decreases the overhead due to offloading the communication processing to a spare core.

We conclude from those experiments that the threaded progression strategy can be beneficial for point-to-point messaging if the progression thread runs on a separate CPU core. The next section analyzes the collective progression behavior where all cores might be busy with computation.



(a) Point-to-point overhead of the different implementation options



(b) NBC_Iallreduce overhead on 32 nodes, one process per node with different progression strategies

Figure IV.16: Open MPI and LibOF Communication CPU Overheads

4.3.4 Collective Overhead

We benchmarked all collective operations with NBCBench and present Allreduce, the most important multi-round operation (cf. [172]). Other important single-round operations, like Alltoall have already been evaluated in the previous section and showed a high potential for overlap. The reduction operations are at the same time the most complicated operations to optimize for overhead. Those operations include, additionally to the communication, a computation step that uses a significant amount of CPU cycles in the reduction operation.

In our analysis we focus on the Allreduce operation which has been found to he hardest to optimize for overlap [11]. All other operations perform (significantly) better (with lower overhead) in all benchmarked cases. The Allreduce implementation in LibNBC uses two different algorithms for small and large messages (cf. [171]). A simple binomial tree with a reduction to rank 0 followed by a broadcast on the same tree is used for messages smaller then $65kiB$. This algorithm, which is described and modeled in Section 3.1.2, has $2 \cdot \lceil log_2 P \rceil$ communication rounds on P processes. The large message algorithm, that is also described and modeled in Section 3.1.2, chops the message into P chunks and performs $2 \cdot P - 2$ communication rounds in a pipelined manner to reduce the data.

Figure IV.16(b) shows the overhead of a non-blocking Allreduce operation on 32 nodes with 1 process per node. The best test strategy was to test every 1024 bytes, but this is significantly outperformed by the threaded implementation due to communication offload to spare cores. Real time threads do not improve performance in this scenario.

These results show that the threaded implementation is able to lower the CPU overhead by one order of magnitude if a spare CPU core is available to offload the computation. However, we showed in [4] that some applications can benefit from using all cores. That means that there might be no “free” cores on today’s dual-, quad- or oct-core systems. Thus, we also have to evaluate our progression strategies in the case where all CPU cores are busy with user computation. However, in general, we assume that especially memory-bound and irregular algorithms, like sparse solvers

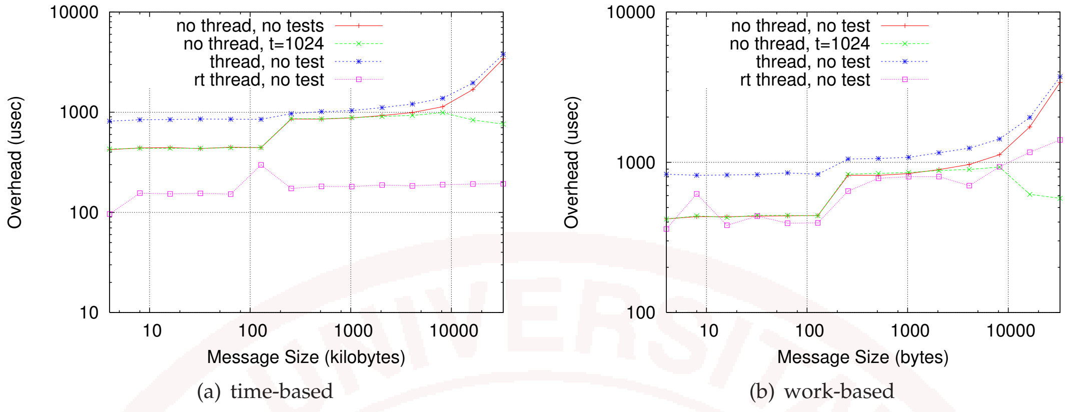


Figure IV.17: NBC_Iallreduce overhead on 32 nodes, 4 processes per node with different progression strategies

or graph problems, can not use the growing number of available cores so that spare communication cores will be available for communication offload in the near future.

4.3.4.1 The Oversubscribed Case

Using MPI exclusively means running n MPI processes on each n-core node. This leads to a 2:1 oversubscription of threads vs. available cores. Other models, like hybrid MPI + OpenMP [173] might lead to only one progression thread per node but are more complicated to program and optimize.

To reflect the oversubscription case, we run 4 threads on each of the 32 test nodes of *Odin*. This leads to 128 processes performing the collective operations. Our results show that our test strategy did not perform well when all cores are busy. The threaded implementation is also not able to decrease the communication overhead significantly due to scheduling problems in the operating system described in Section 4.2.3.1. However, those OS effects can be overcome with a real-time thread that preempts the user computation as soon as an InfiniBand completion event occurs.

The timing-based measurements only accounts for the CPU overhead of the actual communication calls (NBC_Iallreduce and NBC_Wait) but other overheads such as interrupt processing, time spent in the InfiniBand kernel driver and context switching overhead as well as cache pollution are not taken into account and could have a detrimental effect on real application performance. Thus, we analyze the number of context switches, the context switch and interrupt processing overhead in the following.

The number of context switches The implementation allows for more than one event to be processed by the progress thread in a single interrupt. However, the real-time scheduling might cause many context switches. The maximum number of context switches equals to the number of completion notifications, which depends on the transport protocol and the collective algorithm. The eager protocol causes 1 completion on the sender and the receiver and the rendezvous protocol causes 2 completions on the sender the receiver as explained in Section 4.3.2. Thus, the tree-based small message algorithm causes up to $1 \cdot 2 \cdot \lceil log_2 P \rceil$ completion events in the eager case and $2 \cdot$

$2 \cdot \lceil log_2 P \rceil$ completion events in the rendezvous case (some intermediate nodes send and receive $\lceil log_2 P \rceil$ messages). The number of interrupts is thus 14 or 28 in our example with 128 processes. The pipelined large message algorithm causes up to $1 \cdot 2 \cdot (2 \cdot P - 2)$ completions for eager messages and $2 \cdot 2 \cdot (2 \cdot P - 2)$ for rendezvous messages (each node sends and receives a single packet in $2 \cdot P - 2$ rounds. In our example with $P = 128$, this equals 508 or 1016 interrupts.

4.3.4.2 Interrupt and Context Switch Overhead

It is not clear how much a threaded implementation suffers from context switch and system interrupt overhead. In our model, the latency incurred by those operations is less important because we assume that this will be overlapped with computation. The most important measure in our model is the CPU overhead, i.e., how many CPU cycles the interrupt processing and context switch "steals" from the user application. We describe a simple microbenchmark to assess the context switching and interrupt overhead in the following.

The benchmark measures the time to process a fixed workload on every of the c CPU cores. In order to do this, it spawns one computation thread on each core i that records the time t_1^i to compute the fixed problem in a loop. The benchmark has two stages, stage one measures the normal case where no extra interrupts are generated/received[2]. Then, the main thread that has been sleeping so far programs the real time clock interrupt timer to the highest possible frequency $f = 8192\,Hz$ for stage 2. In this stage, the main thread receives those interrupts and thus steals computation cycles from the worker threads that benchmark t_2^i on each core. The results from all cores for the two stages are averaged into $t_1 = \sum_i t_1^i / c$ and $t_2 = \sum_i t_2^i / c$. The difference $t = t_2 - t_1$ is the time that is added by the interrupts and subsequent context switches. The number of interrupts in stage 2 can be estimated with $i = t_2 \cdot 8192\,Hz$.

We ran this benchmark on the *Odin* cluster with 4 computation threads. The average fixed computation of 7 measurements in stage 1 was $t_1 = 19.14626\,s$ and in stage 2 $t_2 = 19.27969\,s$. The number of interrupts in stage 2 was thus $j = t_2 \cdot f \approx 157939$. The j interrupts delayed the work by $t = 133430\,\mu s$, thus yielding a CPU overhead per interrupt per core of $133430/1579394 \cdot 4 = 3.38\,\mu s$.

We show that interrupts and context switching between threads causes about $3.4\mu s$ overhead on our test system and is thus relatively expensive. Based on that, we conclude that frequent context switches increase the overhead significantly. Especially for the large-message Allreduce algorithm that might receive up to 1016 interrupts which would mean an overhead of $3.4ms$ per core. However, this overhead is not reflected in our current time-based computation analysis. Additional other overheads like the time spent in the InfiniBand driver stack and side effects like cache pollution are not modeled so far. To overcome this limitation, we propose a workload-based benchmark that simulates a real-world application with the computation of a constant workload in the different scenarios.

4.3.4.3 Workload-based overhead benchmark

We extend NBCBench with a workload-based computation scheme. The first step to obtain the blocking time t_b remains the same.

[2]"no extra" means only the normal background "noise" in this case

1. Benchmark the time t_b for a blocking communication.
2. Find workload λ that needs t_b to be computed.
   (a) $\lambda = 0$
   (b) while($t_\lambda < t_b$) { increase workload λ by δ;
   t_λ = time to compute workload λ }
3. Start timer t_{ov}.
4. Start communication.
5. Compute fixed workload λ.
6. Wait for communication.
7. Stop timer t_{ov}.

The overhead t_o in this case is the difference between the time for the overlapped case (computation and communication) and the computation time, thus $t_o = t_{ov} - t_c$.

We repeated the benchmarks with the new fixed workload scheme. The results for a single MPI process per node are as expected similar to the results with the time-based benchmark. However, the results in the overloaded case with 4 processes per node are rather different and shown in Figure IV.17(b). This benchmark reveals that even the real time thread is not able to decrease the communication overhead by an order of magnitude as shown with the time-based benchmark. However, the performance improvement is still significant with about a factor of two improvement (note the logarithmic scale of the graph) but mitigated by different sources of overhead, such as context switching, cache pollution and interrupt or driver processing. It is also interesting, that the right test strategy (in this case every 1024 bytes) is able to deliver higher performance for some message sizes due to the relatively low overhead of the test calls.

It has been shown that large messages are easy to overlap with a test-based strategy [11]. Our results with the threaded approach support the previous results and the real-time thread performs an order of magnitude better than the normal threading in the time-based benchmark. This is reduced to a factor of two in the work-based benchmark that takes the different sources of overhead into account. Other collective operations, such as Reduce, Bcast, Alltoall, and Allgather show significantly better results than the complex Allreduce operation because they do not involve computation or they only deliver the result to a single host only. We focused our analysis on the complex Allreduce operation which is also the most important collective operation.

4.3.5 Overcoming the Threading Issues

There are different ways to mitigate or even overcome the problems with threaded progression. The most obvious way would be to limit the number of interrupts by intelligent coalescing. This means that only the events that are important for progression (e.g., no local completions) generate an interrupt and wake the progress thread up. This technique is already used in Myrinet/MX to progress point-to-point communication and is able to at least halve the number of interrupts and thus reduce the overhead significantly.

Another easy change would be to replace the thread-based mechanism with a signal based concept, where the progression code is executed by a signal handler in the same thread. This would save the context switching and scheduler overhead time. However, the current implementation of signals is unreliable and needs to be made reliable to avoid deadlocks.

Another way would be to implement the whole progression engine inside the OS kernel. This would also eliminate context switches, scheduler overhead and also the expensive privilege changes between user- and kernel-space. Since the scheduler design is rather simple, this could be a viable solution to the progression problem. With this, we argue that operating system bypass might not be beneficial in all scenarios. Magoutis et al. also mention several other benefits of kernel-level I/O in [145].

The third and theoretically best but also most expensive way is to implement the whole high-level operation in the network hardware. Approaches to do full point-to-point message progression in the network interface card have been described in Section 4.2.2. This would need to be extended with functionality to handle higher level communication patterns.

5 Conclusions

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiments, it's wrong" – Richard Feynman, (1918-1988) American Physicist

Traditionally, scientists have not been able to fully leverage the performance benefit of overlapping communication and computation because MPI only supports blocking collective operations. In this section, we introduced LibNBC, a portable high-performance library that provides non-blocking versions of all collective operations. Thus, for the first time, fully portable and MPI-compliant non-blocking collective operations are available for overlapping communication and computation.

To assess the latency and overhead of blocking and non-blocking collective operations accurately, we analyzed different benchmarking schemes. We found certain systematic errors in common methods and proposed a new and scalable scheme based on the window mechanism used in SKaMPI. Therefore, we propose and analyze a new scalable group synchronization method for collective benchmarks. Our method is more than 16 times faster on 128 processes and promises to be more accurate than currently used schemes. We implemented a new network-jitter analysis benchmark in the open source performance analysis tool Netgauge. The described scheme to benchmark collective operations has been implemented in the open source benchmark NBCBench. We were able to show that the MPI model to perform blocking collectives in a separate thread is suboptimal.

Then, we analyzed the performance of LibNBC on InfiniBand^{TM} in detail and investigated several options to minimize the communication overhead (in order to maximize communication overlap). We showed that reasonable performance can be achieved at the user level using MPI and appropriate invocation patterns of MPI_Test (to guarantee progress of MPI). However, the invocation patterns depend not only on the MPI implementation but also on the communicator size and data size. We showed that, in general, the programmer would not be able to derive simple and optimal heuristics for progressing MPI. Furthermore, having to manually progress MPI in this way is generally suboptimal because it influences code structure negatively and adds additional overheads.

Based on the requirements of LibNBC, we implemented a low-level interface that uses the OFED verbs API directly to communicate. We propose a new rendezvous protocol that does not require user-level intervention to make independent progress in the network. Furthermore, we show with several microbenchmarks and application kernels that our implementation performs significantly better than blocking communication and non-blocking communication based on Open MPI.

We analyzed different strategies for asynchronous progression of non-blocking collective communication operations in message passing libraries. We analyzed polling and interrupt based threaded implementations and can conclude that polling based implementations are only beneficial if separate computation cores are available for the progression threads. The interrupt-based implementation might also be helpful in the oversubscribed case (i.e., the progress and user thread share a computation core) but this depends on the collective operation as well as on operating system parameters. We found that the progression thread needs to be scheduled immediately after a network event to ensure asynchronous progress. A good way to implement this is the usage of real-time functionality in the current Linux kernel. Our analyses for the most complicated operation, Allreduce, show that it is hard to achieve high overlap. However, further analyses show that other simpler operations, like reductions or broadcasts perform well in our model. We also proposed several mechanisms to mitigate the different sources of overhead that we identified in this chapter. We will also investigate different collective algorithms that can further improve the CPU availability to the user thread.

We have also theoretically shown that all optimization techniques are applicable to the upcoming generation of 10 Gigabit Ethernet adapters that support the iWARP protocol [111] in hardware. The variance in CPU overhead has been discussed in Chapter II.

Chapter V

Optimizing Parallel Applications

"Prediction is very difficult, especially about the future." – Niels Bohr, (1885-1962) Danish physicist, Nobel Prize 1922.

Hunting for performance in distributed-memory parallel scientific applications has long focused on optimizing two primary constituents, namely optimizing (single process or single thread) computational performance and optimizing communication performance. With more advanced communication operations (such as non-blocking operations), it has also become important to optimize the interactions between computation and communication, which introduces yet another dimension in which to optimize. Moreover, these advanced operations continue to grow even more sophisticated (with the advent, e.g., of non-blocking collective operations), presenting the programmer with yet more complexity in the optimization process. As systems continue to grow in scale, the importance of tuning along all of these dimensions grows.

Historically, overlapping communication and computation is a common approach for scientists to leverage parallelism between processing and communication units [137]. Applications with overlapped computation and communication are less latency sensitive and can, up to a certain extent, still achieve good parallel performance and scalability on high-latency networks. The ability to ignore process skew and hide message transmission latencies can be especially beneficial on cluster computers (also known as Networks of Workstations, NOW) and on Grid-based systems. The most straightforward way to improve basic communication performance is to employ high-performance communication hardware and specialized middleware to lower the latency and increase the bandwidth. However, there are limits to the gains to overall application performance that can be achieved just by focusing on bandwidth and latency. Approaches that overlap computation and communication seek to overcome these limits by hiding the latency costs by performing communication and computation simultaneously.

Obtaining true overlap (and the concomitant performance benefits) requires hardware and middleware support, sometimes placing high demands on communication hardware. Some studies, e.g., White et al. [106], found that earlier communication networks did not support overlapping well. However, most modern network interconnects perform communication operations with their own co-processor units and much more effective overlap is possible [5]. To take advantage of this opportunity, specific non-blocking semantics must be offered to the programmer. Many modern library interfaces (even outside of HPC) already offer such non-blocking interfaces, examples include asynchronous filesystem I/O, non-blocking sockets and MPI non-blocking point-to-point communication.

This chapter summarizes and extends results from the articles “Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations” [2], “Leveraging Non-blocking Collective Communication in High-performance Applications” [1], “Sparse Non-Blocking Collectives in Quantum Mechanical Calculations” [7] and “Communication Optimization for Medical Image Reconstruction Algorithms” [14]. The following section analyzes several application kernels for the applicability of non-blocking collective operations and derives general principles like pipelining. Section 2 discusses the usage of the extended collective operations introduced in Chapter III in two real-world applications.

1 Leveraging Non-Blocking Collective Communication

“Research is what I’m doing when I don’t know what I’m doing.” – Wernher von Braun, (1912-1977) German Physicist, National Medal of Science 1975

Another key component to efficient hardware utilization (and therefore to high performance) is the abstraction of collective communication. High-level group communications enable communication optimizations specific to hardware, network, and topology. A principal advantage of this approach is performance-portability and correctness [83].

The use of non-blocking collective operations can avoid data-driven pseudo-synchronization of the application. Iskra et al. and Petrini et al. show in [109] and [165] that this effect can cause a dramatic performance decrease.

It is easy to understand how non-blocking collective operations mitigate the pseudo-synchronization effects and hide the latency costs. However, properly applying those techniques to real-world applications turned out to be non-trivial because code must often be restructured significantly to take full advantage of non-blocking collective operations. In this chapter, we propose a scheme to make non-blocking collective operations much more straightforward to use. In particular, we propose an approach for incorporating non-blocking collectives based on library-based transformations. Our current implementation uses C++ generic programming techniques, but the approach would be applicable in other languages through the use of other appropriate tools.

Overview In this section we address in particular the following issues:

1. We show how scientific kernels can take advantage of non-blocking collective operations.
2. We show which code transformations can be applied and how they can be automated.
3. We analyze the potential benefits of such transformations.
4. We discuss limitations and performance trade-offs of this approach.

1.1 Related Work

Possible transformations to parallel codes to enable overlapping have been proposed in many studies [25, 33, 49, 63]. However, none of them investigated transformations to non-blocking collective communication. Danalis et al. [63] even suggest replacing collective calls with non-blocking send-receive calls. This is clearly against the philosophy of MPI and destroys performance portability and many possibilities of optimization with special hardware support (cf. [139, 158, 190]) completely. Our approach transforms codes that use blocking collective operations and enables the use of non-blocking collective operations and thus combines the performance portability and machine-specific optimization with overlap and easy programmability.

