Dissertation

The Benefits of One-Sided Communication Interfaces for Cluster Computing

vorgelegt von

Dipl.-Inf. Lars Schneidenbach

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
in der Wissenschaftsdisziplin "Informatik"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
Universität Potsdam

angefertigt am
Institut für Informatik
Professur für Betriebssysteme und Verteile Systeme

Betreuung:
Prof. Dr. Bettina Schnor

Potsdam, im März 2009

Contents

1 Introduction **15**
- 1.1 The Design of One-Sided Communication APIs 16
- 1.2 Terms and Definitions 17
- 1.3 Outline and Contribution of This Thesis 18

2 Cluster Computing and Network Technology **21**
- 2.1 Cluster Usage 22
  - 2.1.1 Parallel Applications 22
  - 2.1.2 File Systems 23
  - 2.1.3 Client-Server Applications and Server Load Balancing . . 24
- 2.2 Network Technology for Clusters 24
  - 2.2.1 Ethernet 24
  - 2.2.2 Myrinet 26
  - 2.2.3 InfiniBand 27
  - 2.2.4 Quadrics 32
- 2.3 Summary 32

3 Cluster Communication (Related Work) **33**
- 3.1 Efficiency 34
- 3.2 Interfaces for Communication 35
  - 3.2.1 Sockets 35
  - 3.2.2 VIA 37
  - 3.2.3 Verbs 39
  - 3.2.4 DAPL 39
  - 3.2.5 Myrinet MX 40
  - 3.2.6 Summary 40
- 3.3 Parallel Programming 41
  - 3.3.1 Shared Memory 41
  - 3.3.2 Message Passing 42
- 3.4 Message Passing Interface 43
  - 3.4.1 MPICH2 46

3.4.2 MVAPICH 47
3.4.3 Open MPI 47
3.4.4 MPICH-G2 48
3.4.5 Efficiency of MPI 48
3.5 OpenMP 49
3.6 Further APIs 50
3.6.1 ARMCI 50
3.6.2 LAPI 51
3.6.3 SHMEM 52
3.6.4 GASNet 52
3.6.5 Global Address Space Languages 52
3.6.6 Unified Parallel C 52
3.7 Design Aspects of Efficient Communication 53
3.7.1 Buffer Management 53
3.7.2 Overlap 57
3.7.3 Offload 60
3.7.4 Progress 61
3.7.5 Transport of Data 63
3.8 Interdependencies 66
3.9 Conclusion 67

4 A Model for Communication **69**
4.1 IPC: Producer/Consumer 70
4.2 The Virtual Representation Model 71
4.2.1 Application 72
4.2.2 Communication System 76
4.2.3 How to Apply the Model 79
4.3 Aspects of Efficient Communication 81
4.3.1 Communication Pipeline 81
4.3.2 Sending and Writing 82
4.3.3 Receiving 82
4.3.4 Reading Data 84
4.3.5 Synchronisation 85
4.3.6 Bi-directional Synchronisation 86
4.3.7 Summary 88
4.4 Design Criteria for an Efficient One-Sided Interface 88
4.4.1 Communication 89
4.4.2 Synchronisation 89
4.4.3 Completion 92
4.4.4 Bi-directional Synchronisation 93
4.5 Conclusion 93

5 One-Sided Communication for Parallel Applications 95
5.1 Application Analysis: Cellular Automaton 96
5.1.1 Measurements 97
5.1.2 Point-to-Point 97
5.1.3 One-Sided Communication 102
5.1.4 Summary 103
5.2 The NEON API 104
5.2.1 Introduction 104
5.2.2 Name and Address of Buffers 105
5.2.3 Buffer Announcement 107
5.2.4 Completion 109
5.2.5 Communication 110
5.2.6 Summary 112
5.3 NEON over Sockets 112
5.3.1 General Design 113
5.3.2 Socket Specific Design 114
5.3.3 Implementation 116
5.3.4 Evaluation 119
5.4 NEON over InfiniBand 121
5.4.1 Design 122
5.4.2 Implementation 122
5.4.3 Evaluation 123
5.5 NEON in Shared Memory Environments 125
5.5.1 Applying the Communication Model 126
5.5.2 Synchronisation 126
5.5.3 Data Exchange 126
5.6 What's New 127
5.7 Conclusion 128

6 One-Sided Communication for Server Load Balancing 131
6.1 Server Load Balancing 132
6.1.1 Server Load Balancing Techniques 132
6.1.2 Architecture of Server Load Balancing 133
6.1.3 Quality of Scheduling 134
6.1.4 Load Balancing Algorithms 135
6.1.5 Server Weights 136
6.1.6 Conclusion 137
6.2 A Case for One-Sided Communication 138
6.2.1 Resource Monitoring 138
6.2.2 Characteristics of Resource Monitoring 139
6.3 Credit-Based Scheduling 140

6.3.1 Credits 140
6.3.2 Monitorable Resources 141
6.3.3 Scheduling 144
6.4 Credit Reporting Algorithms 145
6.4.1 Algorithms 145
6.5 Evaluation of Algorithms 148
6.5.1 Factors 148
6.5.2 Primary Factors 149
6.5.3 Secondary Factors 149
6.5.4 Simulation 151
6.5.5 Minimum Number of Credits 154
6.5.6 Single Server 155
6.5.7 Two Servers 156
6.5.8 Summary of Important Impacts 160
6.5.9 Limitations of Simulation Results 162
6.5.10 Summary 163
6.6 Implementations of SlibNet 163
6.6.1 SlibNet: InfiniBand-Based Credit SLB 164
6.6.2 SlibNet: Credit-Based Scheduling 164
6.6.3 SlibNet: Socket-Based Credit SLB 165
6.6.4 Problems with InfiniBand 166
6.6.5 Evaluation 167
6.7 Conclusion 169

7 Conclusions **171**
7.1 Conclusions from the Communication Model 172
7.2 Results from the NEON API 172
7.3 Results from Server Load Balancing 173
7.4 Future Work 174

A Benchmark Testbeds **175**
A.1 Uranus 175
A.2 Einstein 175
A.3 InfiniBand Cluster 176

B Benchmarks **177**
B.1 Micro-Benchmarks 177
B.1.1 MPI Ping-Pong 177
B.1.2 Eins 178
B.2 Application Benchmarks 179
B.2.1 The Cellular Automaton 179

B.2.2 httperf 180
B.2.3 RUBiS 180

C Measurement Data 181
C.1 SlibNet: Simulation of Credit Algorithms 181

D Specification of Units 183

Bibliography 187

Index 205

Acknowledgement

This page is dedicated to all the people who supported this thesis. This is to express my gratitude to them.

First, I want to thank my supervisor Professor Doctor Bettina Schnor. She supported this work with discussions and helpful advises over the years. She offered great opportunities for my scientific work.

One special person has to be mentioned here: Klemens Kittan. Without his commitment to setup and fix problems with hardware or software, this work would not have been done within the next years. I want to thank him for his humor and support here. We are lucky to have such a great administrator.

A great thank goes to Stefan Liske for fruitful advises and bothering me with questions. This was very helpful.

I want to thank all the (former) students of the work group of the professorship for operating systems and distributed systems. A lot of fruitful discussions and fun happened during their masters and diploma thesis.

Thanks to Brendan Boulter for proofreading and checking my *expertise* in English spelling and grammar.

Personally, I want to gracefully thank my wife Sabine. This thesis would not have been possible without her backing. I don't have to forget my family and my friends for their tolerance when I had no time to contact them as regularily as I wanted. Erik Linus and Arne: this work is also for you.

Deutsche Zusammenfassung

Cluster bestehen aus einzelnen Rechnern, die über ein Netzwerk miteinander verbunden sind. Die Einsatzgebiete von Clustern reichen von Servern für vielgenutzte Webseiten und Internet-Dienste bis hin zu den schnellsten Parallelrechnern der Welt. Insbesondere Cluster als Parallelrechner sind seit den 90er Jahren des letzten Jahrhunderts nicht mehr aus der Welt der parallelen Anwendungen wegzudenken. 80 % der Rechner in der Top 500 sind inzwischen der Cluster-Architektur zugeordnet.

Der verteilte Speicher von Clustern erfordert jedoch hochperformante Netzwerkverbindungen und Übertragungsprotokolle, um einen effizienten Datenaustausch ermöglichen. Heutige Technologien wie InfiniBand unterstützen dabei den direkten Zugriff auf den Speicher anderer Prozesse. Um diese *remote direct memory access* (*RDMA*) genannte Technik direkt in Anwendungen nutzen zu können, wurden Programmierschnittstellen zur *one-sided*-communication entwickelt.

Die wohl bekannteste Programmierschnittstelle für parallele Anwendungen ist das *Message Passing Interface* (*MPI*). Seit Version 2.0 werden auch Primitiven für einseitige Kommunikation unterstützt. Bestehende Implementationen dieser Primitiven werden ständig verbessert, dennoch schneiden diese neuen Wege zum Datenaustausch noch immer schlecht ab im Vergleich zur klassischen zweiseitigen Kommunikation mit Sende- und Empfangsaufrufen. Es finden sich auch in der Literatur nur einige wenige Hinweise auf Vorteile beim Einsatz von einseitiger Kommunikation. Diese stammen jedoch aus dem angrenzenden Forschungsgebiet der Programmiersprachen für Globale Addressräume, welche nicht dem Message-Passing-Paradigma folgen.

In dieser Dissertation werden die Ursachen analysiert und gezeigt, dass dies nicht an mangelnder Reife der Implementationen oder gar an Fehlern bei der Implementierung liegt sondern an der Spezifikation der MPI-Schnittstelle. Die erforderliche Synchronisation der Datenübertragung spielt dabei eine wesentliche Rolle.

Anhand eines Kommunikationsmodells – *Virtual Representation Model* genannt – werden Designkriterien für eine effiziente Programmierschnittstelle für parallele Anwendungen hergeleitet. Es wird gezeigt, dass die Spezifikation von MPI-2 one-sided-communication diese Kriterien nicht erfüllt. Mit NEON wird auf Basis der Designkriterien eine Schnittstelle entworfen und für TCP/IP-Sockets und InfiniBand Verbs implementiert und bewertet. Die Ergebnisse zeigen, dass NEON eine effiziente und portable Schnittstelle darstellt. Des Weiteren zeigt sich, dass die Fähigkeit von InfiniBand direkt auf entfernten Speicher zuzugreifen nicht a priori von Vorteil bei der Implementierung von einseitiger Kommunikation ist.

Ein weiteres großes Anwendungsgebiet von Clustern ist so genanntes *Server Load Balancing*. Diese Technik erlaubt es die Anfragelast von Clients auf viele

Server zu verteilen und einen Dienst dadurch skalierbar, flexibel und ausfallsicher zu machen. Während bei parallelen Anwendungen vorrangig homogene Maschinen in einem Cluster arbeiten werden beim Server Load Balancing häufig heterogene Cluster eingesetzt. Dies hat meist historische Gründe, wenn zum Beispiel ein einzelner Server für einen Internet-Dienst nicht mehr ausreicht und um einen oder weitere Rechner ergänzt wird. Dann ist eine Lastverteilung notwendig, die Anfragen von Clients auf die vorhandenen Server verteilt. Bei der Verteilung sollten unterschiedliche Kapazitäten und freie Ressourcen berücksichtigt werden, um eine optimale Auslastung der Rechner zu gewährleisten und den Dienst für möglichst viele Clients gleichzeitig bereit zu stellen.

In der vorliegenden Arbeit wird die Überwachung der Server – das Ressourcen Monitoring – mit Hilfe von einseitiger Kommunikation als eine effiziente Technik vorgestellt. Dabei kommen sogenannte *Credits* zum Einsatz, die eine einfache Metrik für die zukünftige Verfügbarkeit des jeweiligen Servers darstellen. Es stellt sich heraus, dass die Lastverteilung auf der Basis von Credits nahezu ohne Synchronisation auskommt und damit für einseitige Kommunikation gut geeignet ist. Auf dieser Basis wird ein selbst-adaptierendes credit-basiertes Lastverteilungsverfahren vorgestellt, dass sich sowohl an heterogene Cluster als auch an heterogene Anfragen anpasst. Das Verfahren wählt in konstanter Zeit einen verfügbaren Server aus. Es kommt ohne aufwändige Sortierung der verfügbaren Server aus und ist somit hochskalierbar.

Die Arbeit zeigt, dass die Synchronisation den entscheidenden Einflussfaktor auf die Effizienz der Kommunikation zwischen Anwendungen darstellt. Die Kommunikationsleistung von einseitiger Kommunikation wirkt sich auf Anwendungen nur dann positiv aus, wenn ein reduzierter Synchronisationsbedarf besteht.

Abstract

Clusters consist of single computers connected by a network. The field of application of clusters include servers of highly utilised web pages and Internet services as well as the fastest parallel computers of the Top500 list. Clusters have conquered the Top500 list since the 90s of the last century. 80 % of the machines of this list are categorised as clusters.

However, the distributed memory of clusters requires high performance networks and protocols to provide efficient inter-process communication. Current technology like InfiniBand offer direct access to remote memory. This technique is called *remote direct memory access* (*RDMA*). Programming interfaces have been designed to exploit the merits of these network technologies at the application level.

The Message Passing Interface (MPI) is one of the most popular programming interfaces for parallel applications. Version 2.0 of the standard provides one-sided communication. Existing implementations of the standardised one-sided communication primitives are constantly improved. However, at the application level, classical two-sided communication still outperforms one-sided communication. There are only a few indicators from related research fields (global address space languages) that show beneficial use of one-sided communication in special areas. These languages do not follow the message passing paradigm.

This thesis is an investigation to analyse and show the causes of the inferior performance of one-sided communication. The specification of the programming interface is discovered to be a cause. Synchronisation plays a major role in the transmission of data. The inferior performance is not the result of immature or inefficient implementations.

Essential design criteria for an efficient one-sided communication interface are derived from a communication model called *Virtual Representation Model*. These criteria are not completely met by the MPI-2 one-sided communication API. A new interface, called NEON, is proposed, implemented, and evaluated on top of TCP/IP-Sockets and InfiniBand Verbs. The results show that NEON is an efficient and portable approach to one-sided communication for parallel applications. Furthermore, the results show that the availability of remote memory access by

hardware is not *a priori* an advantage for an implementation.

Another important application of clusters is *server load balancing*. This technique increases the flexibility, skalability, and fault tolerance of services. While clusters for parallel applications mostly consist of homogeneous machines, client-server applications often run on heterogeneous clusters. This is because these clusters evolve over time. When the server's capacity no longer meets the service demands, new machines are added to the cluster. A server load balancer component distributes the requests of the clients to an appropriate server in the cluster. The dispatcher should take into account the different capacities of the servers to achieve optimal balanced load of the servers.

This work promotes the monitoring of the server's resources as an efficient technique in the context of server load balancing. So called *Credits* are used to create a simple metric to represent the future availability of the servers. One-sided communication turns out to be suitable for credit reports since the credit-based scheduling works mostly without any synchronisation. A self-adapting credit-based server load balancing is proposed on the basis of one-sided communication. It is able to adopt to heterogeneous clusters and heterogeneous requests. The solution choses a server within constant time. It does not require sorting of available servers. This makes the solution highly scalable.

This thesis shows that synchronisation has a major impact on the efficiency of communication of applications. The performance of one-sided communication is beneficial for applications only if the required amount of synchronisation is limited.

Chapter 1

Introduction and Terminology

The benefits of one-sided communication interfaces for Cluster Computing is the title of this work. This title contains 4 terms requiring explanation before the motivation of this work is given. Starting from the last term *cluster computing* and continuing with *interface* and *one sided communication* (OSC) to the term *benefit*.

The term *cluster computing* includes all computations and processing that is performed with so-called *clusters*. A cluster in the context of this work is a conglomerate of computers interconnected with a network that are '... able to work together collectively as a single, integrated computing resource' [HX97] (page 30).

The term *interface* itself is assumed to be known, but the reason why interfaces are so important to this work requires further explanation. Programming interfaces or APIs are things a programmer has to know and to learn before translating an algorithm into source code. He or she has not only to know the syntax of an API but also its semantics. The more complex an API is, the more effort it takes (especially inexperienced) programmers to efficiently use the API for their purposes. On the other hand, a simple API requires more effort inside the system to efficiently fulfil complex tasks and reduces the flexibility of an API. Furthermore, the API determines the available semantics and, thus, can have an important impact on overall performance of an application. Therefore, it is worth to include the API into the research on communication.

A communication is called *one-sided* if only one of the communication partners determines the parameters of data transfer. This can be the message size, the destination host, the destination address in remote memory, the number of communication operations, or even the kind of operation (e. g. put or get data). Current trends in hardware design and message passing libraries promote *one-sided communication* as a way to improve the performance of applications in comparison to two-sided communication via send/receive [GT05, HSJP05a, GT07].

An API is called *beneficial* for cluster computing if its use improves the performance of an application that can be executed on clusters. Especially, this includes the large class of parallel applications as well as server load balancing for client-server applications. A user will recognise improved performance if the runtime of a given task is reduced, if a more complex or larger problem can be solved within the same amount of time or a new problem can be solved in a reasonable amount time. As a side aspect, an easier usage of an API can be interpreted as a benefit for the user.

1.1 The Design of One-Sided Communication APIs

Many problems cannot be solved within acceptable time if only a single process is used to solve the problem. Therefore, users try to reduce the runtime by exe-

cuting their applications in parallel on multiple processors. If a parallel algorithm requires shared data between at least two processes, inter-process communication (IPC) is required. IPC becomes a limiting factor of the performance in case of high latencies and limited bandwidth of the interconnects. So-called *massively parallel processor* (*MPP*) machines often provide fast interconnects or even shared memory. But these facilities are very expensive. Clusters are a cost effective alternative to massively parallel processors (MPP).

Various publications present poor performance figures [TRH00, LWP04, HSJP05b, Raj05] of one-sided communication using MPI (Message Passing Interface) compared to send/receive-based communication. This thesis investigates the causes and possible solution this poor performance.

Apart from the API, another question motivates this work: Even if one-sided communication is faster than two-sided communication, are there any applications that can make efficient use of OSC? What conditions must be met in order for an application to benefit from OSC?

1.2 Terms and Definitions

This section explains and defines some important terms that are frequently used throughout the document.

API call An *API call* is defined as an instruction inside a program that executes a routine of a library before it returns to the calling process.

asynchronous The term *asynchronous* is the counterpart of synchronous. It describes the processing of an operation in the background. Asynchronous processing does not prevent other operations from being processed.

blocking A *blocking* API call does not return before the requested operation is complete.

buffer A *buffer* is any kind of a limited chunk of memory either on a device or inside the main memory (RAM). The term *memory region* will be used as a synonym.

completion *completion* determines that a non-blocking operation is finished. This is related to *notification*.

host The terms *host*, *computer*, *node*, and *machine* represent a single element of a cluster – an independently working computer attached to a network.

interface An *interface* can be a programming interface (API) or a network interface card (NIC). Usually, the context should explain the intended meaning.

non-blocking A *non-blocking* API call immediately returns to the caller after it has initiated the requested operation. A non-blocking call requires *completion*.

notification A *notification* is sent out to signal the *completion* of an operation. Also after a *notification* is received, an operation can complete.

remote direct memory access (RDMA) *RDMA* is a technique using DMA-based transfer methods to put data into the physical memory of a remote host. This hardware can provide an API to perform RMA operations. Thus, this hardware can be called *RMA-capable*.

remote memory access (RMA) is the process of accessing data in the memory or address space of another process. Despite the term *remote*, the target process can run either on a remote or a local CPU. Although it is unusual to categorise the *direct access* to the data of another thread as *remote memory access*, this direct access can be used to implement an API for *RMA*.

synchronisation point If an application requests operations to synchronise between two or more processes, this is considered as a *synchronisation point*.

synchronous The term *synchronous* is sometimes taken as a synonym for *blocking*. The term *blocking* is more likely to be used in conjunction with API calls (see above). An operation is processed synchronously if the operation is complete after the processing. No other operations can be executed during the processing.

1.3 Outline and Contribution of This Thesis

Chapter 2 and 3 introduce the application of clusters, important network technology, and related work in the area of communication APIs and design aspects of efficient communication.

The *Virtual Representation Model*, presented in Chapter 4, abstracts the layers of the ISO/OSI-Reference Model [JTC94] to application and communication system. This model is inspired by the combination of *producer/consumer* synchronisation between applications and splitting inter-process communication into pipeline steps. The model allows to understand the steps of communication and synchronisation of one-sided and two-sided communication. Important criteria of efficient API design and implementation can be derived from the model. This includes criteria for efficient one-sided communication. The location of intermediate buffers and the network traversal of buffer announcement messages can be derived too.

MPI is the most prominent example of APIs for the large class of parallel applications. This thesis shows the causes of the poor performance of MPI-2 one-sided compared to send/receive-based communication. It is not caused by immature or inefficient implementations of the API. It is caused by the design of the MPI-2 one-sided communication API.

On the basis of the model, an efficient one-sided communication interface for parallel applications, called NEON, is designed and implemented. In Chapter 5, two prototype implementations of the NEON-API are evaluated. Both the InfiniBand and the TCP/IP-Socket-based implementation over Ethernet are able to outperform well designed MPI-2 implementations running a Cellular Automaton.

The main contribution is that separation of notification and completion for non-blocking one-sided communication is beneficial to parallel applications. The NEON-API exploits this aspect. The evaluation confirms that the separation improves the performance of parallel applications. This separation is unique in today's one-sided communication APIs for parallel applications.

Efficient APIs require a parameter to communication calls to distinguish between final and non-final operations. This allows for sending the notification as early as possible. Otherwise, it is impossible to implement the API on top of networks that can only be used efficiently if synchronisation is embedded in the data messages. Since a final operation is not always known in some algorithms, a separate notification call should be available.

An investigation in a Cellular Automaton with a bidirectional synchronisation between neighbouring processes shows that these applications will suffer from an implicit barrier if the notification and the completion are combined in a single API call. In case of the Cellular Automaton, it reduces the process skew tolerance and prevents the notification message to be overlapped with computation.

The limiting factor of the performance of one-sided communication is the synchronisation. This is indicated by the Virtual Representation Model and confirmed by the evaluation of the NEON-API. Applications that require fewer synchronisation benefit from one-sided communication. The other important application of clusters is server load balancing. Chapter 6 identifies advantages of one-sided communication for resource monitoring which is often used in conjunction with server load balancing.

The investigation of Chapter 6 identifies the number of free socket endpoints as a simple and efficient metric to determine the free resources of a server. A load balancing on top of this metric schedules the request according to the current availability of a server in contrast to the current load of a server.

A new credit-based self-adapting server load balancing is proposed and evaluated. A simulation study shows that the best scheduling is achieved if two credit

values are reported. First the servers tell the dispatcher the number of requests[1] they want to handle. The second credit value represents the upper limit of the server. Credits can efficiently be reported using one-sided communication. Using this reporting scheme together with a fast ($O(1)$) round-robin scheduling results in scalable and efficient low overhead scheduling that self-adapts to heterogeneous servers and heterogeneous requests. Since this approach avoids most of the synchronisation messages and allows for efficient use of one-sided communication.

Chapter 7 summarises and concludes this thesis and points out interesting future work.

[1] each request consumes one credit

Chapter 2

Cluster Computing and Network Technology

This chapter gives an overview of common applications and network interconnects of clusters. The communication system and its interfaces can be seen as the link between application and network. Therefore, the applications and the network technologies are the most important constraints to the design of a communication system.

2.1 Cluster Usage

Clusters of workstations are playing an increasingly important role in scientific and industrial applications. Clusters constitute about 81 % of the Top500 list of supercomputers in November 2007 (82 % in November 2008). This contrasts to the situation in November 2003, where clusters only accounted for 42 % of the systems. The number of clusters in the list doubled within 4 years.

The clusters in this list are primarily used for parallel applications. Clusters are also used for applications like parallel file systems to provide faster storage for data, and for server load balancing to build flexible, fault tolerant, and scalable services to serve a large number of clients.

This section gives an overview of the most widespread usage of clusters and presents some examples.

2.1.1 Parallel Applications

In general, parallel applications are used to shorten the time required to solve a given problem or to calculate more complex tasks within an acceptable time. The work is distributed over several processors – the domain is decomposed. There are several APIs and target platforms available to implement parallel algorithms.

2.1.1.1 Bulk-Synchronous Applications

In *bulk-synchronous* applications, the domain is decomposed and the work is distributed amongst the processes of the process group.

The large class of *bulk-synchronous* applications is very important to this work. A significant number of scientific applications fall into this class. According to [WA99], *bulk-synchronous* means that the instances of the parallel applications work individually on their part of the domain for a limited amount of time. Before continuing, the instances are required to synchronise with some or all other instances. Synchronisation is required if the processes of the parallel application need some of the results calculated by other processes.

Examples:

- the Modular Ocean Model (MOM) from Geophysical Fluid Dynamics Laboratory (GFDL) in Princeton and its extensions by the Potsdam Institute for Climate Impact Research (PIK) [Pot04].
- the AMIGA Halo Finder (AHF) that is developed at the Astrophysikalisches Institut Potsdam (AIP) [Ast08].
- Gadget simulates mass interaction in astrophysics. It solves a so-called *N-body problem* in parallel [SYW01, Spr05].

2.1.1.2 Master-Worker Applications

A *master-worker* approach can be considered if the domain cannot be efficiently decomposed or the decomposition or the size of the domain is not *a priori* known. One dedicated process of the group becomes the master. The master splits a job into sub-jobs that are processed by the workers. If a worker is ready, it sends back the results to the master and requests further work.

Examples are:

- The parallel version of the LIDAR tool that retrieves micro-physical parameters from LIDAR measurements and simulations [BMM$^+$05, EAR08]. This software was developed in the context of the EARLINET-ASOS project[1].
- The answer set solving programs *platypus* and *clasp* developed at the department of computer science at the University of Potsdam [GJM$^+$05, GJM$^+$06, Ell08, GKNS07b, GKNS07a].

2.1.2 File Systems

If a single file server is no longer sufficient to fulfil the demands, parallel file systems are a possible solution. These systems provide a consistent view on the files that are physically distributed among a number of disks and machines. If the disk-I/O of parallel applications (checkpointing, logging, results) exceeds the capabilities of a single disk or server, the application will have to wait longer and longer. If the wait time becomes too long, even a parallel processing of an algorithm is not efficient and investigations or investments into application parallelisation and hardware are wasted.

[1] This work was supported by the European Commission under grant RICA-025991 via project EARLINET-ASOS which is gratefully acknowledged.

Popular examples are:

- The Parallel Virtual File System (PVFS) is a free development from Argonne National Lab and Clemson University [PVF03].
- Lustre is a parallel file system developed and maintained by Cluster File Systems, Inc [Clu03]. This company was acquired by Sun Microsystems in October 2007.

2.1.3 Client-Server Applications and Server Load Balancing

In general, the large class of client-server applications is used to deploy services to customers in the worldwide Internet. Primarily, using clusters in this environment is required only if the capabilities of a single server become insufficient. In this case, so-called *server load balancing* techniques are applied to distribute the load of requests to multiple servers. This makes a service more scalable.

Additionally, server load balancing enhances the fault tolerance of a service. If a server fails, other servers are available to the clients. Furthermore, it makes the service flexible since single machines and services can be upgraded, maintained, or extended without interrupting the accessibility of the service. The number of machines can be adopted on demand.

2.2 Network Technology for Clusters

The network is the central component of a communication system required to allow for communication of applications. The type of network usually used for clusters are local area networks and system area networks. In this section, an overview over existing network hardware is given.

More detailed explanation for Ethernet and InfiniBand are presented in this section because this knowledge is important for later use in this document. Coarse performance figures are presented here to show the general behaviour of the specific network. More detailed measurements are presented later.

2.2.1 Ethernet

Ethernet is a commonly used technology for local area networks (and partially metropolitan area networks). It is specified as a standard (IEEE 802.3) [Ins85]. The first standard specified a network with 10 Mbit/s. Subsequent standards of the 802.3 series describe FastEthernet (100 Mbit/s), GigabitEthernet (1 Gbit/s), and 10GigabitEthernet (10 Gbit/s).

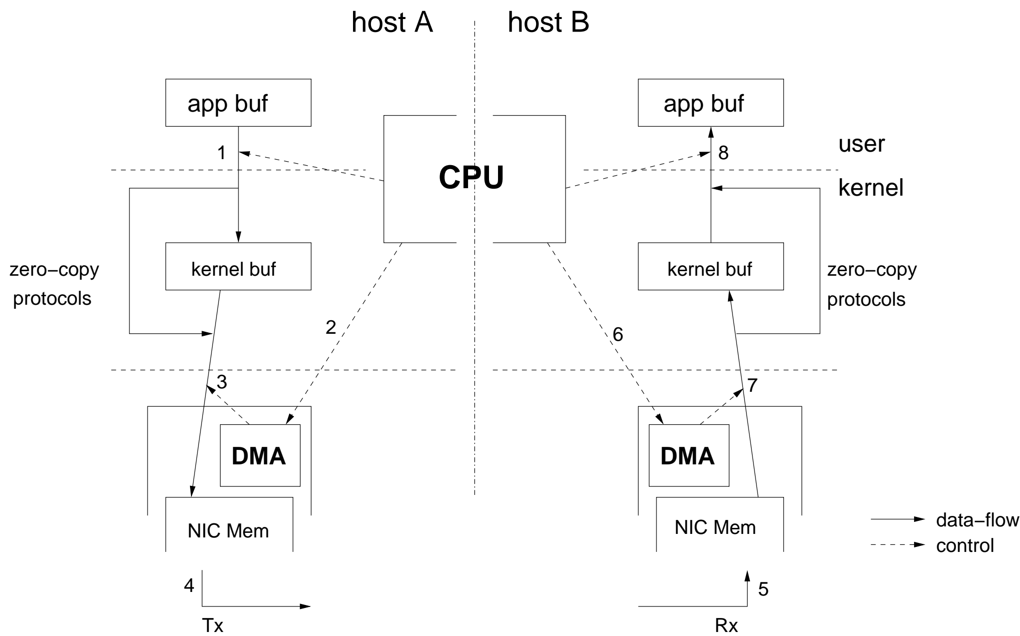

Figure 2.1: General Ethernet data transmission scheme using DMA.

Even though Ethernet is quite old compared to modern interconnect technology it is still an important technology in the field of high performance computing. 282 machines (56.80 %) of the sites in the Top500 list (11/2008) are equipped with GigabitEthernet. There is also ongoing research on Ethernet to improve the network capabilities. [EL06] presents Ethernet-based CLOS topology.

Today's Ethernet network interface controllers (NIC) are able to move data between local memory and NIC memory using DMA engines on the NIC. This allows for data transfers without the host CPU being involved. Some controllers, for example the Broadcom BCM5704 chip, are able to calculate TCP and IP checksums to offload some protocol processing and further reduce CPU overhead of communication.

Ethernet is not able to directly access remote memory, and most of the protocol processing is still required to transfer data. Additionally, the best latency achievable with standard the TCP/IP protocol over Ethernet is very high – compared to local memory-to-memory data exchange. Also the throughput of GigabitEthernet is significantly lower than local memory-to-memory communication (see below).

Performance Details: Since Ethernet is important to this work, some details of performance and internals have to be described. The steps that have to be done to submit data from an application to a remote application are (shown in Figure 2.1):

1. Switching to kernel context due to a send operation call and process the

protocol stack including the Ethernet driver (1).

2. The driver creates a descriptor that describes the packet and its location in memory.
3. The TxDMA engine of the NIC takes a copy of the packet to its own memory (2,3).
4. The packet is transmitted via the link (4, 5).
5. The remote RxDMA copies the packet to host memory and creates a corresponding descriptor (6,7).
6. As soon as the descriptor is created and the data is copied to memory, an interrupt will signal the protocol stack that a new packet has arrived. The receiving can continue.
7. The remote host has to switch to kernel context because of a receive call and processes the protocol stack including the Ethernet driver (8).

Looking at this process, it can be seen that some steps are performed by the host CPU and some are not. The steps in general and the distinction between host processor and other processing units in the system are important for this work, since they influence the pipelining of data and overlapping of communication and computation.

The time spent in particular steps is also important. If this is CPU time, it is not available to the application. Therefore it will be important to know the relation between communication time spent on the CPU and off the CPU.

The link layers of GigabitEthernet can transmit 1 Gbit/s. Taking into account the overhead information (header, trailer, inter-frame gap), a link has a throughput of about 121.9 MB/s. Single bytes have to be processed by the stack within about 8 ns. This is 16 clock tics per byte on a 2 GHz CPU. This is not much and shows the importance of efficient protocol processing.