Several languages, like Split-C [59], UPC [102], HPF [99] or Fortran-D [100], have compilers available that are able to translate high-level language constructs into message passing code.

1.2 Principles for Overlapping Collectives

Many parallel algorithms apply some kind of element-wise transformation or computation to large (potentially multi-dimensional) data sets. Sancho et al. show several examples for such "Concurrent Data Parallel Applications" in [180] and prognose a high overlap potential. Other examples are Parallel Sorting [57], Finite Element Method (FEM) calculations, 3D-FFT [26] and parallel data compression [191].

We chose a dynamic, data-driven application, parallel compression, as a more complex example than the simple static-size transformations that were shown in previous works. The main difference here is that the size of the output data can not be predicted in advance and strongly depends on the structure of the input data. Thus, a two-step communication scheme has to be applied to the problem.

In our example, we assume that the N blocks of data are already distributed among the P processing elements (PE) and each PE compresses its blocks by calling compress(). The data will finally be gathered to a designated rank. Gathering of the compression results must be performed in two steps where the first step collects the individual sizes of the compressed data at the master process, and determines so the parameters of the second step, the final data gathering. This naive scheme is shown in Listing V.1.

```
my_size = 0;
for (i=0; i < N/P; i++) {
  my_size += compress(i, outptr);
  outptr += my_size;

}
gather(sizes, my_size);
gatherv(outbuf, sizes);
```

Listing V.1: Parallel compression naive scheme

```

for (i=0; i < N/P; i++) {
  my_size = compress(i, outptr);
  gather(sizes, my_size);
  igatherv(outptr, sizes, hndl[i]);
  if(i>0) waitall(hndl[i-1], 1);
}
waitall(hndl[N/P], 1);
```

Listing V.2: Transformed compression

This kind of two-step communication process is common for dynamic data-driven computations (e.g., parallel graph algorithms [143]), making it rather challenging to be generated automatically by parallelizing compilers or other tools. None of the projects discussed in the related works section can deal with the data-dependency in this example and optimize the code for overlap. Thus, we propose a library-based approach that offers more flexibility and supports the dynamic nature of the communication.

It seems that two main heuristics are sufficient to optimize programs for overlap: First, the communication should be started as early as possible. This technique is called "early binding" [66]. Second, the communication should be finished as late as possible to give the hardware as much time as possible to perform the communication. In some communication systems, messages can overlap each other. So called communication-communication overlap [34, 137] can also be beneficial.

1.3 Manual Transformation Technique

Listing V.2 shows the transformed code of Listing V.1. This simple scheme enables the communication of the n^{th} element to overlap with the computation of the $(n+1)^{\text{st}}$ element. The call `gather(sizes, my_size)` collects the local data size of all nodes into a single array `sizes` and `igatherv(outbuf, sizes, hndl[i])` starts a non-blocking communication of the buffers, gathering the correct size from every PE. The non-blocking communication is finished with a call to `waitall(hndl, num)` that waits for `num` communications identified by handles starting at `hndl`.

However, our previous works involving overlap, such as optimization of a three-dimensional Poisson solver [2] or the optimization of a three-dimensional Fast Fourier Transform [4] showed that this simple heuristic is not sufficient to achieve good overlap. The two main reasons for this have been found in the theoretical and practical analysis of non-blocking collective operations in Section 2. This analysis shows that the overlap of non-blocking collective operations is relatively low for small messages but grows with message-size. This is due to some constant CPU intensive overheads such as issuing messages or managing communication schedules and the faster "bulk" transfer [29] of larger messages. Furthermore it can be more beneficial to give every communication more time to complete before waiting for it. Another conflicting issue could be that some MPI implementations have problems to manage many outstanding requests and this results in significantly degraded performance. We provide separate solutions to those problems in the following.

1.3.1 Loop Tiling

The fine grained communication can be coarsened by loop tiling. This means that more computation is performed before a communication operation is started. Listing V.3 shows such a transformation.

```
for (i=0; i < N/P/t; i++) {
  my_size = 0;
  for (j=i; j < i+t; j++) {
    my_size += compress(i*t+j, outptr);
    outptr += my_size;
  }
  if (i > w) waitall(hndl[i-w], 1);
  gather(sizes, my_size);
  igather(outbuf, sizes, hndl[i]);
}
waitall(hnld[N/P/t-w], w);
```

Listing V.5: Final compression transformation

```
for (i=0; i < N/P/t; i++) {
  my_size = 0;
  for (j=i; j < i+t; j++) {
    my_size += compress(i*t+j, outptr);
    outptr += my_size;
  }
  gather(sizes, my_size);
  igatherv(outbuf, sizes, hndl[i]);
  if(i>0) waitall(hndl[i-1], 1);
}
waitall(hndl[N/P/t], 1);
```

Listing V.3: Compression after loop-tiling

```
for (i=0; i < N/P; i++) {
  my_size = compress(i, outptr);
  outptr += my_size;
  if (i > w) waitall(hndl[i-w], 1);
  igather(sizes, my_size);
  igatherv(outbuf, sizes, hndl[i]);




}
waitall(hndl[N/P-w],w);
```

Listing V.4: Compression with a window

1.3.2 Communication Window

Allowing more than a single outstanding request at a time gives the opportunity to finish the communication later (possibly at the end) and allow communication/communication overlap. However, too many outstanding requests create a large overhead and can slow down the application significantly. So, the number of outstanding requests must be chosen carefully. This transformation is shown in Listing V.4. Both schemes can be combined efficiently to transform our example as illustrated in Listing V.5.

1.3.3 Tuning the Parameters

These schemes introduce a trade-off that is caused by the pipelined fashion of the transformed algorithm (cf. pipeline theory [94]). The speedup of a pipeline is usually limited by the start-up overhead needed to fill the pipe. Our two-stage pipeline has a start-up time of a single communication. In our special case, this overhead is not a start-up time at the beginning but rather a drain time at the end because the last operation can usually not be overlapped. This means that a high tiling factor can lead to higher drain times without overlap. Thus, the tiling factor has to be chosen big enough to allow for bulk transfers and efficient overlap, but also not too big to cause high pipeline drain times. The window factor is important to give the single communications more time to finish, but a too high window might cause performance degradation in the request matching of the underlying communication system. Both factors have to be optimized carefully to every parallel system and application.

1.4 Programmer-directed Collectives Overlap

The previous section described several code transformation schemes to leverage non-blocking collective operations. However, our experiences show that applying those schemes manually is error-prone and time-consuming. Fully automatic transformation with the aid of a compiler will demand an extremely elaborate data dependency analysis to guarantee inter-loop independence and can simply not handle many cases. We propose a flexible generic approach that does not require external software but only a standard compliant C++ compiler. The basic idea is to separate the communication from the computation and rearrange them while optimizing tiling factor and window-size to get the highest possible performance. The resulting generic communication pattern represents a huge class of applications.

The separation of communication and computation introduces two functors that have to be implemented by the programmer. The interplay between those functors is defined in the template library and can be parametrized in order to achieve the maximum performance. The two functors are called `computation` and `communication`.

`computation(i)` computes step `i` of an iterative parallel application. The input data can either be read in the object or generated on the fly. The results of the computation are written into a `buffer` object, which is selected by the template. A `buffer` object stores the computed data and acts as source and destination buffer for the `communication()` functor. The `communication` functor communicates the data in the buffer it is called with. The `buffer` object itself is not required to store actual data, it can contain references to some other containers to avoid copying. All functors and the buffer object are implemented by the application programmer. Our template library combines those building blocks to enable efficient pipelining. The elements are described in detail in the following.

Computation defines the function call operator that is called for every input element by the template. A user-supplied `buffer` to store the output data is also passed to the function. Listing V.6 shows the computation functor for our compression example.

```
class computation_t {
  void operator() (int i, buffer_t& buffer) {
    buffer.size += compress(i, buffer.ptr);
    buffer.ptr += buffer.size;
} } computation;
```

Listing V.6: Computation functor

Communication defines the parenthesis operator and gets called with a buffer in order to communicate the buffer's contents. The communication functor for the compression is given in Listing V.7.

```
class communication_t {
  void operator() (buffer_t& buffer) {
    gather(sizes, buffer.size);
    igatherv(outbuf, sizes, buffer.hndl);
} } communication;
```

Listing V.7: Communication functor

Buffer is used to store computation results and to communicate them later. All communication bookkeeping is attached to the buffer.

```
std::vector<buffer> buffers[w];
pipeline_tiled_window(N/P, tile_size,
    computation, communication, buffers);
```

Listing V.8: Templated function call

With the two functors at hand, communication overlap of real-world applications can be augmented by tiling and communication windows without modifying the user code. This is demonstrated in Listing V.8.

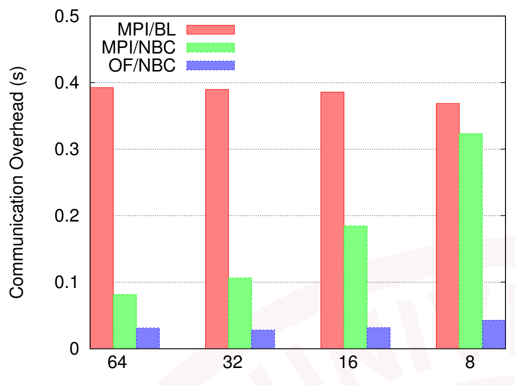
The function template `pipeline_tiled_window` is parametrized in order to tune the tile size - the second function argument - and the number of windows. The latter is defined implicitly by the number of buffers (`buffers.size()`). In the remainder of this section we demonstrate the performance impact of this tuning.

1.5 Performance Results

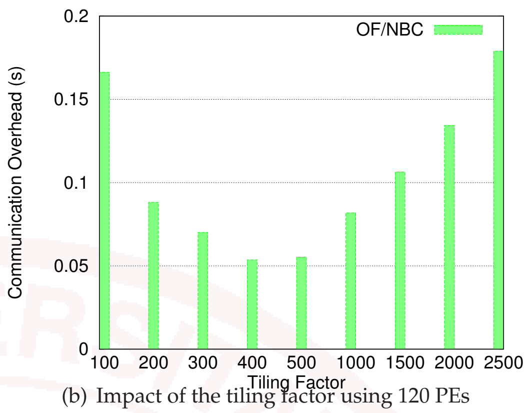
We implemented the example with our template transformation scheme to analyze the parallel performance. We used the g++ 3.4.6 compiler with Open MPI 1.2 [76] and the InfiniBand optimized version of LibNBC to implement the MPI communication and libbzip2 for the data compression. All benchmarks have been executed on the *Odin* cluster at Indiana University. We used only a single processor per node to reflect our communication optimization. The nodes are interconnected with Mellanox InfiniBand^{TM} adapters and a single 288 port switch.

Figure V.1(a) shows the communication overhead for the compression of 122MB random data on different node counts with blocking MPI calls, non-blocking calls with the standard LibNBC (based on MPI_Isend/Irecv calls performing NBC_Test to progress) and our InfiniBand-optimized LibNBC/OF without tests using the wait-on-send implementation (cf. Section 3). The MPI/BL graph shows the performance of the original untransformed code, the MPI/NBC and OF/NBC the optimized code using our template library with and without InfiniBand^{TM} optimizations, respectively.

Our benchmarks show that the communication overhead of the parallel compression example can be significantly reduced with the application of our transformation templates and the subsequent use of non-blocking collective communication in a pipelined way. The compression time on 120 PEs was reduced from $2.18s$ to $1.72s$ which is an application performance gain of 21%. Figure V.1(b) the influence of the tiling factor on 120 PEs.



(a) Different number of PEs (optimal tiling and window)



(b) Impact of the tiling factor using 120 PEs

Figure V.1: Communication overheads using InfiniBand$^{\text{TM}}$ for parallel compression

1.6 Extended Example: Three-dimensional Fast Fourier Transform

A more complex example, the parallel implementation of three-dimensional Fast Fourier Transforms (FFTs), has been studied by many research groups [26, 50, 68, 81]. The application of non-blocking communication resulted in controversial results. The optimized collective all-to-all communication was replaced by a non-blocking point-to-point message pattern. Even if this enabled overlap, the communication was not optimized for the underlying architecture and different effects, such as network congestion, and therefore decreased the performance. Dubey et al. [68] mention the applicability of a non-blocking all-to-all, but they implemented a point-to-point based scheme and did not see a significant performance improvement. Calvin et al. showed performance benefits using the same scheme [50]. However, the test-cases of their study used only four compute nodes which limited the impact of congestion dramatically. We use the optimized non-blocking collective communication operations provided by LibNBC to avoid congestion and ensure proper scaling and overlap.

The three-dimensional FFT can be decomposed into three sweeps of one-dimensional FFTs, which are performed using FFTW [75] in our implementation. The data can be distributed block-wise such that the first two sweeps are computed with the same distribution. Between the second and third sweep the data must be migrated within planes of the cuboids. The computation scheme is depicted on a high level in Listing V.9 and Figure V.2.

The n^3 complex values are initially distributed block-wise in y-direction (our transformation schemes can be used to redistribute the data if this is not the case). The first step consists of n^2 1D-FFTs in z-direction. The calculation in x-direction can still be performed with the same distribution. Before computing the FFT in the y-direction the data in every z-plane must be redistributed, ideally with an all-to-all communication. Performing the calculations within the z-planes in two steps establishes a loop with independent computation and communication so that our pipelining schemes can be applied.

We applied our generic scheme to the parallel 3D-FFT. The `computation()` functor performs a single serial one-dimensional transformation with FFTW and packs the transformed data to a

```
transform all line in z–direction
for all z–planes {
  transform all lines in x–direction
  parallel transpose from x–distr
    to y–distr  /∗ all–to–all ∗/
}
for all y–lines {
  transform line in y–direction
}
```

Listing V.9: Pseudo-code of 3D-FFT

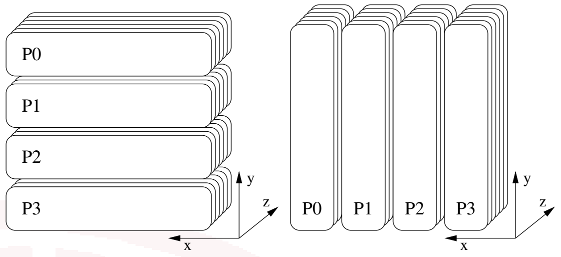
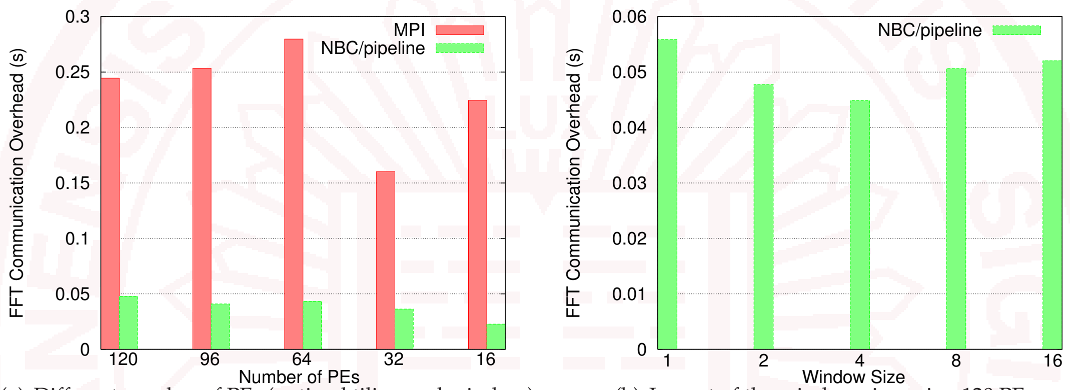


Figure V.2: Block distribution in y-direction (left) and x-direction (right)



(a) Different number of PEs (optimal tiling and window) (b) Impact of the window size using 120 PEs

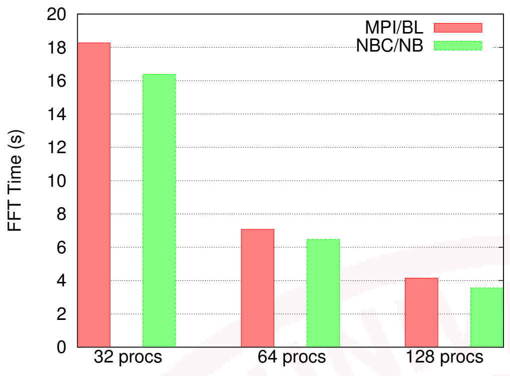
Figure V.3: Communication overheads using InfiniBand™ for parallel 3d-FFT

buffer. The `communication()` uses LibNBC's non-blocking NBC_Ialltoall to initiate the communication. The "wait" function unpacks the received data from the buffer object into the FFT-buffer.

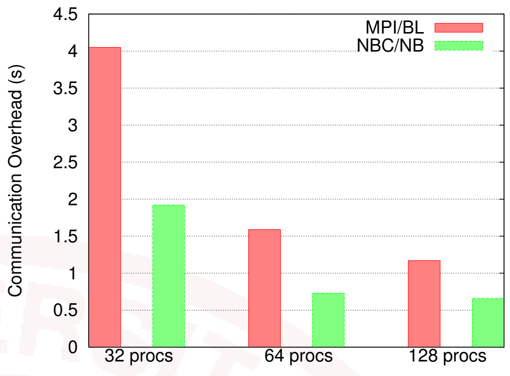
We perform multiple benchmarks on different systems to show the wide applicability and usefulness of our approach. The first series of benchmark results for a weak scaling 3D-FFT (720^3, 672^3, 640^3, 480^3 and 400^3 double complex values on 120, 96, 64, 32 and 16 PEs respectively) using InfiniBand on *Odin* are shown in Figure V.3(a). The communication overhead in this more complex example can also be reduced significantly. Our optimizations were able to improve the running time of the FFT on 120 PEs by 16% from $2.5s$ to $2.1s$.

This first result demonstrates the same performance gain for the parallel FFT as we showed for the parallel compression. Combining window and tiling leads to the highest improvement. Figure V.3(b) shows the communication overhead with regards to the window size (for a fixed optimal tiling).

In a second benchmark, we measure the strong scaling of a full transformation of a 1024^3 point FFT box (960^3 for 32 processes due to memory limitations) on the the Cray XT4, Jaguar, at the National Center for Computational Sciences, Oak Ridge National Laboratory. This cluster is made



(a) Time to solution



(b) Communication overhead

Figure V.4: Blocking and non-blocking 3d-FFT times on the XT4

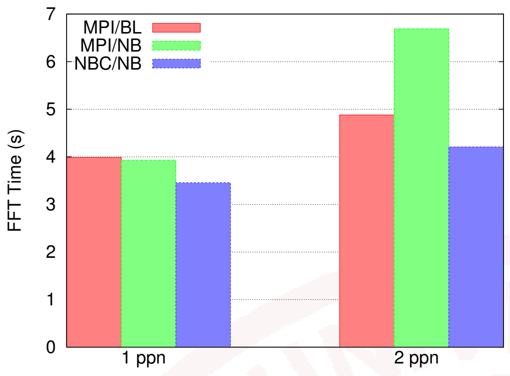
up of a total of 11,508 dual socket 2.6 GHz dual-core AMD Opteron chips, and the network is a 3-D torus with the Cray-designed SeaStar [31] communication processor and network router designed to offload network communication from the main processor. The compute nodes run the Catamount lightweight micro-kernel. All communications use the Portals 3.3 communications interface [43]. The Catamount system does not support threads and can thus not run the threaded implementation. An unreleased development version of Open MPI [76] was used to perform these measurements, as Open MPI 1.2.1 does not provide Portals communications support. However, using the NIC-supported overlap with LibNBC results in a better overall system usage and an up to 14.2% higher parallel efficiency of the FFT on 128 processes, shown in Figure V.4.

A third benchmark was run on the Coyote cluster system at Los Alamos National Labs, a 1290 node AMD Opteron cluster with an SDR InfiniBand network. Each node has two single core 2.6 GHz AMD Opteron processors, 8 GBytes of RAM and a single SDR InfiniBand HCA. The results for runs of the 1024^3 FFT box transformation on 128 processes with either 1 process per node (1ppn) or two processes per node (2ppn) are shown in Fig. V.5 . This effectively compares the efficiency of the MPI approach (perform the non-blocking collectives in a separate thread, cf. Section 1.2) with the LibNBC approach (use non-blocking point-to-point communication). We clearly see the the LibNBC approach is superior on this system. As soon as all available CPUs are used for computation, the threaded approach even slows the execution down. Our conclusion is that with the currently limited number of CPU cores, it does not pay off to invest half of the cores to process asynchronous collectives with the MPI approach; they should rather be used to perform useful computation.

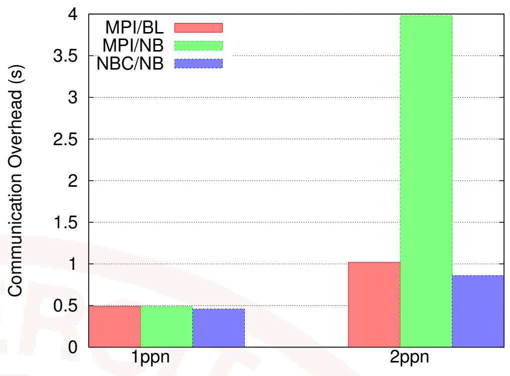
The next section looks at another class of applications where parts of the communication are already independent of the computation and can thus be overlapped easily.

1.7 Applications with Independent Communication

Scientific computing applications are particularly well-suited to benefit from the more abstract expression of parallel communication afforded by collective operations. Moreover, many algorithms



(a) Time to solution



(b) Communication overhead

Figure V.5: Blocking and non-blocking 3d-FFT times using 64 or 128 nodes (128 processes)

in scientific computing, e.g., linear solvers, provide a high potential for overlapping communication and computation. In order to combine the advantages of this overlapping with the advantages of collective communication, we introduce non-blocking collective operations as a natural addition to the MPI standard and demonstrate the performance benefits with a parallel conjugate gradient solver.

1.7.1 Optimization of Linear Solvers

Iterative linear solvers are important components of most applications in HPC. They consume, with very few exceptions, a significant part of the overall run-time of typical applications. In many cases, they even dominate the overall execution time of parallel code. Reducing the computational needs of linear solvers will thus be a huge benefit for the whole scientific community.

Despite the different algorithms and varying implementations of many of them, one common operation is the multiplication of large and sparse matrices with vectors. Assuming an appropriate distribution of the matrix, large parts of the computation can be performed on local data and the communication of required remote data — also referred to as inner boundaries or halo — can be overlapped with the local part of the matrix vector product.

1.7.2 Case Study: 3-Dimensional Poisson Equation

For the sake of simplicity, we use the well-known Poisson equation with Dirichlet boundary conditions, e.g., [89]

$$-\Delta u = 0 \quad \text{in } \Omega = (0,1)\times(0,1)\times(0,1), \tag{V.1}$$

$$u = 1 \quad \text{on } \Gamma. \tag{V.2}$$

The domain Ω is equidistantly discretized. Each dimension is split into $N+1$ intervals of size $h = 1/(N+1)$. Within Ω one defines $n = N^3$ grid points

$$G = \{(x_1, x_2, x_3) | \forall i,j,k \in \mathbb{N}, 0 < i,j,k \leq N : x_1 = ih, x_2 = jh, x_3 = kh\}.$$

Thus, each point in G can be represented by a triple of indices (i, j, k) and we denote $u(ih, jh, kh)$ as $u_{i,j,k}$. Lexicographical order allows for storing the values of the three-dimensional domain into a one-dimensional array. For distinction we use a typewriter font for the memory representation and start indexing from zero as in C/C++

$$u_{i,j,k} \equiv \mathtt{u}[(\mathtt{i}-\mathtt{1}) + (\mathtt{j}-\mathtt{1}) * \mathtt{N} + (\mathtt{k}-\mathtt{1}) * \mathtt{N}^2] \quad \forall 0 < i, j, k \leq N. \tag{V.3}$$

The differential operator $-\Delta$ is discretized for each $x \in G$ with the standard 7-point stencil

$$-\Delta_h u_{i,j,k} = \frac{6u_{i,j,k} - u_{i-1,j,k} - u_{i+1,j,k} - u_{i,j-1,k} - u_{i,j+1,k} - u_{i,j,k-1} - u_{i,j,k+1}}{h^2}.$$

Setting this equation equal to zero for all $x \in G$ provides an approximation of (V.1) on Ω. Considering that the function u is given on the boundary, the corresponding terms can be transferred to the right hand side, e.g., for $-\Delta_h u_{1,3,1}$ the equation reads

$$\frac{6u_{1,3,1} - u_{2,3,1} - u_{1,2,1} - u_{1,4,1} - u_{1,3,2}}{h^2} = \frac{u_{0,3,1} + u_{1,3,0}}{h^2} = \frac{2}{h^2}.$$

The linear operator $-\Delta_h$ can be represented as a sparse matrix in $\mathbb{R}^{n \times n}$ using the memory layout from (V.3), confer e.g., [89] for the 2D case.

1.7.3 Domain Decomposition

The grid G is partitioned into p sub-grids $G_1, \ldots, G_p$ where p is the number of processors. The processors are arranged in a non-periodic Cartesian grid $p_1 \times p_2 \times p_3$ with $p = p_1 \cdot p_2 \cdot p_3$, provided by MPI_Dims_create. In case that N is divisible by $p_i \forall i$ the local grids on each processor have size $N/p_1 \times N/p_2 \times N/p_3$, otherwise the local grids are such that the whole grid is partitioned and the sizes along each dimension vary at most by one.

Each sub-grid has 3 to 6 adjoint sub-grids if all $p_i > 1$. Two processors P and P' storing adjoint sub-grids are neighbors, written as the relation $Nb(P, P')$. This neighborhood can be characterized by the processors' Cartesian coordinates $P \equiv (P_1, P_2, P_3)$ and $P' \equiv (P'_1, P'_2, P'_3)$

$$Nb(P, P') \quad \text{iff} \quad |P_1 - P'_1| + |P_2 - P'_2| + |P_3 - P'_3| = 1. \tag{V.4}$$

Fig. V.6 shows the partition of G into sub-grids and necessary communication.

1.7.4 Design and Optimization of the CG Solver

The conjugate gradient method (CG) by Hestenes and Stiefel [96] is a widely used iterative solver for systems of linear equations when the matrix is symmetric and positive definite. To provide a simple base of comparison, we refrain from preconditioning [89] and from aggressive performance tuning [84]. However, the local part of the dot product is unrolled using multiple temporaries, the two vector updates are fused in one loop, and the number of branches is minimized in order to provide a high-performance base case. The parallelization of CG in the form of Listing V.10 is straight-forward by distributing the matrix and vectors and computing the vector operations and the contained matrix vector product in parallel.

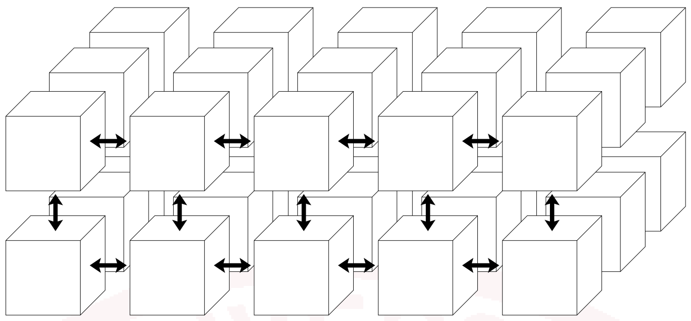


Figure V.6: Processor Grid

```
 1 while (sqrt(gamma) > epsilon * error_0) {
     if (iteration > 1)
       q = r + gamma / gamma_old * q;
     v = A * q;
 5   delta = dot(v, q);
     alpha = delta / gamma;
     x = x + alpha * q;
     r = r - alpha * v;
     gamma_old = gamma;
10   gamma = dot(r, r);
     iteration = iteration + 1;
   }
```

Listing V.10: Pseudo-code for CG method

Neglecting the operations outside the iteration, the scalar operations in Listing V.10 — line 1, 2, 6, 9, and 11 — and part of the vector operations — line 3, 7, and 8 — are completely local. The dot products in line 5 and 10 require communication in order to combine local results with MPI_Allreduce to the global value. Unfortunately, computational dependencies avoid overlapping these reductions. Therefore, the whole potential to save communication time in a CG method lies in the matrix vector product — line 4 of Listing V.10.

1.7.5 Parallel Matrix Vector Product

Due to the regular shape of the matrix, it is not necessary to store the matrix explicitly. Instead the projection $u \mapsto -\Delta u$ is computed. In the distributed case $p > 1$, values on remote grid points need to be communicated in order to complete the multiplication. In our case study, the data exchange is limited to values on outside planes of the sub-grids in Fig. V.6 unless the plane is adjoint to the boundary Γ. Therefore, processors must send and receive up to six messages to their neighbors according to (V.4) where the size of the message is given by the elements in the corresponding outer plane.

However, most operations can be already executed with locally available data during communication as shown in Listing V.11. The first command copies the values of `v_in` needed by

```
void matrix_vector_mult(struct array_3d *v_in,
                         struct array_3d *v_out,
      struct comm_data_t *comm_data)
{
  fill_buffers(v_in, &comm_data->send_buffers);
  start_send_boundaries(comm_data);
  volume_mult(v_in, v_out, comm_data);

  finish_send_boundaries(comm_data);
  mult_boundaries(v_out, &comm_data->recv_buffers);
}
```

Listing V.11: Implementation of parallel matrix vector product

```
void start_send_boundaries(struct comm_data_t *comm_data)
{
  /* Compute displacements */
  if (comm_data->non_blocking)
    NBC_Ialltoallv(sbuf.start, scounts, sdispls, MPI_DOUBLE,
       rbuf.start, rcounts, rdispls, MPI_DOUBLE,
       processor_grid, comm_data->handle);
  else {
    MPI_Alltoallv(sbuf.start, scounts, sdispls, MPI_DOUBLE,
      rbuf.start, rcounts, rdispls, MPI_DOUBLE, processor_grid);
}
```

Listing V.12: Code for starting communication

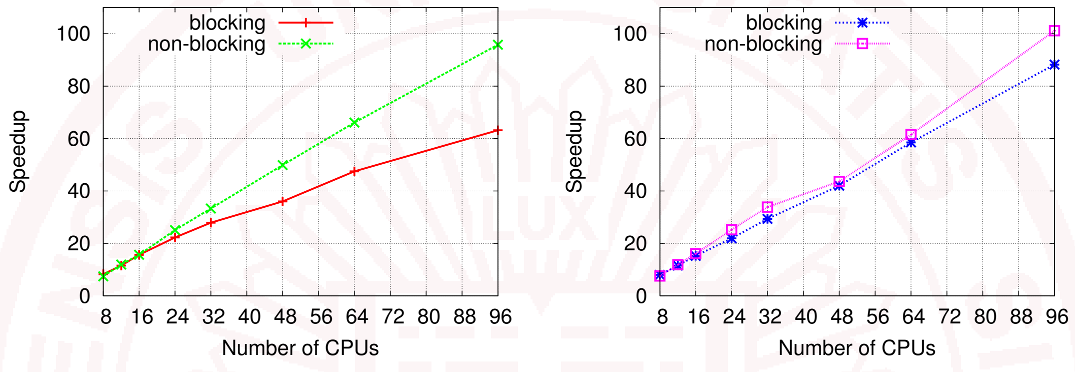
other processors into the send buffers. Then an all-to-all communication is launched, which can be a blocking operation using MPI_Alltoallv or a non-blocking operation using NBC_Ialltoallv, Listing V.12. The last function has the same arguments as the first one with an additional NBC_Handle that is used to identify the operation later. The command `volume_mult` computes the local part of the matrix-vector product and in case of non-blocking communication, NBC_Test is called periodically with the handle returned by NBC_Ialltoallv in order to progress the non-blocking operations. Before using remote data in `mult_boundaries`, the completion of NBC_Ialltoallv is checked in `finish_send_boundaries` with an NBC_Wait on the NBC_Handle, Listing V.13.

```
void finish_send_boundaries(struct comm_data_t *comm_data)
{
   if (comm_data->non_blocking)
     NBC_Wait(comm_data->handle);
   gt2 = MPI_Wtime();
}
```

Listing V.13: Code for finishing communication

1.7.6 Benchmark Results

We performed a CG calculation on a grid of $800 \times 800 \times 800$ points until the residual was reduced by a factor of 100, which took 218 iterations for each run. This weak termination criterion was chosen for practical reasons in order to allow more tests on the cluster. We verified on selected tests with much stronger termination criteria that longer executions have the same relative behavior. The studies were conducted on the *Odin* cluster. Fig. V.7 shows the benchmark results using Gigabit Ethernet and InfiniBand^{TM} up to 96 nodes. The presented speedups are relative to a single-processor run without any communication. We see that the usage of our NBC library resulted in a reasonable



(a) Gigabit Ethernet (b) InfiniBand

Figure V.7: Parallel speedup of a 3d Poisson solver for different network interconnects.

performance gain for nearly all node counts. The explicit performance advantage is shown in Fig. V.8. The performance loss at 8 processors is caused by relatively high effort to test the progress of communication.

The overall results show that for both networks, InfiniBand^{TM} and Gigabit Ethernet, nearly all communication can be overlapped and the parallel execution times are similar. The factor of 10 in bandwidth and the big difference in the latency of both interconnects does not impact the run-time significantly, even if the application has high communication needs. The partially superlinear speedup is due to cache effects in the inner part of the matrix vector product.

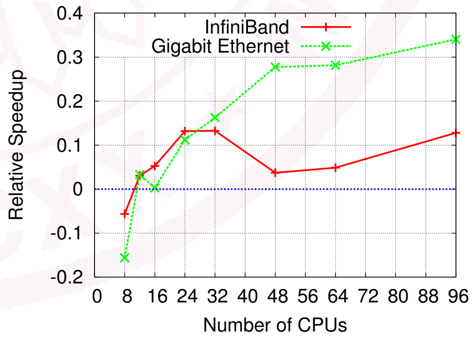


Figure V.8: Speedup using LibNBC

1.7.7 Comparison to non-blocking Point-to-Point Messaging

Due to the our design, non-blocking point-to-point communication would perform almost equally while requiring the user to program the management for multiple communication handlers including the progress enforcement. Using collective communication simplifies programming, increases maintainability and enables portable performance tuning. Possible optimizations for Alltoallv calls include the exploitation of network concurrency (cf. [200]), optimization for special parallel systems like BlueGene/L (cf. [30]) or automatically tuning the communication plans (cf. [71]).

We are aware that the (slow but steady) linear growing of the displacement and count arrays with the communicator size can introduce scalability problems on very large machines (e.g., BlueGene/L). However, the better programmability, high optimization potential, and clearer code-structure (cf. [83]) outweigh those concerns.

Based on those discoveries, we decided to investigate the use of topological collective operations discussed in Section 1 in Chapter III. Those operations enable easy and efficient implementation of sparse collective communication patterns such as nearest neighbor collective operations. We will analyze their impact on programmability as well as on performance with a real-world application using a linear solver in Section 2.2.

1.7.8 Optimization Impact on Other Linear Solvers

The approach described above can be used to optimize other linear solvers in addition to CG. We discuss some ideas for the application of non-blocking collective operations to different algorithms.

As with CG, other Krylov sub-space methods, such as GMRES [177], CG Squared [192], or BI-CGStab [209], have dependencies that similarly limit the potential of overlapping communication and computation for their reduction operations. On the other hand, the preconditioners that are typically used in conjunction with Krylov sub-space iterations often consist of operations that are similar to matrix-vector product, e.g., incomplete LU or Cholesky factorization, and thus have the potential of overlapping.

Classical iterative solvers — Richardson iteration, Jacobi, and Gauß-Seidel relaxation — consist only of operations similar to matrix-vector product. Such iterations therefore offer the potential for significant overlap in contrast to CG and related Krylov sub-space methods that require reduction operations. Unfortunately, due to the slow convergence of these methods, their importance as iterative solvers is limited.

The classical methods are important, however, as “smoothers” in multigrid methods (MG) [202]. In addition to the smoothing process within each level of the multigrid, corresponding sub-grids of adjoint levels are related to each other by interpolation operators. These operators involve communication to interpolate values close to sub-grid boundaries. The amount of communication increases for higher orders of interpolation. Grid values inside the sub-grids can be interpolated with local data — assuming the grids are similarly decomposed — so that the interpolation operators allow for communication overlapping. Particularly on smaller grids, communication becomes a severe bottleneck and non-blocking communication provides the potential for significant improvements. As multigrid methods are solvers with minimal numerical complexity, they are important in scientific computing and we will investigate them in detail in future work.

1.8 Map-Reduce

Another programming-pattern that can be used to express a large class of applications is map-reduce [64, 127]. This generic scheme is used by google to run many of its sevices at large scale.

A map-reduce program is composed of two functions: map and reduce. Both functions are well known in functional programming. Map accepts an input pair, applies the implemented function and emits a number of <*key,value*> pairs. Reduce accepts a key and a number of values and emits a single result.