Comparing the transmission over the network to an internal memory copy, the link speed of GigabitEthernet is still slow. Using an Intel Pentium 4 CPU with 2.8 GHz (see machines IB5 and IB6 in Appendix A.3), transferring a single byte over Ethernet using TCP/IP takes about 440 times the time of a single byte copy from one memory location to another. For larger messages this ratio is about 30. For InfiniBand, the best ratio achieved is about 3.1 to 4.0 for messages larger than 1 MB (all values measured by a back-to-back interconnect).

2.2.2 Myrinet

Myrinet [Ass98, BCF+95] is a local area network technology. It provides low latencies and high bandwidth by using cut-through routing (also called wormhole routing). The host interface cards of Myrinet are equipped with a processor. Therefore the Myrinet software can offload some processing to the NIC.

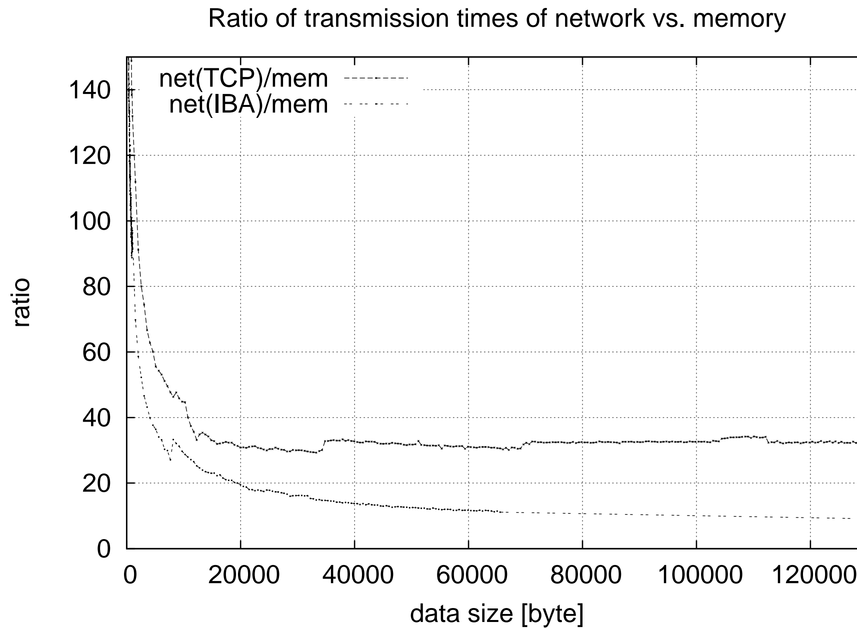

Figure 2.2: Ratio of transmission times (network vs. memory).

Myrinet-2000 hardware achieves a latency of 2.6 μs at user level (MPI) and a maximum throughput of about 240 MB/s. These self-measured data are an outcome of a tutorial at the International Super Computing (ISC) conference 2005 in Heidelberg (Germany).

2.2.3 InfiniBand

InfiniBand [Inf02a] is a modern industry standard developed by the InfiniBand Trade Association (recently renamed to Open Fabrics Alliance) and specifies a *system area network*. The first design also included host internal communication, i.e. the system bus. Today InfiniBand is used as an external network interconnection technology that provides transport layer services (see ISO/OSI model [JTC94, DZ83, Day95]).

InfiniBand distinguishes between processor nodes and I/O nodes. A processor node should be an independent node or host/computer. An I/O node is e.g. a storage subsystem. The network controllers of processor nodes are called *Host Channel Adapter* (HCA). A special type of HCA are *Target Channel Adapters* (TCA) that can be seen as network controllers of I/O nodes like storage systems (see Figure 2.3). The main difference is that HCA provide a standard consumer interface (Verbs), while the interface of TCAs is not specified.

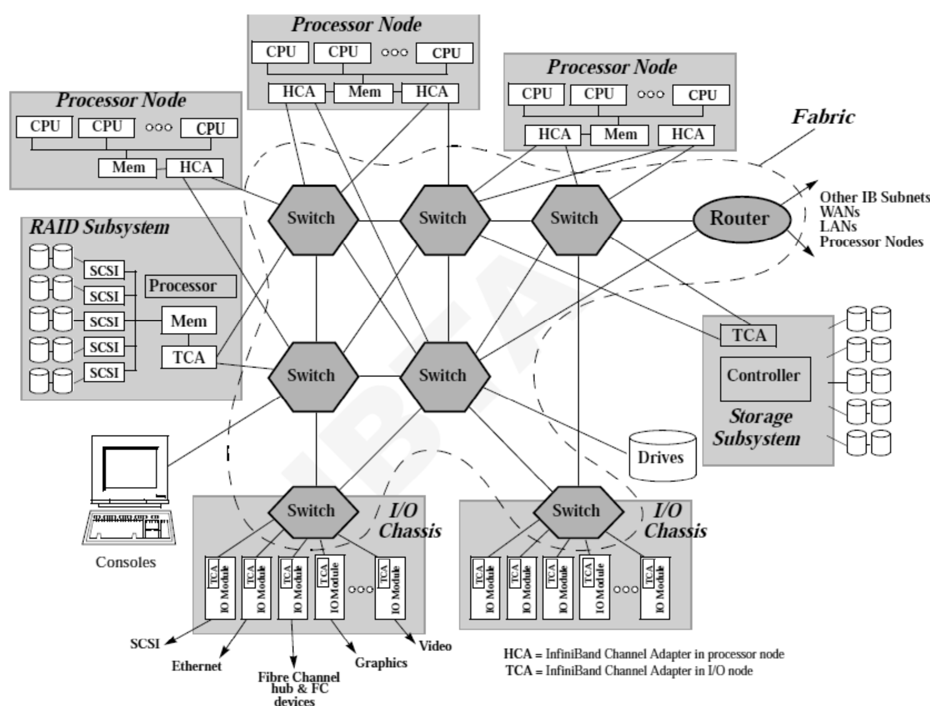

Figure 2.3: InfiniBand network overview [Inf02a].

The address of a node is composed from a global and a local identifier. These identifiers are configured and assigned by a so-called subnet manager (SM). This is a piece of software running once per subnet. To communicate to a remote node, the SM has to be asked for a route or path to the requested host. Afterwards, the host can be contacted.

HCAs are able to execute so-called *work requests* that are initiated by the applications or the software stack. These work requests are collected in queues (work queue - WQ). Each communication endpoint has two queues (a send queue and a receive queue) to hold work queue entries. A communication endpoint of InfiniBand is called *Queue Pair* (QP). If a *work queue entry* (WQE) is complete, it is moved to a *completion queue* (CQ). These queues and requests have to be created, posted, and fetched by the software stack to control InfiniBand's communication.

To connect or contact a remote QP, the general service interface (GSI) has to be contacted first. This is to request the communication manager (CM) for a particular service identifier (SID). Each application has to register available QPs together with a SID. This is similar to the portmapper concept of RPC [Sun88]. The CM returns a free QP to the requesting HCA.

The most prominent software stack for InfiniBand is the Open Fabrics Enterprise Distribution (OFED). It is developed by several companies and vendors

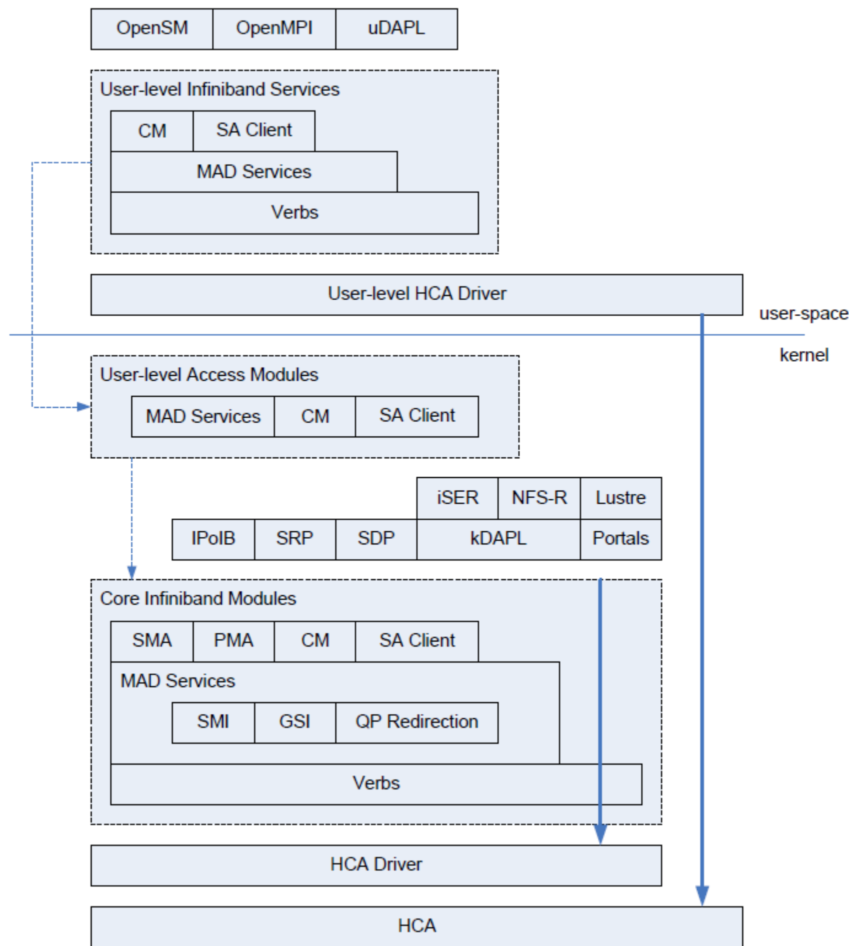

Figure 2.4: InfiniBand Programming Interfaces and Protocols [RnSH05].

(Mellanox, Cisco, Voltaire, IBM, etc.). InfiniBand drivers are included in the standard Linux kernel since version 2.6.11 and are constantly improved and extended.

There are several interfaces to different protocols build on top of the InfiniBand hardware interface (see Figure 2.4). The most fundamental standard interface is the so-called *Verbs*. It provides unified access to the drivers and hardware of different vendors. Nearly all of the interfaces are available for user-level access as well as kernel-level access.

IP over InfiniBand (IPoIB) enables IP-based protocols to communicate via InfiniBand. It has some performance drawbacks due to its additional software overhead. For example, created IP packets have to traverse the InfiniBand stack. However, the achievable throughput (about 300 MB/s) using the OFED 1.1 implementation is still above the throughput of GigabitEthernet. The latency of IPoIB (about 20 µs) is comparable to the latency of GigabitEthernet on Einstein

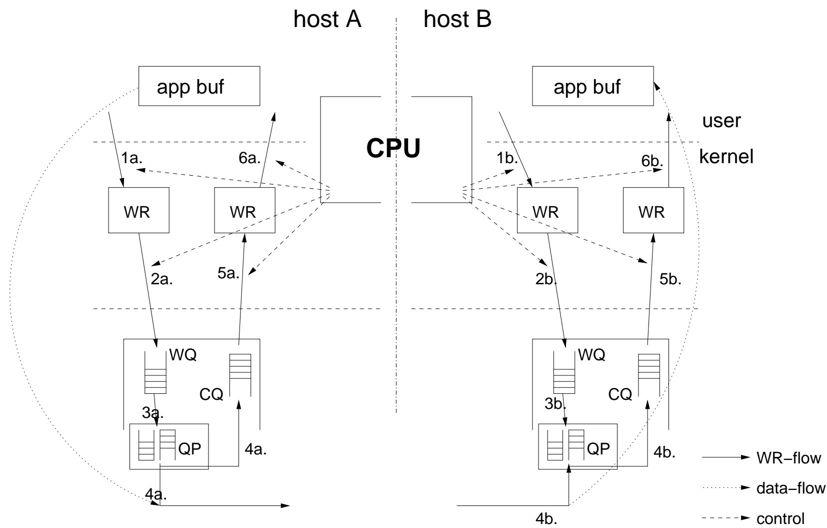

Figure 2.5: Data transmission scheme of InfiniBand (non-RDMA).

hardware (see Appendix A.2) measured with *Eins* [Sch06]. Current efforts in the OFED community try to improve the performance of IPoIB.

In contrast to IPoIB, the sockets direct protocol (SDP) [Inf02b] is part of the InfiniBand standard. It provides a socket interface and is intended to become a selectable address family for socket-based applications. The throughput (up to 900 MB/s) and latency (about 12 µs) in OFED 1.1 are much better with SDP than with IPoIB [Zin07].

Measuring the time to establish a reliable connection via SDP and IPoIB, shows a performance issue of the GSI and QP approach. According to measurements in [Zin07] and [Ryl07], connecting a remote host via IPoIB is much faster (24 µs) than via SDP (353 µs). The reason is found in the OFED code. SDP uses the CM to connect two new QPs by a reliable connection (RC). Each QP is required to step through the states `init`[2], `rtr`[3], `rts`[4], and `rtu`[5]. Every step is an operation on the NIC after traversing a long path of code in OFED. Each of these operations costs about 100 µs. IPoIB uses QPs of the unreliable datagram type. This is a 1:1 mapping of the Internet protocol datagram service. The state transition is done only once during the initialisation of IPoIB. All IP datagrams (and thus transport layer connections) are multiplexed by this single QP.

[2]initialised
[3]ready to receive
[4]ready to send
[5]ready to use

Performance Details The behaviour of InfiniBand is described for the same reason that Ethernet was described in more detail. To transmit data, the following steps have to be executed after creation and initialisation of a QP (see Figure 2.5):

1. The software has to create a work request (WR). This can be a receive (host B), a send (host A), or remote direct memory access (RDMA) routines like put or get (host A) (numbered 1x in the figure). In case of RDMA, host B does not have to create a work request.
2. A context switch to kernel mode[6] has to be made and the software stack is traversed. The work request is submitted to the HCA (2).
3. The work request is processed by the HCA. It is able to directly access the local memory and transmits the data over the wire (3, 4).
4. The work request is complete and is moved to the completion queue (4a).
5. The remote HCA will process a previously and matching receive work request (if any) and is able to put the data to the given memory address.
6. The work request (if any) is complete and is moved to the completion queue (4b).
7. Hosts have to switch context to fetch the completion queue entry and possibly process the arrived data (5, 6). This is not necessary and possible at the receiver in case of RDMA.

Creation, initialisation, fetching, and cleanup of work requests has to be done by the host CPU. The management of queues and work requests and transport of data is done by the HCA. In InfiniBand it is simpler to transmit and receive data without copying from application buffer to intermediate buffers, since the HCA can directly access the RAM (indicated by the *data flow* in Figure 2.5). However, the memory has to be registered with the HCA in advance in order to be accessible. This operation is costly as we will see later.

Several versions of HCAs have their own memory. This is useful to prefetch and buffer data before or after transmission from and to the local RAM. It can also hold internal data like the queues and work requests for faster access of the HCA.

A single speed InfiniBand link achieves a throughput of 2.5 Gbit/s. Today's HCAs have 4 or 8 links and therefore can provide 10 Gbit/s and 20 Gbit/s. Similar to the PCI-bus, the protocol on the links uses 2 redundancy bits for each byte. Therefore the theoretical transfer rate drops to 1 GB/s on a 4x link.

[6]Since all APIs of InfiniBand exist in both the kernel and the user mode, the context switch can happen before or after the creation of the work request.

2.2.4 Quadrics

Quadrics (or better QsNet II) [PcFH+02] is a high speed interconnect that is specialised to build clusters for high performance computing. Four clusters in the Top500 [TOP08] list of November 2008 are equipped with Quadrics networks.

Latencies below 2.0 µs can be achieved using QsNet. This is the lowest latency among the presented network technologies. The achievable bandwidth (900 MB/s) is slightly lower than the bandwidth of InfiniBand with 4 links.

2.3 Summary

Clusters of computers/workstations are a popular and cost-effective alternative to massively parallel processors (MPP). Many kinds of applications are deployed on clusters today. Parallel applications and scalable and fault tolerant client-server applications are the most important applications and, therefore, are considered for this work.

Local area and system area networks are used to build clusters. Several different interconnect technologies exist with different approaches to support communication between hosts – from commodity off-the-shelf interconnects like (Gigabit-)Ethernet to specialised network hardware with support for remote direct memory access like InfiniBand. Some of the important technologies were presented here.

Chapter 3

Cluster Communication (Related Work)

This chapter gives an overview of previous and related research on communication, communication systems and especially on message passing. Some of the presented interfaces can exploit RMA and/or RDMA. Further programming interfaces are described and partially analysed for the large class of parallel applications.

3.1 Efficiency

In this thesis, the term *efficiency* will be used often. The term efficiency is used in different contexts in computer science. Some of these are:

- in conjunction with complexity of algorithms. This will apply to algorithms used inside a communication system.
- a metric for the scalability of a parallel program

More general and out of scope of computer science, the adjective *efficient* is paraphrased by *working well, quickly, and without waste*. In the context of this work, efficiency is related to the influence of the communication system on a running application. A communication system is efficient if it works well, quickly, and with no or small overhead according to the latter definition.

The complexity of algorithms will influence the amount of overhead. In this way, the definition from computer science overlaps with the more general definition.

The impact of the communication system on applications has multiple aspects. The most obvious aspect is the time spent for communication. Another aspect is issued by CPU and memory usage of the communication system. For example each memory region allocated for internal use by the communication system is unavailable to the application.

Another important issue is the interface to the communication system. It determines the semantics and limits the flexibility of usage. Thus, a single interface is not suitable for arbitrary applications. This is also stated in [Zit95]. The communication system is taken as a service that is suitable only for some applications or a special class of applications.

A less important fact in the context of this thesis is the efficiency of an API in terms of productivity. As an example, Cantonnet et. al [CYZEG04] analyse the productivity of *Unified Parallel C* (see Section 3.6.6) and compare it to the Message Passing Interface MPI-2 (see Section 3.4). The authors explain a *program complexity* and a *language complexity*. The program complexity is further divided into syntactic complexity, length, and conceptual/semantic complexity of a program. An example of conceptual complexity is additional communication

and synchronisation. The authors summarise this to ‘manual effort introduced by a given language’ [CYZEG04].

Summarising the above aspects, *efficiency* will mainly focus on the impact of an API on the performance of applications. This includes the impact of the API on the possiblities of an implementation of the communication system and its efficiency.

3.2 Interfaces for Communication

Before digging into the details of communication itself, some application interfaces are described. This is to better understand some of the examples given in Section 3.7. The interfaces are separated in two coarse classes. They are classified weakly by their common usage in the field of parallel applications. For example, the socket interface can be used to implement parallel applications, but since application programmers want to solve their problems with good algorithms, they prefer to use a more specialised and more portable interface for parallel applications like MPI. This allows the user (programmer) to focus on implementing the algorithm and the message exchange instead of dealing with host addresses, ports, or connections. This section describes some general interfaces used for rather low-level data exchange. The efficiency of some of the presented APIs is rated in terms of efficient execution and ease of use.

3.2.1 Sockets

The *Socket*s interface is one of the most popular interfaces to communicate between (remote) processes. It was developed at the University of California in Berkeley.

According to [MBKQ00], the interface design goals where transparency, efficiency, and compatibility. The communication should be transparent in the sense of local and remote processes. Remote and local are not distinguished by the interface. The interface has to be fast and with low overhead, otherwise it would not be used. The compatibility goal had to be fulfilled for the large number of *naive processes* in UNIX. A *naive process* is a process that performs its I/O via files or standard in- and output.

The result of the design is an interface that uses so-called *sockets* as an abstraction of communication endpoints. For compatibility to *naive processes* a socket is a kind of file descriptor. Data can be sent and received from a socket. It does not matter if the partner process is a local or a remote process.

The socket interface is the most common interface used for point-to-point communication.

3.2.1.1 Blocking and Non-Blocking Communication

The common communication calls of sockets are blocking. The calling application is blocked until the requested operation is complete. A blocking `send` call will block the caller until the data is transmitted. A blocking `recv` call is a simple way to wait for incoming data.

What can be done if multiple sockets are used to communicate? If more than one socket has to be observed, a common practise is the use of `select`, `poll`, `epoll` (Linux), or `kqueue` (FreeBSD, OpenBSD). `select` provides signal-based multiplexing of sockets. `poll` permanently consumes CPU cycles because of busy waiting. Two event-based approaches are mentioned here, since `epoll` and `kqueue` are known to be faster and more convenient alternatives to `select`.

Using these APIs, a non-blocking behaviour can only be achieved by using a dedicated thread or process. This thread can wait in a blocking way for communication operations to complete.

3.2.1.2 Asynchronous I/O

Asynchronous I/O is a way to allow non-blocking I/O calls from an application. Linux provides an asynchronous API for I/O [BC05]. The original version used kernel-level threads to ensure asynchronous behaviour. This has been shown to be inefficient. In today's implementation, kernel-level queues are used for Linux Asynchronous I/O (also known as *Linux AIO*).

The authors of [BJL$^+$06] investigate asynchronous I/O in conjunction with processor partitioning. Processor partitioning uses a dedicated processor to make progress on asynchronous operations. The impact of asynchronous I/O is only quantified in combination with processor partitioning in the paper. They implement an asynchronous interface called *Direct User Sockets Interface* (*DUSI*). This paper provides an extensive related work section mentioning many asynchronous I/O interfaces.

Asynchronous I/O allows the overlapping of computation and communication as long as there is a processor to make progress on pending asynchronous operations in the background. Furthermore, the initialisation of operations and the check for completion imply some overhead that has to be taken into account when relying on asynchronous I/O.

3.2.1.3 Efficiency of Sockets

Except for the address handling of remote nodes, the simple usage of sockets increases the efficiency for the programmer (less API overhead). Only a few API calls are necessary. Compared to directly programming a network protocol, the

code will be more portable. Sockets are more widely used because of this. Most client-server applications and *Remote Procedure Calls* (RPC) are build on top of sockets.

Modern libraries like MPICH NEMESIS or Open MPI (see Section 3.4.1) can be configured to run on top of sockets. However, the efficiency of kernel-based sockets is limited. Each communication requires a mode switch between user space and kernel. This takes additional time. The efficiency is further reduced by the processing of pending *interrupt service routines* (ISR) before switching back to user space. Several approaches exist to enhance the performance of sockets [BSWP02]. GAMMAsockets [SSP03] show enhanced socket performance by implementing a socket interface on top of the *lightweight protocol GAMMA* [Cia99] (Genoa Active Message MAchine).

3.2.2 VIA

The *Virtual Interface Architecture* (VIA) [CCC97] is an industry standard developed in 1997 by the Compaq Computer Corporation, the Intel Corporation, and the Microsoft Corporation. It is mentioned here, because the ideas of VIA were adopted into recent interfaces like InfiniBand Verbs (see below).

Communication endpoints are a well-known network abstraction in communication systems and their interfaces. The operating system multiplexes the endpoints to the network hardware. Using the operating system as a multiplexer implies many mode switches between user and kernel. Thus, VIA specifies communication endpoints as a complete *Virtual Interface* (VI). This allows for more bypassing the operating system.

Although the reduction of mode switches motivated the VIA interface in 1997, mode switches still have been an issue in 2002 according to [Ehl03].

Virtual Interfaces (VI) are presented to the application in user space (Figure 3.1). They consist of a send queue and a receive queue (*work queue*s). Each queue has assigned a *Doorbell* to signal events on the queue. The application posts work requests to the work queues to receive or send data. Descriptors specify the kind of request, the address of the application's buffer, and everything else that is required to process the request. A completed request is put into a completion queue that is also associated to the VI. All requests complete asynchronously.

VIA supports two data transfer models, send/receive and RDMA. The send/receive model requires the sender and the receiver to post descriptors to the corresponding work queues. Each send request requires a matching receive request on the destination VI. If no matching receives are posted before data arrives, an error will occur. The requests are ordered in a FIFO. No request can bypass the other inside of a work queue.

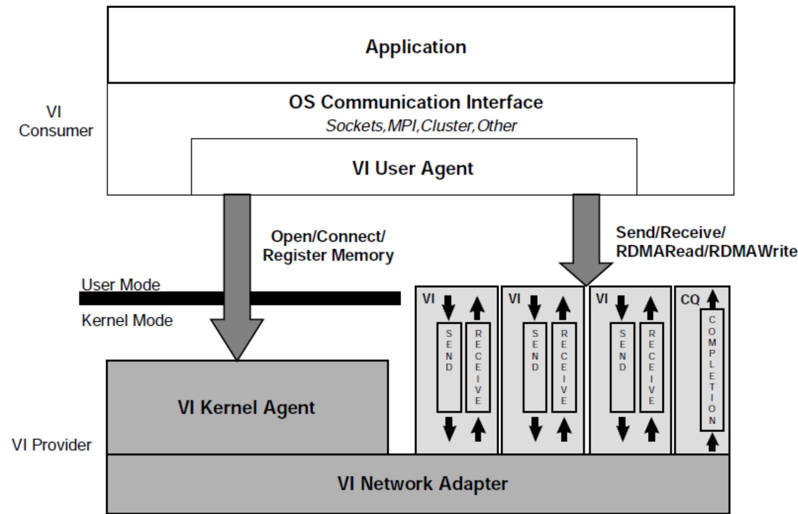

Figure 3.1: VI Architectural Model [CCC97]

The RDMA model specifies an RDMA Write and an RDMA Read operation. Both the source and the destination buffer are determined by the initiator of the operation. The remote memory has to be registered in advance and announced to the initiator. No posted requests on any remote queue are consumed on the remote node. The remote node will not get informed about completion of an RDMA operation. Therefore, extra synchronisation has to be performed if required.

The efficiency of VIA is improved by a kernel bypass. This means the user space application prepares a complete communication command and submits it directly to the driver. No further processing is required in a kernel-level protocol stack. This allows for a more light-weight communication compared to protocols implemented inside the kernel. However, the use of application-level memory is critical. Memory of the user space has to be registered to the network hardware. Depending on the hardware, this operation can be expensive. The additional overhead amortises only if registered buffers are reused or very large buffers are transmitted [SBB+07]. With frequently changing memory regions, the direct access to application buffers becomes inefficient.

It is a good approach to reduce mode-switches to improve the performance by creating user space virtual interfaces. The application can handle incoming and outgoing data on its own. Also the concept of work queues is successfully reused in the follow-up interface InfiniBand Verbs. Problems can occur from the notification and event mechanism. Since notifying and handling events can be expensive operations. This can be seen in the next section.

3.2.3 Verbs

InfiniBand *Verbs* [Inf02a] describe the functionality of an InfiniBand host channel adaptor. The implementation of Verbs specifies the API that is presented to the programmer. The general concepts are derived from VIA. Similar to VIs of VIA, the Verbs API provides *Queue Pair*s (QP) consisting of a send queue and a receive queue. A well known and used implementation of Verbs is contained in the Open Fabrics Enterprise Distribution (OFED) [Ope]. For the remainder of this work, the word *Verbs* defines the implementation of InfiniBand Verbs in the OFED stack.

The OFED implementation needs further improvement to achieve a good efficiency. For example, it is possible to achieve the low latencies promoted with InfiniBand only if established connections are used together with registered memory and RDMA write operations. Several protocols and implementations were analysed by the cluster computing group of the professorship for operating systems and distributed systems at the University of Potsdam. Results in [Ryl07, Zin07, Dav08] show that using InfiniBand and in particular the Verbs API achieves high performance only if these special conditions are met. As long as an application is able to work under these conditions, it can efficiently use InfiniBand Verbs.

Using other provided techniques like event-based signalling of incoming data via completion queues or polling instead, the latency increases dramatically. David Böhme [Dav08] measured a difference of 10 μs between polling and event-based receiving. Compared to the latency of an RDMA write, polling introduces another 10 μs to the latency. The latency of RDMA writes is about $4-5$ μs.

3.2.4 DAPL

The *Direct Access Programming Library* (*DAPL*) is an interface designed in 2007 by *Direct Access Transport* (*DAT*) Collaborative[1], an industry group formed to develop an independent interface for RDMA. Some of the about 40 members include AMD, Intel, IBM, Sun, Oracle, and Mellanox.

DAPL is available as a user space (uDAPL [DAT07b]) and a kernel space interface (kDAPL [DAT07a]). Version 2.0 was published in January 2007. DAPL specifies 'a singe set of ... APIs for *RDMA*-capable Transports' [DAT07b] to exploit RDMA capabilities of interfaces like InfiniBand Verbs or VIA.

The basic model of DAPL is simple (Figure 3.2). A *Direct Access Transport* (*DAT*) consumer contacts a (local) DAT provider. The provider processes the requested communication. Providers enable message oriented or RDMA read and write operations on top of reliable connections. Additional functions provide connection management, memory management, and synchronisation, respectively event handling.

[1]The first specification of uDAPL was ratified in 2002.

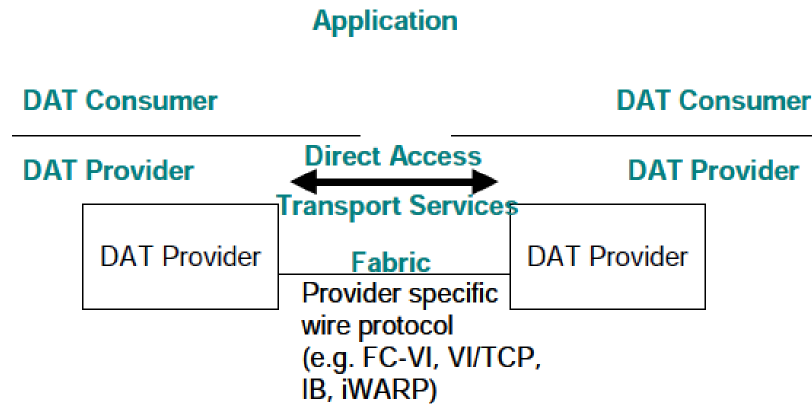

Figure 3.2: Direct Access Transport Framework [DAT07b]

The consumer-provider architecture fits into the model of transport services described by Zitterbart [Zit95]. The provider offers a transportation service to the consumer.

DAPL is not further analysed in this work since it is meant to be an abstraction layer for other RDMA-capable interfaces. Although this allows portable applications, an additional layer is likely to add overhead. For example the project *RDMA Enabled Apache* [DW08] also avoided this API because of slight performance drawbacks. Furthermore, it is out of focus because it is not intended for parallel applications directly.

3.2.5 Myrinet MX

Myrinet MX [Myr05] is a low-level message-passing library for Myrinet networks. The goal is to exploit the special features of Myrinet hardware. In particular, these features are a programmable NIC processor or a matching component (e. g. for MPI described later). The designers' goal is to 'provide exceptional performance for modern middleware interfaces such as MPI or VI' [Myr05]. The API itself is comparable to VIA. The programmer has to create communication endpoints and has to connect these endpoints to communicate between two processes. Routines for management of requests and synchronisation between processes round up the API.

3.2.6 Summary

The interfaces described above show a part of the variety of existing APIs. Starting from the simple Socket-API with its generic concept of abstract communication

endpoints that are mapped to underlying protocols, more complex and specialised APIs like VIA, MX, or InfiniBand Verbs offer a more efficient communication. However, they require more programming overhead. For example, registering memory is not of much interest if the Socket-API is used. Due to the variety of RDMA-APIs, some approaches like DAPL try to create an abstract and more portable interface to unify the use of RDMA. The presented interfaces are not designed to a specific class of application.

3.3 Parallel Programming

Large (scientific) applications often evolve over years. Many developers contribute to the code and improve the algorithms. While the software continuously changes in small steps, the hardware changes in rare and more drastic steps – whenever a cluster or new parallel computer is purchased. Thus, it is advantageous for an API if it can be continuously used and is independent of the hardware. The better portability ensures that an API will be used in ab application. Otherwise, large efforts have to be made to adopt the software to each new hardware.

Another goal of parallel programming APIs is to ease the inter-process communication of parallel applications. If a programming interface for parallel applications is not easy to use, application programmers spend a disproportionate amount of time using the API rather than the problem that they are trying to solve.

According to the survey by Kessler and Keller [KK07], parallel execution and programming relies on two major inter-process communication models: message passing and shared memory or shared address space.

3.3.1 Shared Memory

Shared memory can occur in two ways. The reason for this is the difference between processes and threads. Threads share the address space. This means no special steps have to be taken to access the memory of another thread. In general, processes do not share the address space. Therefore, a special memory area has to be allocated to allow direct access to addresses of another process.