A typical map-reduce program is the search for the number of given strings in a files. The map function accepts an input file (or a part of it) and a vector of strings s. It searches the input for the strings and emits the pair < *s,1*>. The reduce function simply sums all the numbers for a given s together.

```
void map(filem f, keys strs) {
 for(i=0; i<strs.size(); i++) {
  char *ptr=f.start_addr();
  while(ptr<f.end_addr()-strs.len) {
   if(!memcmp(ptr, str[i], strs.len))
    EmitIntermediate(i, 1);
   ptr++;
 } } }
```

Listing V.14: Map Example Function

```
void reduce(key str, values num) {
  int sum=0;
  for(i=0; i<values.size(); i++) {
    sum += values[i];
  }
  Emit(sum);
}
```

Listing V.15: Reduce Example Function

The remaining functionality is offered by the map-reduce library. This programming concept allows automatic parallelization of large input sets. The library can split the input data into chunks that are distributed to multiple processing elements. For example in the case of string-search, the file can be split into multiple parts and the search can be assigned to multiple hosts. The reduce-function sums up the local and distributed counts of the search strings. However, the concept does not offer any data-distribution mechanism, i.e., the programmer of the map-reduce program has to ensure that the data is available to the map function. Optimizations can be supported by the framework, for example parallel filesystem like GFS [79] could be used to map the tasks as close to the data as possible [169]. The library schedules map and reduction tasks to the available PEs and can thus do automatic load-balancing and react to node failures without user intervention.

Map-reduce was not intended to be implemented on top of MPI, and MPI lacks several features that are needed to implement it efficiently. Missing features are for example fault tolerance (i.e., the ability to react to erroneous processes) and variable sized reduction operations (map-reduce reductions can return values that are of a different size than the input values, e.g., string concatenations; MPI only supports fixed-size vectors for reductions). However, if we ignore fault tolerance and enforce only fixed-size reductions (this limits the model but still reflects a large class of applications), we can implement map-reduce on top of MPI.

Map-reduce is a master-worker model, i.e., the computation (map and reduce tasks) is scheduled by a central master process and executed by slave processes. MPI is well suited to implement this concept by using rank 0 as master process and all other $P-1$ ranks as slaves. In this model, it is useful to group a map and a reduce task together because the reduced data (result of the map-task) is already in the right processes' memory. This makes it possible to use collective operations to perform the map and reduce task. The map task can be implemented as a simple scatter operation and the reduce task can use a reduction. Using collective operations forces the reduction operation to accept and return only fixed-size data vectors. But using collective MPI reductions has a high optimization potential. The user can still implement his own reduction scheme over point-to-point if the size of the reduced data changes. However, we assume unchanged data-sizes as we have in our example. The reduction can now be implemented as an MPI Operation (or a predefined operation and datatype can be used which even enables hardware optimizations such as [77]).

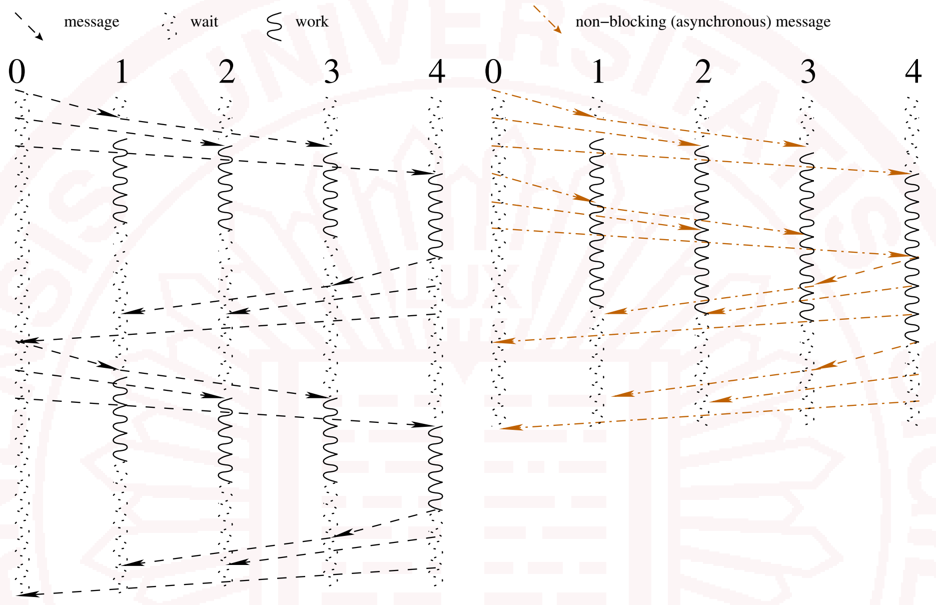


Figure V.9: Map-Reduce Scheme implemented with MPI and NBC Collectives, assuming 5 processes executing 8 tasks and a binomial tree scatter and reduction algorithm. The left part shows the execution in the MPI model and the right part in the NBC model.

1.8.1 Two Simple Map-Reduce Applications

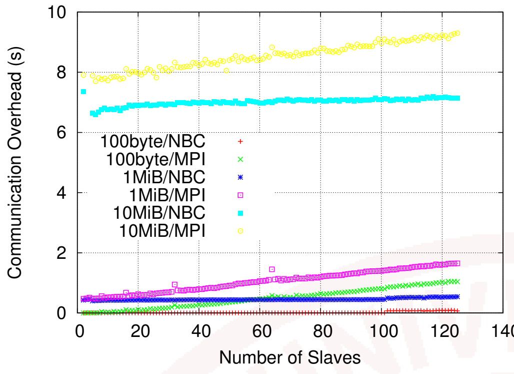
We implemented the simple string-search and an artificial, but more flexible, application in the map-reduce model on top of MPI. The string search was easy to implement and shows the applicability of the MPI-based map-reduce scheme. However, we decided to implement an application that is able to simulate a large class of map-reduce programs. This application accepts four parameters: the number of tasks, minimum (min) and maximum (max) task duration and the size of the reduction operation. The application issues workpackets that take a random time in the interval $[min, max]$. The times are evenly distributed. A slave process retrieves the work-packet and computes a simple loop that takes the received time (emulates the map operation). After the computation is finished, the slave starts the reduce (by calling an MPI reduction). The execution scheme is shown in left part Figure V.9.

We can easily apply non-blocking collective operations to the map as well as the reduce functionality. In the map-case, the master starts w_m non-blocking scatter operations at the beginning before it waits for the first one to complete. It issues the next one after the first one completes. This efficiently creates a "window" of scatter operations that run in the background. Their latency can be ignored if the window size is reasonably large and the work-packets take long enough to compute. The same principle can also be applied if the work-packets contain actual data. The trade-off there is that the memory requirements grow with the window size, i.e., data for all running operations has to be in main memory at the root process. The slave processes similarly start w_m non-blocking scatter operations and re-post new non-blocking scatter until the signal to exit is received. In the reduce case, we have to define a set of w_r buffers to support w_r outstanding non-blocking operations. A similar window-technique as in the map-operation is used to have w_r outstanding NBC_Ireduce operations at any time. If all n buffers are busy, the "oldest" reduction is finished by calling NBC_Wait and the Reduce function is performed locally on the root-node again. The remaining outstanding communications have to be finished and their buffers reduced when all tasks are done. The resulting execution scheme is shown in the right part of Figure V.9.

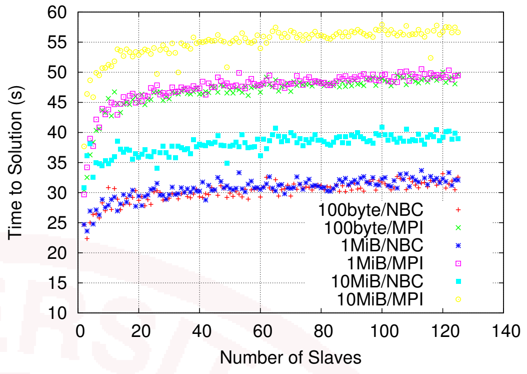
1.8.2 Performance Results

Most map-reduce applications process large amounts of data that have to be read from either the network or local disks. Thus, we assume that the I/O bandwidth is not sufficient to keep multiple processing elements busy. However, most of today's systems are multi-core or SMP systems such that there are idle resources available to offload the communication. We use the threaded InfiniBand-optimized version of LibNBC for all benchmarks. This efficiently results in offloading the Reduce-task to another core (the reduce operation is a part of the MPI_Reduce communication) transparently to the application developer. Benchmarks of the simple string-search example were also covered by the more extensive simulator and delivered exactly the same results. Thus, we only present benchmark results for the different configurations of the simulator.

We benchmarked two different workload-scenarios with 1 to 126 slave nodes with 10 tasks per node. We compared the threaded version of LibNBC with a maximum of 5 outstanding collective operations with Open MPI 1.2.6. We also varied the data-size of the reduction operation (in our example, we used summation as the reduce operation). Figure V.10(a) shows the communication and synchronization overhead for a static workload of 1 second per packet. Using non-blocking collective results in a significant performance increase because nearly all communication can be overlapped. The remaining communication overhead is due to InfiniBand$^{\text{TM}}$'s memory registration which is done on the host CPU. The graphs show a reduction of communication and synchronization overhead of up to 27%. Figure V.10(b) shows the influence of non-blocking collectives to dynamic workloads varying between 1ms and 10s. The significant performance increase is due to avoidance of synchonization and the use of communication/computation overlap. This clearly shows that our technique can be used to benefit map-reduce-like applications significantly. The dynamic example shows improvements in time to solution of up to 25%.



(a) Communication Overhead of Static Workload



(b) Time to Solution of Dynamic Workload

Figure V.10: Overhead and Time to Solution for Static and Dynamic Workloads for different Number of Slaves

2 Case Studies with Real-World Applications

"In the good old days physicists repeated each other's experiments, just to be sure. Today they stick to FORTRAN, so that they share each other's programs, bugs included." – Edsger Dijkstra (1930-2002), Dutch computer scientist, Turing Award 1972

In this chapter, we finally apply the described techniques to real-world applications. All described applications are run with realistic workloads and at reasonable scale, i.e., the communication overheads are not higher than 25-30%. We expect to lower the communication overhead with the application of non-blocking techniques. We have shown in the previous section that this overhead can be significantly reduced from 50% down to 10% in the ideal case. Often, non-blocking techniques can only be applied to some of the operations causing communication overhead. Thus, we expect 5-15% performance increase for the real applications and realistic workloads. We analyze two applications, a software that uses global reduction operations to reconstruct medical images and a quantum mechanical simulation that performs nearest neighbor communication as part of the inner solver.

2.1 Medical Image Reconstruction

Many modern medical methods for diagnosis and treatment require highly accurate, high-resolution 3D images of the inside of a human body. In order to provide the required accuracy and resolution, modern reconstruction algorithms in medical imaging are becoming more complex and time-consuming. In this section, we study Positron Emission Tomography (*PET*) reconstruction, where one of the most popular, but also most time-consuming algorithms—the list-mode OSEM algorithm—requires several hours on a common PC in order to compute a 3D reconstruction. With advanced algorithms that incorporate more physical aspects of the PET process, computation times are rising even further [126]. This motivates the parallelization of the algorithm on multiprocessor clusters [182].

Our current parallel implementation uses MPI collective operations and OpenMP [160, 161]. Collective operations allow the programmer to express high-level communication patterns in a portable way, such that implementers of communication libraries provide machine-optimized algorithms for those complex communications.

To reduce the communication overhead in our case study, we transform our code to leverage non-blocking collective operations offered by LibNBC, which provide—additionally to the overlapping of communication with computation— high-level communication offload using the InfiniBand network. We analyze the code transformations and provide an analytical runtime model that identifies the overlap potential of our approach.

2.1.1 List-Mode OSEM Algorithm

PET is a medical imaging technique that displays metabolic processes in a human or animal body. PET acquisition proceeds as follows: A slightly radioactive substance which emits positrons when decaying is applied to the patient who is then placed inside a *scanner*.

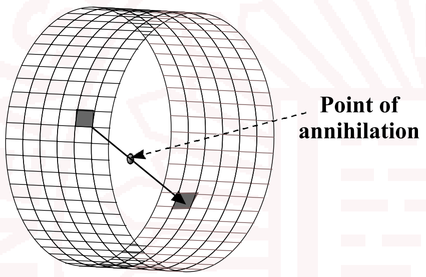


Figure V.11: Detectors register an event in a PET-scanner with 6 detector rings

```
for each(iteration k){
 for each(subiteration l){
  for (event $i \in S_l$){
    compute $A_i$
    compute $c_l += (A_i)^t \frac{1}{A_i f_l^k}$ }
  $f_{l+1}^k = f_l^k c_l$}
 $f_0^{k+1} = f_{l+1}^k$}
```

Listing V.16: Sequential list-mode OSEM algorithm.

The detectors of the scanner measure so-called events: When the emitted positrons of the radioactive substance collide with an electron residing in the surrounding tissue near the decaying spot (up to 3 mm from the emission point), they are annihilated. During annihilation two gamma rays emit from the annihilation spot in opposite directions and form a line, see Fig. V.11. These gamma rays are registered by the involved detectors; one such registration is called an *event*. During one investigation, typically 10^7 to $5 \cdot 10^8$ events are registered, from which a reconstruction algorithm computes a 3D image of the substance's distribution in the body.

In this work, we focus on the highly accurate, but also quite time-consuming list-mode OSEM (Ordered Subset Expectation Maximization) reconstruction algorithm [174] which computes the image f from the m events saved in a list.

The algorithm works block-iteratively: in order to speed up convergence, a complete iteration over all events is divided into s subiterations (see Listing V.16). Each subiteration processes one block of events, the so-called subset. The starting image vector is $f_0 = (1, ..., 1) \in \mathbb{R}^N$, where N is the number of voxels in the image being reconstructed. For each subiteration $l \in 0, ..., s-1$, the events in subset l are processed in order to compute a new, more precise reconstruction image f_{l+1},

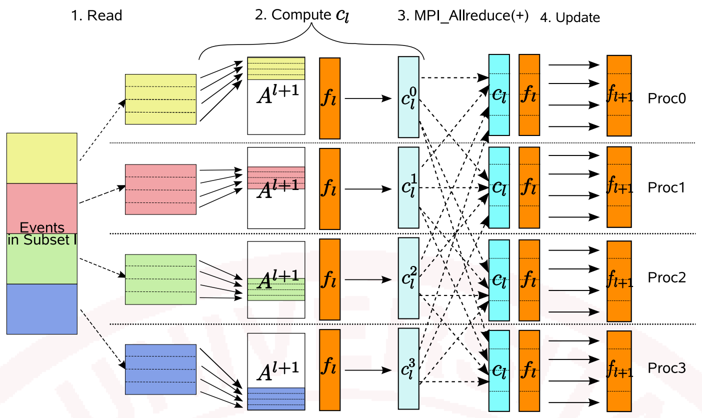


Figure V.12: Parallel list-mode OSEM algorithm on four nodes with the blocking MPI_Allreduce using four OpenMP threads per node

which is used again for the next subiteration as follows:

$$f_{l+1} = \underbrace{\frac{1}{A^t_{norm}\mathbf{1}}}_{:=a} f_l c_l; \quad c_l = \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}, \tag{V.5}$$

where S_l are the indices of events in subset l, $\mathbf{1} = (1, ..., 1)$. For the i-th row A_i of the so-called system-matrix $A \in \mathbb{R}^{m \times N}$, element a_{ik} denotes the length of intersection of the line between the two detectors of event i with voxel k. The so-called normalization vector $a = \frac{1}{A^t_{norm}\mathbf{1}}$ is independent of the current subiteration and can thus be precalculated. In the computation of f_{l+1} the multiplication of $a f_l c_l$ is performed element by element.

After one iteration over all subsets, the reconstruction process can either be stopped, or the result can be improved with further iterations over all subsets (see pseudocode in Listing V.16). Note that the optimal number of events per subset $m_s = m/s$ only depends on the scanner geometry and is thus fixed (for our scanner [181], it is $m_s = 10^6$).

2.1.2 Algorithm Parallelization Concept

Two strategies to parallelize the list-mode OSEM algorithm exist: one distributes the events, the other the 3D image among the processor. In [182] we showed that the first strategy outperforms the second in almost all cases and we therefore chose this first strategy for our parallelization: Since f_{l+1} depends on f_l we parallelize the computations within one subset. We decompose the input data, i.e., the events of one subset into P (=number of nodes) blocks and process each block simultaneously. The calculations for one subset includes four steps on every node j ($\forall\, j = 1, \ldots, P$) (cf. Fig. V.12):

1. Read m_s/P events.
2. Compute $c_{l,j} = \sum_{i \in S_{l,j}} (A_i)^t \frac{1}{A_i f_l}$. This includes the on-the-fly computation of A_i for each event in $S_{l,j}$.
3. Sum up $c_{l,j} \in \mathbb{R}^N$ ($\sum_j c_{l,j} = c_l$) with MPI_Allreduce.
4. Compute $f_{l+1} = f_l c_l$.

We implemented steps 1 and 3 (i.e., the reading of data and the actual communication of the parallel algorithm) using MPI_File_Read and blocking MPI_Allreduce. We start one process per node and support SMP clusters by additionally parallelizing steps 2 and 4 using OpenMP.

2.1.3 Parallel Algorithm with Non-Blocking Collectives

In order to optimize the parallel algorithm, we reduce the overhead arising from the Allreduce step by overlapping its communication with computations that are independent of the communicated data.

We overlap the reading of events for subset l and the computation of the corresponding submatrix A^l (which is composed of rows $i \in S_l$) with the communication of c_{l-1} of the preceding subset (see Fig. V.13). Hence, the non-blocking parallel algorithm on nodes j ($\forall\, j = 1, \ldots, P$) reads as follows:

1. Read m_s/P events in the first subset.
2. Compute $c_{l,j} = \sum_{i \in S_{l,j}} (A_i)^t \frac{1}{A_i f_l}$. This includes the on-the-fly computation of A_i for each event in $S_{l,j}$ in the first subset. Beginning from the second subset, rows A_i have already been computed in parallel with NBC_Iallreduce.
3. Start NBC_Iallreduce for $c_{l,j}$ ($\sum_j c_{l,j} = c_l$).
4. In every but the last subset, each node reads the m_s/P events for subset $l+1$ and computes A_i for subset $l+1$.
5. Perform NBC_Wait to finish NBC_Iallreduce.
6. Compute $f_{l+1} = f_l c_l$.

Note that in this approach, A^l has to be kept in memory. If not enough memory is available, one part A^l can be computed as in the original version in step 2 and the other part in step 4. Also, since A_i is precomputed, the computation of $c_l = (A_i)^t \frac{1}{A_i f_l}$ could cause CPU cache misses that influence the performance.

2.1.4 Analyzing the Overlap Potential

In order to identify the overlap potential of our approach, we develop an analytical runtime model for the overlappable computations. We denote the sequential time to compute the m_s rows of A^l by $t^1_{A_l}(m_s)$ and the time to read each node's m_s/P events by $t^P_{read}(m_s/P)$. If we assume that $t^P_{read}(m_s/P) \approx t^P_{read}(m_s)/P$, we obtain a computational overlap time per subset with one thread on each of the P nodes of

$$t^P_{CompOver} = t^P_{read}(m_s/P) + t^1_{A^l}(m_s)/P \approx (t^P_{read}(m_s) + t^1_{A^l}(m_s))/P \qquad \text{(V.6)}$$

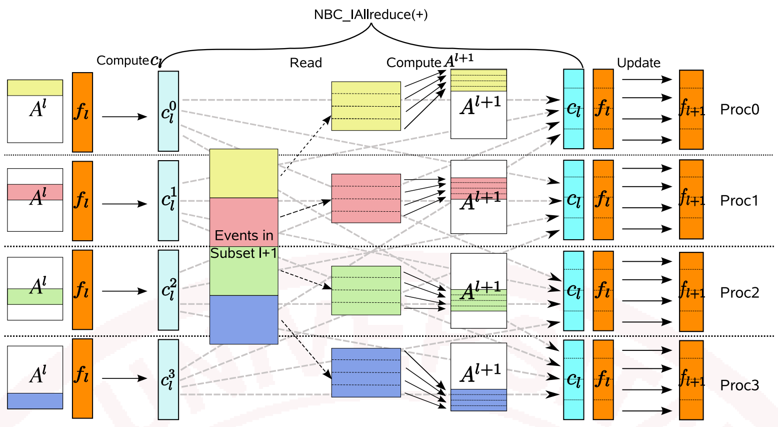


Figure V.13: Parallel list-mode OSEM algorithm on four nodes with the non-blocking NBC_Iallreduce using four OpenMP threads per node

We will verify our model (V.6) with experiments in Section 2.1.5.

On q cores per node, the ideal parallel efficiency with our OpenMP parallelization would be $\alpha(q) = t^1_{A_i}/(t^q_{A_i} \cdot q) = 1$. However, with an increasing number of threads sharing the cache, cache misses increase considerably and thus our OpenMP implementation scales worse than ideally on multi-core machines. For example, on a quad-core processor, efficiency is $\alpha(4) = 0.5$.

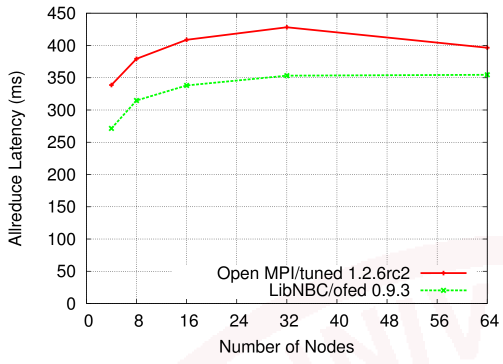
Note that on systems where file I/O and MPI communication share the same network, the overlapping of reading of data and communication might be limited due to the network's bandwidth. Hence, in the worst case, with the network fully loaded by MPI communication, $t^P_{CompOver} = t^1_{A^l}(m_s)/P$.

Fig. V.14(a) shows a comparison of the "blocking performance"[1] of LibNBC 0.9.3 with the "tuned" collective module of Open MPI 1.2.6rc2. The measurements were done with NBCBench [15] on the *odin* cluster at Indiana University. *Odin* uses NFSv3 over Gigabit Ethernet as file system and the Intel compiler suite version 9.1. LibNBC's Allreduce uses multiple communication rounds (cf. [11]). This requires the user to ensure progress manually by calling NBC_Test or run a separate thread that manages the progression of LibNBC (i.e., progress thread). Fig. V.14(b) shows the communication overhead with and without a progress thread under the assumption that the whole communication latency can be overlapped with computation (i.e., the overhead is a lower bound) and the progress thread runs on a spare CPU core (the overhead with a progress thread is constantly $3\mu s$, due to the fully asynchronous processing, and thus at the bottom of Fig. V.14(b)).

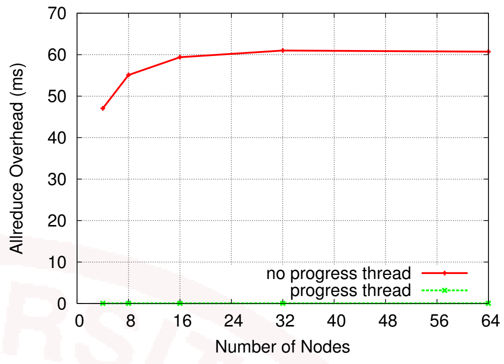
2.1.5 Benchmark Results

In our benchmarks, we study the reconstruction of data collected by the *quadHIDAC* small-animal PET scanner [181]. We used 10^7 events divided into 10 subsets and performed one iteration over all events. The reconstruction image has the size $N = (150 \times 150 \times 280)$ voxels. We ran a set of different benchmarks on the *Odin* system. We compared the non-threaded and threaded versions of LibNBC using the InfiniBand™ optimized transport. We progressed the non-threaded version

[1]NBC_Iallreduce immediately followed by NBC_Wait



(a) Comparison of the Allreduce Performance of LibNBC and Open MPI



(b) Comparison of the Allreduce Overheads of different LibNBC Options

Figure V.14: Allreduce Performance Results for a 48MiB Summation of Doubles

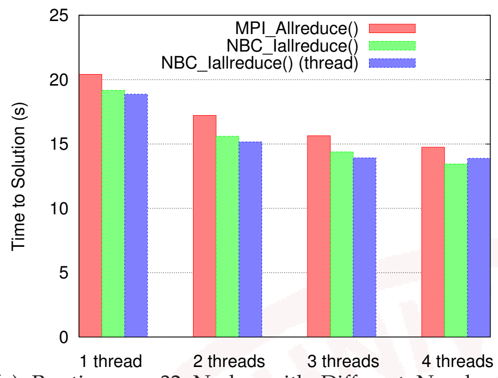
with $4 \times P$ calls to NBC_Test that are equally distributed over the overlapped time. The threaded version of LibNBC is implemented by using InfiniBand's blocking semantics and the application did not call NBC_Test at all. We benchmarked all configurations of LibNBC and the original MPI implementation on 8, 16 and 32 nodes with 1, 2, 3 or 4 OpenMP threads per node three times and report the average times across all runs and processes (the variance between the runs was low).

Computational Overlap The computational overlap time per subset $t^P_{CompOver}$ decreases—as expected from our model—linearly with increasing number of processes P. The average time was $833.5\,ms$ on 8, $469.9\,ms$ on 16 and $241.8\,ms$ on 32 nodes. With reading time t^P_{read} ranging from $55.4\,ms$ on 8 to $11.2\,ms$ on 32 nodes and computation time $t^P_{A_l}$ ranging from $778.1\,ms$ to $230.6\,ms$ on 8 and 32 nodes, respectively, we are able to verify our model (V.6) with an error of about 6%.

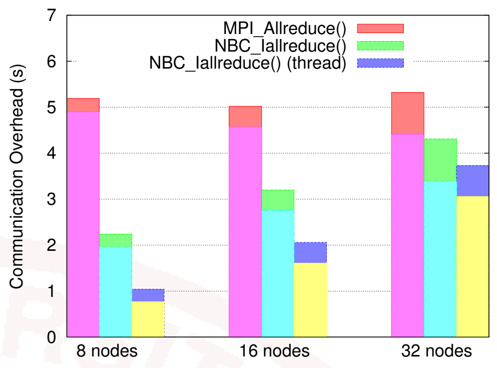
Fig. V.15(a) shows the application running time on 32 nodes with different numbers of OpenMP threads per node. We see that the non-threaded version of LibNBC is able to improve the running time in every configuration. However, the threaded version is only able to improve the performance if it has a spare core available because of scheduler congestion on the fully loaded system. The performance gain also decreases with the number of OpenMP threads. This is because we studied a strong scaling problem and the overlappable computation time gets shorter with more threads computing the static workload and is eventually not enough to overlap the full communication. Another issue for smaller node-counts is that our transformed implementation is, as described in Section 2.1.4, slightly less cache-friendly which limits the application speedup at smaller scale.

Fig. V.15(b) shows the communication overhead for different node counts with one thread per node[2]. Our implementation achieves significantly smaller communication overhead for all configurations. However, the workload per node that can be overlapped decreases, as described above, with the number of nodes, while the communication time of the $48\,MiB$ Allreduce remains nearly constant. Thus, the time to overlap shrinks with the number of nodes and limits the performance gain of the non-blocking collectives.

[2]the lower part of the bars denotes the Allreduce overhead



(a) Runtime on 32 Nodes with Different Number of OpenMP Threads



(b) Communication Overhead for Different Node Counts and a single Thread

Figure V.15: Application Benchmark for different number of OpenMP threads and nodes

2.2 Octopus - A Quantum Mechanical Software Package

Ab-initio quantum mechanical simulations play an important role in nano and material sciences as well as many other scientific areas, e. g., the understanding of biological or chemical processes. Solving the underlying Schrödinger equation for systems of hundreds or thousands of atoms requires a tremendous computational effort that can only be mastered by highly parallel systems and algorithms.

Density functional theory (DFT) [123, 175] is a computationally feasible method to calculate properties of quantum mechanical systems like molecules, clusters, or solids. The basic equations of DFT are the static and time-dependent Kohn-Sham equations:[3]

$$\boldsymbol{H}\boldsymbol{\varphi}_j = \varepsilon_j \boldsymbol{\varphi}_j \qquad\qquad i\frac{\partial}{\partial t}\boldsymbol{\varphi}_j(t) = \boldsymbol{H}(t)\boldsymbol{\varphi}_j(t) \tag{V.7}$$

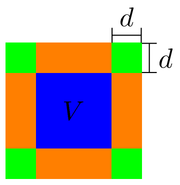
The electronic system is described by the Hamiltonian operator

$$\boldsymbol{H} = -\frac{1}{2}\nabla^2 + \boldsymbol{V}, \tag{V.8}$$

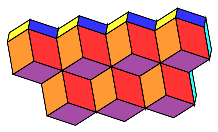
where the derivative accounts for kinetic energy and $\boldsymbol{V}$ for the atomic potentials and electron-electron interaction. The vectors $\boldsymbol{\varphi}_j$, $j = 1, \ldots, N$, are the Kohn-Sham orbitals each describing one of N electrons.

The scientific application `octopus` [51] solves the eigenvalue problem of Eq. (V.7, left) by iterative diagonalization for the lowest N eigenpairs $(\varepsilon_j, \boldsymbol{\varphi}_j)$ and Eq. (V.7, right) by explicitly evolving the Kohn-Sham orbitals $\boldsymbol{\varphi}_j(t)$ in time. The essential ingredient of iterative eigensolvers as well as of most real-time propagators [52] is the multiplication of the Hamiltonian with an orbital $\boldsymbol{H}\boldsymbol{\varphi}_j$. Since `octopus` relies on finite-difference grids to represent the orbitals, this operation can be parallelized by dividing the real-space mesh and assigning a certain partition (domain) to each node as shown in Fig. III.1(a).

[3] i denotes the imaginary unit $i = \sqrt{-1}$ and t is the time parameter.



(a) An area V enlarged by d.



(b) Rhombic dodecahedrons, consisting of twelve rhombic faces, stacked to fill up space.

Figure V.16: Message sizes and number of neighbors.

The potential $\boldsymbol{V}$ is a diagonal matrix, so the product $\boldsymbol{V}\boldsymbol{\varphi}_j$ can be calculated locally on each node. The Laplacian operator of (V.8) is implemented by a finite-difference stencil as shown in Fig. III.1(b). This technique requires to send values close to the boundary (gray shading in Fig. III.1(b)) from one partition (orange) to a neighboring one (green).
The original implementation of $\boldsymbol{H}\boldsymbol{\varphi}_j$ is:

1. Exchange boundary values between partitions.
2. $\boldsymbol{\varphi}_j \leftarrow -\frac{1}{2}\nabla^2\boldsymbol{\varphi}_j$ (apply kinetic energy operator).
3. $\boldsymbol{\varphi}_j \leftarrow \boldsymbol{\varphi}_j + \boldsymbol{V}\boldsymbol{\varphi}_j$ (apply potential).

In this article, we describe a simplified and efficient way to implement and optimize the neighbor exchange with non-blocking collective operations that are defined on topology communicators.

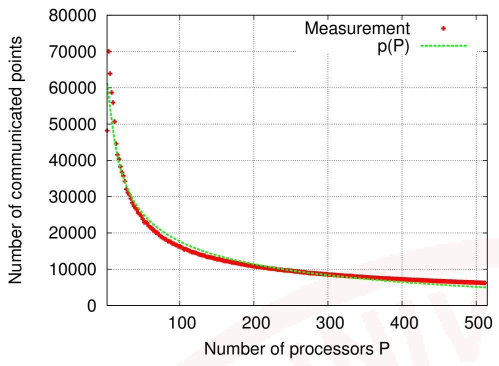
2.2.1 Parallel Implementation

This section gives a detailed analysis of the communication and computation behavior of the domain parallelization and presents alternative implementations using non-blocking and topology-aware collectives that provide higher performance and better programmability.

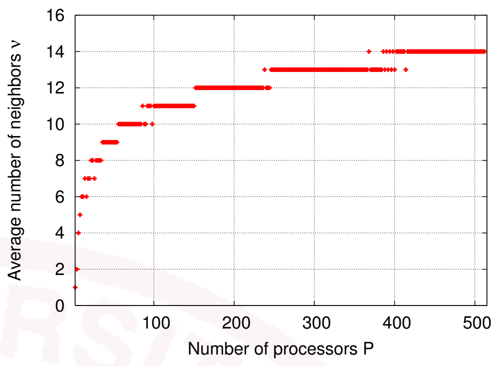
2.2.1.1 Domain parallelization

The application of the Hamiltonian to an orbital $\boldsymbol{H}\boldsymbol{\varphi}_j$ can be parallelized by a partitioning of the real-space grid. The best decomposition depends on the distribution of grid-points in real-space which depends on the atomic geometry of the system under study. We use the library METIS [117] to obtain partitions that are well balanced in the number of points per node.

The communication overhead of calculating $\boldsymbol{H}\boldsymbol{\varphi}_j$ is dominated by the neighbor exchange operation on the grid. To determine a model to assess the scaling of the communication time which can be used to predict the application's running time and scalability, we need to assess the message-sizes, and the average number of neighbors of every processor. Both parameters are influenced by the discretization order d that affects how far the stencil leaks into neighboring domains, and by the number of points in each partition. Assuming a nearly optimal domain decomposition, NP points in total, and P processors we can consider the ratio $V = NP/P$ as "volume" per node. The number of communicated points is $p(P) = V_d - V$ with V_d being the volume V increased by the



(a) Exchanged points as a function of the number of processors for a large grid.



(b) Average and maximum number of neighbors ν per partition.

Figure V.17: Communicated points and neighbor-count for different numbers of processors.

discretization order d and reads

$$p(P) = \alpha d^3 + \beta d^2 \sqrt{V(P)} + \gamma d \sqrt[3]{V(P)^2} \tag{V.9}$$

with coefficients α, β, γ depending on the actual shape of the partitions. The derivation of (V.9) is sketched schematically in Fig. V.16(a) for a 2D situation: an area V is increased by d in each direction. The enlargement is proportional to d^2 (green) and $d\sqrt{V}$ (red). In 3D, the additional dimension causes these terms to be multiplied by d and leads to one more term proportional to $d\sqrt[3]{V^2}$. Fig. V.17(a) shows the number of exchanged points measured for a cylindrical grid of 1.2 million points and the analytical expression (V.9) fitted to the data-points.

Since the average number of neighbors (ν) depends on the structure of the input system, we cannot derive a generic formula for this quantity but instead give the following estimate: METIS minimizes edge-cut which is equivalent to minimization of surfaces. This can be seen in Fig. III.1(a) on 64 where the partition borders are almost between the gray Carbon atoms, the optimum in this case. In general, the minimal surface a volume can take on is spherical. Assuming the partitions to be stacked rhombic dodecahedrons as approximation to spheres, shown in Fig. V.16(b), we conclude that, for larger P, ν is clearly below P because each dodecahedron has at maximum twelve neighbors. This consideration, of course, assumes truly minimum surfaces that METIS can only approximate. In practice, we observe an increasing number of neighbors for larger P, see Fig. V.17(b). Nevertheless, the number of neighbors is an order of magnitude lower than the number of processors.

Applying the well-known LogGP model [29] to our estimations of the scaling of the message sizes and the number of neighbors ν, we can derive the following model of the communication overhead (each point is represented by an 8 byte double value):

$$t_{comm} = L + o\nu + g(\nu - 1) + G(\nu \cdot 8\, p(P)) \tag{V.10}$$

We assume a constant number of neighbors ν at large scale. Thus, the communication overhead scales with $O\left(\sqrt{NP/P}\right)$ in P. The computational cost of steps 2 and 3 that determines the potential to overlap computation and communication scales with NP/P for the potential term and $\alpha d^3 + \beta d^2\sqrt{NP/P} + \gamma d\sqrt[3]{(NP/P)^2} + \delta NP/P$ for the kinetic term.[4] We observe that our computation has a similar scaling behavior as the communication overhead, cf. Eq. (V.10). We therefore conclude that overlapping the neighbor exchange communication with steps 2 and 3 should show a reasonable performance benefit at any scale.

Overlapping this kind of communication has been successfully demonstrated on a regular grid in [2]. We expect the irregular grid to achieve similar performance improvements which could result in a reduction of the communication overhead.

Practical benchmarks show that there are two calls that dominate the communication overhead of `octopus`. On 16 processors, about 13% of the application time is spent in many 1 real or complex value MPI_Allreduce calls caused by dot-products and the calculation of the atomic potentials. This communication can not be optimized or overlapped easily and is thus out of the scope of this work. The second biggest source of communication overhead is the neighbor communication which causes about 8.2% of the total runtime. Our work aims at efficiently implementing the neighbor exchange and reducing its communication overhead with new non-blocking collective operations that act on a process topology.

2.2.1.2 Implementation with NBC_Ialltoallv

The original implementation used MPI_Alltoallv for the neighbor exchange. The transition to the use of non-blocking collective operations is a simple replacing of MPI_Alltoall with NBC_Ialltoallv and the addition of a handle. Furthermore, the operation has to be finished with a call to NBC_Wait before the communicated data is accessed.

However, to achieve the best performance improvement, several additional steps have to be performed. The first step is to maximize the time to overlap, i. e., to move the NBC_Wait as far behind the respective NBC_Ialltoallv as possible in order to give the communication more time to proceed in the background. Thus, to overlap communication and computation we change the original algorithm to:

1. Initiate neighbor exchange (NBC_Ialltoallv).
2. $\varphi_j \leftarrow \boldsymbol{v}\varphi_j$ (apply potential).
3. $\varphi_j \leftarrow \varphi_j - \frac{1}{2}\nabla^2\varphi_j^{inner}$ (apply kinetic energy operator to inner points).
4. Wait for the neighbor exchange to finish (NBC_Wait).
5. $\varphi_j \leftarrow \varphi_j - \frac{1}{2}\nabla^2\varphi_j^{edge}$ (apply kinetic energy operator to edge points).