Since inter-process communication via shared memory is becoming more and more important for cluster computing due to the development of multi- and many-core CPUs, shared memory is discussed in more detail below.

3.3.1.1 Shared Address Space

If shared address space is available, then one part of the application can directly access and modify the data of another part of the application. As long as the

accessed memory is disjoint, synchronisation is not required.

If the memory is not disjoint, access has to be synchronised to keep consistent data. The usual synchronisation that is used are semaphores or monitors. *Mutual exclusion* is required as the basic synchronisation pattern. The semantics of *producer/consumer* and *readers and writers* can be applied if necessary.

3.3.1.2 Shared Memory

If two processes in separate address spaces have to exchange data and shared memory is available, they can allocate their data in this memory area. In this case, the access and modification of data is similar to the shared address space method.

If the data does not reside in shared memory areas (e. g. if the shared memory is provided by a library), the data has to be copied or mapped into and out of the area to exchange data. In this case, the shared memory area is used similar to a buffer of the transport system.

3.3.2 Message Passing

There are two available kinds of *message passing* between two processes: two-sided communication (point-to-point communication) and one-sided communication (remote memory access). Furthermore, communication can occur between two or more processes. The details of 1-to-n or n-to-m communication are out of the scope of this work. Unless stated otherwise, all communication is 1-to-1.

3.3.2.1 Two-Sided

Point-to-point communication or *two-sided* communication is a well-known way to exchange data between processes. One of the processes, the source, has to call a routine to submit data. The other process, the destination, has to initiate a receiving function. The source determines the size and the content of the message including the identifier of the destination process. The destination determines the memory address of the destination buffer and the number of bytes to receive to that location. Usually, the number of bytes sent and received have to match if the communication is based on messages. For message passing, the number of `send` and `receive` calls also have to match.

The key property of two-sided communication is that both involved processes have to participate in the actual data transfer.

3.3.2.2 One-Sided

One-sided communication or *remote memory access* (*RMA*) enables *direct access* to the address space of another process through a network. Only one process has to call a communication routine (e. g. `put` or `get`) to determine the parameters of a data transfer. This source process (source of the operation, not necessarily the source of data) specifies the source and destination buffer as well as the size of the message and the identifier of the remote process.

The key feature of *one-sided* communication is that only one process has to actively participate in the actual data transfer. This does not include synchronisation as stated in the next section. Also the number of RMA operations is not predefined.

Using the RMA `get` operation, the definitions of data source and destination are interchanged compared to `put` or `send` calls. Since this makes the description more complex, the `get` operation is mostly omitted in this document. Additionally, `get` operations have shown to be slower due to their request/reply-based behaviour [BU03]. Therefore, the latency of a `get` operation is expected to be at least twice the latency of the network. The `get` operation can be implemented as a (remotely triggered) `put` operation at the destination process (the data source in this case). This solution performs better for some transmission methods (e. g. see [VBR+04] where reading from PCI-bus is significantly slower than writing). However, the semantics of this is *not* the same since the data source is actively involved in the communication.

3.3.2.3 Hybrid

Today's hardware often consists of multi- or many-core computation nodes interconnected by high speed networks. These trends in hardware have imposed so-called *hybrid* parallel programming models where message passing is used for inter-node communication and shared memory paradigms are applied to intra-node communication (e. g. a mix of MPI and OpenMP).

The hybrid approach has benefits and drawbacks. For example, the creation of threads introduces some overhead. Using a single process per SMP node to communicate via MPI reduces the memory usage of the communication system and the overall number of MPI processes.

3.4 Message Passing Interface

The *Message Passing Interface* (*MPI*) [MPI95] is developed as a successor and competition to the *Parallel Virtual Machine* (PVM) in 1994. It is designed to ease the development of parallel applications primarily following the SIMD-Model.

The processes of a parallel application are grouped together in a so-called *process group*. Each process is assigned a unique *rank* in the group (an integer value starting from 0). This simplifies the addressing of the processes and is independent of the transport protocol. Network addresses, ports, or Queue Pairs are not required. The rank is sufficient.

PVM is still used, but it has no interface for one-sided communication. Therefore, it is not considered within this work.

```
#include "mpi.h"

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myId);

    MPI_Barrier(MPI_COMM_WORLD);
    if ((myId%2) == 0) {

        /* processes with even rank */
        MPI_Send(buffer, length, ..., MPI_COMM_WORLD);
        MPI_Recv(buffer, length, ..., MPI_COMM_WORLD, ...);

    } else {

        /* processes with odd rank */
        MPI_Recv(buffer, length, ..., MPI_COMM_WORLD, ...);
        MPI_Send(buffer, length, ..., MPI_COMM_WORLD);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
}
```

Listing 3.1: Simple MPI Ping-Pong Example.

An example MPI program is shown in Listing 3.1. First, `MPI_Init` is called to initialise the process group and prepare communication. Communication is performed by calling primitives for point-to-point communication (`MPI_Send`, `MPI_Recv`) or collective operations (`MPI_Barrier`). MPI offers primitives for blocking, non-blocking, synchronous, or buffered point-to-point communica-

```
     Process A:                                    Process B:
 MPI_Win_fence()                                MPI_Win_fence()
     ...                                            ...
         MPI_Put() / MPI_Get() / MPI_Accumulate()
     ...                                            ...
 MPI_Win_fence()                                MPI_Win_fence()
```

Figure 3.3: Fence synchronisation mechanism of MPI-2 [MPI97].

```
     Process A:                                    Process B:
                                                 MPI_Win_post()
 MPI_Win_start()
    ...
 MPI_Put() / MPI_Get() / MPI_Accumulate()     ...
    ...
 MPI_Win_complete()
                                                 MPI_Win_wait()
```

Figure 3.4: Post-Start-Complete-Wait synchronisation of MPI-2 [MPI97].

tion. `MPI_Finalize` closes the process group and performs does the cleanup of communication.

MPI-2 [MPI97] is an extension to the MPI standard published in 1997. It includes specifications for *dynamic process management*, new routines for *parallel I/O*, and functions to exploit *one-sided* communication. In the following, the interface to *one-sided* communication is discussed.

One-sided communication was introduced to the MPI-API to allow an application to make use of *remote memory access* (*RMA*). The design of the one-sided communication API is derived from programming models of shared-memory architectures. The semantics are different (e. g. the progress rules). Thus, these approaches can hardly be compared to MPI-2. Approaches like SHMEM from Cray [Shm01, Sil08] rely on a fast synchronisation mechanism. A read or write operation to data structures of another process can be synchronised (e. g. as a kind of a barrier) to assure that the operation is complete for all participating processes. The absence of fast synchronisation mechanisms in cluster environments[2] is one reason for a different semantics of MPI-2's one-sided communication.

The MPI-2 standard specifies one-sided communication as non-blocking com-

[2]cluster in the sense of Beowulf clusters or network of workstations without special networks for synchronisation

munication with explicit synchronisation. The synchronisation has to be performed via special and additional messages. This implies communication overhead. Synchronising multiple communication operations with a single synchronisation message is an advantage [SSO$^+$95]. MPI-2 one-sided communication provides two variants of synchronisation (see Figure 3.3 and 3.4): two kinds of *active target synchronisation* where the target process is involved in the synchronisation. A *passive target synchronisation* is also available where the target process is completely uninvolved in synchronisation. The passive target has just to announce a buffer and withdraw the announcement to open and close the epoch.

Beside the implementations described below, several approaches exist to make MPI-2 – and especially MPI-one-sided communication – aware of specific features of hardware. In [TRH00], Träff, Ritzdorf, and Hempel present an implementation for NEC SX machines. Another example is MVAPICH [MVA07] for InfiniBand described below.

3.4.1 MPICH2

MPICH2 is one of the most common and freely available implementations of the MPI standard developed at Argonne National Labs. The central design component is an *Abstract Device Interface* (*ADI*). The application uses the upper layer which provides the MPI-2 standard-compliant interface. The upper layer translates the API calls into calls to the ADI. For example, this includes special algorithms to map collective operations to point-to-point communication. The ADI finally maps the upper layer calls onto the specific hardware features.

The ADI abstracts the network and protocols. To support a new network or protocol, the ADI has to be implemented. An overview of the general architecture is shown in Figure 3.5. All boxes enclosed by the ADI box represent implementations of the ADI. The *channel device* provides a simplified interface to ease the implementation of the ADI. Therefore, many network modules are implementations of the channel device.

MPICH2 [MPI07] is a redesigned and full featured MPI-2 implementation derived from MPICH. Like its predecessor MPICH, it is still build around the *Abstract Device Interface* (*ADI*) [GL94, GLDS96]. The ADI has been extended to exploit RDMA features of underlying hardware to support MPI-2 one-sided communication.

Since version 1.0.4, a new architecture named Nemesis [BMG06a, BMG06b] is introduced. The goal of Nemesis is to benefit from shared-memory-based interprocess communication exposed by multi- and many-core processors.

Our own measurements could not confirm the advantage of Nemesis over the previous design using a point-to-point ping-pong benchmark (see Appendix B.1.1) on Uranus-hardware (see Appendix A.1). This advantage is promoted in [BMG06b].

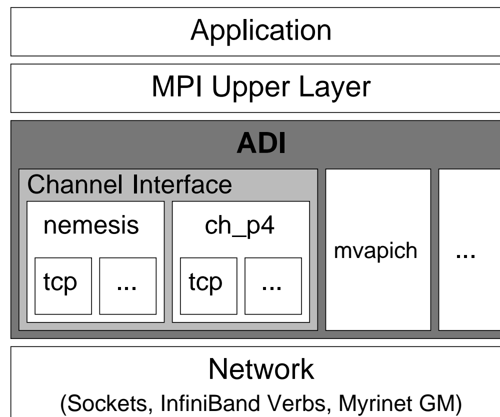

Figure 3.5: Architecture of MPICH and MPICH2

An implementation of MPICH2-nemesis over IPv6 and IPv6-enabled Open MPI is presented in [KKF+08]. An IPv6-enabled version of MPICH is described in [SS05]. Both implementations could be done straightforward due to the modular design of MPICH. However, several issues with the address handling by the runtime environment of MPICH had to be solved.

3.4.2 MVAPICH

MVAPICH [MVA07] is derived from *MPICH2* and is one of the free implementations of *MPI-2* that directly uses *RDMA*-features of *InfiniBand*. It is an implementation of the *ADI* [LWP04]. MVAPICH uses RDMA-based communication of InfiniBand for all kinds of remote process interaction. The reason is that the performance will suffer if any different mechanism provided by *Verbs* is used (see Section 3.2.3).

Due to its good performance MVAPICH will be used as a reference for measurements with the self designed one-sided communication interface presented in this thesis in Chapter 5. MVAPICH was compared to *Open MPI* and a vendor MPI of HP in a student thesis [EA07] at the University of Potsdam. Parts of the results were published in [SS07a].

3.4.3 Open MPI

Open MPI [GFB+04, OMP07] is another free implementation of the *MPI-2* standard. It is the successor of several MPI-related projects. According to their own presentation, the initial Open MPI was the result of a merge of FT-MPI [FD00], LA-MPI [ADD+04], LAM [BDV94, SL03], and PACX [GRBK98]. The basic design of Open MPI is different from MPICH. Everything is build around three

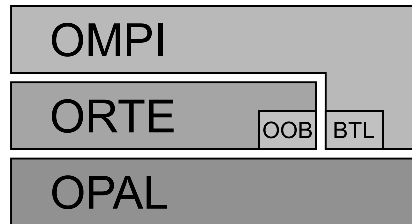

Figure 3.6: Abstraction Layers of Open MPI.

abstraction layers. Figure 3.6 shows the abstraction layers *Open Portable Access Layer* (OPAL), *Open Run-Time Environment* (ORTE), and *Open MPI* functionality (OMPI). Open MPI offers the possibility to use multiple networks or protocols in parallel. This can be used either to communicate in heterogeneous environments or to increase the available bandwidth between the computing nodes.

As mentioned in Section 3.4.1, Kauhaus et. al. implemented an IPv6-enabled Open MPI. This work is presented in [KKPF07, KKF+08].

3.4.4 MPICH-G2

If a single cluster is insufficient for a large scale application, multiple clusters are combined to work on a problem. This is the area of multi-cluster environments and grid computing. To run an application across multiple clusters, MPICH-G2 was developed [KTF02] in the context of the Globus Toolkit 2. The main extension made to MPICH is the addressing. It was extended by topology information. Each process is assigned a so-called *colour*. Prior to communication, the colour of the destination process is compared to the local process to choose the best protocol to communicate. For example destinations inside the local cluster can communicate by using a special vendor-MPI. The most common case will be a TCP/IP-based communication method to contact processes on remote clusters.

3.4.5 Efficiency of MPI

The API of MPI can be described as efficient from the point of view of a programmer. MPI is easy to use (compared to socket based message passing), portable, and specialised to the requirements of parallel application. According to the definition of *efficiency* in 3.1, a few efforts of programming have to be made to write parallel programs. On the one hand, the programmer has the full control over the inter-process communication. On the other hand, the programmer is responsible for all communication. The efforts strongly depend on the algorithm and there is some criticism to MPI-2 [Geo06].

A specific evaluation will be done in the corresponding context. For example MPI-one-sided communication exploits the benefits of explicit synchronisation that are theoretically analysed in [SSO+95]. The authors propose decoupling of synchronisation and data transfer to improve the performance of message passing systems. The achievable efficiency of implementations of the one-sided communication interface of MPI-2 is addressed in Chapter 4. Performance measurements of MPICH2 are presented in Chapter 5.

The highest-level approach to optimise the behaviour of an implementation is to modify the application itself. This approach violates the goal of MPI to allow portable applications. The modifications can counteract when running on a different implementation. Saif and Parashar [SP04] describe such an approach to avoid the blocking behaviour of non-blocking large message transfers in MPICH 1.2.5 and IBM MPI. These optimisations are based on implementation issues that expose the non-compliance to the standard. Both implementations obviously block `MPI_Isend`-calls of large messages until the corresponding `MPI_Wait`-call of the receiver occurs. This behaviour was not reproducible with version MPICH 1.2.7.

A more portable approach is to exploit the features of the underlying hardware to optimise the performance of an application (i. e. improve the efficiency of MPI). This requires the API to be efficiently implementable.

Research has been and is done on the optimisation of MPI. On the one hand, it concerns general approaches like the management of connections [YGP06] or the use of eager- and rendezvous protocols to transmit small and large messages efficiently. On the other hands, special features of the underlying network are exploited [CC99, Cia, AAC+04]

3.5 OpenMP

OpenMP [Ope05] supports the data parallel model [WA99] to allow the parallelising of algorithms. The programmer is responsible for just some pre-compiler statements to tell the compiler how to parallelise a part of the code. In this way, the parallelising is finally performed by the compiler. The programmer specifies how the compiler should handle the data.

OpenMP provides a way to implement parallel algorithms. Additionally, it offers a low-effort method for existing sequential code to exploit multi-processor machines. Pre-compiler statements are ignored by non-OpenMP compilers. Thus, in most cases, the code will compile and run even if the compiler is not aware of OpenMP.

Due to the increased availability of multi-core processors, OpenMP becomes more and more attractive to programmers. For example *hybrid* programs are an interesting approach for clusters of multi-core machines. Hybrid programs make

use of OpenMP for local IPC and employ MPI to communicate between cluster nodes.

OpenMP is efficient in shared memory architectures due to its programming model that is based on threads with a global address space. OpenMP will suffer from the overhead required for synchronisation in distributed memory environments.

3.6 Further APIs

There are many APIs available to create parallel programs. In this section some of these APIs will be described to some extend.

3.6.1 ARMCI

The Aggregate Remote Memory Copy Interface (*ARMCI*) is designed to support global address space for distributed memory using one-sided communication. ARMCI was presented by Nieplocha and Carpenter in 1999 [NC99]. The authors describe ARMCI as an interface to exploit high performance remote memory access.

According to [NTKP06], the design of ARMCI has several aspects in the focus:

- First, the *communication progress* rules have to be simple. All operations should complete independently from the actions of the remote process. A dependency would reduce the responsiveness and increases latencies. In addition, dependencies can result in communication deadlocks if the programmer is not aware of the progress rules.

- The second aspect is the communication and computation *overlap*. The independence of remote actions allows for remote overlap of computation and communication, while local overlap is provided by returning from communication calls as soon as possible. Let the underlying network (hardware) complete the communication.

- The third aspect includes non-contiguous data transfers. There are a number of ways to perform non-contiguous transfers. Using a memory copy to gather and scatter the data, or using multiple communication calls to transfer each element separately. While the first is not suitable for RMA communication and involves the remote host, the second requires multiple start-up costs. Non-contiguous data transfers are out of the scope of of this work.

- To avoid problems with unregistered memory regions, ARMCI forces the application to use provided memory allocation functions. This allows the program to use the fastest access and avoids additional checking of memory status (registered or unregistered).

The synchronisation of data transfers is explicitly done by wait and other synchronisation calls. ARMCI distinguishes between local and remote completion by providing different synchronisation calls for this purpose.

A unique feature of ARMCI are so-called *aggregated handles*. These can be used to combine multiple handles of communication calls to a single handle. This handle can be used to check the completion state of all aggregated operations for example. With this way of synchronisation, ARMCI does not use the concept of epochs like MPI-2.

The article cited above contains many interesting approaches to optimise communication. Depending on the costs of memory registration, a local memory copy is used to transfer small messages. The data is copied to a pre-registered memory area. Transferring larger messages in chunks copied to the pre-registered area can be a benefit if memory registration costs are high. To avoid waiting on the intermediate memory, multiple pre-registered buffers can be employed and used alternating. In this way, the communication is pipelined.

The authors present an interesting alternative to implement a `get`-operation if a `get` is inefficiently supported by the underlying network. They propose to send a request to trigger a more efficient remote put or write operation. The result is a faster communication that conflicts with the idea of RMA: the remote host should not involved in the communication. This is still true at the application level. Therefore, it can reduce the overlapping effect that may be intended by the programmer.

In Chapter 5, we will see that this API is very close to NEON – a new one-sided communication interface presented in this thesis.

3.6.2 LAPI

LAPI is an interface designed by IBM [SNM+98] for use with the RS/6000 SP that came out in 1995. The API provides communication primitives for one-sided communication. The synchronisation can be performed by MPI-like calls (e. g. fence).

LAPI offers an additional way to signal completion of messages. It uses counters at the origin and at the target process to check and signal for completion of data transfers. The counters can be specified by a parameter to the `put` call. This allows the target process to check if a data transfer is complete without waiting for a specific synchronisation message required by the MPI-2 interface.

Since counters rely on a kind of active message at the target, they cannot be considered as fully one-sided. The target is involved in the communication to increase the target counter.

3.6.3 SHMEM

SHMEM [Sil08, Shm01] is a widely known communication interface designed by Cray for its shared memory machines. It provides one-sided communication operations and synchronises via barrier-like operations.

3.6.4 GASNet

GASNet enables the illusion of a global address space regardless of hardware supported shared memory. This allows running of Global Address Space Languages over distributed memory machines like clusters.

3.6.5 Global Address Space Languages

There are several new languages developed and still under development. These languages have in common that they rely on a global address space to exploit parallel execution [BD03].

3.6.6 Unified Parallel C

Unified Parallel C (*UPC*) [UPC05] is an example for so-called *global address space language*s (GAS). It is developed at the University of California Berkeley by the group of Katherine Yelick. The current 1.2 specification is dated 31st May 2005.

UPC is an extension of ANSI C to create parallel programs. The parallel execution is intended to work with threads on shared-memory-based architectures. In this way, it is similar to OpenMP (Section 3.5) except that OpenMP is introduced as pre-compiler statements. Similar to High Performance Fortran (HPF), UPC can express affinity between data and thread to exploit data locality.

Analyses of the application and runtime efficiency can be found in [EGC01, EGC02]. A productivity analysis of UPC is presented in [CYZEG04]. UPC is compared to MPI by number of lines and characters to parallelise the kernels of the NAS Parallel Benchmarks [BBB+91, NAS07]. The authors conclude that UPC requires less 'manual efforts' [CYZEG04] to write parallel algorithms than MPI. Since UPC requires a global address space, a special communication system is required to provide global address space on clusters with distributed memory.

The communication interface *GASNet* [GAS06] provides global address space in a network independent way.

3.6.6.1 X10

This Java-like parallel language designed by IBM in 2006 tries to enable a programmer to exploit parallelism [SN08]. As described for *UPC*, X10 also includes expressions for data locality and parallel execution on machines with non-uniform memory access.

3.6.6.2 Chapel

Chapel [Inc08, CCZ07] is developed by Cray. According to the project homepage, the development of Chapel is still in progress. Chapel is intended as an approach to improve the productivity of parallel programming by including the expression of parallelism into the language itself. Thus, it has similar intentions to UPC or X10.

3.6.6.3 Fortress

Fortress [ACH+08] is another example of a *global address space language*. This language is derived from Fortran and developed by Sun Microsystems. The current specification is dated March 2008. Fortress is intended to ease the programming of math. A compiler could also generate LaTeX code to create a document. The language assumes parallel execution by default. Programmers explicitly have to specify sequential code.

3.7 Design Aspects of Efficient Communication

This section will give an overview over existing research on design aspects of *communication system*s for efficient data transport. Important aspects like *buffer management*, efficient transport layers, the impact of *offloading* and *overlapping*, and the issues and benefits of the combination of aspects is the focus of this section.

3.7.1 Buffer Management

Efficient use of internal buffers[3], bypassing internal buffers, or handling application buffers, reduction of the memory footprint of a communication library are

[3] buffers inside the communication system

important aspects of buffer management.

3.7.1.1 Zero Copy

Each memory copy introduces additional overhead to the transmission of data. The impact of additional intermediate copies of data is analysed in [Cia99, VBR+04]. Kurmann, Rauch and Stricker [KRS01] present an approach to use speculative techniques to eliminate copies from the TCP/IP stack. Chu [Chu96] proposes a way to implement *zero-copy* TCP/IP in the OS *Solaris*.

The overhead of a memory copy is hard to evaluate. One can measure the time or the CPU cycles required to copy data, but the impact on performance of an application can not be derived from this measurement only. The cache will become polluted by a copy (*cache pollution*). And this can result in unpredictable decrease of application performance [JCB96, Bru99, VBR+04].

3.7.1.2 Memory Registration

Modern network hardware (e. g. InfiniBand, Quadrics) supports *remote direct memory access* (*RDMA*) to the application buffers . While this provides hardware-based zero-copy, it includes some uncomfortable side issues. Since two components of the computer concurrently access the main memory, additional synchronisation and access mechanisms have to be implemented.

Some hardware offers NIC-based translation look-aside buffers (TLB) to access pages in memory (e. g. Quadrics). This allows the hardware to handle/access pages that are swapped out. This requires the operating system to be aware of a second TLB in the system. Special hardware drivers have to be used.

Hardware without a separate TLB has to rely on the application or the communication system software to prevent memory from being swapped out. The memory has to be registered. The registration can be an expensive operation. An analysis of this process in the Mellanox InfiniBand software stack is presented in [MRB+06]. According to the measurements, the registration of a 2 kB buffer takes about 1.8 ms. This can make a memory copy more efficient than insisting on zero-copy via direct access. Optimised versions of *memory registration* use a registration cache to avoid registering of previously registered pages [TOHI98, BB03, WW05].

3.7.1.3 Minimal Copy

If the underlying hardware and/or the operating system does not allow for zero-copy protocols, so-called *minimal copy* protocols can be used.

Additional copies can even improve performance if the overhead is too high for managing zero-copy. This depends on the hardware features and the application. For example, when using asynchronous communication in MPI it can make sense to send small messages immediately even if the remote application did not provide the corresponding destination buffer. The message will be stored in an intermediate buffer on the remote side – the so-called *eager buffer*. If the receiver becomes ready, only a fast local copy is required to complete the transfer.

3.7.1.4 Early Sender Problem

In this thesis, the term *early sender problem* describes the general effect of early called operations to transmit data without the receiver to be ready. This issue is known from message passing. It is also called *late receiver problem*. A distinction between both names is necessary if load imbalances and delays have to be discovered in the application (see SCALASCA [Jül08]). The point of view of the communication system is the same in both cases since both names describe the fact that the receiver is not prepared. Therefore, only the *early sender problem* is discussed.

The early sender problem is a major issue of buffer management to be solved by a *communication system*. If a sender is ready to send the data before the receiver has prepared the destination buffer, the communication system has to handle this situation properly and efficiently.

- The communication system can block the sender until the receiver is ready.
- If asynchronous communication is requested by the application, the communication system can defer the transmission until the receiver signals the availability of the buffer. In this case, the application will not be blocked until the *synchronisation point* (e. g. a call of `MPI_Wait`).
- The communication system can copy the data to a local buffer. This buffer is transmitted if the receiver is ready.
- The communication system can send the data eagerly to the receiver. The receiver has to manage the early arrival of data. These messages are called *unexpected messages*.

3.7.1.5 Expected and Unexpected Messages

The distinction between *expected* and *unexpected messages* has been made to describe the early sender problem more generic. A message is called expected, if the receiver is prepared and the destination buffers are known to the communication system. Expected messages can immediately be delivered to the destination buffer. This is the best case for the communication system. Additional blocking

of an application or buffering data is not required. It should be optimised as a so-called *fast path* for performance reasons.

MPI implementations handle unexpected messages [MPI07, OMP07] by pre-allocated buffers for short messages in order to buffer early arriving data at the receiver. A threshold determines the difference between short and long messages. This threshold is used to switch between *eager mode* or *rendezvous mode* to efficiently handle short and long messages.

Whether a message is considered to be large, strongly depends on the underlying hardware and protocols. Assuming an *MTU* of 32 Bytes (the MTU on IBM BlueGene/P), a message size of 1000 Bytes can already be considered as a large message. While 1000 Bytes will usually be considered as small if the MTU is 1500 Bytes like in Ethernet networks.

3.7.1.6 Eager Mode

In *eager mode*, the sender relies on a sufficient number and size of intermediate buffers (*eager buffer*s) at the receiver. An early sender will not be blocked due to an unprepared receiver and data is immediately (eagerly) transmitted. This allows the sender to speed up transfers of short messages without waiting for the receiver to be ready. It also allows the receiver to omit potentially required memory registration. However, it forces an additional memory copy at the receiver and allocates memory that is no longer available to applications.

The number of buffers is limited. The limit becomes a crucial parameter in applications that run on a large number of processes. If a large number of processes send eager messages to the same process, the number of eager buffers should be large enough to be prepared for at least one message from each communication partner. Otherwise, even the first message of some processes is rejected and has to be resent later. This introduces further overhead to communication. Scaling the number of eager buffers proportional to the number of processes will reduce the amount of memory available to the application. It reduces the efficiency of memory usage.

Another solution to this problem is not to limit the number of buffers but allocate memory on demand (e. g. the MPICH implementation). If this is combined with *memory pooling* (also described as segregated free lists [WJNB95]) the average overhead of allocating memory on demand can be reduced. Due to memory limitation, this solution is not always applicable efficiently.

MPICH uses unexpected message lists to manage eager messages. The lists are extended dynamically without a fixed limit. The threshold is configurable. The default depends on the network module used. It is set to 128 kB for non-local communication over Nemesis. For the *shm* module it is limited to about 10000 Bytes.

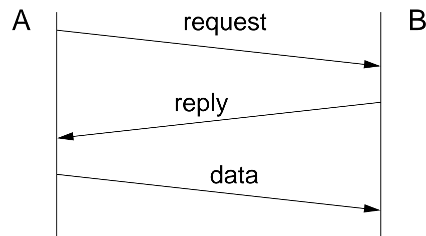

Figure 3.7: Basic rendezvous transmission scheme.

In summary, tuning is required to determine the threshold. When determining the threshold, the *Maximum Transfer Unit* (*MTU*), the available memory per node, and the number of processes should be taken into account. The MTU determines the number of packets to send. The available memory limits the size and the amount of intermediate buffers. The number of processes introduces an upper limit to the number of messages that can concurrently arrive at a single process.

3.7.1.7 Rendezvous Mode

The *rendezvous mode* is used to transfer large sized messages Therefore, it is also called large message transfer (*LMT*). Usually, this is transparently performed by the communication system. An efficient API should not expose this to the programmer.

Figure 3.7 shows the steps of the basic *rendezvous protocol*. The sending component *A* submits only the meta-data to the receiving component *B* and waits for *B* to signal the availability of a sufficient destination buffer. This special protocol to transfer large messages is used to avoid memory copies of large amounts of data. Furthermore, large messages are a candidate for pipelining of transfers (see next section). Therefore, the transmission of large messages can be optimised to exploit special features of the underlying network or hide memory registration costs [BU03].

3.7.2 Overlap

Along the transmission path of data, several independent processors may participate in the communication. The result is a communication *pipeline*. Figure 3.8 shows an example consisting of a sending and a receiving application and network interface card (NIC). The first step of this 4-step pipeline is an interaction between the application and the NIC. The processor running the application has to contact the NIC to either transmit the data or initiate a data transfer controlled by the NIC. A similar procedure will happen at each subsequent step of the pipeline.

Pipelining is a way to process some work in parallel using multiple processors. Pipelines are suitable if the same or similar work has to be done several times and can be split into subsequent steps. This is the case with communication. The sender has to initiate the transfer. The protocol stack has to process the data and prepends some headers. The network interface cards have to transmit the data. The receiver has to process the data in a similar but reverse order of the steps.

Pipelines are only efficient in parallel applications if each step costs the same amount of time. Otherwise, some of the processors have to wait and are not efficiently used. The same problem exists for communication. There are faster and slower steps in the pipeline. Additionally, the fact of being slow or fast depends on various parameters (e. g. the message size, the number of messages, the number of communication partners).

The message has to be split into several small messages (fragments) to use one effect of a communication pipeline for large messages. The fragments are subsequently processed and submitted through the pipeline. In Figure 3.9, the reduction of transmission time can be seen schematically for a 4-step pipeline with equally fast steps.

The authors of [WKM$^+$98] analyse the impact of fragments on the overall transmission time. Additionally, they optimise the fragment size in case of different speeds at different steps of the pipeline. The basic ideas of the pipeline model can be adopted to communication since multiple independent processors are available in many networks.

A complete implementation of the optimal solution is hardly possible. The prerequisite of the optimal solution is complete knowledge of performance characteristics of each pipeline step. According to [WKM$^+$98], the characteristics are determined by two parameters: the *per byte overhead* and the *initial overhead* (depicted as small black bars in the figures) to transmit a zero byte fragment. The latter parameter is generally called *latency*. Both overheads are expressed in units of time. Since these parameters depend on the message size, the characteristics have to be measured for each message size.

Multiple processors in the communication path enable another effect of pipelining. This is known as overlapping of communication and computation (short *overlap*). Brightwell et. al. [BRU05] discuss the impact of overlap on the application performance. They conclude that overlap can improve the performance

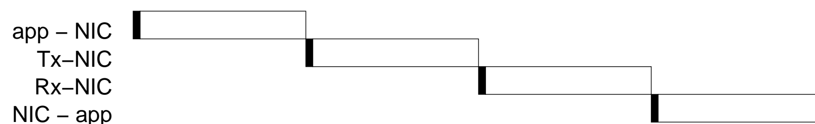

Figure 3.8: Communication pipeline example.

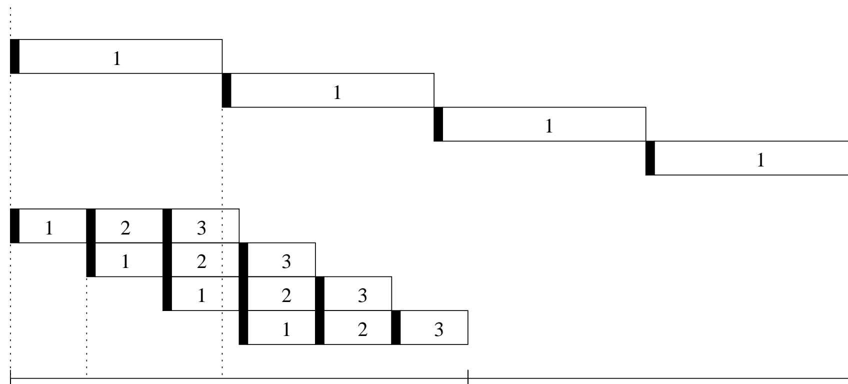

Figure 3.9: Effect of sending fragments through a pipeline.

if the application is able to make use of non-blocking communication. Further improvements are possible if overlap is combined with offload and/or independent progress (see the corresponding sections below).