We initiate the exchange of neighboring points (step 1) and overlap it with the calculation of the potential term (step 2) and the inner part of the kinetic energy, which is the derivative of all points that can be calculated solely by local points (step 3). The last step is waiting for the neighbor-exchange to finish (step 4) and calculation of the derivatives for the edge points (step 5).

[4]The derivation of this expression is similar to (V.9) except that we shrink the volume by the discretization order d.

A usual second step to optimize for overlap is to introduce NBC_Test() calls that give LibNBC the chance to progress outstanding requests. This is not necessary if the threaded version of LibNBC is running on the system. We have shown Chapter IV that the a naively threaded version performs worse, due to the loss of a computational core. However, for this work, we use the InfiniBand optimized version of LibNBC which does not need explicit progression with NBC_Test() if there is only a single communication round (which is true for all non-blocking operations used in `octopus`).

As shown in Section 2.2.1.1, the maximum number of neighbors is limited. Thus, the resulting communication pattern for large-scale runs is sparse. The MPI_Alltoallv function, however, is not suitable for large-scale sparse communication patterns because it is not scalable due to the four index arrays which have to be filled for every process in the communicator regardless of the communication pattern. This results in arrays mostly filled with zeros that still have to be generated, stored and processed in the MPI call and is thus a performance bottleneck at large scale. Filling those arrays correctly is also complicated for the programmer and a source of common programming errors. To tackle the scalability and implementation problems, we use the new collective operations, proposed in Chapter III, that are defined on the well known MPI process topologies [6]. The following section describes the application of one of the proposed collective operations to the problem described above.

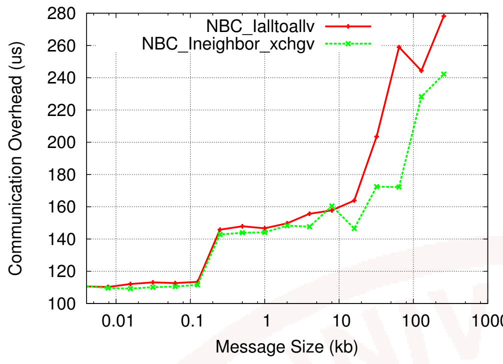
2.2.1.3 Implementation with Topological Collective Operations

We use a graph communicator and the newly proposed collectives introduced in Chapter III to represent the neighborship of partitions generated by METIS for the particular input system. MPI_Graph_create is used to create the graph communicator. Due to the potentially irregular grid (depending on the input system), the number of points communicated with each neighbor might vary. Thus, we used the vector variant NBC_Ineighbor_xchgv to implement the neighbor exchange for `octopus`.

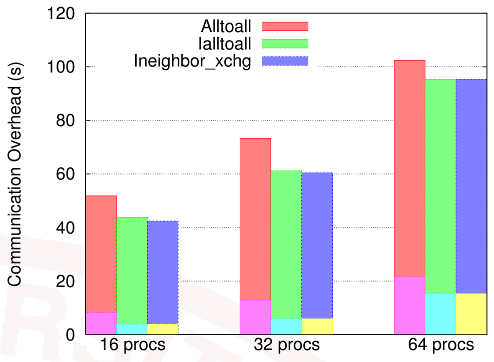
2.2.2 Performance Analysis

We benchmarked our implementation on the CHiC supercomputer system (a cluster computer consisting of nodes equipped with dual socket dual-core AMD 2218 2.6 GHz CPUs, connected with SDR InfiniBand and 4 GB memory per node). We use the InfiniBand-optimized version of LibNBC [9] to achieve highest performance and overlap. Each configuration was ran three times on all four cores per node (4-16 nodes were used) and the average values are reported.

Fig. V.18(a) shows the microbenchmark results for the overhead of NBC_Ialltoallv and NBC_Ineigbor_xchgv of *NBCBench* [11] with 10 neighbors under the assumption that the whole communication time can be overlapped. The overhead of the new neighbor exchange operation is slightly lower than the NBC_Ialltoallv overhead because the implementation does not evaluate arrays of size P. Fig. V.18(b) shows the communication overhead of a fixed-size ground state calculation of a chain of Lithium and Hydrogen atoms. The overhead varies (depending on the technique used) between 22% and 25% on 16 processes. The bars in Fig. V.18(b) show the total communication overhead and the tackled neighbor exchange overhead (lower part). We analyze only the overhead-reduction and easier implementation of the neighbor exchange in this work.



(a) Overhead of LibNBC on 64 Processes on 16 Nodes.



(b) `Octopus` Communication Overhead on 4-16 Nodes.

Figure V.18: LibNBC and `octopus` communication overhead.

The application of non-blocking neighbor collective operations efficiently halves the neighbor exchange overhead and thus improves the performance of `octopus` by about 2%. The improvement is smaller on 64 processes because the time to overlap is due to the strong scaling problem much smaller than in the 32 or 16 process cases. The gain of using the nearest neighbor exchange collective is marginal at this small scale. Memory restrictions prevented bigger strong-scaling runs.

3 Conclusions

Although non-blocking collective communication operations offer significant potential for improving application performance, they need to be used appropriately within applications to actually provide performance improvements. In this chapter, we presented different programming patterns to enable efficient overlapping of collective communication for a wide class of scientific applications. Applying non-blocking collective communication with our approach allowed applications to scale far beyond the parallelism of simple non-blocking point-to-point communication schemes.

The implementation of our transformation scheme as a generic library function template allows these techniques to be used without requiring the programmer to explicitly encode them. Compression and FFT benchmarks using our template demonstrate a significant decrease of communication overhead for both applications. The communication overhead was reduced by more than 92% for the parallel compression and 90% for the parallel 3D-FFT which led to an application speedup of 21% and 16% respectively.

We also demonstrated the easy use of the LibNBC and the application principle of non-blocking collectives to a class of application kernels that exhibits explicit data parallelism. We were able to improve the parallel application kernel running time by up to 34% with minor changes to the application.

The programming scheme Map-Reduce was also analyzed. We noted that Map-Reduce does not ideally fit the MPI model. However, we analyzed the large subset of Map-Reduce applications that match the MPI model. The use of collective operations for the Map and Reduce tasks seems natural for this class of applications. We demonstrated reasonable performance gains for dynamic and static workloads by using non-blocking collective operations and the simple window/pipelining techniques that we described before.

In an application study, we applied non-blocking collective operations to the mixed OpenMP/MPI parallel implementation of the list-mode OSEM algorithm and analyzed the performance gain for a fixed problem size (strong scaling) on different setups of MPI processes and OpenMP threads and a real-world input. The conducted study demonstrates that the overlap optimization potential of non-blocking collectives depends heavily on the time to overlap (amount of work to do while communicating) which usually decreases while scaling to larger process counts. However, even in the worst case (smallest workload) of our example, running 128 threads on 32 nodes, LibNBC was able to reduce the communication overhead from 40.31% to 37.3%. In the best case, with one thread on 8 nodes (highest workload per process), the communication overhead could be efficiently halved from 12.0% to 5.6%.

In a second application study, we analyzed the ease of use and performance potential of our newly proposed topological collectives. We showed the application of the new operations to the quantum mechanical simulation program `octopus`. The communication overhead of the neighbor exchange operation was efficiently halved by overlapping of communication and computation improved the application performance.

In general, we have been able to prove our assumptions with real-world codes and workloads. However, we have to say that the correct application and performance tuning of non-blocking collective is non-trivial and requires a lot of expert knowledge and fine tuning. Manual approaches to do this are cumbersome and we showed a semi-automatic approach. Simplifying the use and tuning of this new class of operations is subject of ongoing research.

Chapter VI

Summary and Conclusions

"I hope we'll be able to solve these problems before we leave. And when I say 'before we leave', I mean 'before we die.' " – Paul Erdös, (1913-1996) Hungarian Mathematician

In this work, we analyze today's parallel programming models and principles for their applicability to large-scale systems and algorithms. We focused on the Message Passing Interface (MPI) which is the most used model on today's large-scale systems. We strongly believe that only gradual change can be successful in practice. Thus, we focus on extending MPI to improve its performance at large-scale.

In order to model the reality as accurately as possible, we analyze a particular network that is often used in High Performance Computing, InfiniBand. We implemented the extensive open-source benchmark Netgauge to measure, compare and model all of InfiniBand's different data transmission modes. Based on this, we were able to verify the LogGP model for large messages and extend it for more accurate modelling of small messages in the InfiniBand architecture.

We show that collective optimization of communication patterns is the key to portable performance at large scale. MPI offers the collective operation mechanism to enable the user to express dense communication patterns abstractly. The new model enabled a step back into theory such that we could derive two new principles to optimize collective communication in MPI.

Based on the previous experiments, we reasoned that the performance metric of large-scale networks, their bisection bandwidth, is not accurate to predict performance at scale. We analyzed the largest unclassified existing InfiniBand installations and introduced a new measure, *effective bisection bandwidth*, which seems more accurate from an application perspective. We also extrapolate those results to larger future systems with up to 10,000 endpoints. An extensive analysis of real-world application patterns shows that the new measure also provides a better assessment for different communication patterns such as tree, dissemination and nearest neighbor. Based on this, we point out that optimized collective implementations could take advantage of the additional information such as routing. We propose an optimization scheme based on a remapping strategy. We were able to approximate good solutions with a genetic algorithm at small scale, however, large-scale simulations showed that this approach is not viable due to the large (factorial) search space.

Based on those experiments and our belief that large-scale communication patterns have to be optimized abstractly, we proposed two extensions to the existing MPI collective operations. The first extension are topological collective operations that enable sparse communication patterns to be modeled at a high level (e.g., with graph topologies) which is a huge optimization potential. This enables the expression of common sparse communication schemes, such as nearest neighbor, in terms of collective operations. With this technique, the sparse operations can be optimized by the library, and the programmer's task is simplified while the scalability of simple point-to-point is maintained. The second extension is a non-blocking variant of all MPI collective operations. The obvious benefit here is that they offer an interface that can be used to overlap communication and computation to hide the latency and use the network bandwidth more efficiently. Another not so obvious benefit is more important at large-scale: the mitigation of the detrimental effects of process skew and load imbalance by moving the data-driven pseudo-synchronization of collective operations into the background. Both extensions are under consideration for the third generation MPI standard.

Overlapping of computation and communication is widely discussed and there seems to be no clear conclusion among scientists. We show that the drawbacks lie mainly in the implementation of communication libraries which limit the practical use of overlapping techniques. We model the potential performance gain of non-blocking collective operations with our newly developed models and are able to draw two main conclusions: first, the theoretical performance gain grows with the size of the exchanged data and system size. It enables a reduction of the communication overhead of up to three orders of magnitude for large messages on large systems. At smaller (today's realistic) scale, the performance benefits vary between 80% and 99%. Second, the choice of optimal algorithms to implement non-blocking collective operations is fundamentally different from the implementation of blocking collectives because CPU overhead and progression becomes more important than latency in this context. Thus, we conclude theoretically that a simple threaded implementation using MPI-2 features would not perform optimally and has other disadvantages, for example, programmability.

To demonstrate the disadvantages, we compared an implementation based in threaded blocking MPI collective operations with an implementation based on point-to-point messages and algorithms optimized for overlap. Based on the results, we decided to continue the development of the point-to-point-based implementation due to its higher flexibility and performance. The resulting implementation, LibNBC, supports all colletive operations defined in MPI and all topological collectives proposed in this work with a non-blocking interface. In order to gain higher measurement accuracy, we extended a well-known benchmark scheme for blocking collective operations to enable higher accuracy, higher scalability and the measurement of communication overheads of non-blocking collective operations. The benchmark's source-code, called NBCBench, is publicly available. In order to show principles that can be used to implement a low-overhead version of non-blocking collectives, we optimized our implementation for the InfiniBand network and were able to demonstrate near-optimal performance. Our fully-threaded version can be used to offload the whole communication and the communication-related computation to a separate processing element. We also discuss operating system issues that become significant when the progression thread is run on a loaded CPU. We also discuss possible solutions to all discovered problems.

The biggest remaining problem is the use of the new techniques in real applications. We showed several basic principles to optimize parallel algorithms in the extended model. We implemented those ideas as generic C++ templates which can be used to rearrange computation and communication in order to achieve highest performance. We show the the optimization of three application kernels: 3-dimensional Poisson Solver, a 3-dimensional Fast Fourier Transform and a Parallel Compression. The communication overhead of all kernels could be decreased by up to 92%. We also analyzed the applicability to the parallel programming scheme Map-Reduce and showed that performance of static and dynamic workloads can be significantly enhanced.

Two real-world applications, a medical image reconstruction algorithm and a quantum-mechanical computation were also analyzed. The communication overhead of the optimized operations could be halved in the best case. However, we also saw that non-blocking collective operations can not be applied to all algorithms and might need rather huge changes to the underlying algorithm or mathematical representation. We still show performance improvements for both application examples. We see that optimization is an expert task and requires significant tuning and knowledge about the application and the communication network.

We expect that our results will impact mathematical thinking and algorithm design. Parallel algorithms will need to be designed with high-level communication patterns in mind and enable overlap of communication and computation wherever possible. Thus, the designer needs to think in terms of data- and functionality parallelism. We expect that new algorithms and architectures will be designed with those principles in mind.

All results of this thesis and programs used to gather them are available as source code. Programs were either implemented in Open MPI or in one of the software packages developed as tools for the research. Those tools, namely LibNBC, Netgauge, LibAllprof and NBCBench, comprise about comprise about 41,000 source lines of C, C++ and python code[1]. All tools are available to the public at the personal homepage of the author.

[1]counted with David A. Wheeler's 'SLOCCount'

Appendix

"Our two greatest problems are gravity and paper work. We can lick gravity, but sometimes the paperwork is overwhelming." – Wernher von Braun, (1912-1977) German Physicist, National Medal of Science 1975

A References

A.1 Publications Containing Results of this Thesis

[1] T. Hoefler, P. Gottschling, and A. Lumsdaine. Leveraging Non-blocking Collective Communication in High-performance Applications. In *SPAA'08, Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, pages 113–115. Association for Computing Machinery (ACM), June 2008.

[2] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, September 2007.

[3] T. Hoefler, R. Janisch, and W. Rehm. Parallel Scaling of Teter's Minimization for Ab Initio Calculations. November 2006. Presented at the workshop HPC Nano in conjunction with the 2006 International Conference on High Performance Computing, Networking, Storage and Analysis, SC06.

[4] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. A Case for Standard Non-Blocking Collective Operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI 2007*, volume 4757, pages 125–134. Springer, October 2007.

[5] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, March 2007.

[6] T. Hoefler, F. Lorenzen, D. Gregor, and A. Lumsdaine. Topological Collectives for MPI-2. Technical report, Open Systems Lab, Indiana University, February 2008.

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

[8] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, October 2008.

[9] T. Hoefler and A. Lumsdaine. Optimizing non-blocking Collective Operations for InfiniBand. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2008.

[10] T. Hoefler and A. Lumsdaine. Overlapping Communication and Computation with High Level Communication Routines. In *Proceedings of the 8th IEEE Symposium on Cluster Computing and the Grid (CCGrid 2008)*, May 2008.

[11] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, November 2007.

[12] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A Network Performance Measurement Framework. In *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782, pages 659–671. Springer, September 2007.

[13] T. Hoefler and W. Rehm. A Communication Model for Small Messages with InfiniBand. In *Proceedings PARS Workshop 2005 (PARS Mitteilungen)*, pages 32–41. PARS, June 2005. (Awarded with the PARS Junior Researcher Prize).

[14] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication Optimization for Medical Image Reconstruction Algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, September 2008.

[15] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately Measuring Collective Operations at Massive Scale. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2008.

[16] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, October 2008.

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

[18] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, and W. Rehm. Non-Blocking Collective Operations for MPI-2. Technical report, Open Systems Lab, Indiana University, August 2006.

[19] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine. A Case for Non-Blocking Collective Operations. In *Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops*, volume 4331/2006, pages 155–164. Springer Berlin / Heidelberg, December 2006.

[20] T. Hoefler, C. Viertel, T. Mehlan, F. Mietke, and W. Rehm. Assessing Single-Message and Multi-Node Communication Performance of InfiniBand. In *Proceedings of IEEE International Conference on Parallel Computing in Electrical Engineering, PARELEC 2006*, pages 227–232. IEEE Computer Society, September 2006.

[21] Torsten Hoefler, Lavinio Cerquetti, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPP'05)*, pages 562–569, June 2005.

[22] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte - CSR-04-03*, 2004.

[23] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. Fast Barrier Synchronization for InfiniBand. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.

[24] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. LogfP - A Model for small Messages in InfiniBand. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (PMEO-PDS 06)*, April 2006.

A.2 Computer Science and Technical References

[25] Tarek S. Abdelrahman and Gary Liu. Overlap of computation and communication on shared-memory networks-of-workstations. *Cluster computing*, pages 35–45, 2001.

[26] A. Adelmann, W. P. Petersen A. Bonelli and, and C. W. Ueberhuber. Communication Efficiency of Parallel 3D FFTs. In *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, volume 3402 of *Lecture Notes in Computer Science*, pages 901–907. Springer, 2004.

[27] Saurabh Agarwal, Rahul Garg, and Nisheeth Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *12th Annual IEEE International Conference on High Performance Computing*, Goa, India, December 2005.

[28] Sadaf R. Alam, Nikhil Bhatia, and Jeffrey S. Vetter. An exploration of performance attributes for symbolic modeling of emerging processing devices. In Ronald H. Perrott, Barbara M. Chapman, Jaspal Subhlok, Rodrigo Fernandes de Mello, and Laurence Tianruo Yang, editors, *HPCC*, volume 4782 of *Lecture Notes in Computer Science*, pages 683–694. Springer, 2007.

[29] 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.

[30] George Almasi, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, Jose E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[31] Robert Alverson. Red Storm. In *Invited Talk, Hot Chips 15*, 2003.

[32] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Readings in computer architecture*, pages 79–81, 2000.

[33] Francoise Baude, Denis Caromel, Nathalie Furmento, and David Sagnol. Optimizing metacomputing with communication-computation overlap. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 190–204, London, UK, 2001. Springer-Verlag.

[34] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An Evaluation of Current High-Performance Networks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 28.1, Washington, DC, USA, 2003. IEEE Computer Society.

[35] V. E. Benes. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[36] M. Bernaschi and G. Iannello. Quasi-Optimal Collective Communication Algorithms in the LogP Model. Technical report, University of Naples (DIS), 03 1995.

[37] 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.

[38] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

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

[40] Thomas Bönisch, Michael M. Resch, and Holger Berger. Benchmark evaluation of the message-passing overhead on modern parallel architectures. In *Parallel Computing: Fundamentals, Applications and New Directions , Proceedings of the Conference ParCo97*, pages 411–418, 1997.