As long as the implemented algorithm is suitable, the application can continue with the computation immediately after initiating the communication. This reduces the visible communication time (*latency hiding*). Knowing the large gap between processor speeds and communication latencies of modern hardware, this approach will continue to play an important role for efficient communication (especially for non-blocking communication which is intended by MPI-2 one-sided communication). This is also stated in [NTKP06], where this trend is also prognosticated for the future.

Initialising the communication includes all work to allow the succeeding processor in the pipeline to continue with communication. Usually, this work includes provision of information about the involved application buffers and/or possible data copies to special memory areas or NIC memory.

Doerfler and Brightwell [DB06] present a method to measure application availability in an overlap test. The measurements show that InfiniBand is bad for large messages in this context, while Myrinet and Quadrics perform well. This picture is reversed for small messages (80% vs. 90% availability). Unfortunately, each network is tested with a vendor-specific MPI-implementation. This can be a major source of different behaviour and is not further explained in the paper! Therefore, the results can not be used to compare the networks. In conjunction with non-blocking communication, the availability test is interesting for the studies in this work.

Some options to implement OSC in Open MPI are analysed in [BSL07]. The authors recommend to make use of RMA-capable hardware to improve the performance. In this case, the API can be directly implemented as a simple translation to

the API of the hardware. Especially, this is recommended in terms of overlapping communication and computation. When implementing the explicit synchronisation, the authors describe a problem in conjunction with Open MPI's capability to use multiple networks simultaneously. In this case, an explicit synchronisation message could pass the data messages. Thus, synchronisation messages have to be deferred and will be unblocked after the last data message is sent. Since a last message cannot be specified by the MPI-2 API, the synchronisation message is deferred to the synchronisation call (`MPI_Win_complete`).

Applying the pipeline model offers a way to improve the performance of applications by enabling overlap. However, the effect of overlap strongly depends on exact knowledge of the introduced overhead and which component/processor has to process this overhead. The authors of [KKZL03] have experienced a performance degradation in conjunction with overlap. The receiving process is interrupted by the operating system in an adverse way. The performance was improved by using blocking instead of non-blocking receives. They analysed several performance issues in a molecular dynamics application (NAMD) [PBW+05] on up to 3000 CPUs. Unfortunately, the improvement of using blocking receives is not explicitly shown. They improved the original version by several modification inside the Quadrics' communication library. Even though the effect is not quantified, one can derive that overlap is not *a priori* a benefit.

3.7.3 Offload

Modern network interfaces allow the processing of parts of the protocol stack on the NIC. This kind of processing is called *offloading*. There is a common assumption that protocol offload should improve the performance of applications because the network hardware is designed to process network traffic. Already in 1996, Hennessey and Patterson [HP96] pointed out that NIC processors do not *a priori* speed up communication. 'Fallacy: Adding a processor to the network interface card improves performance' [HP96](page 623). Often, NIC processors are much slower than the host CPU. Thus, protocol processing on the NIC can be slower and performance is lost.

In [BRU05, BU04], the benefit of offload is analysed using the NAS Benchmark Suite [BBB+91, NAS07]. The hardware (Quadrics, ASCI Red) is able to independently complete pending `send` operations. This allows for good overlap of computation and communication. The application can continue to work and more CPU cycles are left to the application. An additional MPI-tag matching of the NIC further reduces CPU occupancy of the communication.

Transport and network layer protocols (TCP/IP) are also offloaded even in commodity off-the-shelf hardware. Several authors exploit this feature [SC03, Mog03, FHL+05]. The Broadcom BCM570x series and other modern Ethernet

NICs support checksum calculation. Offloading checksum calculation has a small risk to transmit incorrect packets if the data is corrupted by failures on the system bus (PCI, PCI-Express). These errors cannot be detected by the NIC.

3.7.4 Progress

The existence of pipelines [WKM+98] allows to overlap calculation and communication by using non-blocking communication calls. If the transmission of data has to be stopped because the target is not prepared (e. g. no reply to a rendezvous request, no buffer announcement available for an RMA operation), there has to be some component to continue the transmission later but efficiently. The influence of this progress is analysed by Brightwell, Riesen, and Underwood in [BRU05]. They conducted several experiments to measure the impact of dependent and *independent progress* on the application performance. Progress is independent if the communication can continue independently of the remote application's calls to the communication library. The authors found two crucial factors during their experiments:

1. Using independent progress makes beneficial use of the communication pipeline. Although, without any offload or overlap, it implies a higher risk of cache pollution and context switches. This is due to frequent checks and processing required to make progress on both sides.

2. Applications that can not make use of non-blocking communication can suffer from the overhead introduced to implement independent progress.

These statements are confirmed by measurements with the NAS Parallel Benchmarks [BBB+91, NAS07] on different hardware and different implementations of MPI. The results of the paper indicate improved efficiency if overlap, offload, and independent progress are combined. Some experiments in [BRU05] confirm that the combination can improve the performance even more than expected from the sum of the particular techniques.

Independent progress in MPI can be visualised with tools like *jumpshot*. In Figure 3.10, the eager and *rendezvous mode*s of MPICH are compared. The code is shown in Listing 3.2.

```
1 for (i = 0; i < tries; i++) {
2     MPI_Barrier(MPI_COMM_WORLD);
3
4     if ((myId%2) == 0) {
5
6         timea=MPI_Wtime();
```

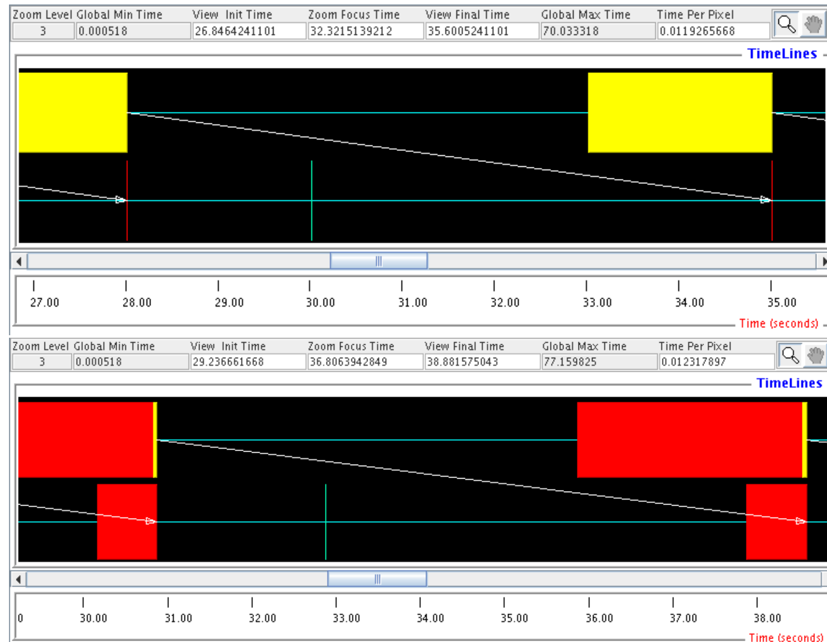

Figure 3.10: Comparing eager (upper) and rendezvous (lower) mode of MPI 1.2.7.

```
7            req = MPI_Isend(buffer, length, ... );
8            timeb=MPI_Wtime();
9        } else {
10           sleep(2);              /* delay the receive */
11           req = MPI_Irecv(buffer, length, ... );
12       }
13       MPI_Wait(req, ...);
14   }
```

Listing 3.2: MPI example to show progress.

Process 0 (upper) sends data to process 1 using `MPI_Isend`. Node 1 waits for 2 seconds before calling `MPI_Irecv`. This is to force an *unexpected message* or an *early sender problem*. Both nodes wait for additional 5 seconds before calling wait. There is a barrier (`MPI_Barrier`) before and after the experiment. The `send` operation can not finish in rendezvous mode since the receiver is not ready. Data is instantly transferred to intermediate buffers at the receiver in eager mode. The eager mode version spent most of the communication time in the barrier (yellow) while the wait (red) takes a long time in the rendezvous version. A reason for different times to wait is the instant completion of eager operations. The rendezvous variant can only send the rendezvous request. The data transfer does not start before the receiver calls `MPI_Wait`. Why the transfer cannot

start when the receiver is calling MPI_Irecv? Because MPI_Irecv could be blocked. The rendezvous reply could be sent, but since data has to be actively received (experiment is done for Sockets/Ethernet without a communication thread) MPI_Irecv has to wait for the data itself too. Otherwise, the communication channel can become congested. The communication cannot continue before the receiving application calls MPI_Wait.

3.7.5 Transport of Data

Most of the above issues describe general concepts to improve the efficiency of communication. These concepts are implemented on top of a transport protocol. The transport protocol itself has to be efficient too. This section presents some mechanisms to improve communication efficiency by reducing overheads or transmission times of the transport protocol.

3.7.5.1 Reliability

The programmer of a parallel application is not interested in checking the correct transmission of data. This task has to be done by the communication system. The communication system has to reliably transfer messages. Reliability requires additional effort and this overhead reduces the efficiency. A reliable and light-weighted mechanism for Ethernet networks is presented in [Cia99, CES02]. The approach performs better than common implementations on top of TCP/IP. Since TCP is a protocol for WAN, more overhead is introduced to assure correct transmission. This overhead can be omitted for LAN environments due to their more reliable transfer of data. Compared with this, Verstoep, Bal, et al [VBR+04] analysed the impact of reliability provided by Myrinet hardware. If configured, Myrinet NICs retransmit corrupted or lost packets. This has negligible impact on throughput. If the host CPU has to do the retransmission, sending small messages requires more time. This effect is less important than expected for large messages. The two different examples show that the impact strongly depends on the underlying protocol and the hardware.

3.7.5.2 Packet Size and MTU

The *packet size* is limited by the protocols inside the protocol stack. A maximum sized IP-packet can be 64 KiB. Larger messages have to be sent in chunks. This fragmentation is done by the communication system transparently.

In combination with the *Maximum Transfer Unit* (*MTU*) of network hardware, the packet size is important. In [CES02], Ciaccio, Ehlert, and Schnor analyse the difference in throughput of GigabitEthernet between *Jumbo-frames* and standard

MTU (1500 Bytes). The MTU has a significant impact on the throughput of mid-sized messages on certain hardware. One of the reasons is an inefficient use of the communication pipeline.

Generally, a smaller MTU reduces the relative amount of payload due to the header required in each packet ($size_{header}$). This reduces the maximal throughput of the network link B_{link} to B_{max} (see Equation 3.1).

$$B_{max} = \frac{MTU - size_{header}}{MTU} * B_{link} \tag{3.1}$$

Verstoep and Bal et. al. [VBR+04] have analysed the impact of the MTU in Myrinet networks. The result was a slight improvement of the throughput for large messages. The larger the MTU, the smaller is the improvement. More important: the number of buffers N_{buffer} on the NIC depends on the MTU (see Equation 3.2).

$$N_{buffer} = \frac{Mem_{NIC}(bytes)}{MTU(bytes)} \tag{3.2}$$

A large MTU reduces the number of buffers. If an application sends a lot of small messages, the NIC can run out of buffers. This has a more significant impact on performance than the size of the MTU itself. The authors could not find an impact on latency. This is an expected result.

3.7.5.3 DMA-based Copy

Hardware provides several ways to transfer data from application buffers to the local network hardware. Verstoep and Bal et. al. [VBR+04] analysed the impact of using programmed I/O (PIO) or *direct memory access* (*DMA*) mechanisms to communicate via *Myrinet* networks.

PIO-based transfers require the host CPU to copy data via the system bus (PCI) between main memory and NIC memory. This is a fast way to perform transmission but steals CPU cycles from the application. It requires neither initialisation nor acknowledgement of the transfer.

DMA-based copying is performed by special hardware components in the system or on the NIC. It requires an initialisation and a synchronisation phase. However, the host CPU can continue to process the application. The synchronisation is needed to signal a transfer is complete (usually done by an interrupt).

One of the conclusions in [VBR+04] is to use PIO for short messages due to the reduced overhead of initialisation and synchronisation. However, in the test environment of the authors, *PIO* made no sense at the receiver. The reason is the slow read operation from PCI devices. Depending on the platform and the message size, PIO can improve the performance.

Using DMA offers more availability of the CPU to the application if the message size is large. In the testbed in [VBR+04], using DMA forces the receiver to copy data. This is because incoming data is stored in a special memory area. An alternative implementation (also tested in the study) is able to directly access the destination buffer. This implies the above mentioned drawbacks of (remote) *direct memory access*: registration or extra TLB. Further, the Myrinet hardware is able to access a single page only. Therefore, the *MTU* setting has an impact on this feature. If the MTU is above the page size, the receiver's overhead of a packet will be doubled.

Similar to the decision of using eager- or rendezvous mode, the recommendation is as follows: *As long as the per message overhead is high, compared to the overall transmission time, use the faster PIO method even if it requires some more CPU cycles! Otherwise, use DMA transfer!* What is fast and what is high overhead strongly depends on the hardware capabilities. This is a tuning parameter that has to be determined for a particular environment to achieve the best application performance. The decision is also influenced by the application's way of using the communication system.

3.7.5.4 Interrupts and Polling

Interrupts allow asynchronous processing of components of a system controlled by a central processor (CPU). A task is given to a device (often also initiated by an interrupt). If the task is complete, the device raises an interrupt. The CPU is interrupted and the interrupt will be handled by an interrupt handler. Often, network communication is also working with interrupts, e. g. to signal incoming data. Especially, interrupts are problematic in high speed networks [MR97]. Due to the high packet rate, a lot of interrupts have to be handled in very short time. This can occupy the CPU completely and running applications cannot continue (interrupt livelock). To avoid livelocks, interrupts can be disabled [Cia99, CES02]. Polling is used instead. This reduces the latency since no interrupt handler has to be called. Additionally, the code locality is improved, cache pollution is reduced, and the application is the beneficiary. However, if non-blocking communication is used by the application, the latency can increase. Packets are fetched from the NIC not before the application polls for data. Thus, the latency is influenced by the software. Further, polling consumes CPU cycles. Thus, polling is not recommended if non-blocking communication is requested to overlap computation and communication.

The so-called NAPI (New API) [SOK01] proposes a hybrid approach. The first incoming packet triggers an interrupt. This interrupt is handled while further interrupts are deactivated temporarily. After handling the interrupt, the system will poll the interface for further packets. If a maximum number of packets is

fetched or a given number of polling cycles are done, interrupts are enabled again. This leads to a high responsiveness if applications communicate sparsely. Large amounts of data or frequent communication allow fast processing without handling thousands of interrupts.

3.8 Interdependencies

The previous sections presented several aspects of efficient communication. This section explains why many of these aspects have to be considered in combination. In short, this is because they depend on each other. Some interdependencies are summarised here from the above sections.

Zero-Copy and Memory Registration Zero-copy transmission is a nice feature of RDMA hardware like InfiniBand. As described in Section 3.7.1.2, this requires the memory to be pinned or registered. Since this is often an expensive operation, it can be more efficient to write data to intermediate, preregistered buffers and copy the data to the application buffer afterwards. If registration of memory is considered as overhead in the corresponding pipeline steps, this helps to decide if intermediate copies are acceptable or not.

Zero-Copy and Overlap Forcing zero-copy will not always result in good performance if overlap of communication and computation is employed. Making use of the pipeline characteristics can improve the application performance more than avoiding copies, especially at the destination process. Avoiding copies can prevent a pipeline step from processing further data if there's no (intermediate) buffer space available. If the usage of overlap is essential to the performance, forcing zero-copy is not recommended.

Overlap and Progress Computation and communication can easily be overlapped as long as there is an independent processor to process the local steps of the pipeline. If this asynchronous communication cannot be handled by the underlying hardware, progress on pending asynchronous operations can be made by threads or by deferring the transmission.

A thread is able to make independent progress on pending operations. It is able to efficiently use the communication pipeline. Using a thread offers the best potential to overlap communication and computation. However, it steals CPU cycles and pollutes the cache of the application. Therefore, threads are avoided to avoid these drawbacks.

Deferring transmission will result in inefficient usage of the pipeline and reduce the possibility to overlap communication and computation. For example, MPICH and MVAPICH defer the transmission if a non-blocking `send` cannot be completed. Only if the process is *in the library* (i. e. after an API call), they try to progress on pending operations. The reason is to avoid a thread that interrupts the application.

The challenge is to decide which of the *work-arounds* has the better impact on the performance of the application. In multi-core environments, one of the cores can be dedicated to a progress thread. In this case, cycle stealing and cache pollution problems will disappear.

Overlap and Offload Overlap of computation and communication is easier to realise if there is an independent processor to process pending communication as noted above. Some steps of the pipeline can be offloaded to that processor if a processor is available to offload at least parts of the protocol processing. This eases the implementation of overlap and increases the availability of the CPU to the application.

Offload and Progress It was already mentioned above that a dedicated processor to offload protocol functionality is helpful to make independent progress on pending communication operations.

Keeping in mind that the network processors to offload protocol functionality provide usually less performance than the host CPUs. Therefore, offloading the functionality of making independent progress is not *a priori* beneficial (see the fallacy in Section 3.7.3 on Page 60).

3.9 Conclusion

This chapter presented programming interfaces for communication in general and for parallel programming (see Section 3.2 and 3.3). Programming interfaces allow the programmer to express inter-process communication or even parallelism of code or data. There will be a higher potential of application's benefit from a programming interface if this interface is efficiently implementable on top of available network technology.

Section 3.7 explained various aspects of efficient transport of data. The implementation of an API can exploit these aspects. The implementor has to be careful. Several interdependencies exist between single aspects (see Section 3.8).

The overall recommendation of this section is to *be careful when applying a specific recommendation to a communication system.* Many things depend on the behaviour and requirements of the application and not everything works on

every hardware because of the different capabilities of the used technology. The specific aspects and demands of one-sided communication will be analysed in the next chapter.

Chapter 4

A Model for Communication

This chapter introduces a communication model called *Virtual Representation Model* that is derived from the *producer/consumer* like inter-process communication known from operating systems. This chapter shows that there are several similarities between *one-sided* and *two-sided* communication. The applicability of the model is analysed for different interconnection architectures (network, network with *RDMA*, shared memory, distributed memory). This model is then used to develop an efficient one-sided communication interface.

4.1 IPC: Producer/Consumer

The *producer/consumer*-based inter-process communication is known from synchronisation in operating systems [Tan01]. This synchronisation scheme has to be applied if a process (producer) produces data and transmits this data to another process (consumer) where each transmission has to be processed (consumed).

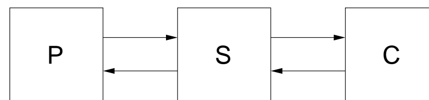

Figure 4.1: Producer/Consumer Example.

This interaction is very common in distributed applications. Otherwise, there would be no need to communicate. Because of its importance for this thesis, the required steps of synchronisation are explained here. Figure 4.1 shows the producer P, the shared buffer S, and the consumer C. The arrows show the interactions between the processes and the buffer.

1. Before the producer can deliver data to the buffer, it has to be sure that the buffer is available. This information is provided by the consumer.
2. The consumer will announce the availability of the buffer.
3. If the consumer wants to consume data from the buffer, it has to know if the producer has completed writing data to the buffer. This information can only be provided by the producer.
4. The producer will notify the consumer that the data in the buffer is available for processing.
5. Since both processes access a shared buffer, the access has to be synchronised. This is performed by mutual exclusion.

4.2 The Virtual Representation Model

Figure 4.2 shows the central property of the *Virtual Representation Model* (also referred as *communication model* afterwards): it models the task of the communication system to virtually represent the remote process to the process that calls communication functions (any transmission, receive, or synchronisation).

This abstracts the 7 layers of the *ISO/OSI Model* [DZ83, JTC94]. The application layer (7) is mapped to each process. The *communication system* can be seen as an agglomeration of the remaining layers. The communication between process A and B is virtually a direct communication in both the ISO/OSI-Model and the *Virtual Representation Model*. This abstraction is chosen to focus on the benefits for the application.

First, the *Virtual Representation Model* is explained from the application point of view. Afterwards, different mappings to interconnection architectures are described as the view of the communication system.

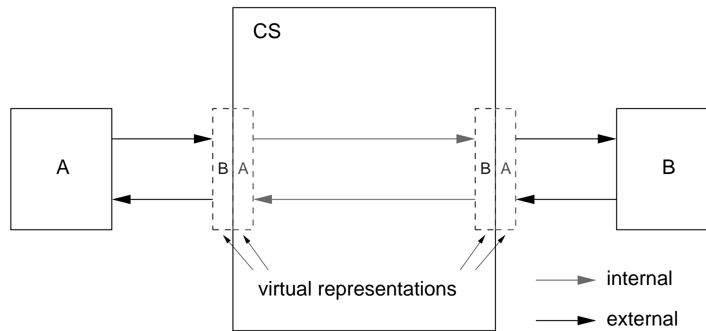

Figure 4.2: Communication model of virtually provided remote process.

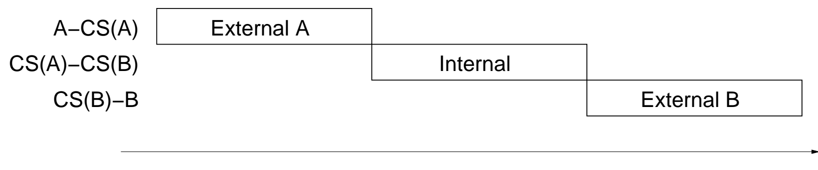

Figure 4.3: Pipeline steps of the communication model.

Figure 4.2 indicates two further important aspects of communication. The *producer/consumer*-based synchronisation between process A and B (see Section 4.1) and the steps of a communication pipeline. Figure 4.3 shows the three pipeline steps that are indicated by external and internal interactions of Figure 4.2. The first step is an interaction between process A and the communication system (A

– CS(A)). The next step represents internal interaction between parts of the communication system at process A and B (CS(A) – CS(B)). The last step is the interaction between the communication system and process B (CS(B) – B).

4.2.1 Application

Since the model abstracts communication to be an interaction between one process and the virtual representation of its communication partner, the communication system becomes a black box that provides access to the representation of the remote process. The virtual representation is mainly the representation of the memory regions that are involved in communication. These buffers are accessed to exchange data.

What does *access* mean in the context of the model and the view of the application? Each active interaction between the application and a (virtual) representation of a remote process is considered as access. An access can be classified as *direct access* and *indirect access*. Access is considered as indirect, if intermediate steps are required to exchange data between two processes. Additionally, the transmission from process A to B can require process B to actively read data or not. This excludes synchronisation that is explained separately (see below).

Therefore, the *access* can be classified into 4 classes:

- *indirect access* to the address space of B without process B involved: If the address space of A and B is different, *real* communication will be necessary. Otherwise, one would use direct access to exchange data. Communication is defined as *real* if there is any kind of network transfer required to move data from process A to B. An example is one-sided communication. Either the communication system or the processes are required to transfer the data into and out of the communication system (read or write). This involves only one of the processes.
- *indirect access* with both processes involved: Both processes are required to interact with the communication system, i. e. virtually with the other process, to complete the data exchange. This reflects the classical message passing mechanism via send/receive or a central shared memory area in the model. This kind of interaction abstracts from the fact that the memory of the other process is accessed.
- *direct access* to the address space of B without process B involved: This requires a shared address space between A and B (e.g. threads or shared memory at application level). In this case, the representation is no longer virtual.
- The last case, *direct access* with both processes involved, does not exist. Process B has no influence on the data transfer if the data is directly written or read by process A. Therefore, process B cannot be involved.

The *synchronisation* of the data transfer from process A to process B (see Figure 4.2) is abstracted to the following steps:

1. Data from process A can only be delivered to the virtual representation of process B if there is an available buffer to store the data. This buffer can be either the destination buffer of process B or an intermediate buffer inside the communication system.
2. Process B has to provide information about the buffer. This will be called *buffer announcement*. This has to happen in advance to the transmission of data to the destination buffer of process B.
3. If process B is interested in all transmitted data, it has to know when data is present in the (virtual) representation of process A. One-sided and two-sided communication slightly differ in this step. In case of two-sided communication, data is actively read from the virtual representation of process A. In case of one-sided communication, the data is silently delivered to the process by the communication system. Process B has to read the status of the delivery from the (virtual) representation of process A. This is step will be called *completion*. This is only the completion of the receiving process!
4. Only process A can decide when one or multiple data transfers are complete. Thus, process A has to notify the virtual representation of process B that the data transmission is complete. This will be called *notification*.
5. In the model, two processes access the same black box independently. Therefore, each access has to be synchronised by mutual exclusion. If the process' access to the virtual representations is done via API calls, the access can be synchronised inside the black box.

It can be seen that these steps are the same steps as described for the *producer/-consumer* synchronisation described in Section 4.1.

One additional step is required since the representation of the remote process is virtual. Process A needs to know about the *real* delivery to process B. This can be called the transmission side *completion*. Concerning the semantics, the important aspect of completion is that process A can *safely* reuse its transmission buffer after *completion*. Safely means that the data is already delivered to the destination or the communication system is able to reliably deliver the data from its intermediate buffers.

4.2.1.1 Two-Sided Communication

The classic two-sided communication via a network is represented by the model if both processes use indirect access to the communication system.

The sender will initiate a `send` operation by calling an API. Depending on the used API, it will specify either the virtual representation of the receiver or the

address of the receiver itself. It will tell the communication system the location of the source buffer. Depending on the implementation of the communication system, the data can be copied into the communication system by the sender or by the communication system. This is the first step of the pipeline – the external interaction between process A and the communication system (see Figure 4.3).

Somehow the communication system has to transfer the data to a virtual representation of process A that is accessible by process B (details can be found in Section 4.2.2). This is the second step of the communication pipeline – the internal interaction.

Process B is required to read the data from the virtual representation of process A. It calls a `receive` function to tell the communication system the process from which it wants to read the data. It announces the destination address for the data. The transfer of data can be done by the communication system or by process B. This is the last step of the pipeline – the external interaction between process B and the communication system.

The synchronisation steps of this transfer are included in the API calls. Here, a distinction has to be made between blocking and non-blocking two-sided communication.

1. If the communication system has internal buffers, the sender can immediately start transferring the data. Step one of the pipeline may happen without waiting for an announcement of a receive buffer. If no internal buffers are available, the sender is blocked (in case of blocking communication) or a pending `send` is waiting for a buffer announcement processing inside the communication system.
2. If the receiver calls `receive`, this includes the announcement of the buffer. The communication system knows about the destination now. While non-blocking `receive` calls just include the *buffer announcement*, blocking calls additionally include the waiting for *completion* (see below).
3. Using a blocking `receive` call, the receiver implicitly waits for *completion* of the transfer. Since each `receive` call has to match a corresponding `send` call, the completion is implicitly signalled if the message is delivered. Non-blocking `receive` calls require an extra API call to assure completion (e. g. `MPI_Wait`).
4. The sender implicitly notifies the receiver about the completion of the transfer. Each `send` call requires a matching `receive` call (in case of message passing interfaces), therefore the complete transmission of the data of a single `send` call implies the *notification* of the receiver.
5. Mutual exclusion can be handled by the communication system because all interaction is done via API calls. These calls can trigger mutual exclusion inside the communication system if necessary.

The *completion* of a blocking `send` call is included in the call. Non-blocking `send`s need an extra API call to assure completion (e. g. `MPI_Wait`).

Both processes are required to interact with the virtual representation of the communication partner to perform the transmission of data. Hence, this is called two-sided communication. Since process B will pick up the data, it is sufficient for process A to specify the destination process. However, omitting the specification of a destination buffer is dangerous. It requires the messages to be send and received in correct order and number. MPI avoids buffer mismatch by using tags as an abstract name of a buffer address. This allows unordered transmission of messages with different tags. Tags are given by the programmer as compile-time knowledge making internal negotiation of abstract names unnecessary. Nevertheless, each `send` requires a matching `receive`. This can be a major disadvantage of send/receive data exchange between processes.

4.2.1.2 One-Sided Communication

How does one-sided communication fit into the model? First, a remote write operation (`put`) is explained. A `get` is briefly described afterwards. According to most of the existing one-sided communication APIs (e. g. MPI-2 [MPI97] or ARMCI [NTKP06]), one-sided communication calls are considered as non-blocking operations.

There is only one difference between a non-blocking one-sided `put` and non-blocking two-sided communication: one-sided communication omits the active reading of data from the virtual representation. The data is delivered by the communication system. Therefore, one-sided communication allows an arbitrary number of operations on the remote data buffer.

The same steps of synchronisation still need to happen (see above). Thus, the operations cannot start or complete before the *buffer announced*. Process A has to notify process B to complete the operations at process B. Except from the communication calls at process A (which can include the waiting for buffer announcement), all synchronisation steps require an explicit API call (buffer announcement, completion at process A and B). For example the MPI-2 *active target synchronisation* API exposes all steps of synchronisation (except mutual exclusion) to the application.

A non-blocking `get` operation at process A tries to read data from the virtual representation of process B. It has to wait for the buffer announcement of process B and triggers the communication system to retrieve data from the memory of process B. After the data is read by process A it can notify process B about completion. The `get` call can complete locally if all requested data is available. Since process A specified the number of bytes to read, it can decide about local completion.

To transmit data, only one of the processes is required to interact with the virtual representation of the other process. This makes the communication one-sided. Since the communication system delivers the data to process B, process A has to specify the destination process and the address of the buffer. These can also be abstract names. As noted above, this allows an arbitrary number of operations on the remote buffer. This can be a benefit if the application can make use of multiple transfers within a single *producer/consumer* synchronisation [GT07].

4.2.2 Communication System

Looking inside the black box is the subject of this section. The *Virtual Representation Model* is applied to existing architectures.

All data transfers can be decomposed in two types of interaction: *external interaction* and *internal interaction* (see Figure 4.2). External interaction happens between processes and the communication system and can occur as noted above in Section 4.2.1. Internal interaction is required if the communication system is separated into parts residing on separate address spaces (e. g. different hosts or CPUs). In this case, data has to be moved internally from one representation to another. This can be network transmission or data copies via internal memory regions.

The *buffer announcement* and the *notification* can only be specified by the communicating processes. Both of the synchronisation steps have to be exposed to the application. A further propagation of this information depends on the requirements of the hardware, the implementation, and the API. For example if the API requires process A to specify the exact virtual memory address of the destination buffer, the buffer announcement has to be propagated to process A. The synchronisation messages of the *buffer announcement* have to be transferred at least to the nearest *neighbouring* representation if process B is not involved in the data transfer (one-sided communication). Otherwise, the destination is not known by the (virtual) representation that delivers the data to process B.

Since process B needs to know about the *completion* of the communication, *notification* messages of process A always have to be propagated to process B.

4.2.2.1 Network without RDMA

If the underlying network is not able to perform direct delivery to the remote process (e. g. Gigabit Ethernet), the communication system has to implement this function in software. Otherwise, one-sided communication cannot be performed over this type of network.

If the model is applied to this kind of network, all virtual representations will stay virtual. No physical mapping of an address space is possible. Thus, the

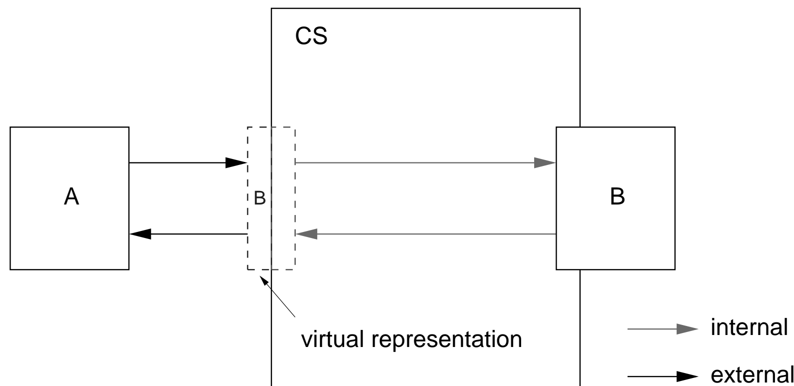

Figure 4.4: Physical mapping of process B into the communication system.

general model will apply (see Figure 4.2).

Internal communication is represented by the transfer of data and synchronisation over the network. External interaction can be implemented as CPU-based memory copies between the process and the memory of the network interface (if available).

If the communication system implements intermediate buffers (internal buffers), the messages for buffer announcement of process B don't have to be propagated to process A.

A typical example for the propagation or non-propagation of buffer announcement messages is the implementation of the *eager*- and *rendezvous* protocols. To allow eager transmission of small messages, the communication system provides internal buffers at process B. If a message is larger than the size of the internally provided buffers, process A has to wait for the propagation of the buffer announcement from process B. Usually, process A requests for the propagation of the buffer announcement. Therefore, it is called *rendezvous* protocol. According to the Virtual Representation Model, the *rendezvous* request message is not required.