[41] Rajendra V. Boppana and Suresh Chalasani. A comparison of adaptive wormhole routing algorithms. *SIGARCH Comput. Archit. News*, 21(2):351–360, 1993.

[42] Ron Brightwell, Sue Goudy, Arun Rodrigues, and Keith Underwood. Implications of application usage characteristics for collective communication offload. *Internation Journal of High-Performance Computing and Networking*, 4(2), 2006.

[43] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, 1999.

[44] Ron Brightwell, Rolf Riesen, and Keith D. Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl.*, 19(2):103–117, 2005.

[45] Ron Brightwell and Keith D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305, New York, NY, USA, 2004. ACM Press.

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

[47] Sally Anne Browning. *The tree machine: a highly concurrent computing environment*. PhD thesis, Pasadena, CA, USA, 1980.

[48] Darius Buntinas, Dhabaleswar K. Panda, and Ron Brightwell. Application-bypass broadcast in mpich over gm. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 2, Washington, DC, USA, 2003. IEEE Computer Society.

[49] Pierre-Yves Calland, Jack Dongarra, and Yves Robert. Tiling on systems with communication/computation overlap. *Concurrency - Practice and Experience*, 11(3):139–153, 1999.

[50] C. Calvin and F. Desprez. Minimizing communication overhead using pipelining for multidimensional fft on distributed memory machines, 1993.

[51] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, and E. K. U. Groß andA. Rubio. phys. stat. sol (b). *243(11):2465–2488*, 2006.

[52] A. Castro, M. A. L. Marques, and A. Rubio. Propagators for the time-dependent kohn-sham equations. *The Journal of Chemical Physics*, 121(8):3425–3433, 2004.

[53] B. Chandrasekaran, Pete Wyckoff, and Dhabaleswar K. Panda. Miba: A micro-benchmark suite for evaluating infiniband architecture implementations. *Computer Performance Evaluation / TOOLS*, pages 29–46, 2003.

[54] Chapel Consortium. *Chapel Language Specification 0.775*. Cray Inc., 2008.

[55] Hsiang Ann Chen, Yvette O. Carrasco, and Amy W. Apon. MPI Collective Operations over IP Multicast. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 51–60, London, UK. Springer-Verlag.

[56] F. R. K. Chung, Jr. E.G. Coffman, M.I. Reiman, and B. Simon. The forwarding index of communication networks. *IEEE Trans. Inf. Theor.*, 33(2):224–232, 1987.

[57] Mark J. Clement and Michael J. Quinn. Overlapping computations, communications and i/o in parallel sorting. *J. Parallel Distrib. Comput.*, 28(2):162–172, 1995.

[58] C. Clos. A study of non-blocking switching networks. *Bell System Technology Journal*, 32:406–424, 1953.

[59] David Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[60] David Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast Parallel Sorting under LogP: from Theory to Practice. In *Proceedings of the Workshop on Portability and Performance for Parallel Processing*, Southampton, England, July 1993. Wiley.

[61] 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.

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

[63] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 58, Washington, DC, USA, 2005. IEEE Computer Society.

[64] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[65] Lloyd Dickman. An introduction to the pathscale infinipath htx adapter. *Pathscale White Papers*, November 2005.

[66] R. Dimitrov. *Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations*. PhD thesis, Mississippi State University, 2001.

[67] Zhu Ding, Raymond R. Hoare, Alex K. Jones, and Rami Melhem. Level-wise scheduling algorithm for fat tree interconnection networks. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 96, Tampa, Florida, 2006. ACM.

[68] Anshu Dubey and Daniele Tessera. Redistribution strategies for portable parallel FFT: a case study. *Concurrency and Computation: Practice and Experience*, 13(3):209–220, 2001.

[69] Eric Allen and David Chase and Joe Hallett and Victor Luchangco and Jan-Willem Maessen and Sukyoung Ryu and Guy L. Steele Jr. and Sam Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0*. Sun Microsystems, Inc, 2008.

[70] 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.

[71] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of MPI collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM Press.

[72] 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.

[73] 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.

[74] Andrew Friedley, Torsten Hoefler, Matthew L. Leininger, and Andrew Lumsdaine. Scalable High Performance Message Passing over InfiniBand for Open MPI. In *Proceedings of 2007 KiCC Workshop, RWTH Aachen*, December 2007.

[75] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[76] 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.

[77] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, M. E. Giampapa P. Coteus, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2):195–213, 2005.

[78] Mario Gerla, Prasasth Palnati, and Simon Walton. Multicasting protocols for high-speed, wormhole-routing local area networks. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 184–193, New York, 1996. ACM Press.

[79] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[80] 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.

[81] S. Goedecker, M. Boulet, and T. Deutsch. An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. *Computer Physics Communications*, 154:105–110, August 2003.

[82] 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.

[83] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[84] Peter Gottschling and Wolfgang E. Nagel. An efficient parallel linear solver with a cascadic conjugate gradient method. In *EuroPar 2000*, number 1900 in LNCS, 2000.

[85] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22(6):789–828, 1996.

[86] William Gropp and Ewing L. Lusk. Reproducible measurements of mpi performance characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, 1999. Springer-Verlag.

[87] William D. Gropp and Rajeev Thakur. Issues in developing a thread-safe mpi implementation. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*, pages 12–21. Springer, 2006.

[88] 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.

[89] Wolfgang Hackbusch. *Iterative solultion of large sparse systems of equations*. Springer, 1994.

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

[91] 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.

[92] S. Hanna, B. Patel, and M. Shah. Multicast Address Dynamic Client Allocation Protocol (MADCAP). RFC 2730 (Proposed Standard), December 1999.

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

[94] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1st edition, 1990.

[95] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[96] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Natl. Bur. Stand.*, 49:409–436, 1952.

[97] M. C. Heydemann, J. C. Meyer, J. Opatrny, and D. Sotteau. Forwarding indices of k-connected graphs. *Discrete Appl. Math.*, 37-38:287–296, 1992.

[98] M. C. Heydemann, J.c. Meyer, and D. Sotteau. On forwarding indices of networks. *Discrete Appl. Math.*, 23(2):103–123, 1989.

[99] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0*. Houston, Tex., 1993.

[100] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of compiler optimizations for fortran d on mimd distributed memory machines. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 1–14, New York, NY, USA, 1992. ACM.

[101] R. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[102] Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, New York, NY, USA, 2003. ACM.

[103] Costin Iancu, Parry Husbands, and Paul Hargrove. Hunting the overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[104] G. Iannello. Efficient algorithms for the reduce-scatter operation in loggp. *IEEE Trans. Parallel Distrib. Syst.*, 8(9):970–982, 1997.

[105] G. Iannello, M. Lauria, and S. Mercolino. Logp performance characterization of fast messages atop myrinet, 1998.

[106] J.B. White III and S.W. Bova. Where's the Overlap? - An Analysis of Popular MPI Implementations, 1999.

[107] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. Loggps: a parallel computational model for synchronization analysis. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 133–142, New York, NY, USA, 2001. ACM Press.

[108] Intel Corporation. Intel Application Notes - Using the RDTSC Instruction for Performance Monitoring. Technical report, Intel, 1997.

[109] Kamil Iskra, Pete Beckman, Kazutomo Yoshii, and Susan Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedings of Cluster Computing, 2006 IEEE International Conference*, 2006.

[110] Curtis L. Janssen, Ida B. Nielsen, Matt L. Leininger, Edward F. Valeev, and Edward T. Seidl. The massively parallel quantum chemistry program (mpqc), version 2.3.0, 2004. Sandia National Laboratories, Livermore, CA, USA.

[111] Jeff Hilland and Paul Culley and Jim Pinkerton and Renato Recio. *RDMA Protocol Verbs Specification (Version 1.0)*. Hewlett-Packard Company and Microsoft Corporation and IBM Corporation, 2003.

[112] Terry Jones, Shawn Dawson, Rob Neely, William G. Tuel Jr., Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing*, page 10, November 2003.

[113] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[114] Arkady Kanevsky, Anthony Skjellum, and Anna Rounbehler. MPI/RT - an emerging standard for high-performance real-time systems. In *HICSS (3)*, pages 157–166, 1998.

[115] 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.

[116] Richard M. Karp, Abhijit Sahay, and Eunice Santos. Optimal broadcast and summation in the logp model. Technical report, Berkeley, CA, USA, 1992.

[117] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[118] Chamath Keppitiyagama and Alan S. Wagner. Asynchronous mpi messaging on myrinet. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 50, Washington, DC, USA, 2001. IEEE Computer Society.

[119] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1176–1183, London, UK, 2000. Springer-Verlag.

[120] 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.

[121] Sushmitha P. Kini, Jiuxing Liu, Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Fast and Scalable Barrier Using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, pages 369–378, 2003.

[122] Gregor Kjellstrom. Evolution as a statistical optimization algorithm. *Evolutionary Theory*, 11:105–117, 1996.

[123] W. Kohn and L.J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev*, 140:A1133, 1965.

[124] T. Kohno, A. Broido, and KC Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

[125] Matthew J. Koop, Sayantan Sur, Qi Gao, and Dhabaleswar K. Panda. High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 180–189, New York, NY, USA, 2007. ACM Press.

[126] Thomas Kösters, F. Wübbeling, and F.Natterer. Scatter correction in PET using the transport equation. In *IEEE Nuclear Science Symposium Conference Record*, pages 3305–3309. IEEE, October 2006.

[127] Ralf Lämmel. Google's mapreduce programming model — revisited. *Sci. Comput. Program.*, 68(3):208–237, 2007.

[128] 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.

[129] William Lawry, Christopher Wilson, Arthur B. Maccabe, and Ron Brightwell. COMB: A Portable Benchmark Suite for Assessing MPI Overlap. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002), 23-26 September 2002, Chicago, IL, USA*, pages 472–475. IEEE Computer Society, 2002.

[130] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. In *In ACM Trans. Math. Soft., 5 (1979), pp. 308-323*, 1979.

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

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

[133] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, 1985.

[134] 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.

[135] David Levine. Users Guide to the PGAPack Parallel Genetic Algorithm Library. Technical report, Argonne National Laboratory, 1996.

[136] X. Lin, Y. Chung, and T. Huang. A multiple lid routing scheme for fat-tree-based infiniband networks. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS04)*, page 11a, Sana Fe, NM, 04 2004.

[137] G. Liu and T.S. Abdelrahman. Computation-communication overlap on network-of-workstation multiprocessors. In *Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1635–1642, July 1998.

[138] 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.

[139] J. Liu, A. Mamidala, and D. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. Technical report, OSU-CISRC-10/03-TR57, 2003.

[140] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance rdma-based mpi implementation over infiniband, 2003.

[141] Jiuxing Liu, Balasubramanian Chandrasekaran, Weikuan Yu, Jiesheng Wu, Darius Buntinas, Sushmitha Kini, Dhabaleswar K. Panda, and Pete Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, January 2004.

[142] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, 2004.

[143] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[144] 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.

[145] Kostas Magoutis, Margo I. Seltzer, and Eran Gabber. The case against user-level networking. In *Proceedings of Workshop on Novel Uses of System-Area Networks (SAN-3)*, Madrid, Spain, 2004.

[146] A. Mamidala, J. Liu, and D. Panda. Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms, 2004.

[147] A. R. Mamidala, H. Jin, and D. K. Panda. Efficient Hardware Multicast Group Management for Multiple MPI Communicators over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, volume 3666 of *Lecture Notes in Computer Science*, pages 388–398. Springer.

[148] J. C. Martínez, J. Flich, A. Robles, P. López, and J. Duato. Supporting fully adaptive routing in infiniband networks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 44.1, Washington, DC, USA, 2003. IEEE Computer Society.

[149] Giulio Iannello Massimo Bernaschi. Collective communication operations: experimental results vs. theory. *Concurrency - Practice and Experience 10*, 5:359–386, 1998.

[150] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. In *ACM Trans. on Modeling and Computer Simulations*, 1998.

[151] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. 1995.

[152] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Euro-Par 2006 Parallel Processing*, pages 124–133. Springer-Verlag Berlin, August 2006.

[153] 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.

[154] Philip J. Mucci, Kevin London, and John Thurman. The MPIBench Report. Technical report, CEWES/ERDC MSRC/PET, 1998.

[155] S.J. Murdoch. Hot or not: revealing hidden services by their clock skew. *Proceedings of the 13th ACM conference on Computer and communications security*, pages 27–36, 2006.

[156] Lionel M. Ni. Should Scalable Parallel Computers Support Efficient Hardware Multicasting? In *International Conference on Parallel Processing, Workshops*, pages 2–7, 1995.

[157] Natawut Nupairoj and Lionel M. Ni. Benchmarking of Multicast Communication Services. Technical Report MSU-CPS-ACS-103, Department of Computer Science, Michigan State University, 1995.

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

[159] Interagency Working Group on Information Technology Research and Development. Grand challenges: Science, engineering, and societal advances requiring networking and information technology research and development. Technical report, National Coordination Office for Networking and Information Technology Research and Development, 1996.

[160] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, version 1.0 edition, 1998.

[161] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, 2008.

[162] Scott Pakin. Reproducible network benchmarks with conceptual. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*, pages 64–71. Springer, 2004.

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

[164] F. Petrini, J. Fernandez, E. Frachtenberg, and S. Coll. Scalable collective communication on the asci q machine. In *Hot Interconnects 12*, 08 2003.

[165] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8, 192 Processors of ASCI Q. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom*, page 55. ACM, 2003.

[166] Fabrizio Petrini and Marco Vanneschi. K-ary n-trees: High performance networks for massively parallel architectures. Technical report, 1995.

[167] G. Pfister, M. Gusat, W. Denzel, D. Craddock, N. Ni, W. Rooney, T. Engbersen, R. Luijten, R. Krishnamurthy, and J. Duato. Solving hot spot contention using infiniband architecture congestion control. In *Proceedings HP-IPC 2005*, Research Triangle Park, NC, 6 2005.

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

[169] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[170] Jelena Pjesivac-Grbovic. *PhD thesis*. PhD thesis, The University of Tennessee, Knoxville, 2007.

[171] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance Analysis of MPI Collective Operations. In *Proceedings*

of the 19th International Parallel and Distributed Processing Symposium, 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 05), Denver, CO, April 2005.

[172] Rolf Rabenseifner. Automatic MPI Counter Profiling. In *Proceedings of 42nd CUG Conference*, 2000.

[173] Rolf Rabenseifner. Hybrid parallel programming on hpc platforms. In *In proceedings of the Fifth European Workshop on OpenMP, EWOMP'03*, Aachen, Germany, 2003.

[174] A. J. Reader, K. Erlandsson, M. A. Flower, and R. J. Ott. Fast accurate iterative reconstruction for low-statistics positron volume imaging. *Phys. Med. Biol.*, 43(4):823–834, 1998.

[175] E. Runge and E. K. U. Gross. Density-functional theory for time-dependent systems. *Phys. Rev. Lett.*, 52(12):997, 1984.

[176] Rachid Saad. Complexity of the forwarding index problem. *SIAM J. Discret. Math.*, 6(3):418–427, 1993.

[177] Y. Saad and M.H. Schultz. GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems. 7(3):856–869, July 1986.

[178] Subhash Saini, Robert Ciotti, Brian T. N. Gunney, Thomas E. Spelce, Alice E. Koniges, Dob Dossa, Panagiotis A. Adamidis, Rolf Rabenseifner, Sunil R. Tiyyagura, Matthias Müller, and Rod Fatoohi. Performance evaluation of supercomputers using hpcc and imb benchmarks. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[179] J.C. Sancho, J. Flich, A. Robles, P. Lopez, and J. Duato. Performance evaluation of up*/down* routing using virtual channels for infiniband networks. In *Actas de las XII Jornadas de Paralelismo*, Valencia, España, 2001.

[180] Jose Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. MPI tools and performance studies—Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 125, New York, NY, USA, 2006. ACM Press.

[181] K. P. Schäfers, A. J. Reader, M. Kriens, C. Knoess, O. Schober, and M. Schäfers. Performance evaluation of the 32-module quadHIDAC small-animal PET scanner. *Journal Nucl. Med.*, 46(6):996–1004, 2005.

[182] M. Schellmann and S. Gorlatch. Comparison of two decomposition strategies for parallelizing the 3d list-mode OSEM algorithm. In *Proceedings Fully 3D Meeting and HPIR Workshop*, pages 37–40, 2007.

[183] Michael D. Schroeder, A. Birell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: A high-speed, self-configuring local area network using pointto -point links. *IEEE Journal on Selected Areas in Communications*, 9(8), 10 1991.

[184] 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.

[185] Galen M. Shipman, Tim S. Woodall, Rich L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. Infiniband scalability in open mpi. In *Proceedings of IEEE Parallel and Distributed Processing Symposium*, April 2006.

[186] Galen Mark Shipman, Tim S. Woodall, George Bosilca, Ri ch L. Graham, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[187] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 57–57, New York, NY, USA, 2001. ACM Press.

[188] M. Shro and R. Geijn. Collmark mpi collective communication benchmark, 2001.

[189] Christian Siebert. Efficient Broadcast for Multicast-Capable Interconnection Networks. Master's thesis, Chemnitz University of Technology, 2006.

[190] 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.

[191] M. E. Gonzalez Smith and J. A. Storer. Parallel algorithms for data compression. *J. ACM*, 32(2):344–373, 1985.

[192] Peter Sonnefeld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 10:36–52, 1989.

[193] 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, 2004.

[194] V. S. Sunderam. Pvm: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[195] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *PPoPP '06:*

Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 32–39, New York, NY, USA, 2006. ACM.

[196] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105, New York, NY, USA, 2006. ACM.

[197] Mellanox Technologies. Infiniband - industry standard data center fabric is ready for prime time. *Mellanox White Papers*, December 2005.

[198] Paul Terry, Amar Shan, and Pentti Huttunen. Improving application performance on hpc systems with process synchronization. *Linux J.*, 2004(127):3, 2004.

[199] The InfiniBand Trade Association. *Infiniband Architecture Specification Volume 1, Release 1.2*. InfiniBand Trade Association, 2003.

[200] Vinod Tipparaju and Jarek Nieplocha. Optimizing all-to-all collective communication by exploiting concurrency in modern networks. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2005. IEEE Computer Society.

[201] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. pages 13–21, 1995.

[202] Ulrich Trottenberg, Cornelis Oosterlee, and Anton Schüller. *Multigrid*. Academic Press, 2000.

[203] Dave Turner and Xuehua Chen. Protocol-dependent message-passing performance on linux clusters. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 187, Washington, DC, USA, 2002. IEEE Computer Society.

[204] Dave Turner, Adam Oline, Xuehua Chen, and Troy Benjegerdes. Integrating new capabilities into netpipe. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, volume 2840 of *Lecture Notes in Computer Science*, pages 37–44. Springer, 2003.