4.2.2.2 Network with RDMA

Figure 4.4 shows the mapping of the model to implementations of a communication system with RDMA. For process A, the memory of process B can be directly mapped into the communication system at process B. Note, this mapping only represents the communication system's view on the communication initiated by process A. Buffer announcement calls from process B still require external interaction between communication system and process B according to Figure 4.2.

This mapping shows that *buffer announcement*s have to traverse the internal link (this is the network!). Since the communication system of process A can directly access the memory of process B (RDMA), it needs to know the exact

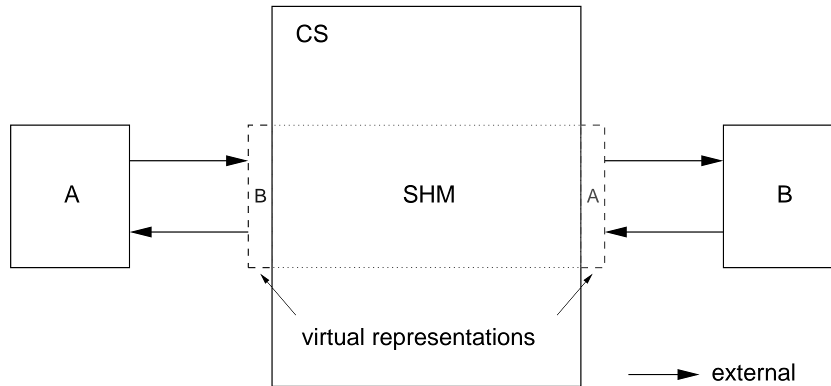

Figure 4.5: Physical mapping in case of shared memory as transport.

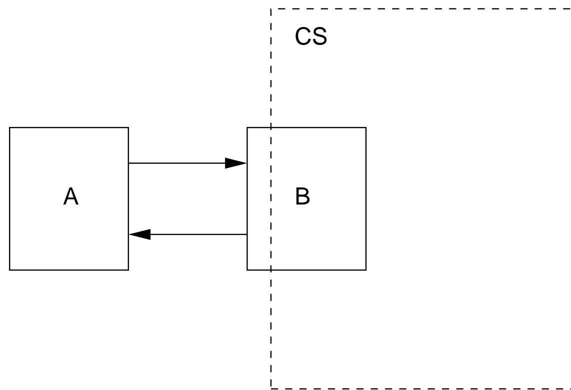

Figure 4.6: Physical mapping in case of shared address space.

memory address of the buffers. Since the speed of the networks is still slower than the speed of the interaction between process B and the communication system at process B, the buffer announcements can become a performance issue for RDMA.

4.2.2.3 Shared Memory inside the Communication System

If the communication system uses a shared memory region to transport data between process A and B (A and B have separate address space), no internal interaction is required (see Figure 4.5). The data and synchronisation messages have to be copied into and out of the memory of the communication system (external interaction).

4.2.2.4 Shared Address Space

Figure 4.6 shows the mapping if process A can directly access the memory of process B. The term *process* should be replaced by *thread* here to reflect the implementation in practise. All virtual representations can be realised as physical representations.

Process A has to know the address or the abstract name of the destination buffer. Since the address space is shared among all involved processes, the destination process directly announces the buffer to process A.

The communication system is still there. Under the hood, at least the hardware has to transfer data over memory buses or has to keep caches coherent. These tasks introduce similar external and internal interaction as the network transfers.

For example, if A and B are threads and a thread writes data into a shared variable and the result should be processed by the other thread running on the second CPU of an SMP node, the hardware has to copy the data from the cache of one CPU to the cache of the other CPU. At this level of detail, there is a communication system working and the representation of B to A is just virtual too.

The assignment of variables can also be seen as an API call to tell the communication system about the source and destination buffer and the size of the data. However, this is not recognised as an API call. The synchronisation still requires special calls. For example programming languages like C/C++ or Java lack a kind of *buffer announcement* for simple assignments. Therefore, the synchronisation is done via mutual exclusion or barriers in these environments. If *producer/consumer* semantics are required, barriers are an appropriate solution. A barrier implicitly includes the *notification* and *completion* as well as a *buffer announcement* for future access.

4.2.3 How to Apply the Model

The previous sections provided examples on how the *Virtual Representation Model* is applied to specific environments. This section provides some description how to apply the model to a general inter-process communication scenario.

Virtual Representations are the determining components of the model. If an API is given, the virtual representations are specified by this API. The model shows that process A accesses the (virtual) representation of process B. This access is determined by the API. In this case, the application of the Virtual Representation Model helps to implement efficient communication. If no API is given, the model can be a guide to an efficient API. Identify potential representations from the details and classification below! After a distinction between external

and internal communication steps, the model helps to specify and implement an efficient API.

Details of external and internal communication have to be identified. Answer the following questions:

- Where has the data to be copied or processed? This will result in a more detailed view on the pipeline and the pipeline steps.
- Which steps can be classified as local, transfer, or remote steps from the viewpoint of process A? A transfer step is moving data from a local buffer at process A to a remote buffer at process B. Local steps move data from a local buffer of process A to another local buffer at process A.
- What are the characteristics of these pipeline steps? Latency and per byte costs are important parameters. Consider the transfer characteristics of both data messages and synchronisation messages.
- Where are the bottleneck steps of the pipeline? This will help to decide about buffer management inside the communication system.

Classification of the access depends on the number of pipeline steps and the type of the last step to deliver the data to the destination buffer.

- *indirect access with involved remote process* requires at least two pipeline steps to have an indirect access. A prominent example with two steps is two-sided communication between two processes via shared memory. The first step is local to process A. The second step is local to process B (see Figure 4.5 at Page 78).
- *indirect access without involved remote process* requires at least two pipeline steps to have an indirect access. The last step has to be a transfer step to the memory of process B. Otherwise, the remote process B would be involved. An example with 2 steps is one-sided communication using RDMA. However, one-sided communication over Ethernet uses a 3-step pipeline. The communication system at process B has to deliver the data.
- *direct access without involved remote process* allows only one pipeline step to deliver data to process B. This is only possible if A and B have a shared address space. For example data exchange between two threads.

If an API has to be designed, the required communication and synchronisation steps have to be identified and mapped to appropriate calls.

4.3 Aspects of Efficient Communication

This section explains aspects of efficient communication, particularly from the point of view of the application. It can also be seen as recommendations or requirements to a communication system to provide efficient inter-process communication. Every recommendation is described in the context of the Virtual Representation Model. Details of internal communication are hidden in the black box called communication system. The communication pipeline is considered to have three steps.

Sending and receiving data is described separately. This is because receiving data is a bit more complex than sending. This is because receiving data depends on the destination process providing the buffer and the data to arrive at the virtual representation in the communication system.

4.3.1 Communication Pipeline

If the communication has to traverse a network, the second step of the communication pipeline will be the bottleneck step in today's hardware (see Figure 4.3 at Page 71). The speed of network communication is still below the speed of data transfers inside a host. According to [WKM+98], this step has to be employed as early as possible to achieve a short communication time.

This results in two recommendations: send messages (including data and synchronisation) as early as possible and store them as close as possible to the destination buffer. Sending data early will assure that the bottleneck is employed as early as possible. However, the sender also has to be sure that there is any buffer to store the data since sending data without intermediate buffering requires the destination buffer to be available. If the receiver is not prepared (*early sender problem*), the communication system can defer the transmission or provide intermediate buffers. If intermediate buffers are used, the communication system should store the data *as close as possible* to the destination buffer. *As close as possible* means if the receiver becomes ready, the delivery of data requires only the traversal of fast (non-bottleneck) pipeline steps.

Since the buffering of data requires memory, the designer of a communication system has to be very careful to not violate the *memory constraints*. The communication system should leave as much memory as possible to the application. Even if the CPU is available to quickly process the pipeline steps, the memory and the cache should be used carefully. CPU-based access to intermediate buffers also increases the *cache pollution*. This is known to hamper the performance of applications. The application data is replaced by the communicated data and has to be reloaded to the cache afterwards.

4.3.2 Sending and Writing

A process can actively transmit data in several ways. It can use two-sided communication to send data to another process. One-sided communication is a second kind to perform active transmission of data. Both of these ways can be requested as blocking or non-blocking communication. However, one-sided communication operations are non-blocking in the mentioned interfaces of Chapter 3.

If a blocking transmission is requested, the application is blocked until the operation is complete. Therefore, blocking operations are time-critical operations. The processing of the algorithm has to wait for the completion. Therefore, the CPU is available[1] to the communication system to process the requested and pending communication operation.

The application wants blocking transmissions to be processed by the fastest mechanism that is available. From the application's point of view, the processing is allowed to make arbitrary use of the CPU (see Section 3.7).

Non-blocking communication offers the possibility of overlapping computation and communication. Therefore, non-blocking communication is not as time-critical as blocking communication if two prerequisites are fulfilled: first, the computation requires more time than the communication (this is a non-trivial decision!). Second, the communication partners are not waiting for the data. This is also known as the *late sender problem* [Jül08] and can become a serious issue of performance and scalability.

If the application calls a blocking synchronisation function to wait for the completion of non-blocking transmissions, the communication operation becomes blocking. The application is blocked until the communication completes.

In many APIs, the semantics of blocking and non-blocking communication prevent the application from accessing the buffer until the operation is completed. Thus, for both kinds of transmission, the communication system has access to the involved application buffer. In case of non-blocking communication, special care has to be taken on the availability of the buffer. The operating system can schedule other tasks and page out parts of the buffer (see Memory Registration in Section 3.7.1.2 on Page 54). This can prevent parts of the communication system from accessing the buffer (e. g. if the network hardware directly transmits the data from the application's buffer).

4.3.3 Receiving

The process of receiving data is different in one-sided and two-sided communication. Therefore, a distinction is made in this section.

[1] assuming that only one process or thread is running on a single CPU

Both types of communication have the common goal to efficiently deliver incoming data to the buffer of the application.

4.3.3.1 Two-Sided Communication

If the application calls a blocking or non-blocking `receive` routine, the communication system can start to deliver incoming data to the destination buffer.

Blocking `receive` calls are time-critical because the application cannot proceed. Therefore, the fastest mechanism should be applied to deliver the data. The same considerations apply as for blocking `send` and `write` operations. The CPU is available to process the data, but the memory usage and potential cache pollution have an impact on the application performance.

4.3.3.2 One-Sided Communication

The communication system of the destination process of one-sided communication is also considered as a receiving communication system. However, the application is not actively involved in receiving data – the communication system is involved.

First, the term *receive* has to be explained for a destination process of one-sided communication. The *buffer announcement* determines the the earliest time that the communication system can start delivering the data to the destination buffer. The buffer announcement is comparable to the initiation of a non-blocking `receive`. However, the one-sided buffer announcement is not restricted to a single remote operation.

Similar to a non-blocking `receive`, the communication becomes blocking if the application starts waiting for *notification* to complete the one-sided communication operations on the announced buffer.

4.3.3.3 Two-Sided and One-Sided Communication

In general, there are three possible cases when the *buffer announcement* is issued by the application: all data is locally available, some data is available, or no data is available.

1. **All data is available**: Data is completely received to a local buffer of the communication system. This requires buffering inside the communication system and a sufficient amount of internal buffers to receive the transmitted data.

   Often, this local copy at the receiver is avoided for performance reasons (see Section 3.7.1). However, the pure speed of local copies is very high

compared to network transmission. If the application becomes ready to receive the incoming data, only a fast and local copy is required instead of a much slower transmission over the network. If it is allowed by the application semantics, the sending or writing process can complete before the receiver called `receive` in this case. This reduces the impact of the *early sender problem*.

Receiver side buffering is not possible if RDMA is used to implement (one-sided or two-sided) communication because of the physical mapping of the memory to the communication system (see Section 4.2.2.2). Therefore, this case is not possible with RDMA as a transport.

2. **Some data is available**: If some of the data is received to an intermediate buffer, the communication system can continue to receive data into this buffer and make a copy afterwards. However, this is not efficient. After the communication system knows the address of the application buffer, future incoming data should be received directly. This reduces the number of bytes to copy. The data in intermediate buffers can be copied afterwards. This kind of *hybrid* receiving is implemented in GAMMAsockets [SSP03] and NEON (see Chapter 5).

   According to the explanation of the first case, this is not possible if RDMA is used to transfer data.

3. **No data is available**: In this case, the communication system should directly use the application buffer to receive the data. This helps to avoid additional copies. This is the only possible case for RDMA-based transports (see above) or if the communication system does not provide internal buffers.

4.3.4 Reading Data

A one-sided `get` or `read` has to send a request for data. Therefore, it will suffer from the *early sender problem* if the remote buffer is not announced. The request for data has the same requirements as a non-blocking `send`. The reply has the requirements of a non-blocking `receive`.

The combination of request and reply results in an extended pipeline: three steps for the request and three steps for the reply. The bottleneck step of this pipeline is the second step of both request and reply. Therefore, the request for data should be send as early as possible, to allow the reply to start as early as possible.

If the `get` is initiated, no data will be available at the communication system of the initiating process. Thus, `get` operations will always find the last case of the above (see Section 4.3.3.3).

If the completion of a `get` is requested, the operation becomes a sequence of a blocking `send` and a blocking `receive`. If the request is already sent, then just a blocking `receive` will have to be completed.

4.3.5 Synchronisation

There is not much an API can do to improve the synchronisation of blocking communication since communication and synchronisation are triggered by a single API-call. With non-blocking communication, the synchronisation becomes more variable. An API can include some of the synchronisation steps into communication calls or provide explicit calls. At least completion must be a separate call. Otherwise, the communication is blocking.

Two-sided non-blocking communication includes the *buffer announcement* and the *notification* into the communication calls. The *completion* is done via explicit routines. This allows an application to announce a buffer as early as possible and to defer the completion to the latest time. At the receiver side the application can make use of communication and computation overlap. This helps to hide communication time behind computation and improves the performance of applications.

At the sending process, the notification is included in the communication and the completion is done by a separate call. This allows the sender to initiate the communication as early as possible and defers the completion to a later time. This also allows overlapping communication and computation. Furthermore, the early notification also allows the receiver to complete earlier.

One-sided communication interfaces require the same receiver-side synchronisation as for two-sided non-blocking communication. The sender is allowed to perform multiple communications to match a single *buffer announcement*. Therefore, the *notification* cannot be implicitly performed with a message by default.

The one-sided communication interface of MPI-2 and all other APIs presented in Chapter 3 prevent the sender-side (in case of a `put` operation) from an early notification because the *notification* is included in the *completion* call. Thus, an early notification and a late completion is impossible. In case of MPI-2, this is true with the `post-start-complete-wait` and the `fence` synchronisations.

4.3.6 Bi-directional Synchronisation

The combination of notification and completion introduces another critical issue discovered in MPI-2 if two processes bi-directionally communicate like the Cellular Automaton described in Section 5.1 of the next chapter.

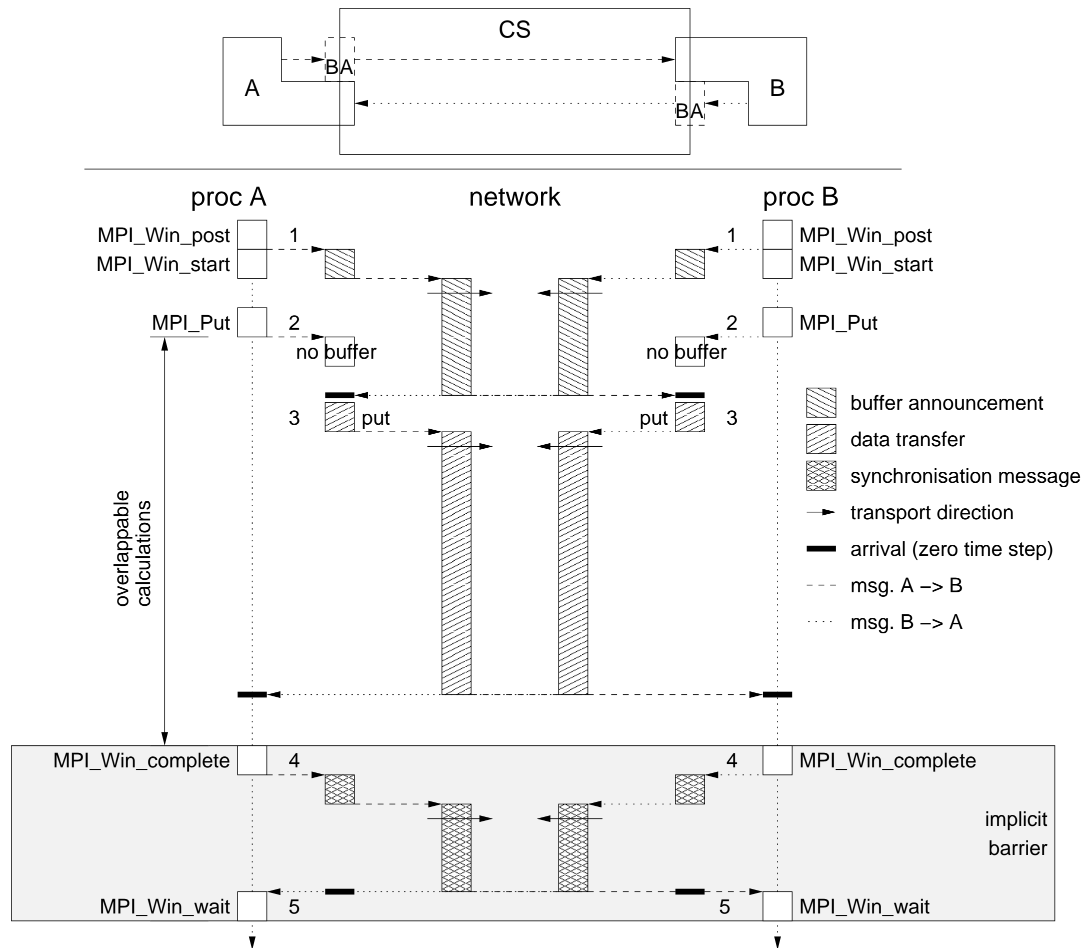

Figure 4.7: Implicit barrier with bi-directional synchronisation.

The problem of the MPI-2 interface is depicted in Figure 4.7. It shows the bi-directional interaction of two processes based on one-sided communication on top of RDMA. This means a direct data placement through a physical mapping of each of the remote processes. Any access to the communication system is indirect (API calls). Therefore, the processes A and B still access the virtual representation of each other.

The figure shows one iteration of a *bulk-synchronous* application using the MPI-2 *post-start-complete-wait* synchronisation (since the start is often imple-

mented to do nothing [GT05], it is omitted here). The dashed lines show the data flow from A to B. The dotted lines represent the flow from process B to A. First, a buffer is announced (post) and a non-blocking `write` operation to the remote process is initiated (`put`). Then some calculation is done. Synchronisation and completion are performed at the end. The synchronisation (complete) is a combination of notifying the remote process and completion of `put`. This is the critical issue. Finally, both processes wait for completion of the remote access (completion of `MPI_Win_post`).

If both processes use explicit notification together with the corresponding completion, an *implicit barrier* is created. This barrier can introduce several major performance penalties. The barrier is implicit because it is the result of a notification and a completion. On their own, both operations do not indicate any barrier-like behaviour. If ever, only experienced programmers will be aware of it.

The performance penalties are:

- Increased synchronisation time is the result of two processes waiting for a signal from each other. If the synchronisation and completion is the last step of an iteration, the completion has to wait for a full traversal of the communication pipeline (sync time).
- If the beginning of an iteration is the announcement of the buffers (`post`) and the initiation of a data transfer (`put`), more early sender problems will occur. As we know from Section 4.2.2.2, the announcement has to traverse the network. If the traversal takes more time than any calculation between the post and the `put`, the data transfer has to be deferred. The result is an inefficient usage of the communication pipeline.
- A deferred data transfer reduces the potential to overlap communication and computation. The non-blocking behaviour of `put` is counteracted.
- If the notification and completion of `put` is moved to and earlier time inside the iteration, the synchronisation can overlap the computation. The potential to overlap the data transfer itself is reduced. This counteracts the non-blocking behaviour of `put`.
- The barrier itself reduces the overall process skew tolerance of non-blocking one-sided communication. Any delay in a process results in a delayed synchronisation message and in increased iteration time of both processes.
  A process skew would allow the leading process to announce the buffer and avoid an early sender problem at the slower process. Then, the slower process can make full use of communication overlap to potentially catch up the other process. A process delay at the leading process does not influence the overall runtime. A detailed analysis of this effect is presented in Section 5.1.

Two solutions are possible for a programmer to reduce the negative impact of the barrier. First, deferring the `put` operation to avoid early sender problems. Second,

synchronise the `put` operation earlier, to allow the barrier to be overlapped with computation. Both effects are measured in the context of NEON in the diploma thesis of David Böhme [Dav08]. Both solutions provide only half of the possible time to overlap. Additionally, the programmer needs to know the *half* of the iteration.

4.3.7 Summary

The recommendations from the application's point of view are:

1. Use the fastest mechanism to transport data if time-critical communication is pending!
2. Communication becomes time-critical if any process is waiting for the communication to complete!
3. Do not make use of the CPU if no time-critical operations are initiated.
4. Be careful when considering the memory as available to the communication system!
5. Make as much use as possible of the applications buffers!
6. Respect the characteristics of the pipeline!
7. Send data as early as possible!
8. If data has to be buffered, store the data as close as possible to the destination!

The serious impact on bi-directional interaction of processes shows the issues of combined *completion* and *notification* in MPI-2. To solve this problem, the completion of `put` and the notification have to be separated. Synchronisation has to happen as early as possible to enable overlap of the synchronisation message. Completion has to happen as late as possible to increase process skew tolerance and the potential to overlap communication and computation.

The most important recommendation is to not force the implementation of the recommendations. Several recommendations have orthogonal goals. For instance sending data as early as possible will collide with the recommendation to never consider the memory as available, because early sending requires buffering if the receiver is not ready.

4.4 Design Criteria for an Efficient One-Sided Interface

This section proposes the basic concepts for the design of an efficient one-sided communication interface. The principles of the communication model (see Sec-

tion 4.2) are applied to the API.

An API is considered to be efficient if both the programmer can easily use it and the API is efficiently implementable on top of different transports (hardware and software) with different capabilities.

The basic functionality consists of a communication function to write data to remote processes and synchronisation operations for buffer announcements, completion, and notification.

4.4.1 Communication

A non-blocking `put` operation is sufficient for a basic API. A `put` call requires the source buffer, the number of bytes to transmit, the name of the destination process, the address in the destination buffer, and the synchronisation flag for implicit synchronisation.

Because the (passive) destination process just announces the starting address of the buffer, the active process has two ways to specify an arbitrary memory address in the destination buffer:

- provide the exact address or
- split the exact address into the start address of the remote buffer and an offset.

Since the second mechanism is common practise in current APIs and there's no difference in the amount of translation operations, the second method is used for the one-sided communication API presented below.

If abstract names are used to address remote buffers, the Virtual Representation Model helps to determine the virtual representation that has to know the destination address. At least the step that directly accesses the destination buffer (if any) has to know this address. This is also the location where *buffer announcement*s have to be delivered at least (see above). For example if RDMA capable hardware can directly access the remote destination buffer, the communication system at the active communication partner has to know the destination address. This means that buffer announcements have to traverse the network, if RDMA is used.

4.4.2 Synchronisation

Focusing on non-blocking communication operations, the synchronisation is an important aspect of the API. Synchronisation can be implicit (embedded in the communication call) or explicit (a separate API call). The central question is: which synchronisation steps can be implicit and which can be explicit?

The completion of non-blocking communication has to be explicit. Otherwise, the operation will become blocking.

The buffer announcement of one-sided communication is always an explicit call, since the destination process does not initiate any communication.

The only decision that remains is whether to use implicit or explicit notification.

4.4.2.1 Implicit or Explicit Notification

A portable API has to use *implicit notification*. An *explicit notification* cannot be efficiently implemented if the communication system's performance is improved by including the synchronisation into data messages. For example an implementation of *explicit notification* on top of Ethernet either will have to send an explicit synchronisation message or to defer the transfer of data to the *synchronisation point*. The deferral is inefficient in terms of the pipeline model. An additional message is inefficient in terms of the communication overhead.

While explicit notification cannot be mapped efficiently to the synchronisation message embedded into the data message, implicit notification can be mapped to both embedded and explicit synchronisation messages.

Therefore, implicit notification is preferred in the design of this API, even if the underlying network or protocol is more efficient with explicit notification. For example InfiniBand RDMA transfers are much faster than any other available mechanisms of the current InfiniBand OFED-Stack [Ope]. In this case, implicit notification can be efficiently implemented by sending data and notification messages separately.

Synchronisation of multiple operations with a single synchronisation can be a challange if message ordering is not guaranteed by the underlying protocol. For example, Open MPI is able to communicate over multiple communication paths. In this case, all synchronisation has to be done carefully since the synchronisation message can arrive before one of the previous unsynchronised messages [BSL07]. This will have a bad impact on the application.

If the focus is on the learning efforts of users: if *direct access* is available, implicit notification will force the application to access data via an API call instead of just directly accessing the data. Thus, if the goal is to design an API to encourage users of *direct access* APIs to use APIs with *indirect access*, explicit notification can reduce the efforts to port applications. This may be a benefit to users and usability but not to performance.

4.4.2.2 Synchronisation Functions

The API has to offer a way to implement a *producer/consumer*-based interaction between processes. This requires 5 synchronisation operations. Depending on the API, a maximum of four of these operations have to be exposed to the programmer. Mutual exclusion can be handled inside the communication system since each interaction is triggered by calling the API. The required steps were already analysed in Section 4.1 and 4.2.1.

Wait for Buffer Availability: An API should not expose this to the user. If the communication system is able to queue and defer communication requests or is able to buffer messages, this synchronisation can be handled transparently. Omitting this function is different from MPI-2 where operations like `MPI_Win_fence` or `MPI_Win_start` expose this functionality to the user. Some implementations implement the `MPI_Win_start` routine to do nothing [GT05].

A `put` can be deferred internally. A deferral is not new. The MPI-2 standard allows a deferral of synchronisation and communication messages. This deferral is not preferred. But since the remote buffer is not known, it may become necessary (e. g. InfiniBand RDMA can not proceed without the remote address and key).

Buffer Announcement: Only the application itself knows about buffer availability. Therefore, an API has to provide a special call to announce a buffer (*buffer announcement*). Additionally, since the destination process does not initiate communication, this call cannot be embedded into communication.

A typical method in parallel applications is to reuse buffers, e. g. an update in each iteration. If the API should match this pattern, it should provide two kinds of announcement functions. A complete announcement including address and size of the buffer. A light-weighted version just to signal the *re-availability* of the buffer. Whether the underlying communication system makes a difference or not, is unimportant. However, it simplifies the usage if buffers are reused frequently.

Wait for Notification: This is the *completion* of an announcement. For convenience there should be a blocking and a non-blocking version of this function, because probing for a status is a common requirement. This is a very common API call (`MPI_Wait`, `MPI_Win_wait`).

Notification: As seen in Section 4.3.5, this signal must be separated from the completion of communication operations. According to the above analysis, an implicit signalling is proposed since the API should not anticipate the

best synchronisation method of the communication system. To implement implicit notification, an API has to expose a flag to the programmer in order to inform the communication system to notify or not. Thus, all communication calls have to include a synchronisation parameter.

4.4.3 Completion

One additional operation is required to wait for the *completion* of communication operations. The completion of the buffer announcement has already been explained since it is a required step of the *producer/consumer* synchronisation. The completion of the communication operations has to be available too. It is proposed to use the same API call as for the completion of buffer announcements.

For a basic API, the completion based on local events is sufficient if the communication system provides reliable transmission of messages.

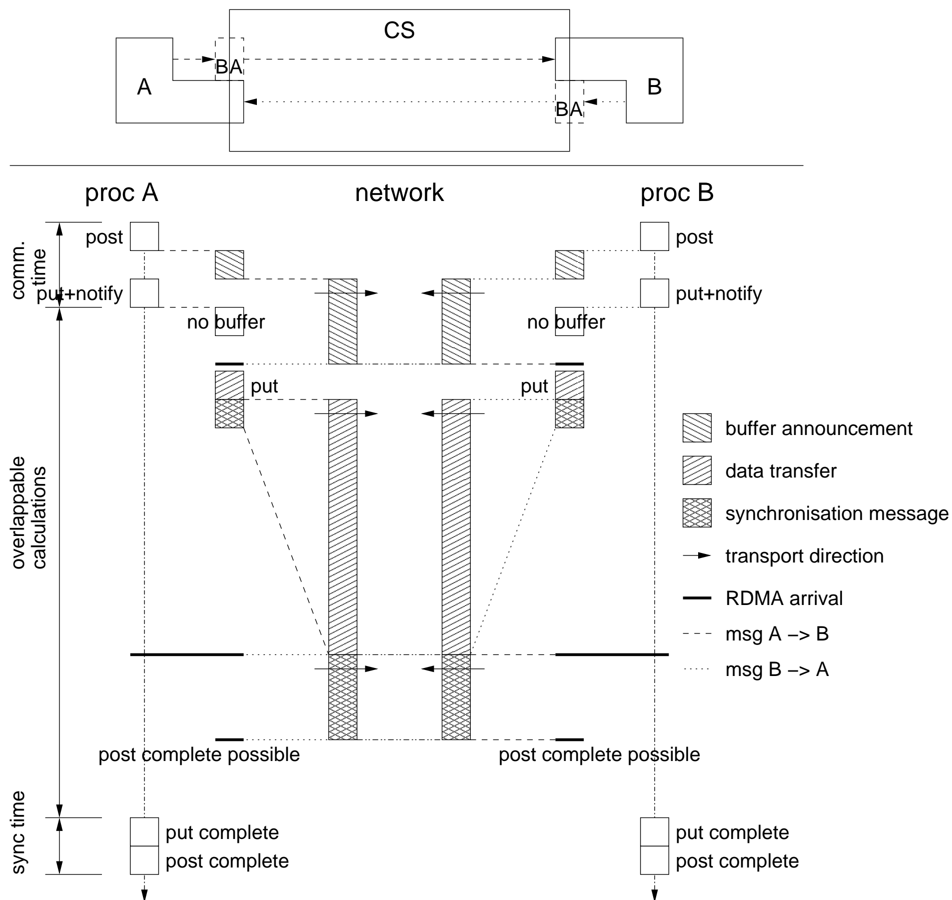

Figure 4.8: Bi-directional synchronisation in the proposed API.

4.4.4 Bi-directional Synchronisation

Figure 4.8 shows the bi-directional interaction between two processes that use the proposed API with separated notification and completion. The notification is included in the communication call (`put`+notify). The main difference between this Figure and Figure 4.7 is that the buffer announcement (`post`) can complete earlier (post complete possible).

The time between the possible completion of `post` and the call to complete the `post` is the maximum of a process skew in process A without delaying process B. This time is significantly increased compared to Figure 4.7.

Compared to Figure 4.7, the time for synchronisation at the end of the iteration is reduced to local completion (the completion of the `put` operation is considered to be local in both figures).

4.5 Conclusion

The presented *Virtual Representation Model* is based on an abstraction of the ISO/OSI reference model and the *producer/consumer* synchronisation. The model allows the comprehension of the required steps of communication and synchronisation of one-sided and two-sided communication.

By mapping the model to a certain transport protocol or hardware, it can give hints to an efficient design and implementation of the communication system. For example mapping the model to networks with RDMA (like InfiniBand) tells the implementer that buffer announcement messages have to traverse the network. This is not required for networks without RDMA (like Gigabit Ethernet).

A drawback of the model is that it can only represent the view of one of the processes at a time if physical representations are applied.

The differences between one-sided and two-sided communication are only related to the transfer of data. The *producer/consumer* synchronisation indicates that applications will not benefit from one-sided communication if they require *producer/consumer*-based synchronisation of single communication operations. Both two-sided and one-sided communication have to perform the five synchronisation steps either by the application or by the communication system. One-sided communication can reduce the number of overall synchronisations if the application can make use of multiple communication operations within a single synchronisation. There are some examples for non-contigous data [MS05] and the NAS FT benchmark [BBNY06] that could benefit from one-sided communication.

The synchronisation in two-sided and one-sided communication is compared. The most important difference between one-sided and two-sided communication in the MPI-2 interface is that *notification* and *completion* is combined into a sin-

gle API-call in case of active target synchronisation of one-sided communication. This is also true for all analysed APIs presented in Chapter 3. However, the application's view on non-blocking communication requires that notification and completion have to be separated to achieve the performance of non-blocking two-sided communication. This also applies if multiple operations are synchronised by a single notification and completion.

The basic design criteria for efficient one-sided communication was derived from the presented model. Implicit notification is more portable to the capabilities of underlying networks. On the basis of the proposed functionality, a one-sided communication interface will be designed and implemented in the next chapter.

Chapter 5

One-Sided Communication for Parallel Applications

In this chapter, the benefits of one-sided communication for parallel applications are exploited based on Chapter 4. This API is expected to be efficiently portable to different transport protocols with different capabilities. The API is called **N**ew **E**fficient **On**e-sided communication interface (*NEON*). NEON is implemented and evaluated on top of the Socket interface over Ethernet (see Section 5.3) and the Verbs API over InfiniBand (see Section 5.4).

Before NEON is presented, the behaviour of a *bulk-synchronous* parallel application is analysed in Section 5.1. This application is a Cellular Automaton.

5.1 Application Analysis: Cellular Automaton

The Cellular Automaton is an example for the large class of *bulk-synchronous* parallel applications. It iteratively calculates a so-called *stencil*. This is similar to the well known game of life by John Horton Conway.

Since the original version is based on MPI with two-sided communication, this section analyses the two-sided versions with blocking and non-blocking communication first. This analysis is required to better evaluate the reasons of the reduced performance of the version with MPI-2 one-sided communication.

Figure 5.1 shows the one-dimensional domain decomposition. Due to the neighbourhood dependency of the (9-point-)stencil calculations, the cells at the topmost and undermost borders have to be exchanged between the members of the process group. Using a one-dimensional decomposition simplifies the communi-

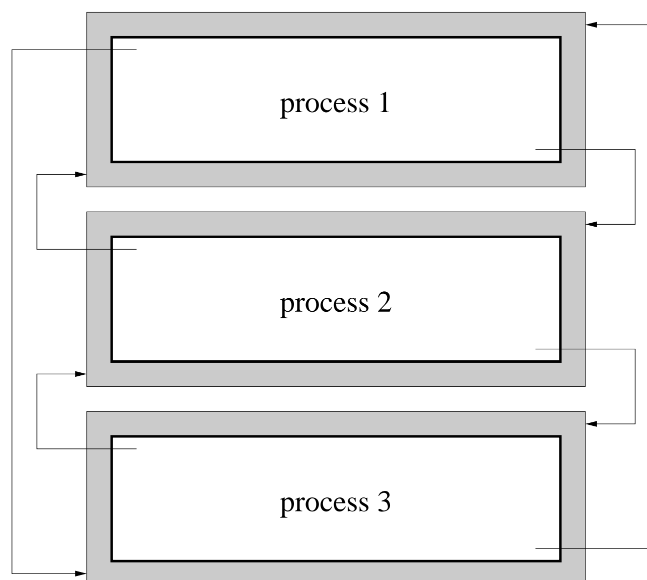

Figure 5.1: One-dimensional domain decomposition of the Cellular Automaton.

	20 lines		1024 lines	
Implementation	time [s]	comm.time (%)	time [s]	comm. time (%)
Sendrecv	29.97	83 %	379.72	9 %
ISend/Irecv	15.22	67 %	356.74	1.8 %
OSC pscw	29.82	83 %	371.56	6.5 %

Table 5.1: Runtime and of communication time of the Cellular Automaton.

cation pattern by reducing the number of messages to send. It further increases the efficiency to exchange the cells at the border, since only one dimension of an array can be contiguously represented in today's memory. Thus, the topmost and undermost line can be transferred as a contiguous block of data.

5.1.1 Measurements

Three implementations of the Cellular Automaton are measured. The blocking variant makes use of the point-to-point primitive `MPI_Sendrecv` to exchange cells. The non-blocking Cellular Automaton with two-sided communication initiates communication with `MPI_Isend` and `MPI_Irecv`. The synchronisation is done via `MPI_Wait`. The implementation with one-sided communication uses the `post-start-complete-wait` *active target synchronisation* of MPI-2.

To evaluate different ratios of communication and computation, two different sized Cellular Automata are measured. First, a Cellular Automaton is run with 10 % of the cells communicated (20 lines per process). The second experiment uses a Cellular Automaton with 0.195 % of the cells to communicate (1024 lines per process).

Each measurement retrieves the overall runtime of the Cellular Automaton. Table 5.1 shows the average of 3 runs with 30000 iterations on 8 nodes of the Einstein cluster (see Appendix A.2). The size of the transferred messages is constant (4104 Bytes).

The results are the basis of the discussion in the following sections.

5.1.2 Point-to-Point

The blocking version of the Cellular Automaton uses `MPI_Sendrecv`. Internally, this is a sequence of `MPI_Irecv - MPI_Send - MPI_Wait` in case of the used MPICH2 version. This means that it uses non-blocking communication. However, this is just to avoid a deadlock when all processes call the same sequence of `receive` and `send`.

Using a non-blocking receive has a further advantage. Each process signals the available buffer and starts sending data. This allows the communication system

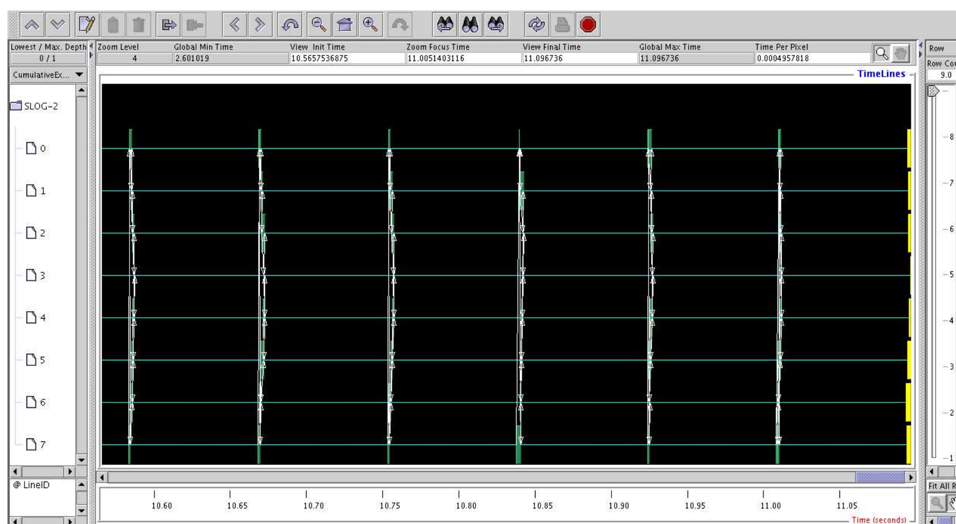

Figure 5.2: Jumpshot visualisation of the CA with MPI_Sendrecv.

to optimise communication. Since it is not known which process is ahead[1], this scheme is more efficient than a fixed send/receive pattern.

In general, this sequence of primitives introduces a point of synchronisation or a barrier between neighbouring processes. Since all processes build a logical ring, a *front of synchronisation* is created. This front is visualised by a jumpshot picture in Figure 5.2.

If the non-blocking Cellular Automaton is used (see Listing 5.1), all calculations can run in parallel to the communication, except for two lines. The first action is to announce an available buffer by calling `MPI_Irecv`. The first and the last line of each process' domain is calculated. Then, `MPI_Isend` is called to submit the topmost and the undermost line. This implicitly notifies the communication system (and finally the receiver) that this is the last (virtual) indirect access to the remote buffer. After the calculation is done, the completion of all non-blocking operations is forced by calling `MPI_Wait`.

```
for (all iterations) {
  MPI_Irecv(firstrow, previousPE);
  MPI_Irecv(lastrow, nextPE);

  /* calculate first and last row */
  simulate();

```

[1]The runtime of an iteration can vary due to interference of the operating system or load imbalances of the calculations.

```
8     MPI_Isend(firstrow , previousPE);
9     MPI_Isend(lastrow , nextPE);
10
11    /* calculate remaining cells */
12    simulate();
13
14    /* synchronise */
15    MPI_Wait (...);
16  }
```

Listing 5.1: Non-Blocking Cellular Automaton

Measuring the Cellular Automaton with 20 lines per process shows a 16 % higher amount of communication time if `MPI_Sendrecv` is used instead of non-blocking communication. The blocking Cellular Automaton runs 1.969 times (96.9 %) slower than the non-blocking version. This effect disappears if the amount of communication is low (see right column in Table 5.1). The impact of the communication disapears.

The runtime of a sequential version t_{seq} is calculated as the sum of the time t_i for each iteration i out of n:

$$t_{seq} = \sum_{i=1}^{n} t_i$$

If m processes work in parallel and exchange the required cells at the end of each iteration, the overall runtime of the blocking Cellular Automaton t_{sync} is the sum of all of the longest iteration times (see Equation 5.1). Due to non-deterministic delays, the slowest process differs from iteration to iteration. At the call to `MPI_Sendrecv`, faster processes have to wait for completion of the bidirectional blocking communication. Figure 5.2 shows the typical flow chart created with jumpshot when using `MPI_Sendrecv`. n represents the number of iterations.

$$t_{sync} = \sum_{i=1}^{n} max\{t_{i_1}, t_{i_2}, \ldots, t_{i_m}\} \tag{5.1}$$

Using non-blocking communication, the time to wait is theoretically shortened or even nullified. Analysed traces of the Cellular Automaton show that process skew also leads to wait times. A call to `MPI_Wait` will take more time if one of the communication partners is not able to transmit the required data in time. Thus, the non-blocking version also becomes (partially) synchronous. In the following, a non-blocking Cellular Automaton will be called to be in *partially synchronous* state if a `wait` call cannot return immediately due to missing remote data.

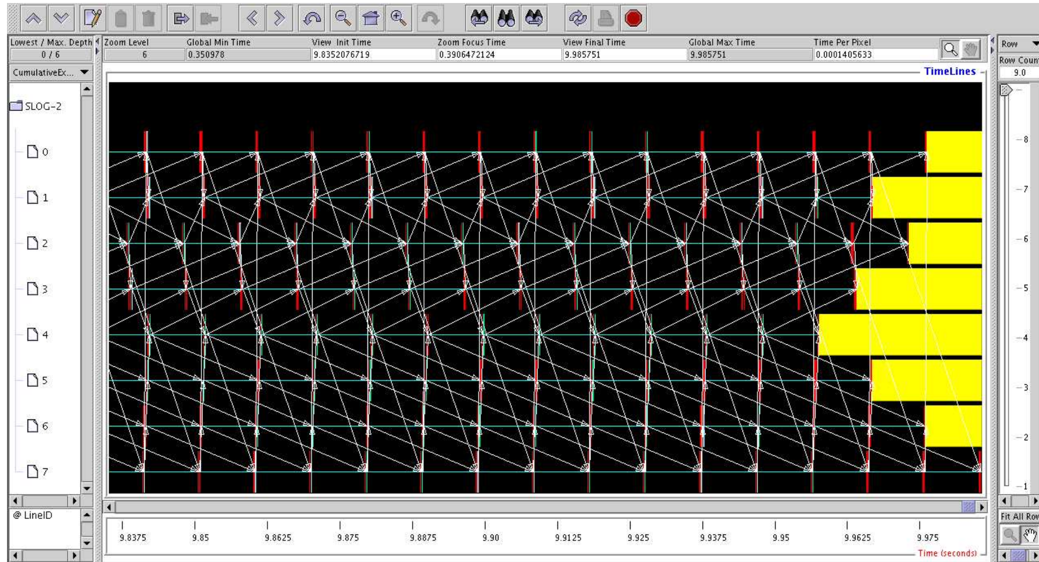

Figure 5.3: Time spent in the barrier at the end.

Even though the partially synchronous behaviour in the long term, the overall runtime is better than the runtime of the blocking version. Why? First, the possibility to overlap computation and communication can reduce the communication time. Second, overall runtime t_{async} is no longer the sum of all slowest iterations, but (mostly) equals the overall runtime of the overall slowest process l (see Equation 5.2). All other processes have to wait at the end of the calculations (see barrier wait times in Figure 5.3).

$$t_{async} = \sum_{i=1}^{n} t_{i_l} \qquad (5.2)$$

Process skew is possible due to impacts of hardware[2], scheduling, interrupts, or background processes. Thus, each iteration will potentially have a different slowest process. A study of the impact of system noise on parallel applications in large scale is presented in [PKP03].

Using non-blocking communication these differences in runtime can be (partially) buffered. The time buffer is as large as the time between initiation of non-blocking communication and the corresponding enforced completion (approximately one iteration in case of the Cellular Automaton). After this buffer is consumed, the Cellular Automaton enters a partially synchronous state and the overall runtime will be prolonged. The wait times can were detected in visualised traces. They also have been measured by counting `MPI_Test` loops that were used instead of `MPI_Wait`. If any of the waiting processes is slowed down, the partial

[2]Even identical CPUs may have slightly different timings.

synchronous state can end.

5.1.2.1 Shift of the Slowest Process

Equation 5.2 assumes that the slowest process never has to wait because all other processes run ahead. This assumption is not generally valid.

For example:

1. Starting from a non-blocking Cellular Automaton in a partial synchronous state due to process k.
2. This may result in wait times at its neighbours $j = k \pm 1$.
3. Due to system noise in process j, j is no longer ahead of k and becomes the slowest process l at the end of the calculations.
4. The overall runtime is the runtime of l and it contains wait times from a previous partial synchronous state caused by k.

After this example, Equation 5.2 can only be used as a lower boundary to the runtime of a non-blocking Cellular Automaton.

5.1.2.2 Runtime Difference Between Blocking and Non-Blocking:

Summarising the above explanations, the runtime difference is the result of 3 effects:

1. Most obviously, the communication time of the non-blocking Cellular Automaton is reduced due to the possibility to overlap computation and communication. The measured communication time is reduced by 16% compared to the blocking variant.
2. The blocking Cellular Automaton suffers directly from delays in one of the processes, caused by the synchronisation at the end of each iteration. The non-blocking implementation tolerates process skew up to a certain amount of time without introducing any wait time.
3. Because of the synchronised processing, all parts of the blocking application initiate the communication at about the same time. This may lead to traffic bursts in the network infrastructure. If the non-blocking processes are skewed, their communication can occur more scattered over runtime. This effect can hardly be quantified and will not be quantified here. It is expected to have a minor contribution to the overall runtime (especially with the low number of used processes). Some approaches to respect the distribution of network traffic can be found in [HC07].

5.1.2.3 Summary

Process skew has an impact on the performance of parallel applications (the Cellular Automaton in this case). Using computation and communication overlap compensates short delays. The overall impact is hard to predict and cannot be directly influenced by the programmer. An application developer can only try to extend time to overlap and thus increase the ability to compensate process skew. However, this depends on the algorithm.

5.1.3 One-Sided Communication

The behaviour of the MPI-2 one-sided communication is studied by using the `put` primitive together with the post-start-complete-wait synchronisation (see pseudo-code in Listing 5.2).

```
MPI_Win_Create(upper, lower);

/* iteration start */
MPI_Post(upper, lower);
MPI_Start(upper, lower);

/* calculate boundary cells */

MPI_Put(upper, to_upper);
MPI_Put(lower, to_lower);

/* calculate remaining cells */

MPI_Complete(upper, lower);
MPI_Wait(upper,lower);
/* iteration end */
```

Listing 5.2: Outline of Cellular Automaton with MPI-2 RMA primitives.

The used implementation of the Cellular Automaton uses a technique called *double buffering*. During an iteration, one buffer contains the input data. The output is stored in the second buffer. The buffers are switched at the end of each iteration. This technique has an influence on the usage of one-sided communication within the Cellular Automaton. The origin process has to determine the correct buffer for the next action.

One-sided communication in MPI-2 is non-blocking by intention. The synchronisation is performed by explicit API calls. The Cellular Automaton uses

bidirectional communication and therefore bidirectional synchronisation is required. Explicit bidirectional synchronisation results in a kind of barrier when closing the access epoch (see Section 4.3.6). The runtime of the OSC Cellular Automaton is the result of a non-blocking Cellular Automaton with a partial barrier[3] at the end of each iteration. Because of this barrier, the overall runtime will be extended by (nearly) any process skew that occurs.

If the runtime of one process k is extended due to system noise, the neighbours $j = k \pm 1$ of process k will be delayed at the *synchronisation point*. Now process k and j are behind the other processes and their neighbours will have to wait. This will continue until all processes have suffered from the delay of process k.

The overall runtime can be approximated by the same calculations as for the blocking Cellular Automaton (see Equation 5.1). The runtime t_{osc} of n iterations is the sum of the longest iteration times among all m processes (see Equation 5.3).

$$t_{osc} = \sum_{i=1}^{n} max\{t_{i_1}, t_{i_2}, \ldots, t_{i_m}\} \tag{5.3}$$

The overall runtimes of the blocking Cellular Automaton and the one-sided communication variant differ only by the communication time. This can be seen in the measurements of the Cellular Automaton in Table 5.1:

- There's no difference in the runtime between the Cellular Automaton with blocking two-sided and non-blocking one-sided communication in case of 20 lines. The communication cannot be overlapped because of the short calculation.
- If 1024 lines are used, the communication time of the one-sided Cellular Automaton is reduced by about 10 s (2.5 %) compared to the blocking two-sided Cellular Automaton. The overall runtime is reduced by about 8 s (2.1 %). This indicates that the reductionn in runtime is only the result of the reduced communication time.

5.1.4 Summary

Non-blocking communication helps to improve the performance of parallel applications like the Cellular Automaton. This application was analysed in detail. The reduction of communication time is caused by allowing computation and communication overlap. Furthermore, non-blocking communication enables a (partial) compensation of process skew.

The measurements show that MPI-2 one-sided communication may only help to reduce the communication time. Non-blocking two-sided communication of

[3] synchronising each process with its neighbours

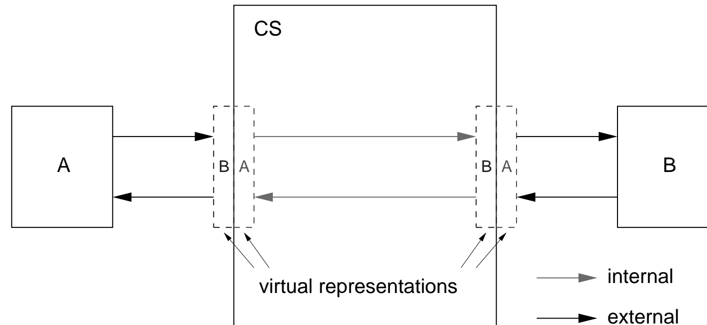

Figure 5.4: Communication between two processes.

MPI also tolerates process skew. This significantly improves the runtime of the Cellular Automaton. The NEON API will be able to exploit the process skew tolerance.

5.2 The NEON API

The main lesson learnt from the analyses of the Cellular Automaton is that using one-sided communication is not the main reason for inferior performance of non-blocking communication in parallel applications. The explicit synchronisation and the combination of notification and completion into a single API call impose a significant reduction of performance. In this section, a new one-sided communication API (NEON) is proposed.

5.2.1 Introduction

The design of the API is based on the API requirements presented in Section 4.4. Figure 5.4 shows the interaction scheme between process A and B again. The following definitions will have a main influence on the design and the explanations:

- Descriptions are based on the *put* operation. This is due to the fact that a lot of measurements with *get* operations show a lower performance due to its request/reply nature. Furthermore, a `get` operation can be added later.
- The communication is based on one-to-one communication between the members of a process group of a parallel application.
- The source process of the data is known. There is no communication with unknown processes. This is because of the buffer announcement required for one-sided communication.
- One CPU is assigned to one process – a CPU is not shared.

The advantage of one-sided communication is that once the destination buffer is announced, the source can arbitrarily access this buffer as long as the *producer/-consumer* semantics are not violated. This means that there has to be a notification if the source wants the destination to consume current data.

Since the synchronisation of `put` operations has to be implicit but the communication itself must be processed asynchronously, NEONs communication calls will be locally tested and completed by using blocking and non-blocking completion operations. These operations are well known from non-blocking point-to-point communication of MPI. Furthermore, they are independent of the context[4] to simplify the API.

The API is described by explaining buffer announcement and handling, synchronisation, completion, and communication.

5.2.2 Name and Address of Buffers

A buffer is assigned to a process. Therefore, the process has to be the a part of an address. In MPI, the members of a process group are identified by their rank, which is a simple and portable way, and will be adopted to NEON.

The virtual address of a buffer is not known before running the application. The programmer cannot work with runtime information. Thus, an abstract name – a *tag* – is chosen to identify a destination buffer. The communication system has to map abstract names to virtual addresses at runtime. This will be a portable approach since not all communication protocols and network interconnects support remote memory access (RDMA).

Compared to MPI-2, this approach simplifies the API. The creation of a *memory window* is a collective operation in MPI-2 and returns a handle to the *window*. The handle is another abstract identifier of the buffer, but the programmer has to use an additional API call to get it. Using a tag, the programmer can directly assign an identifier to a buffer to provide compile-time knowledge to the communication system.

Similar to MPI, all communication operations have to provide the complete address of the destination buffer. This is the rank and the tag in case of NEON.

Using abstract names to address a remote buffer has a disadvantage for shared-memory systems, where the destination address would be directly available. However, it represents the more portable approach and introduces only a singular overhead at the time the buffer is associated with the tag. In comparison to MPI-2, an explicit and collective call like `MPI_Win_create` is not required. Thus, the NEON API lacks an explicit routine to create windows.

[4] in contrast to post-start-complete-wait or fence synchronisation

Some network hardware (e. g. InfiniBand) requires the source and destination buffers to be registered to be accessible to the *Host Channel Adapter* (*HCA*). Usually, the programmer is not interested in those internal requirements. The first version of the NEON API did not contain any memory related routines. Unfortunately registering memory on demand often implies overhead that makes communication inefficient (see measurements in [Dav08] and see Section 3.7.1.2). Strategies to reduce the overhead of memory registration are presented in [MRB+06]. To allow an efficient implementation on top of InfiniBand, explicit routines to register and unregister memory areas are introduced in the current version of NEON.

5.2.2.1 NEON_Register

Syntax:

```
neon_memhandle NEON_Register (address,
                              size,
                              flags)
```

Description:
This routine registers `size` bytes of memory starting from `address`.

Return Value:
On success, the call returns a memory handle that is required to unregister the region. On error, the error state is returned.

Semantics:
The call allows the communication system to prepare the given memory region for communication operations. Each registered memory area may be used by arbitrary buffers. Therefore, the related API calls have the parameters `buffer`, `offset`, and `memory`

Annotation:
If the underlying hardware or software does not require any preparation, the implementation can use empty routines. Since this is a transport protocol dependent operation, the upper layer of NEON directly invokes the protocol specific implementation of this routine.

5.2.2.2 NEON_Unregister

Syntax:

```
neon_memhandle NEON_Unregister (neon_memhandle)
```

Description:
This routine unregisters the memory area specified by `neon_memhandle`.

Return Value:
On success, the call returns zero. On error, the error state is returned.

Semantics:
The call allows the communication system to remove any internal information of the given memory region.

Annotation:
If the underlying hardware or software does not require any preparation, the implementation can use empty routines. Since this is a protocol dependent operation, the upper layer of NEON directly invokes the protocol specific implementation of this routine.

5.2.3 Buffer Announcement

The destination process of one-sided communication has to announce its buffer for remote access for two reasons. First, the abstract address and size have to be known by the source. Second, the re-availability of the buffer must be announced if the buffer is reused. It makes sense to have two separate API calls to fulfil these tasks (NEON_Post, NEON_Repost).

5.2.3.1 NEON_Post

Syntax:

```
neon_handle NEON_Post (address,
                       size,
                       num_ranks,
                       list_of_ranks,
                       tag,
                       memory,
                       &status)
```

Description:
This routine is required to completely announce the buffer `address` of size

size to num_ranks remote hosts in list_of_ranks. The tag tag is assigned. The full announced buffer has to be covered by the registered memory region memory.

Optional/optimal: Optionally, this call could support immediate completion. The parameter status contains a success code then. For remote put operations, this is possible only if the communication system uses buffering techniques to allow early transmissions (e.g. socket based implementation of NEON in Section 5.3). The data has to be copied and the buffer announcement can return in complete state.

Return Value:
This routine returns a handle to a job that has to be used to check for completion of operations. It returns zero on error. The status parameter contains the state of completion or error conditions.

Semantics:
After calling this function, the local process will know about address and contact information of the remote processes in list_of_ranks. The content of the buffer has to be taken as invalid until a synchronisation routine returns successful completion of all remote communication operations that correspond to this announcement.

If there are multiple processes accessing a single buffer, the system will not guarantee a special ordering of memory access from different sources. The programmer is forced to synchronise processes manually if ordering is required.

Annotation:
An implementation of an ANY_SOURCE wildcard known from MPI is possible but not intended, since it requires synchronisation with every process in the process group (e. g. communicator in MPI), even if it is not interested in accessing the buffer.

5.2.3.2 NEON_Repost

Syntax:

```
int NEON_Repost (neon_handle,
                 &status)
```

Description:
This call is the light-weight version of NEON_Post. It takes a previously cre-

ated job `neon_handle` (using the `NEON_Post` call) to announce the reavailability of a buffer. The buffer is attached to the job by the `NEON_Post` call. `NEON_Repost` will reuse the buffer information except the state.

Return Value:
This routine returns an error code or zero on success.

Semantics:
`NEON_Repost` is a non-blocking call to announce a buffer. It works much the same way as `NEON_Post`, except the creation of a new job and association of tag, address, and job can be omitted.

5.2.3.3 NEON_Unpost

Syntax:

```
int NEON_Unpost (neon_handle)
```

Description:
This routine removes the association between the tag, the memory, and the application buffer created by `NEON_Post`.

Return Value:
The return value is zero after successful operation. In case of errors, the error code is returned.

Semantics:
Any internal structures will be removed. The library can assure that buffers are unposted only if pending jobs on this tag are complete. The programmer has to assure that there is no further NEON-based access to an unposted buffer.

5.2.4 Completion

Operations to check for completion of operations are essential to synchronise the communication and the communicating processes. In case of NEON, only two routines are introduced to complete non-blocking operations, – a blocking and a non-blocking version of a completion check.

5.2.4.1 NEON_Wait

Syntax:

```
int NEON_Wait (neon_handle,
               flags)
```

Description:
The caller is blocked until the given job (neon_handle) is completed. It is comparable to the MPI_Wait call of the MPI standard.

Optional/optimal: Optional flags can be used to specify the type of synchronisation in the future. For example a STRONG-flag can force the call to block until the data is completely delivered to the destination buffer, while a WEAK-flag allows to continue after local completion of the communication.

Return Value:
The routine returns zero if the operation was successfully completed. Otherwise an error code is returned.

Semantics:
The local content of buffers is valid after successful return from neon_Wait. All communication partners of the list_of_ranks given in NEON_Post have to send their notification before this call can return.

5.2.4.2 NEON_Test

Syntax:

```
int NEON_Test (neon_handle,
               neon_flag_t flags)
```

Description:
This routine represents the non-blocking version of neon_Wait.

Return Value:
See NEON_Wait.

Semantics:
If the return value signals completion, the content of buffers is valid.

5.2.5 Communication

The preferred way to exchange data is the put operation. A get will suffer from the fact that remote data has to be requested before data can be transferred (request/reply). Therefore, the focus is on the put operation.

5.2.5.1 NEON_Put

Syntax:

```
neon_handle NEON_Put(buffer,
                     size,
                     dest_rank,
                     tag,
                     offset,
                     flags,
                     memory,
                     &status)
```

Description:

This routine writes `size` bytes starting from `buffer` to the remote buffer `tag` of process `dest_rank`. The first byte of the buffer is written to the remote buffer starting from position `offset`. After the routine has returned, `status` contains the current state of the operation.

The `flags` parameter is used to prevent implicit synchronisation. If no flags are given, NEON assumes that this is the final operation on this destination buffer. This causes the communication system to signal completion to the remote process. Setting the flag to `NON_FINAL`, this call does not notify the destination process. A single check for completion via test or wait should be sufficient to check the final and all initiated `NON_FINAL` `put` operations.

Return Value:

The caller obtains a job handle to check for completion afterwards.

Semantics:

This is a non-blocking initiation of communication. The communication is not forced to start immediately. In case of a non-buffering implementation, the operation has to wait for a matching buffer announcement. If a buffer announcement is available, data transmission can start at any time after the `put` call. At the latest, it has to start when the application calls wait. This operation can start immediately without waiting for the announcement if the communication system has sufficient internal buffers to hold the message.

The local buffer must not be modified until the local completion is signalled by successful return of a `NEON_Wait` or `NEON_Test` call.

5.2.5.2 NEON_Get

Syntax:

```
neon_handle NEON_Get(buffer,
                     size,
                     source_rank,
                     tag,
                     offset,
                     flags,
                     memory,
                     &status)
```

Description:
This is the pendant to NEON_Put which is described for completeness. However, it is not implemented. It will read `size` bytes from the buffer `tag` at the `source_rank` starting from `offset`. The data is written to `buffer` which has to be inside the registered `memory`.

Return Value:
The operation return a handle to the job. This can be used to check for completion.

Semantics:
This is a non-blocking call. The content of local buffer is invalid until successful completion.

5.2.6 Summary

The NEON API is intended to show the applicability of the API requirements in Chapter 4. It is designed to ease the usage of one-sided communication in *bulk-synchronous* parallel applications with an (extended) *producer/consumer* synchronisation.

5.3 NEON over Sockets

This section presents a Linux-based implementation of NEON on top of TCP/IP-Sockets over Ethernet. First, the general design of the NEON implementation is explained. The goals are to provide an efficient communication system while keeping in mind to extend the implementation for further network technologies.

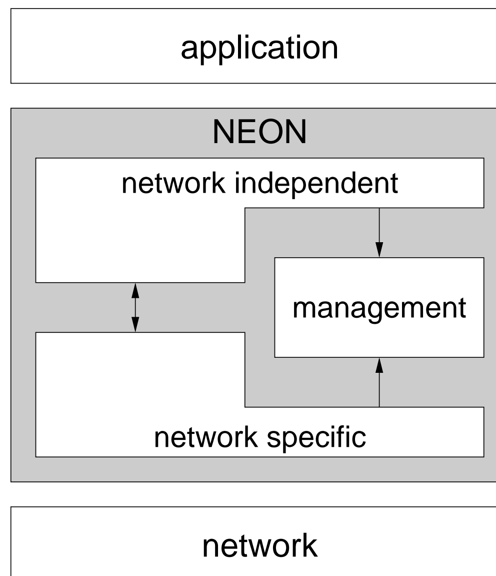

Figure 5.5: Architecture of the NEON implementation (derived from [Dav08])

5.3.1 General Design

The implementation of one-sided communication on Ethernet networks requires a mapping of memory semantics to send/receive semantics, since Ethernet has no support for RDMA. The architecture of the NEON implementation is modular and exists of 3 layers. Figure 5.5 shows the layers of NEON. The assumption of this architecture is that the network specific parts (modules) of NEON have to do the major work. This allows an implementation to make use of all the features of the underlying network.

The network independent layer is intended as a thin wrapper to check parameters and do some general initialisation. After setting up the required management structures (if any), a call to the corresponding network specific module is done.

Network specific modules are responsible for the mapping of the NEON-API to the API and capabilities of the network. This is described below for TCP/IP-Sockets over Ethernet.

The management layer is intended to provide data structures for all components of NEON. A typical procedure will be as follows. The network independent layer will create a so-called *job* in the management layer for a new operation. Then, the network module is triggered. The network module picks up a job from the management and tries to process it. The state of the job is set according to the progress that could be made. And NEON returns to the network independent layer. Now, the state of the operation can be checked and the control is given back to the application.

5.3.2 Socket Specific Design

The one-sided communication of NEON has to be mapped to the send/receive-based semantics of the Socket interface. The mapping of the following operations and events are explained in more detail: the buffer announcement (`NEON_Post`), the data transfer (`NEON_Put`), and the notification.

5.3.2.1 Buffer Announcement

When looking at the communication model in Chapter 4, it becomes clear that the buffer announcement does not have to traverse the Ethernet network. This is for two reasons.