[205] 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.

[206] UPC Consortium. *UPC Language Specifications, v1.2* . Lawrence Berkeley National Lab, 2005.

[207] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

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

[209] Henk van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for for the solution of nonsymmetric linear systems. 13:631–644, 1992.

[210] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 156–165, Washington, DC, USA.

[211] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 96, Washington, DC, USA, 2002. IEEE Computer Society.

[212] Vijay Saraswat and Nathaniel Nystrom. *Report on the Experimental Language X10, Version 1.7*. IBM Research, 2008.

[213] A. Vishnu, M. Koop, A. Moody, A. R. Mamidala, S. Narravula, and D. K. Panda. Hot-spot avoidance with multi-pathing over infiniband: An mpi perspective. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 479–486, Washington, DC, USA, 2007. IEEE Computer Society.

[214] Adam Wagner, Darius Buntinas, Dhabaleswar K. Panda, and Ron Brightwell. Application-bypass reduction for large-scale clusters. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 404–411. IEEE Computer Society, December 2003.

[215] Thomas Worsch, Ralf Reussner, and Werner Augustin. On benchmarking collective mpi operations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK, 2002. Springer-Verlag.

[216] 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.

[217] W. Yu, S. Sur, and D. K. Panda. High Performance Broadcast Support in La-Mpi Over Quadrics. *International Journal of High Performance Computing Applications*, 19(4):453–463, 2005.

[218] 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.

[219] Weikuan Yu, Darius Buntinas, and Dhabaleswar K. Panda. High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2, 2003.

[220] X. Yuan, S. Daniels, A. Faraj, and A. Karwande. Group Management Schemes for Implementing MPI Collective Communication over IP-Multicast, 2002.

[221] Eitan Zahavi. Optimized infiniband fat-tree routing for shift all-to-all communication patterns. In *Proceedings of the International Supercomputing Conference 2007 (ISC07)*, Dresden, Germany.

B Glossary

ANSI	*American National Standards Institute*	79
AO	*Atomic Operations*	9
API	*Application Programming Interface*	10
BSP	*Bulk Synchronous Protocol*	15
C90	*ANSI C Standard from 1990*	79
CB	*Crossbar*	53
CPU	*Central Processing Unit* The main processing unit in a computer system	3
EBNF	*Extended Backus Naur Form*	80
GM	*Glenn's Messages*	29
GUID	*Global Unique Identifier*	59
HCA	*Host Channel Adapter*	11
HPC	*High Performance Computing*	1
ILP	*Instruction Level Parallelism*	3
ISA	*Instruction Set Architecture*	87
LibNBC	*Non-blocking Collective Library*	79
LibOF	*Open Fabrics Communication Library*	97
MC	*Multicast*	10
MPICH2	*MPI Chameleon Version 2*	28
MTU	*Maximum Transfer Unit*	9
MVAPICH	*MPICH-based MPI for InfiniBand*	30
NBC	*Non-blocking Collectives*	79
NTP	*Network Time Protocol*	88
OFED	*Open Fabrics Enterprise Edition*	97
OSEM	*Ordered Subset Expectation Maximization*	135
PE	*Processing Element*	132
PET	*Positron Emission Tomography*	135

PMB	*Pallas MPI Benchmarks*	*23*
ppm	*parts per million*	*89*
PRTT	*Parametrized Round Trip Time*	*23*
QP	*Queue Pair*	*9*
RAW	*Raw InfiniBand Transport*	*9*
RC	*Relieable Connection*	*9*
RD	*Relieable Datagram*	*9*
RDMA-R	*Remote Direct Memory Access Read*	*9*
RDMA-W	*Remote Direct Memory Access Write*	*9*
RDTSC	*Read Time Stamp Counter*	*87*
RTR	*Ready to Receive* QP Ready to Receive state or Ready to Receive message in the rendezvous protocol	*98*
RTT	*Round Trip Time*	*11*
SCI	*Scalable Coherent Interface*	*29*
SIMD	*Single Instruction Multiple Data*	*3*
SM	*Subnet Manager*	*53*
SQ	*Send Queue*	*98*
SSE	*Streaming SIMD Extensions*	*3*
TCP	*Transmission Control Protocol*	*28*
TSC	*Time Stamp Counter*	*88*
UC	*Unrelieable Connection*	*9*
UD	*Unrelieable Datagram*	*9*
VAPI	*Verbs API*	*10*
WR	*Work Request*	*98*
x86	*Intel i386 compatible architecture*	*88*

C LibNBC API Definition

This section defines the internal API to add new non-blocking collective algorithms and new collective functionality (e.g., new operations) to LibNBC and the external API for end-users to use the non-blocking operations. All functions return NBC error codes, defined in `nbc.h`.

C.1 Internal API - Building a Schedule

This section describes all the functions that can be used to build a schedule for a collective.

- `NBC_Sched_create(NBC_Schedule* schedule)` allocates a new schedule array.
- `NBC_Free(NBC_Handle *handle)` deallocates a schedule and all related allocated memory.
- `NBC_Sched_send(void* buf, int count, MPI_Datatype datatype, int dest, NBC_Schedule *schedule)` adds a non-blocking send operation to the schedule
- `NBC_Sched_recv(void* buf, int count, MPI_Datatype datatype, int source, NBC_Schedule *schedule)` adds a non-blocking receive operation to the schedule
- `NBC_Sched_op(void* tgt, void* src1, void* src2, int count, MPI_Datatype datatype, MPI_Op op, NBC_Schedule *schedule)` adds a blocking local reduction operation of the non-overlapping buffers `*src1` and `*src2` into `*tgt` to the schedule.
- `NBC_Sched_copy(void *src, int srccount, MPI_Datatype srctype, void *tgt, int tgtcount, MPI_Datatype tgttype, NBC_Schedule *schedule)` adds a blocking local copy operation from `*src` to `*tgt` to the schedule.
- `NBC_Sched_barrier(NBC_Schedule *schedule)` ends the current round in the schedule and adds a new round.
- `NBC_Sched_commit(NBC_Schedule *schedule)` ends the schedule. The commit function could be used to apply further optimization to the schedule, e.g., to add automatic segmentation and pipelining (currently not done).

C.2 Internal API - Executing a Schedule

This section describes all internal functions that are used to execute a schedule.

- `NBC_Start(NBC_Handle *handle, MPI_Comm comm, NBC_Schedule *schedule)` starts the execution of a schedule.
- `NBC_Progress(NBC_Handle *handle)` progresses the instance identified by the handle. Returns NBC_OK if the instance (collective operation) is finished or NBC_CONTINUE if there are still open requests.

C.3 Internal API - Performing Local Operations

This section describes all internal functions that are used to perform local (blocking) operations.

- `NBC_Copy(void *src, int srccount, MPI_Datatype srctype, void *tgt, int tgtcount, MPI_Datatype tgttype, MPI_Comm comm)` copies a message from a source to a destination buffer (from `*src` to `*tgt`).
- `NBC_Operation(void *tgt, void *src1, void *src2, MPI_Op op, MPI_Datatype type, int count)` performs a reduction operation from `*src1` and `*src2` into `*tgt`.

C.4 External API

This section defines the external API for non-blocking collective operations:

- Non-blocking collective communications can be nested on a single communicator. However, the NBC implementation limits the number of outstanding non-blocking collectives to 32767. If a new non-blocking communication gets started, and the NBC library has no free resources, it fails and raises an exception.
- User-defined MPI reduction operations are not supported.
- NBC collective operations do not match MPI collective operations.
- The send buffer must not be changed for an outstanding non-blocking collective operation, and the receive buffer must not be read until the operation is finished (e.g, after NBC_Wait).
- Request test and wait functions (NBC_Test, NBC_Wait, NBC_Testall, NBC_Testany, ...) similar to their MPI conterpart, described in Section 3.7 of the MPI-1.1 [151] standard, are supported for non-blocking collective communications.
- NBC_Request_free is not supported.
- NBC_Cancel is not supported.
- The order of issued non-blocking collective operations defines the matching of them (compare the ordering rules for collective operations in the MPI-1.1 standard and MPI-2 standard in threaded environments).
- Non-blocking collective operations and blocking collective operations can not match each other. Any attempts to match them should fail to prevent user portability errors.
- progress is defined similar as for non-blocking point-to-point in the MPI-2 standard
- operations are not tagged to stay close to the current MPI semantics for collective operations (in threaded environments) and to enable a simple implementation on top of send receive (an implementation could simply use negative tag values to identify collectives internally)
- NBC request objects are used to enable mixing with poit-to-point operations in operations like NBC_Waitany. The authors do not see a problem to add this third class of requests (the two classes right now are point-to-point requests and generalized requests).
- Status objects are not changed by any call finishing a non-blocking collective because all the information is available in the arguments (there are no wildcards in collectives).

Now, we describe some routines in the style of the MPI standard. Not all routines are explained explicitely due to the similarity to the MPI-standardized ones. The new features are summarized in “Other Collective Routines”.

C.4.1 Barrier Synchronization

NBC_IBARRIER(comm, request)

IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int NBC_Ibarrier(MPI_Comm comm, NBC_Handle* request)
```

NBC_IBARRIER(COMM, REQUEST, IERR)
INTEGER COMM, IERROR, REQUEST

NBC_Ibarrier initializes a barrier on a communicator. NBC_Wait may be used to block until it is finished.

> *Advice to users.* A non-blocking barrier sounds unusable because barrier is defined in a blocking manner to protect critical regions. However, there are codes that may move independent computations between the NBC_Ibarrier and the subsequent Wait/Test call to overlap the barrier latency.
>
> *Advice to implementers.* A non-blocking barrier can be used to hide the latency of the barrier operation. This means that the implementation of this operation should incur only a low overhead (CPU usage) in order to allow the user process to take advantage of the overlap.

C.4.2 Broadcast

NBC_IBCAST(buffer, count, datatype, root, comm, request)

INOUT	`buffer`	starting address of buffer (choice)
IN	`count`	number of elements in buffer (integer)
IN	`datatype`	data type of elements of buffer (handle)
IN	`root`	rank of the broadcast root (integer)
IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm, NBC_Handle* request)
```

NBC_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERR)
<type> BUFFER(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR, REQUEST

> *Advice to users.* A non-blocking broadcast can efficiently be used with a technique called “double buffering”. This means that a usual buffer in which a calculation is performed will be doubled in a communication and a computation buffer. Each time step has two independent operations - communication in the communication buffer and computation in the computation buffer. The buffers will be swapped (e.g., with simple pointer operations) after both operations have finished and the program can enter the next round. Valiant’s BSP model [208] can be easily changed to support non-blocking collective operations in this manner.

C.4.3 Gather

NBC_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

IN	`sendbuf`	starting address of send buffer (choice)
IN	`sendcount`	number of elements in send buffer (integer)
IN	`sendtype`	data type of sendbuffer elements (handle)
OUT	`recvbuf`	starting address of receive buffer (choice, significant only at root)
IN	`recvcount`	number of elements for any single receive (integer, significant only at root)
IN	`recvtype`	data type recv buffer elements (handle, significant only at root)
IN	`root`	rank of receiving process (integer)
IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int NBC_Igather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm, NBC_Handle* request)
```

NBC_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,RECVTYPE, ROOT, COMM, REQUEST, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR, REQUEST

C.4.4 Neighbor Exchange

NBC_INEIGHBOR_XCHG(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

IN	`sendbuf`	starting address of send buffer (choice)
IN	`sendcount`	number of elements in send buffer (integer)
IN	`sendtype`	data type of sendbuffer elements (handle)
OUT	`recvbuf`	starting address of receive buffer (choice, significant only at root)
IN	`recvcount`	number of elements for any single receive (integer, significant only at root)
IN	`recvtype`	data type recv buffer elements (handle, significant only at root)
IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int NBC_Ineighbor_xchg(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm, NBC_Handle* request)
```

NBC_INEIGHBOR_XCHG(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECV-COUNT,RECVTYPE,
COMM, REQUEST, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR, REQUEST

C.4.5 Other Collective Routines

All other collective routines in MPI can be executed in a non-blocking manner as shown above. The operation MPI_<OPERATION> is renamed to NBC_I<OPERATION> and a request-reference is added as last element to the argument list. All collective routines are shown in Table C.4.5.

NBC_IBARRIER	NBC_IBCAST
NBC_IGATHER	NBC_IGATHERV
NBC_ISCATTER	NBC_ISCATTERV
NBC_IALLGATHER	NBC_IALLGATHERV
NBC_IALLTOALL	NBC_IALLTOALLV
NBC_IALLTOALLW	NBC_IREDUCE
NBC_IALLREDUCE	NBC_IREDUCE_SCATTER
NBC_ISCAN	NBC_IEXSCAN

Table A.1: Proposed non-blocking collective functions

C.5 Examples

This section presents two examples. Section C.5.1 is intended for implementers who want to implement new non-blocking collective algorithms using the NBC scheduler. Section C.5.2 is intended for MPI programmers which want to use the non-blocking collective operations to optimize their program.

C.5.1 Implementing Collective Algorithms in LibNBC

New collective algorithms can easily be added to LibNBC. We present an example of the implementation of the dissemination barrier, that has been proposed in [93] and a pseudo-code is given in [22]. The implementation with the NBC scheduler can be found in `nbc_ibarrier.c` in the LibNBC sources. An excerpt of the code is show in Listing A.1
Line 13 creates a new schedule. The algorithm consists of $log_2 P$ rounds and processes synchronize pairwise in each round. This means that each process sends and receives to/from other processes each round and it has to wait until the messages have been received. The rank-specific schedule is built in the do-loop (Line 17-32). The schedule is committed in Line 36 and ready for use afterwards. Line 27 starts the scheduler to execute the schedule (non-blocking).
A schedule of rank 0 in a 4-process communicator would consist of two rounds. Each round consists of a send and a receive operation, and its representation in memory is shown in Figure A.1.

send to 1	recv from 3	end	send to 2	recv from 2	end

Figure A.1: Schedule for Rank 0 of 4 for a Dissemination Barrier

```
int NBC_Ibarrier(MPI_Comm comm, NBC_Handle* handle) {
  int round, rank, p, maxround, res, recvpeer, sendpeer;
  NBC_Schedule *schedule;

  res = MPI_Comm_rank(comm, &rank);
  res = MPI_Comm_size(comm, &p);

  schedule = malloc(sizeof(NBC_Schedule));

  round = -1;
  handle->tmpbuf=NULL;

  res = NBC_Sched_create(schedule);

  maxround = (int)ceil((log(p)/LOG2)-1);

  do {
    round++;
    sendpeer = (rank + (1<<round)) % p;
    /* add p because modulo does not work with negative values */
    recvpeer = ((rank - (1<<round))+p) % p;

    /* send msg to sendpeer */
    res = NBC_Sched_send(NULL, 0, MPI_BYTE, sendpeer, schedule);

    /* recv msg from recvpeer */
    res = NBC_Sched_recv(NULL, 0, MPI_BYTE, recvpeer, schedule);

    /* end communication round */
    if(round < maxround){
      res = NBC_Sched_barrier(schedule);
    }
  } while (round < maxround);

  res = NBC_Sched_commit(schedule);

  res = NBC_Start(handle, comm, schedule);

  return NBC_OK;
}
```

Listing A.1: Dissemination Barrier in LibNBC (error checks removed)

```
int function() {
  buffer comm, comp;
  NBC_Handle handle;

  do {
    /* do some computation on comm buffer */

    NBC_Ibcast(comm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD, &handle);

    /* do some computation on comp buffer */

    NBC_Wait(&handle);
  } while (problem_not_solved);
}
```

Listing A.2: Code example with non-blocking NBC_Ibcast

```
int function() {
  buffer comm, comp;
  NBC_Handle handle;

  do {
    /* do some computation on comm buffer */

    NBC_Ibcast(comm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD, &handle);

    /* do some computation on comp buffer */

    NBC_Test(&handle);

    /* do some computation on comp buffer */

    NBC_Wait(&handle);
  } while (problem_not_solved);
}
```

Listing A.3: Code example with non-blocking NBC_Ibcast with NBC_Test

C.5.2 Using Non-Blocking Collective Operations from a MPI Program

The transition from blocking MPI collective operations to non-blocking NBC collective operations is simple. The NBC interface is similar to the blocking MPI collective operations and adds only a NBC_Handle as last parameter. A code using NBC_Ibcast is shown in Listing A.2. To ensure progress in the background, the user should call NBC_Test on active handles as shown in Listing A.3.

D Vita

Torsten Hoefler was born in Magdeburg, Germany on March 31st 1981. He finished his *Abitur* (university-entrance diploma) at the *Gymnasium Oelsnitz* as the third-best student of his year in 1999. He served (mandatory) as a computer expert in the *Bundeswehr* (German army) until 2000, when he started his studies of Computer Science at the *Technische Universität Chemnitz* (University of Technology Chemnitz). He finished his *Diplom* (similar to a Master's degree) in Computer Science, among the first of his entry-class, in early 2005 with the highest predicate *"sehr gut"* ("very good"). His thesis was awarded with the *Universitätspreis 2005* (best student award 2005).

He started his Ph.D. studies in Chemnitz in the group of Prof. W. Rehm with support by a grant from AMD Saxony to enhance quantum mechanical computations on clusters of workstations. He won the *PARS Nachwuchspreis 2005* (Junior researcher award) from the *Gesellschaft für Informatik* (German Computer Society). He was also awarded with a fully funded HPC-Europa stay at the *CINECA Consorzio Interuniversitario* (Italian High-Performance Computing Center) in Caseleccio di Reno, Italy in November/December 2005. The results of his stay were named as a "scientific highlight" of the exchange program in the year 2005. Beginning from 2006, he received a full scholarship/stipend from the *Sächsisches Staatsministerium für Wissenschaft und Kunst* (Saxon Ministry of Science and the Fine Arts). He was part of the scientific design and advisory council to design the *CHiC* supercomputer system (rated #117 on the Top500 list in June 2007). He also advised eight *Diplom* theses during his time in Chemnitz.

In the summer (March-September) of 2006, he was invited by Prof. A. Lumsdaine as a visiting scholar to Indiana University Bloomington. In January 2007, he worked under Gilles Zérah at the Département de Physique Théorique et Appliquée, Commissariat á l'Énergie Atomique (CEA-DAM) in Bruyères-le-Châtel, France in order to improve the performance in quantum mechanical computations on cluster systems.

In 2007, he transferred with his Ph.D. program to the Computer Science Department at Indiana University in order to work with Prof. A. Lumsdaine. He is member of the IEEE and ACM and the Message Passing Interface (MPI) Forum. He co-founded and co-organizes the yearly German KiCC workshop series on *Kommunikation in Clusterrechnern und Clusterverbundsystemen* (Communication in Cluster Computers and Clusters of Clusters) since 2005.