- First, Ethernet does not provide RDMA. The destination buffer cannot be physically mapped into the communication system in a way that allows the remote communication system to write the data. The communication system of the destination has to write the data to the destination buffer or the application has to pick it up. In this case, matching the tag to a virtual address has to be done by the communication system at the destination anyway.
- Second, the overhead of sending small messages[5] is high and requires protocol processing in the communication system of both processes. Since there is generally no separate processor available to process the protocol, the host CPU is involved and the application is disturbed. Thus, the implementation of NEON over Ethernet avoids explicit buffer announcement messages and does destination-based matching of the tag and the virtual address of the announced buffer.

The buffer announcement for the Ethernet implementation is therefore a local operation. The application lets the local communication system know which buffer is available. This information is kept by the management layer of NEON to perform a matching operation on incoming data.

5.3.2.2 Data Transfer

Data has to be transferred by using the send and receive operations provided by the Socket API. The problem with these routines is their blocking behaviour. Furthermore, the TCP transport layer implements a streaming protocol. This allows the source and the destination to send and receive the message in arbitrary chunks.

[5] An announcement will be just a few bytes in size

The source initiates a remote write operation (`NEON_put`). This is mapped to a `send` call that will not block but may transfer only a part of the whole message. The remaining data has to be transferred later because the `put` is a non-blocking call.

The slow internal communication of Ethernet introduces a bottleneck step in the communication pipeline. This requires the source to transfer the data as soon as possible. The NEON implementation distinguishes between short and long messages by using an eager protocol for short messages and a rendezvous protocol for long messages.

Up to a given threshold (`EAGER_RENDEZVOUS_THRESHOLD`), the data is sent immediately. The system relies on the buffering and flow control of the TCP layer. TCP will stop sending data if the receiver has no more buffers left.

The destination announces the destination buffer. Since the buffer announcement does not traverse the network, the *early sender problem* at the source process turns into an *unexpected message*s problem at the destination. If the destination has no matching buffer posted, the message has to be stored to intermediate buffers.

If the communication system becomes aware of the real destination buffer, further incoming data is written there. The data that is already received is copied afterwards. This *hybrid* approach of buffering and direct receive is applied to comply with the rules of efficient usage of the communication pipeline.

5.3.2.3 Notification

An advantage of one-sided communication – to perform subsequent operations on the same remote memory area without synchronisation – is mapped to the Sockets' send/receive by the introduction of a flag that tells the remote side whether this is a final message or not. This requires message ordering from single processes or at least the final message to be transmitted last.

The notification will be embedded in the header of the final message. Otherwise, a short message has to be transmitted which would introduce unnecessary overhead.

The notification and the offset given by the source make the difference between single NEON communication and classic send/receive interaction. The notification is involved to let the source determine the number of bytes received. The offset determines the virtual address at the destination inside the announced buffer.

5.3.3 Implementation

Asynchronous progress on pending operations is one of the central problems to solve with the Socket API. This is essential to implement efficient external communication – the interaction between application and communication system.

During the overall design of NEON, the blocking behaviour of Sockets is ignored. The design assumes a kind of *processor* which is able to process a queue of work in the background. A crucial question is: Who does the work in the background?

Several mechanisms exist to trigger a processing in the background. However, on Linux-based Ethernet platforms, there is no real background processing available. The abilities of the network processors are limited (see Section 2.2.1). Thus, most of the background processing has to be done by the host CPU. Some of the available mechanisms are:

- dedicated communication thread
- library-based progress
- asynchronous I/O
- signals
- timers

Asynchronous I/O is already discussed in Section 3.2.1.2 on Page 36. Signals and timers introduce a high processing overhead. Measurements in [AGSZ06] showed that the handling of signals and timers is slower than the context switch to another thread. Therefore, asynchronous I/O, signals, and timers are no longer considered for the implementation of NEON. Future implementations of asynchronous I/O in Linux can make asynchronous I/O an interesting alternative to the communication thread.

Dedicated Communication Thread Using a thread for asynchronous progress is a good way to perform any operation independently of the application. A thread can immediately respond on network activity and application events/calls. It makes the implementation to comply with the pipeline model by sending data as early as possible and thus, allows to overlap parts of the communication with computation.

The major drawback of threads is that they consume resources required by the application, especially the host CPU. They require synchronisation of shared data structures. This introduces additional overhead. Finally, they require scheduling. Scheduling is coarse-grained compared to the average latencies achievable in Ethernet networks (a few 10 µs vs. 1 ms to 10 ms). This can increase the latency of the protocol.

Library-Based Progress Some implementations of MPI (MPICH, Open MPI) can use this technique to check if pending requests can be further processed[6]. Compared to a thread, it does not suffer from scheduling overhead and cycle stealing. On the other hand, the performance and behaviour depends on the application's way to call the API. Communication can be delayed, the pipeline is inefficiently used, and overlap is inhibited.

5.3.3.1 Hybrid Approach

The implementation of NEON uses a *hybrid* approach. A separate thread is used for pseudo-background communication, the thread is suspended as often as possible. The thread works as a kind of *progress engine* that tries to make progress on pending operations and incoming data or requests. No thread activities will disturb the application under the following conditions.

- The thread is suspended inside `epoll` (see below) until a request for a new connection or data is arriving.
- The thread is denied to enter the progress engine if the application uses blocking communication or synchronisation. This helps to reduce additional latencies and overhead introduced by the granularity of scheduling and context switches.

The thread is not allowed to actively poll for any incoming data since this would consume an unacceptable amount CPU cycles. Therefore, an appropriate interface has to be used to suspend and resume the thread if no communication happens. Using Linux, there are `select` and the more efficient and modern `epoll` available[7]. `epoll` is an improved and scalable kind of `select`. It returns a list of file or socket descriptors with pending data.

Some self-defined private data is attached to each file descriptor. This data is used to determine the next action to do if the descriptor becomes active. In the current implementation of NEON, this private data contains the handler to call and internal states.

The progress engine and important parts of the Socket module were developed during the Diploma thesis of Hynek Schlawack in [Sch06].

5.3.3.2 Data Structures

The *job* is the main data structure of NEON. A job contains information about the buffer address, the size, the state of transfers, and the communication partners. It is separated in two parts: a general part and a network specific part. The network

[6] Open MPI also can be configured to use a thread.

[7] By using `epoll` the implementation is restricted to Linux.

specific part can be optionally used by a network module to keep specific data. In case of the Socket implementation, this is a file descriptor, the connection state, and the number of transferred bytes.

A system type and a user type of job exists. This is because parts of the protocol require to create a job on demand by the system (e. g. if incoming data does not match a buffer announcement). The user type is to mark jobs as created by user and to prevent the system from destroying these jobs before the user requests for removal.

There are two central data structures to organise jobs. One is for operations to submit data. The other contains the buffer announcements created by calling `NEON_Post`.

Sending or writing data to a remote process has to be ordered at least to assure correct notification. In case of a single Ethernet link per node, this is no limitation. Therefore, the data structure for data operations is designed to keep operations to the same destination in a FIFO queue. Currently it uses a static array of list heads with the destination rank as the index.

Checking for finished updates of data to complete a buffer announcement will be based on the tag. Thus, the data structure for buffer announcements is designed to speed up the search for the tag. The current implementation limits the tag to a number between 1 and 255. It uses a static array to hold the posted buffer announcements and their state.

The current implementation of the data structures for transmission is socket specific and provides $O(1)$ access for required operations. It does not consider buffer announcement signals that are submitted to the source process. These operations can be triggered by the API or by the network (in case of incoming data). This structure can be moved into the socket module in future versions.

It is easy to see from Figure 5.5 that the management structures are shared data and the access has to be synchronised between the communication thread and the application. Therefore, if any structure has to be manipulated, mutual exclusion is required, except from updating the state of a job. Even if the check of the state interrupts a concurrent update, this is not synchronised. The check would fail but the data is still consistent. Therefore, the next check will be successful.

5.3.3.3 Data Transfer

The socket module uses plain `send`- and `recv`-calls to transfer data. Each transfer is split into a header and a data transmission. All headers are sent in a blocking manner. Although, this contradicts the non-blocking semantics of the API, it is assumed that sending a few bytes will rarely block. Furthermore, sending a header is expected to block only for short time. This is considered to be acceptable in the current implementation.

Data is sent by using the streaming of TCP. It is tried to send as much data as possible without blocking the sender. This implies the possibility of splitting data messages into several TCP transmissions.

There is no priority for headers. If a data message could not be send completely, this pending transfer will prevent NEON from sending any headers (e. g. rendezvous request or reply). Only one job to the same destination is processed at a time. Jobs to different destinations are processed in a round-robin fashion.

The restricted sequence of headers and data messages eases the handling of incoming messages at the receiver. Receiving a header puts the receiver into the corresponding state given in the header. If a header specifies a succeeding data message, the receiver prepares for receiving a data message. It expects the next header only if the data message is completely received.

For Ethernet the NEON implementation uses two transfer modes: *eager* and *rendezvous*. The reason is that buffering inside the communication system has to make a trade-off. A trade-off between sending data as early as possible (to meet the requirements of the pipeline model) and the amount of internal buffers to store early received data (*unexpected message*s).

For the eager protocol, the system has to check if there are other pending transfers to the same destination. If not, the header is created and the transmission immediately starts. If there are pending transfers to the same destination, the new transfer is appended to the list of pending transfers. This is required to maintain message ordering to the same destination.

In case of messages that are larger than the `EAGER_RENDEZVOUS_THRESHOLD`, just a header with a rendezvous request and the size of data is fed into the transmission queue. The threshold is configurable and is currently set to 16 KiB. After the reply, the real data is sent.

If the source receives a rendezvous request, it looks out for an appropriate buffer announcement. If there is none, it tries to allocate a sufficient buffer and returns a rendezvous reply in order to allow the source to start transmission as early as possible.

An optimisation that is not analysed and implemented yet is to send a rendezvous reply at the time of the buffer announcement. This can avoid the sending of a request and speed up the communication. However, the problem is that the destination cannot determine the size of a data message from the size of the destination buffer. This would result in useless (announcement) messages and overhead. Investigating in this optimisation is subject of future work.

5.3.4 Evaluation

The evaluation of the NEON implementation on top of TCP/IP-sockets over Ethernet is performed by measuring latency and bandwidth using a ping-pong bench-

Implementation	Latency [µs]	Bandwidth [MiB]	Bandwidth [MiB]
MPICH2 P2P	40.75	15.27 (1KB)	88.01 (1MB)
MPICH2 OSC	50.81	12.20 (1KB)	89.22 (1MB)
NEON	46.71	9.53 (1KB)	87.49 (1MB)

Table 5.2: Latency and Bandwith of NEON and MPICH2.

mark (see Appendix B.1.1) and the Cellular Automaton (see Section 5.1 and Appendix B.2.1). The results are compared to MPICH2 [MPI07]. This implementation was chosen because the Cellular Automaton performs better than with Open MPI [OMP07]. The results were obtained in a student research project at the Potsdam University [AS06].

Both comparisons to one-sided and two-sided communication are performed. All measurements are performed in the same environment. The Einstein cluster at the Potsdam University (see Appendix A.2).

5.3.4.1 Micro-benchmark Results

Comparing NEON to MPICH2, the latency and bandwidth are good over Gigabit Ethernet. The experiments were run between two nodes of the Einstein-cluster.

It can be seen from Table 5.2 that NEON performs worse than non-blocking point-to-point communication of MPICH2 (version 1.0.4p1). The main reasons are the impact of the thread and a less mature and less optimized implementation of NEON. The ping-pong with one-sided communication suffers from the additional synchronisation message.

The latency of MPI-OSC is higher than the latency of NEON. However, the bandwidth of MPI-OSC for a 1 kB message is better.

5.3.4.2 The Cellular Automaton

A comparison between the Cellular Automaton based on MPICH2 two-sided communication (p2p), MPICH2 one-sided communication (osc) and NEON is presented in Table 5.3. The table shows the runtime of 30000 iterations over MPI and NEON on 4 nodes of the cluster. The number of lines per node is varied to change the ratio of communicated to computed cells. For instance, 4 lines means 50 % of the computed cells and lines have to be transferred to adjacent processes. The message size is determined by the size of a line due to the one-dimensional domain decomposition described in Section 5.1.

It can be seen that the NEON cellular automaton performs slightly worse than the mature implementation of non-blocking two-sided communication. One could expect an inferior performance of NEON compared MPI-2 one-sided communica-

Lines/Node & Version	Transferred Message Size (Byte) 136	520	4104	16392
(4 lines) P2P	1.70	2.97	7.84	23.09
OSC	3.30	5.30	10.64	29.45
NEON	2.83	3.77	8.85	23.53
(16 lines) P2P	1.70	2.99	8.70	33.53
OSC	3.18	5.22	13.65	42.16
NEON	2.92	3.79	9.51	35.43
(128 lines) P2P	2.27	5.58	51.07	154.94
OSC	3.87	8.35	55.74	162.75
NEON	2.89	6.62	52.57	157.22
(512 lines) P2P	5.58	20.57	155.59	563.83
OSC	7.93	24.21	161.86	571.37
NEON	6.16	21.38	156.09	557.64

Table 5.3: Runtime comparison (in seconds) of the cellular automaton using MPICH2 one-sided and two-sided communication and NEON.

tion for the Cellular Automaton with messages above 1 kB (see Table 5.2). However, NEON outperforms the version with MPI-2 one-sided communication in all scenarios.

Similar results were published for FastEthernet on the Uranus-Cluster (see Appendix A.1) in [SS07a].

The NEON implementation can strictly adhere to the pipeline model since the API permits the sending of data and synchronisation messages as early as possible. The communication system can avoid extra synchronisation messages because the API makes use of implicit synchronisation in the (last) `put` call.

The NEON version of the Cellular Automaton tolerates as much process skew as the MPI-point-to-point version. It avoids an implicit barrier at the end of each iteration. This is described in Section 4.3.6.

The effect of the deferred transfers can be seen from the experiment with 4 lines. The difference between one-sided MPI and NEON increases with the message size because the messages are rarely overlapped with computation. NEON can overlap at least parts of the communication.

5.4 NEON over InfiniBand

In [Dav08], NEON is implemented on top of InfiniBand OFED. The most important aspects of the work of David Böhme are presented here. This implementation will show whether applications can benefit from one-sided communication on top

of RDMA capable hardware or not. Especially, the aspects of synchronisation and buffer announcement are of particular interest.

The results of the concepts and the InfiniBand implementation are published in [SBS08].

5.4.1 Design

The overall design can be implemented in a straight-forward manner since the NEON API can be directly mapped to the InfiniBand Verbs.

The original NEON API had to be extended to let the programmer specify which memory regions have to be registered or unregistered. The destination process' memory could be registered at the time the post call is initiated. The separation between post and re-post is very comfortable for this task. However, the communication calls like `put` and `get` require the memory regions at calling process to be registered too. The API extension was introduced since registering on demand requires sophisticated methods of cached registrations [MRB+06] or modified system library calls of `malloc` and `free`. This was considered to be too much effort for the first proof of concept implementation.

5.4.2 Implementation

As noted above, NEON is implemented on top of InfiniBand Verbs (see Section 3.2.3). InfiniBand specifies methods of implicit synchronisation by using RDMA with immediate data. This creates an entry in the remote completion queue. This event can be checked by the application at the destination process. Thus, the first implementation made use of event-based RDMA as a direct mapping of the NEON API.

The buffer announcements can arrive from any member of the process group. Therefore, a straight-forward implementation should use the reliable datagram transport of InfiniBand for buffer announcements and connection establishment. Unfortunately, the current OFED stack does not support this transport type. Thus, the first implementation uses the unreliable datagram transport. A fault tolerance protocol was intended.

Using RDMA write requires a reliable connection transport type to transfer data. An unreliable datagram queue pair was used to establish connections on demand. However, this method was changed into a full mesh of established connections at startup after the evaluation of the unreliable datagram service (see evaluation below).

Although InfiniBand HCAs are equipped with a special purpose network processor, the *early sender problem* has to be solved by the software. Thus, a communication thread is also considered for this implementation. However, the overhead

of the thread can be significant since InfiniBand has much lower latencies than Ethernet.

After an evaluation of the event-based notification via the immediate data feature of InfiniBand RDMA, the thread was removed from the implementation. Polling the completion queue is much more efficient. However, a polling thread is unsuitable to make asynchronous progress on non-blocking communication. This would consume too much of the host CPU. Without the thread, data transfers have to be deferred to later API calls in case of an early sender. This will violate the pipeline model. Unfortunately, there was no mechanism found in InfiniBand to solve this issue.

5.4.3 Evaluation

The latency of the first implementation was very high. About 40 µs compared to 5 µs of MVAPICH2. The reasons were found in slow event handling of RDMA with immediate data and the buffer announcement via the unreliable datagram transport.

It was surprising to see that sending data via the unreliable datagram service is about 10 µs slower than via the reliable connection transport. Thus, the implementation was changed to use reliable connections.

The overhead of announcements and notifications raises dramatically if the completion queue is configured to signal incoming data. Due to this measurable effect, the last step of the pipeline becomes a bottleneck because of the high setup costs, especially in case of event-based completion queue operations. The implementation gains about 10 µs if the completion queue is polled for new entries.

After removing these issues by the use of polling the completion queue and using reliable connection transport for buffer announcements, the latency dropped to 10 µs.

The latency is still twice the latency of MVAPICH2. The major reason is that MVAPICH2 uses RDMA for all communication including buffer announcements and notifications. MVAPICH checks for incoming synchronisation messages at the time the communication call is executed. If there is no announcement available, the transmission is deferred to the completion. In conjunction with the pipeline model, this has a drawback: the pipeline is filled late.

Measurements of the bandwidth show that NEON performs slightly worse but comparable at least for mid-sized and large messages.

Now, the Cellular Automaton was measured with different ratios of communication and computation. 10, 100, and 1000 lines were calculated per process. This results in 20 %, 2 %, and 0.2 % of communicated cells out of all calculated cells.

API	CA10	CA100 (ms)	CA1000
MPI-2	0.0484	0.177	1.46
NEON	0.0309	0.147	1.43

Table 5.4: Time per iteration of Cellular Automaton.

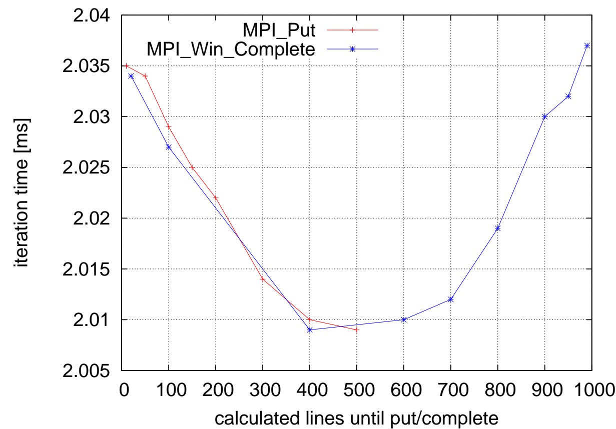

Figure 5.6: Impact of early notification and implicit barrier in MPI-2.

As shown in Table 5.4, the Cellular Automaton over NEON performs better even though the communication of NEON at 1024 Bytes is still 45 % slower than MVAPICH2. Two reasons were found.

1. Early notification of NEON leads to better process skew tolerance. This is because the remote processes don't have to wait for the explicit notification at the end of the iteration. The MVAPICH2 variant has to wait.
2. The bi-directional communication and synchronisation in the Cellular Automaton imposes an *implicit barrier*. This problem is described in Section 4.3.6 and 5.1.3. This further reduces the *process skew tolerance* and inhibits overlapping of computation and communication. Furthermore, the barrier results in a higher risk for the *early sender problem* since the *buffer announcement* and the `put` are very close at the beginning of each iteration.

These two impacts were measured by two experiments using MVAPICH2 and the MPI-2 version of the Cellular Automaton with 1000 lines per process.

- To increase the process skew tolerance, the call of `MPI_Win_Complete` is moved away from the end of the iteration towards the beginning by changing the number of lines calculated before and after the call. In Figure 5.6 the result is visible.

Moving the completion towards the middle of the iteration improves the overall runtime. The benefit of the overall runtime is only 2 %. However, the ratio of communicated to computed cells is 0.2 %. The average runtime of a single iteration is shortened by about 25 µs while transferring 1024 Bytes twice takes about 10 µs.

Moving the completion too close to the `put` call makes the communication more and more blocking and the overall runtime increases. If `MPI_Put` and `MPI_Win_complete` are called successively, a blocking behaviour is implemented.

- The second experiment moves the `put` call away from the buffer announcement. This should reduce the *early sender problem*. MVAPICH2 defers the data transfer to the completion call if the buffer announcement has not arrived at the time of the `put` call. Therefore, avoiding early sender problems should increase the ability to overlap computation and communication.

  The `MPI_Put` measurements in Figure 5.6 show that this effect is measurable. Similar to the first experiment, moving the `put` towards the middle of an iteration improves the overall runtime.

The process skew tolerance and non-blocking communication perform best if they are moved to the middle of the iteration. Unfortunately, the put and the completion cannot be moved to the middle of the iteration since this will result in blocking communication. Additionally, this implies that the middle of the iteration can be determined. This is easy for the Cellular Automaton but may not be easy for other parallel applications. Thus, the NEON approach is advantageous because it notifies the destination as early as possible.

5.5 NEON in Shared Memory Environments

An implementation of NEON on top of shared memory libraries or shared memory in common is out of focus of this thesis. In this section, some aspects of NEON are explained to check the possibilities of an efficient implementation of NEON over shared memory.

There are two kinds of shared memory available:

1. completely shared address space. This means free access to remote processes. This is only available to threads. Therefore it is not of much interest here.

2. shared memory in terms of available addresses in memory that can be accessed by all processes. This requires the data buffers either to be allocated

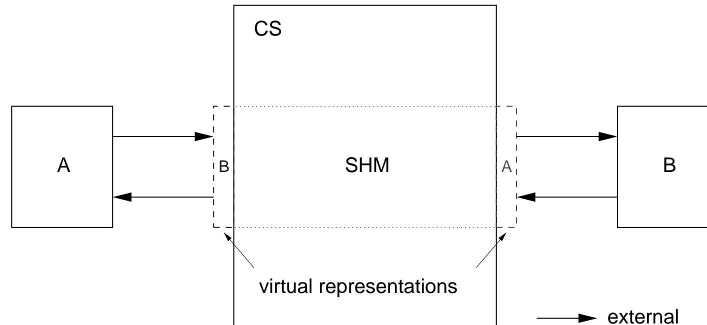

Figure 5.7: Application of the Communication Model to Shared Memory.

in a shared memory region or to reside inside the communication system which requires copying the data to internal shared memory areas and back. Using shared memory as a transport is the topic of this section.

5.5.1 Applying the Communication Model

When shared memory is applied to the communication model of Chapter 4, external communication is represented by copying data from and to the shared memory region. Internal communication does not occur or is transparently performed by hardware to keep caches and memory of different CPUs coherent. Figure 5.7 visualises this inter-process communication scheme.

5.5.2 Synchronisation

The popular shared memory API SHMEM [Shm01, Sil08] uses explicit synchronisation to signal the completion of data transfers. While the benefits of overlap play a minor role in shared memory architectures due to the fast (or non-existent) internal communication, the kind of synchronisation will have an impact.

As explained in Section 5.1.3, a barrier will prevent the application from tolerating process delays. Therefore process skew cannot happen. (Mostly) all delays in any process will increase the overall runtime. Using non-blocking data transfers in combination with separation of completion and notification will enable an application to be less sensitive to process skew.

5.5.3 Data Exchange

The current implementation of the NEON API has a very small network independent layer. It does not force any data copies or buffer pre-allocation. A lightweight

shared memory implementation of the NEON transport layer will benefit from the lightweight upper layer of NEON.

Since there is no direct access to the memory of the destination process, the buffer announcement has to traverse the external communication step at the destination process (i. e. the buffer announcement has to be signalled to the communication system).

Early senders should not be such a big issue in shared memory environments because the time for communication is very short compared to network-based communication. Therefore, a deferral of message transfers has a smaller impact on the overall performance. Furthermore, the data transfer in memory will be performed by memory copies. If no special hardware is available to do this, the CPU will have to perform at least the external communication. Thus, asynchronous progress introduces the same issues as for the socket implementation. It requires CPU cycles.

The impact of the pipeline usage and the overlap of communication and computation will be less important than for network-based implementations. The internal communication is performed by the hardware, therefore it will not be a bottleneck of the communication pipeline.

The usage and size of the shared memory regions will be a possible bottleneck for an implementation. The communication system cannot acquire arbitrary amounts of shared memory to keep the transferred data. Therefore, an implementation has to take care for the memory requirements of the communication system.

5.6 What's New

This section compares the NEON API and implementations to existing APIs and implementations for one-sided communication. A comparison to MPI-2 one-sided communication is already done in the previous sections and Chapter 4. Since NEON is designed to fit into the message passing paradigm, it will not be compared to global address space languages.

ARMCI is a message passing-based interface that is very similar to NEON. The main difference to NEON is that ARMCI does the same combination of notification and completion like the MPI-2 API. Similar to NEON, ARMCI does not use synchronisation epochs.

In [DBP+08], Danalis et. al. present a companion library for MPI called *Gravel* that allows a higher level specification of the communication pattern. This is done by separating meta-data (buffer announcements, notification, completion, etc.) from the data transfers. Every step of communication can be expressed by a corresponding Gravel call. This allows the user to maximise the potential to

overlap computation with communication and synchronisation.

The NEON API can be mapped to Gravel since Gravel provides all required steps in separate calls. Unlike NEON, it uses explicit notification. Although the Gravel interface is intended to be used by skilled users. The NEON API is designed to be easy to use. Both APIs will require a user to understand the concept of early non-blocking communication and notification and late completion in order to make efficient use of their features.

5.7 Conclusion

NEON is currently the only one-sided communication API that enables non-blocking communication with early notification and late completion. This is achieved by separating notification from completion in the API. Furthermore, the notification can be embedded into the communication call. This allows efficient implementations on different networks and increases the portability of NEON.

The Socket-based implementation of NEON shows that one-sided communication over Gigabit Ethernet can be improved compared to MPI-2. This is achieved by applying the rules derived from the *Virtual Representation Model* and by avoiding the transmission of additional synchronisation messages. The implementation on top of Sockets proves the applicability of the communication model in Chapter 4. For example, it shows that the buffer announcement does not have to traverse the network (internal communication path) in case of buffering inside the communication system at process B.

Except the usage of reliable datagram transports, the basic concepts of NEON could be mapped to the capabilities of InfiniBand. Unfortunately, they could not be mapped efficiently. The current OFED stack is optimised for RDMA. This can be seen from the measurements of buffer announcements over unreliable datagram and notification with events.

The InfiniBand implementation of David Böhme works well. Even though it has a higher latency than MVAPICH2, it outperforms the MPI-2 implementation in the Cellular Automaton benchmark. This indicates that the concept of separating completion from notification is beneficial for non-blocking one-sided communication.

According to the applied communication model and the synchronisation issues investigated in conjunction with Sockets and InfiniBand, an implementation of NEON for shared memory architectures is expected to be straightforward.

Explicit synchronisation has to be used with care by the programmer. If the application communicates in two directions, there is a high risk of implementing an implicit barrier synchronisation. This barrier inhibits the compensation of process skew even if non-blocking communication is used. The impact of implicit

barriers and overlap were measured with MVAPICH2 in Section 5.4.3.

Asynchronous progress on pending data transfers and remote memory access is easier with InfiniBand compared to Ethernet. The hardware is capable to process pending work requests. Transferring messages with the Socket API required a lot of effort to cope with the streaming semantics of TCP.

Either with InfiniBand or with TCP/IP-sockets, solving the *early sender problem* requires a lot of efforts. This is a task of event handling or polling. Both have proofed to be inefficient in InfiniBand. The same problems occurred in the Socket implementation.

The most important conclusion from the implementations is that both of the implementations suffer similar problems. Building an implementation that applies the pipeline model without hampering the performance by using inefficient mechanisms of communication is currently not possible with both InfiniBand and TCP/IP-sockets over Ethernet. InfiniBand supports the required mechanisms but their implementation offers slow performance and too much overhead. Ethernet lacks the support for remote memory access and the streaming semantics of TCP require additional overhead for message passing.

Parallel applications cannot benefit from one-sided communication if synchronised single communication operations are used. Two-sided communication offers simpler semantics and better performance with the current MPI API and implementations.

The separation of notification and completion is proposed to the *MPI Forum* as a contribution to the *MPI-3* Remote Memory Access standard.

Chapter 6

One-Sided Communication for Server Load Balancing

Server load balancing is the other important application of clusters focused in this thesis. This chapter analyses whether server load balancing can benefit from one-sided communication or not.

After a general introduction to server load balancing, the synchronisation requirements are analysed and compared to the *Virtual Representation Model* (see Chapter 4). In Sections 6.3 to 6.6 a new credit-based scheduling is proposed and evaluated.

6.1 Server Load Balancing

Server load balancing [Bou01] is a technology to distribute the traffic of a site among a number of servers and to improve the availability and quality of a service by building a scalable, reliable, and flexible service environment. A server load balancing system allows for the inclusion of more servers to adopt to increasing traffic or service demands. Particular servers can be maintained or removed from the system without interrupting the service and, thus, make the service very flexible. If some servers crash, the service will still be available.

The main tasks of a server load balancing component are [Bou01]:

- distribute the traffic of the site
- select an appropriate server for individual requests
- maintain an up to date list of available servers
- redundancy of the dispatcher to avoid a single point of failure
- enable connection or session oriented services to work, even if the forwarded packets are not aware of these

Traffic distribution, server selection, and maintaining a list of available servers are the focused tasks. The redundancy of the dispatcher is subject of future work.

6.1.1 Server Load Balancing Techniques

Apart from server load balancing, there are other techniques to balance the traffic of a site. Some of these technologies will be presented here briefly. These techniques are illustrated more in detail in [Bou01].

6.1.1.1 DNS-based Load Balancing

DNS-based load balancing is a common technique that uses the domain name system (DNS) to balance the traffic. It is also known as DNS round-robin. Clients translate a URL into an IP address by DNS requests. The DNS server has a list of

available servers and returns one of the IP addresses at each request in a round-robin manner.

This approach is quite simple and works well. But it has some major drawbacks. One problem is the concept of DNS caching. The clients store the answer from DNS servers in a local cache that is reused if the same URL is requested again. Thus, the DNS-based load balancing is bypassed if there is a DNS cache hit at the client. Further, DNS-based load balancing can hardly take server availability into account. Failed servers have to be removed from the DNS database. While this may be done quickly, it will take a while to be propagated to all the caches. Therefore, a lot of clients will try to access the failed server.

6.1.1.2 Global Server Load Balancing

This is a variant of load balancing that distributes the load depending on the location. Clients from the same or similar geographical area are routed to the same site. If one of the sites is not available the service is provided by the servers of another region. This would increase the latencies of the service and the load of the servers, but the service itself will be available.

6.1.1.3 Clustering

Clustering is a technology to distribute the load at application level instead of manipulating network packets. The servers divide the '…tasks amongst themselves…' [Bou01].

6.1.2 Architecture of Server Load Balancing

A *server load balancer* is a special component that intercepts the traffic of a site and distributes it among several servers. Generally, a server load balancer can work at any ISO/OSI-layer. The chosen layer depends on the type of service. If the service requires a connection-aware server load balancer (e. g. a TCP/IP-based service like www), layer 4 should be preferred. In the following, the term server load balancer will refer to a site-local hardware or software that performs the tasks described above. The machine that distributes the requests can also be called *dispatcher* or *frontend server*.

A popular technique is to assign the dispatcher a *virtual IP* (*VIP*) to make the service available to clients. Additionally, a TCP or UDP port is assigned to specify the provided service. At least one real server (also called *backend server*) has to be attached to the VIP, to run the service(s) on. While one real server is required to enable the service at all, the service becomes more reliable and flexible with multiple backend servers.

6.1.2.1 Flat-Based Server Load Balancing

Based on the classification presented in [Bou01], two architectures are distinguished by IP address configuration. In *flat-based* server load balancing systems, the dispatcher is involved in incoming and outgoing traffic. The server load balancer and the backend servers are located in the same subnet. A packet is modified according to the following steps:

1. The dispatcher rewrites the destination address of an incoming packet and forwards the modified packet there. The address of the backend server is chosen by the scheduling algorithm.
2. The response of the server will go back to the load balancer, since it is the default route of the server for responses to the source address.
3. At the dispatcher, the source address of the outgoing packet is rewritten to the virtual IP of the server load balancer and the packet is send to the client.

Flat-based setups can be used together with *bridge-path*, *route-path*, or *direct server return* setups. These are the three classes of the return-path-based classification of load balancing systems according to [Bou01]. A further classification used in [Bou01] is the physical connectivity that can be one-armed or two-armed. With flat-based configurations a one-armed setup makes sense, since the dispatcher and the backend servers are on the same subnet.

6.1.2.2 NAT-Based Server Load Balancing

NAT-based server load balancers are used if the real servers should be in a different subnet, e. g. for security reasons. In these cases, *two-armed* setups make sense and neither bridge-path nor direct server return are possible. In the following, NAT-based server load balancing implies route-path and two-armed setups. The steps required to answer a client request are the same as for flat-based configurations.

6.1.3 Quality of Scheduling

The quality of a load balanced service is determined by the view of the client. The best distribution will result in the best service quality parameters. In this thesis, the quality is defined by three measurable parameters.

Dropped Requests The number of rejected or *dropped requests* has to be low if good quality is a goal. If a client request is dropped by the load balancing system, the service is not available to that client. This means a lost client and potentially a lost customer. Therefore, the number of requests dropped is an important indicator of the quality of a service.

A request can be rejected for many reasons along the network path from the client to the server. For the evaluation of a server load balancing system, only the server-based reasons are considered. These are limits of the dispatcher or the backend servers e. g. a socket-based application can only open about $64K$ connections at the same time. Further connections have to be rejected. Other limits are discussed in Section 6.3 where several metrics for load statistics are presented.

Answer Time If the service is available, the shortest average answer time of a request represents the second quality parameter.

If the dispatcher selects the *least-loaded* server, the average answer time of the service is expected to be the shortest. The answer time consists of two parts: the *round trip time* (*RTT*) of the network and the time to process the request. The design and implementation of the server load balancing system has an impact on both.

The processing time is influenced by the capabilities of the processing machine and the number of requests that are already waiting for processing.

The dispatcher's scheduling algorithm will also have an impact on the round trip time. The algorithm chooses the backend server and influences the load of the dispatcher itself. Considering a large number of backend servers, an $O(1)$ scheduling algorithm is expected to be more scalable than $O(n)$ algorithms. Since the scheduling of requests requires processing, a high load of the dispatcher is expected to increase the round trip time.

Burst Length While the first two parameters can be measured by a single client, the length of a burst is only measurable at the dispatcher. The length of a traffic burst is an indicator of availability. Here, the *length* is measured as the number of requests that is contained in the burst. How many requests can be handled without having to drop a request if a the request arrival rate is equal or above the maximum capacity of the sum of all backend servers? This is the main question to anwer by the burst tests.

The longer a burst can be without dropping a request, the better the load is distributed among the available servers. Since network traffic is known to be bursty in nature [JD05], this measure is also important to evaluate the load balancing system.

6.1.4 Load Balancing Algorithms

Many load balancing algorithms are used in practise. A short description of some implemented algorithms of the *Linux Virtual Server* (*LVS*) project is presented

below. A complete description of the algorithms can be found in [Lin07a]

Round-robin This is the adoption of the classical scheduling scheme known from operating systems. The requests are distributed equally among the servers regardless of their current load.

Least Connections This algorithm counts the number of open and inactive[1] connections. The next request is scheduled to the server with the least number of connections. According to the source code (Linux 2.6.19), inactive connections are weighted $\frac{1}{256}$ compared to active connections.

Shortest Expected Delay This algorithm is very similar to the least connections algorithm. It count the number of active connections only. The next request is scheduled to the server where the number of connections plus one is minimal.

Source Hashing The source address of a packet is put into a hash function. The outcome is the number of the server the packet is scheduled to.

Destination Hashing The destination address is put into the hash function. The outcome is the number of the destination server. This is useful if a service is accessible via several different virtual IP addresses.

Never Queue The basic idea of this algorithm is to distribute packets only to servers that will not queue the request. Regardless of the speed of the server, queueing the request is expected to be slower than handling it immediately by a slow server. If no idle server is available, shortest expected delay is applied.

6.1.5 Server Weights

Most of the above algorithms are able to use weights to reflect heterogeneous servers. This is a common technique. The *speed factor* in Equation 6.1 represents the speed of a machine in relation to the slowest machine. In the literature, sometimes the speed factor α is defined reverse to this definition. Following the specification of server weights in the popular server load balancing system LVS [Zha00, Lin07b], the variant in Equation 6.1 is used. A weight of 2 represents a server that is twice as fast as a server with weight 1.

$$\alpha = \frac{\text{slowest avg. service time}}{\text{avg. service time}} \tag{6.1}$$

[1] closed connections are considered inactive for a certain amount of time

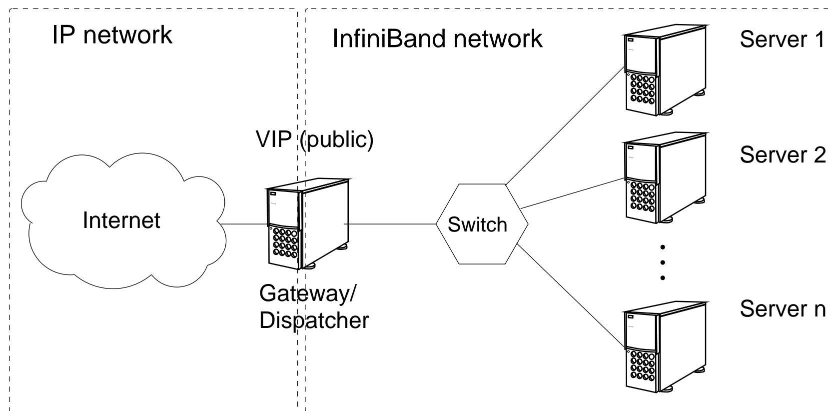

Figure 6.1: Architecture of SlibNet.

The capability of a server strongly depends on the service. For example, if a service requires processing only, the weights should be calculated from the CPU speed. Any other resource of a machine can be the basis of weights (see Section 6.3.2).

Instead of given resources, the calculation of weights can rely on measurements. This will approximate the exact weight by firing a workload to the servers and use the results to determine the weights. This has two drawbacks. First, it takes more effort to determine the weights. Second, the result is the weight for the chosen workload. Even if the workload consists of traces from real traffic of the site, this workload will be different from future workloads created by real clients. In general, requests are inhomogeneous and this can not be reflected by predefined weights. Thus, the weights can never be exact.

Weights have shown to be a critical issue, since their exact determination is impossible. Inexact weights significantly reduce the quality of the distribution [SLP94].

In [SSZL08], an example is described with two Apache backend servers differing in CPU speed and memory. One machine has only half the CPU speed and the half memory of the other. Thus, one would expect the correct server weights to be 1:4. The result was 3:4 when separately measuring the performance of the servers using *RUBiS*-traces [CCE+03] with *httperf* [MJ98]. Finally, using *Weighted Round-Robin* with the weights 1:2 performed best with both servers together.

6.1.6 Conclusion

Server load balancing is used to increase the reliability and scalability of services. The homogeneity of the cluster is less important than for parallel applications.

Therefore, heterogeneous clusters are used often. Server weights are a common technique to adopt to heterogeneous servers. These weights are hard to determine and only reflect the heterogeneity of the servers. The problems of heterogeneous workload can not be solved by server weights.

SlibNet has to use a NAT-based approach, because the InfiniBand interconnect between dispatcher and real servers implies a different subnet from the public Internet or classic Ethernet LAN technology. This is shown in Figure 6.1. Hence, the return path will be route-path-based and the dispatcher will be two-armed.

The most important factors for the evaluation of the load balancing system are the answer time, the number of dropped requests, and the length of tolerated bursts.

6.2 A Case for One-Sided Communication

The term one-sided communication was primarily used in the context of MPI and parallel applications. This section introduces one-sided communication to resource monitoring and reporting in conjunction with server load balancing.

6.2.1 Resource Monitoring

The dispatcher can rely on current load statistics of the servers to improve the *quality* of the service. Three classes of scheduling algorithms can be identified using the kind of *resource monitoring* as a classification criteria:

1. **No resource monitoring at all:** This class of algorithms does not use resource monitoring and distributes the load according to a fixed pattern. This pattern can be created from given knowledge about the service (e. g. server weights). LVS uses this concept for the *round-robin* (*RR*) scheme to distribute requests.

   Using no resource monitoring at all is a fast and simple approach to distribute incoming requests. However, it does not respect imbalances resulting from inhomogeneous requests. An advantage of ignoring the current load is that the scheduling decision easily can be implemented by a simple algorithm with low complexity. No calculation or search has to be done on the arrival of requests.

2. **Dispatcher-based monitoring:** If the dispatcher itself collects and calculates statistics of the servers to choose a server for the current request, this is named as *dispatcher-based monitoring*. The *least connections* (*LC*) algorithm of LVS is an example since the dispatcher counts the current number

of connections to a server and uses the server with the least number of connections.

3. **Backend-based monitoring:** Using *backend-based monitoring*, the dispatcher requires the backend servers to provide statistics. These statistics can be pushed by the backend servers or pulled by the dispatcher to use the values for the determination of the server to send the request to. This concept is implemented by the *feedbackd*-extension [Ker08, Ker03] available for LVS. This extension reports statistics from the backend servers upon request of the dispatcher (pull mechanism).

Additionally, a combination of dispatcher- and backend-based resource monitoring is possible. The dispatcher generates statistics that are updated or adjusted by statistics from the backend servers.

The backend servers are a good place to gather load statistics since they have the best knowledge of their current load. The drawback of backend-based systems is that the statistics have to be transmitted to the dispatcher. This implies additional traffic in the backend network and additional processing overhead at the dispatcher. Since the dispatcher is a central component, the processing may become a bottleneck that will limit the scalability of the service. As noted in Section 6.1, a higher load can result in increased round trip time.

6.2.2 Characteristics of Resource Monitoring

Resource monitoring applications do not require remote synchronisation. The process interaction is *different* from the *producer/consumer* scheme. Thus, there is no need for *notification* and *completion* (see Chapter 4). Remote synchronisation is unnecessary because the monitoring component is only interested in the most recent value. For example, data can be silently written to the dispatcher's memory. The RMA routines of an API need to be used only.

In case of concurrent access to the data, mutual exclusion is the only required synchronisation. This is only a local synchronisation at the dispatcher.

Before the data can be written, the remote process has to know about the target buffer. This buffer has to be announced. However, since the buffer does not change and is always writable, this *buffer announcement* has to occur only once.

Server load balanced systems are not highly dynamic environments of resource monitoring. New backend servers are added manually and their registration to the dispatcher is not time critical. Furthermore, the frequency of machine failures cannot be higher than the inclusion of new and restarted machines. Otherwise, the service would fail completely after a time. This results in a more or less slowly or rarely changing number of communication partners (backend servers)

of the dispatcher. This means that buffer announcements are rare and non-time-critical events.

Another issue of resource monitoring is the resource consumption. Since the monitoring requires processing, it consumes some of the resources that are monitored. For example, by measuring and transferring the CPU idle time, CPU cycles are consumed and adulterate the measurement. These effects also have to be considered for an efficient resource monitoring.

Assuming resource monitoring to be a case for InfiniBand RDMA, the *Virtual Representation Model* has to be applied as an indirect access model without the remote process involved. If the servers shall access the memory of the dispatcher, the following steps of synchronisation are required:

- The *buffer announcement* has to be sent once from the dispatcher to a new registering server. Since the virtual representation of the dispatcher at the server delivers the data directly to the memory of the dispatcher, the announcement has to be sent over the network. Since this is only a single event that is not time-critical, the performance does not matter.
- *Waiting for an announcement* is done at the server. During the registration, the server has nothing to do. Therefore, this can even be a busy waiting (provided that only a single service runs on the server).
- Sending *notification* messages is only required if the server is re-registered. This means, if the dispatcher took the server out of the schedule and has to be notified to take it back into the schedule. Even this notification can be omitted if the dispatcher frequently checks out-of-schedule servers.
- *Waiting or testing for a notification* can be omitted as long as there are no servers taken out of the schedule. Otherwise waiting or testing for a notification is required.

Fast data delivery without the destination's CPU being involved in the communication is required together with infrequent and non-time-critical buffer announcements. Thus, the InfiniBand Verbs API fits well.

6.3 Credit-Based Scheduling

This section explains the *credit-based scheduling* and finds a metric to calculate the number of credits. The basic scheduling algorithm is presented afterwards.

6.3.1 Credits

The basic idea behind a *credit-based scheduling* is to let the servers tell the scheduler how many requests they can handle. Thus, credits will be a metric that rep-

resents the availability of a server application. This simple metric can be used to run a fast scheduling algorithm on top of it.

Credits are a common technique used for flow control of network protocols [Cia99, Inf02a]. In a similar sense, one can map the intention of credits to server load balancing. Credits are used to avoid forwarding requests to a server that would drop the request due to overload. Primarily, they represent the availability of a service.

Credits are self-adapting and can remove the need to specify weights. The key is that a credit represents a free resource of a server. If a server processes requests faster, it will have more free resources. Thus, it will report more credits than a slower machine and tells the dispatcher that it can handle more requests. This makes the number of credits self-adapting to heterogeneous servers.

Credits also self-adapt to heterogeneous requests. If a server has to process requests that consume more resources, it has fewer free resources to report. Thus, it will tell the dispatcher to forward fewer requests by reporting a smaller number of credits.

In this thesis, one server will run only one single service. Nevertheless, it is expected that the presented concepts are applicable to multiple services per machine due to the self-adaptability of the credits.

6.3.2 Monitorable Resources

Modern operating systems provide several monitoring facilities. Some examples of resources that can be monitored are idle time of the processor, available memory, or available disk space. But, how can these figures be used to calculate the number of further requests a particular server can handle? This number is hard to determine in conjunction with a service that is requested by an unpredictable number of clients. Depending on the service, each request consumes an unpredictable amount of resources.

The following resources are analysed concerning the possibilities to use them as credits or to determine the number of credits:

- CPU
- memory
- run-queue length of the CPU
- number of established connections

CPU and Memory: CPU and memory are often the main resources used by applications. Sometimes a limiting factor of a service is disk I/O or I/O in general. While this increases the service time of single requests, it does not limit the

number of clients at all. The CPU can process other requests during I/O if asynchronous I/O, multiple processes, or threads are used to handle multiple clients simultaneously.

The limited availability of memory and CPU will somehow limit the availability of a service. Can these metrics be used to determine the number of credits to be reported to the dispatcher? The answer is: not generally and not well. This is because the number and type of resources consumed by a request is unpredictable. For example, the question, how many requests will fit in 50 MB available RAM, can hardly be answered. Maybe it can be answered for fully analysed and static services with a fixed type of request. However, this ignores unpredictable resource consumption by the operating system itself.

A service dependent solution is not appreciated for a general purpose server load balancing system. Therefore, available CPU and memory are unfavourable metrics to determine credits, though they are still relevant.

Run-queue Length of CPU: Simulation studies have shown that the run-queue length of the CPU is a good metric for load balancing in homogeneous systems [Kun91]. In heterogeneous environments, the run-queue length has to be normalised by *speed factor*s. As mentioned before, these factors strongly depend on the application [SPL96].

Can the results of this research area be mapped to the calculation of credits? The run-queue may be a good metric to choose the least loaded server. However, it does not help to determine the number of free resources, unless one could specify a maximum length of the run-queue. Specifying an upper limit is the only way to tell the dispatcher how many requests it can safely forward to this server.

Established Connections: The number of established connections is a critical metric of connection oriented client-server applications, since the maximum number of connections is limited. Some commonly used scheduling algorithms in LVS rely on the number of connections (least connections, shortest expected delay). The main problem of this limit is that it neither reflects the server's capabilities nor prevents the service from overload.

Since the limit is just a theoretical value, using the number of established connections has the same drawback as the length of the run-queue. The number of future available connections cannot be determined from this load indicator. Additionally, an established connection is a very fuzzy load indicator because of (generally) unpredictable behaviour of the clients.

6.3.2.1 Communication Endpoints

A more general view on connections leads to the consideration of communication endpoints. Client-server-based services require a *communication endpoint* at both server and client sides[2]. The type of endpoint depends on the protocol layer and the kind of protocol. For example, the endpoints of the InfiniBand-Verbs layer (ISO/OSI-layer 4) are named *Queue Pair* (QP) while communication endpoints of socket applications are called *Socket* (layer 5). From the point of view of a TCP/IP socket-based application, an endpoint is a socket with a unique descriptor and an associated IP address and TCP port.

The basic assumption to use communication endpoints as a metric to calculate credits is that *an application consumes free communication endpoints at the same speed as it is able to process requests* (as long as there are incoming requests). This is reverse to the idea of the least connections algorithm, where an application releases connections at the speed of processing of requests.

The number of communication endpoints is usually limited by hardware, software, or protocol specific constraints. For example, the number of open TCP/IP sockets of an application is limited by the number of possible TCP ports[3], which is 65535. This is where credits come into place. A credit represents a free communication endpoint. Thus, the dispatcher cannot distribute more requests than available credits, which means not more than available communication endpoints. Therefore, SlibNet is able to drop or reject requests already at the dispatcher. This avoids the load of the backend servers getting worse under heavy load.

Reporting the number of free communication endpoints has a major drawback: it is not always possible to determine the number of free endpoints. For example, it will be an expensive operation, if not impossible, to determine the number of unused socket descriptors in a server application. An easier way to calculate credits is provided by the InfiniBand-Verbs layer. Each Queue Pair has to be allocated and registered by a central component, the communication manager (CM). Therefore, the CM is good candidate to count allocated but unused Queue Pairs that can be reported as credits. A credit-based load balancing on top of InfiniBand and Queue Pairs is developed in two masters theses [Fri06, Ryl07] and is also presented in [SS07b].

The socket-based approach is investigated in a masters thesis [Zin07]. This work presents a credit-based approach for TCP socket-based applications. The approach is designed, implemented, and evaluated. Credits are calculated from the free entries of the *socket backlog queue*. This queue contains the pending

[2]Since the client is not part of the load balancing system, the focus will be on the server side communication endpoint here.

[3]if no other operating system boundaries like the maximum number of open file descriptors further limit the amount open sockets

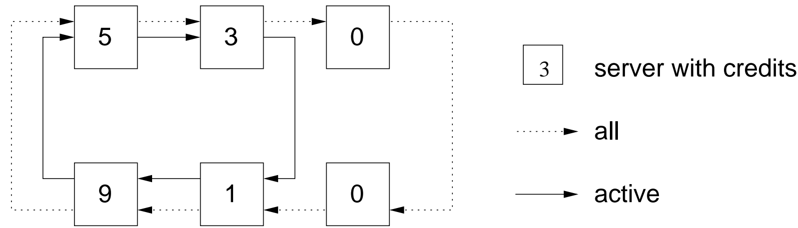

Figure 6.2: Rings of registered and available servers.

connection requests that are not accepted by the application. The increase and decrease of this queue indicates the current ability of the service to process requests. The length cannot exceed a limit that is specified by the application itself. Further requests will be dropped. This makes the socket backlog queue a good basis for credits. This is also published in [SSZL08].

6.3.3 Scheduling

One of the main design goals of SlibNet was to achieve a scalable system. In theory, this is achieved by the consequent use of $O(1)$ approaches for the scheduling and credit handling components. The scheduling can be described in short as round-robin among servers with available credits. This is illustrated in Figure 6.2 by the *active ring*. The outer ring (*all ring*) contains all servers that are registered. The scheduler decrements the credits of a chosen server by one. If there are no more credits available for a server, it is taken out of the schedule until it reports new credits.

SlibNet uses a combination of dispatcher-based and backend-based resource monitoring together with a push strategy to update credits. The backend servers calculate the number of credits periodically and report the resulting value. This makes the system a backend-based monitoring. Since the dispatcher removes consumed credits, it performs a kind of dispatcher-based monitoring.

Since the scheduling is based on two states of a server (available or not) and is independent of the absolute amount of credits, the reporting of credits will play the central role for the quality of the distribution. Some credit reporting strategies are analysed and evaluated by simulation below in Section 6.4.

The availability check can be an issue in conjunction with the use of RMA, since a new or newly available server has to be detected. An RMA write to the dispatcher memory will not be noticed by the dispatcher. Therefore, a detection of a status change from zero credits to non-zero or back has to be done. This is the only case for a notification message from the back-end servers to the dispatcher.

However, it can also be solved by the scheduler or by an extra detection/monitoring component in the background. For example, the scheduler can do a fixed

number of checks on each server selection triggered by an incoming request. This would keep the scheduling algorithm at the complexity of $O(1)$, while still allows for the use of RMA without notification. The dispatcher based check for new non-zero credits can cause delays in the detection of new servers or requires a kind of polling. The outcome will be a trade-off between overhead and delay in server detection. On the other hand, if the credit update is done by two-sided communication, every update would require data processing at the dispatcher. This reduces the scalability of the service.

6.4 Credit Reporting Algorithms

The current scheduling ignores the absolute amount of remaining credits at the dispatcher. Scheduling just over the availability of credits strongly depends on the algorithm used to report credits. This component of SlibNet is the focus of this section. The section presents some algorithms to report credits used to schedule requests. A simulative approach is used to compare various aspects of reporting credits.

6.4.1 Algorithms

Five algorithms to report credits are presented in this section. The algorithms are compared by their expected behaviour. A detailed analyses of their behaviour and the quality of distribution is deferred to Section 6.5.4.

6.4.1.1 Plain Full Credits Reporting

The simplest method reports every available resource. To determine the credits, a so-called *lookup* is done after a given number of processed requests. This algorithm uses a fixed *lookup interval* (li). The number of credits is calculated according to Equation 6.2. The queue of a server is limited by q_{max}. The current length of the queue is $q_{current}$.

$$credits = q_{max} - q_{current} \tag{6.2}$$

This number is reported to the dispatcher. Thus, for Plain Full Credits Reporting the *report interval* is equal to the lookup interval. The only tuning parameter of the algorithm is the lookup interval.

6.4.1.2 Low Watermark Reporting

This algorithm also calculates available credits within a fixed lookup interval (same as the Plain Full Credits Reporting). At each lookup, the server decides

to report or not. A report is triggered if the number of *expected credits* at the dispatcher is below a threshold (*low watermark*). In case of a triggered report, all available resources are reported according to Equation 6.2. This algorithm is explained in detail in the masters thesis of Sven Friedrich [Fri06].

The servers calculate the number of *expected credits* by decreasing the number of reported credits themselves if a request is taken from the queue. In this way, the servers anticipate the number of credits at the dispatcher. Since there is a skew between decreasing the credits at the dispatcher and at the server, the value is called *expected credits*.

Additionally, a minimum number of credits is specified to avoid trashing. This can happen if the number of credits is short above or equal to the low watermark, which results in more frequently triggered reports. Compared to the Plain Full Credits Reporting, this algorithm reduces the number of reports, while providing fine-grained reporting interval under heavy load.

The behaviour of reporting will not be as good as one could think, since the number of reports increases during heavy load phases. The additional overhead of reports will further exacerbate the load. This is why the minimum number of credits is introduced. On the other hand, more frequent reporting increases the server's availability and reduces the overall number of drops.

6.4.1.3 Soft Credits Reporting

This algorithm extends the simple report by reporting two credit values within a fixed interval. It reports so-called *soft credits* to tell the dispatcher how much work it wants. This can be seen as a recommendation to the dispatcher. The so-called *hard credits* represent the upper limit the server could handle if no other server has soft credits reported to the dispatcher. The number of hard credits is calculated according to the Plain Full Credits Reporting (see Equation 6.2).

The number of soft credits is based on the relation (r) between the lookup interval (li) and new requests (n) since the last lookup (see Equation 6.3). r is similar to the load factor ρ known from operational analysis [Jai91] and reflects the server's current request processing capability.

$$\begin{aligned} n &= q_{current} - (q_{old} - li) \\ r &= \begin{cases} \frac{li}{n} & \text{if } n \neq 0 \\ 1.0 & \text{otherwise} \end{cases} \end{aligned} \quad (6.3)$$

$$c_s = c_h * min(1.0, r) \quad (6.4)$$

$$c_{corr} = c_s * \frac{c_h}{q_{max}} \quad (6.5)$$

Equation 6.4 works well if $r \geq 1.0$. If r is smaller, the value of r does not influence the number of reported credits. This means that a server, which is temporarily under low load ($r < 1.0$) but still has a long queue, will report full credits and there will be no difference between soft and hard credits. To smooth out this process, the number of reported credits can be additionally reduced by a ratio calculated from the maximum length and the current length of the queue (see Equation 6.5).

In contrast to the Plain Full Credits Reporting and Low Watermark Reporting, the dispatcher has to be extended in order to check the soft credits first and fall back to hard credits if no soft credits are available. This is more complex, but still has the complexity of $O(1)$[4].

The Soft Credits Reporting is expected to improve the distribution of the load. The reduced number of reported credits will result in earlier *out-of-soft-credits*-situations for single servers. These servers are considered again after they report new soft credits or if no more servers have soft credits reported. The additional reported hard credits are necessary to provide all available resources to the dispatcher under heavy load.

6.4.1.4 Dynamic Lookup Soft Credits Reporting

This algorithm extends the Soft Credits Reporting by a dynamic lookup interval. The interval is defined to be proportional to the length of the queue. This reduces the number of lookups and reports under heavy load and improves the responsiveness under low load. A heavily loaded server with a long queue will have some more time to process the requests. A short lookup interval can be tolerated if the server has fewer requests to process. Equation 6.6 shows the calculation of the dynamic lookup interval. The correlation factor cf is a tuning parameter. li_{min} is used to avoid reporting after each requests under low load.

$$lookup = MAX\left(li_{min}, q_{current} * cf\right) \qquad cf \in (0.0; 1.0] \tag{6.6}$$

It is expected that the dynamic report interval will improve the quality of the distribution because loaded servers are taken from the schedule for a longer time. But the long report interval is also used for hard credits. Therefore, this algorithm is expected to drop more requests under heavy load.

6.4.1.5 Dynamic Pressure Relieve Algorithm

The *Dynamic Pressure Relieve Algorithm* is the result of the above analyses of the lookup and report interval and the use of soft and hard credits. This algorithm uses

[4] The round-robin ring that contains all available servers with soft credits will be empty and the next server will be chosen from the round-robin ring of hard credit available servers.

a fixed interval to report hard credits and a dynamic interval to report soft credits. In case of a soft credit report, hard credits are also reported and the counter for the interval of hard credits is reset. In this way, the algorithm works like a pressure relieve valve that additionally can be triggered dynamically by hand. Hence, the name *dynamic pressure relieve* algorithm.

This algorithm is expected to perform best among all presented algorithms. Due to the dynamic reporting interval for soft credits, a good distribution should be achieved. The fixed interval to report hard credits will reduce the number of dropped requests under heavy load. The only drawback of this algorithm is the increased number of reports because more or less two separate reports are performed.

6.5 Evaluation of Algorithms

Each algorithm is implemented, evaluated, and compared to other algorithms. Since all credit algorithms and the weighted round-robin scheme are $O(1)$, the complexity is the same. The evaluation focuses on the quality of the distribution as described in Section 6.1.3.

6.5.1 Factors

Before the algorithms are analysed with a simulation, the experimental setup is prepared by determining the primary factors that affect the performance and quality. The following factors are found:

- request arrival rate or distribution
- request processing rate or distribution
- the algorithm
- lookup interval (absolute, relative for dyn-lookup)
- report interval (if different from lookup)
- server weights (for comparison with weighted round-robin)
- server speeds
- number of servers
- low watermark
- maximum length of the queue

The evaluation of the credit report algorithms is done by comparing the report algorithms to the Plain Full Credits Reporting and to weighted round-robin. A comparison to more sophisticated algorithms like *least connections* or *shortest expected delay* is not yet done because these algorithms use a less scalable $O(n)$

scheduling algorithm. Although, this comparison should be subject of future work. Round-robin and weighted round-robin use an $O(1)$ algorithm. Thus, the credit algorithms are compared to round-robin.

6.5.2 Primary Factors

The goal is to compare the quality of the distribution using different algorithms in homogeneous and heterogeneous environments. Therefore, the only factors to be varied are the server speed and the algorithm.

server speed The server speeds determine the type of environment. Homogeneous environments are represented by the same speed for each machine. To simulate heterogeneous environments, the server speeds are unequal. The comparison to weighted round-robin requires experiments with exact and non-exact weights to see the benefits of the credit-based load balancing if weights are not exactly determined. This is done by using fixed weights of 1 : 1 for homogeneous and 1 : 2 for heterogeneous environments. The server speeds are set to 1 : 1 and 1 : 2. The simulation of inexact weights is done by using 1 : 1.05 and 1 : 1.1 for homogeneous and 1 : 2.1 and 1 : 2.2 for heterogeneous environments. This simulates deviations of 5 % and 10 %. Note that the faster server will get the additional amount of requests in the heterogeneous environment with inexact weights.

The number of setups sums up to 5 for weighted round robin and 2 for each credit algorithm since differing weights can not be specified for the credit algorithms.

algorithm All algorithms are evaluated. Including the weighted round-robin algorithm, this results in 6 experiments per environment.

As a result of the specified primary factors and their values, 16 experiments have to be conducted.

6.5.3 Secondary Factors

All algorithms will be evaluated using a 100 % loaded system. The arrival interval of requests is fixed at 10000 units of time to have the same granularity of time for all experiments. Thus, the request processing time is adjusted respectively. The processing time of 2 homogeneous servers will be internally calculated as 20000 units of time to get 100 % load. If heterogeneous servers have speed factors of 1 : 2, their average processing times are 15000 and 30000. Arrival intervals and processing time of requests are generated according to a negative exponential distribution. The number of servers is fixed and set to 2.

Maximum Length of the Queue The maximum length of the queue determines the maximum number of credits. It makes no sense to report more credits than available slots in the queue. To make calculations more convenient the maximum is set to 100. This is fixed for all servers in all experiments of the simulation.

Lookup Interval The overhead of a lookup includes the costs to determine the number of credits to report. Some examples of such overheads are the required calculations, system calls, or management overhead. Thus, the lookup interval is one of the important impacts on the efficiency of the credit report algorithm. A heavily loaded server should use a coarse lookup interval to leave as many resources as possible to the service. But a coarse-grained lookup is unfavourable during phases of low load. The system will slowly react to rapid changes of load. Thus, a short lookup interval is favourable under low load. A fixed lookup interval can only be a compromise between overhead and reactivity. Therefore, a dynamic lookup interval is implemented in some algorithms.

The absolute value of a fixed interval is set to 10. This is chosen to lookup for credits to report after 10 % of the maximum number of credits. The dynamic interval is set according to Equation 6.7.

$$li = max(10, q_{current} * 0.5) * q_{max} \tag{6.7}$$

In case of the Dynamic Pressure Relieve Algorithm, the 0.5 in Equation 6.7 is set to 1.0. This results in early *out-of-soft-credits* situations and thus a better distribution of requests. In contrast to the Dynamic Lookup Soft Credits Reporting, it should not increase the overall drops because of the fixed interval for hard credits.

Report Interval The number of reports differs from the number of lookups if the Low Watermark Reporting decides not to report any credits. Basically, the overhead of a report is the cost of the network transfer and protocol processing. Each server has to transmit the number of credits on each report. Here, the same considerations as for the lookup interval apply.

While the server has to process its own credit reports, the dispatcher has to process the credit reports of all servers. Hence, the number of reports will have an impact on the scalability of the system. This impact is the case for RDMA, since the reports via RDMA will not require any processing at the dispatcher.

Minimum Number of Credits This factor is only relevant for the Low Watermark Reporting (see Section 6.4.1). The number of reports can be reduced if a minimum of credits to report is specified. If there are not enough free resources, no credits can be reported until the lookup detects a sufficient amount of available

parameter	values
algorithm	5 credit algorithms, round-robin
server weights	1:1, 1:1.05, 1:1.1, 1:2, 1:2.1, 1:2.2
server speeds	homogeneous (1:1), heterogeneous (1:2)
arrival rate	10000
avg. processing rate	10000
lookup interval	10 or dynamic
number of servers	1, 2
low watermark	10
minimum credits	20 (Low Watermark Reporting only)
max queue length	100

Table 6.1: Parameter overview of simulations.

resources. A smaller threshold can result in faster toggling between zero and non-zero credit situation for this particular server. The duration of zero and non-zero credit situations will be extended by a high threshold. Using a value of 20 is a result of a separate study with the Low Watermark Reporting in Section 6.5.5.

All parameters used in the simulation are summarised in Table 6.1.

6.5.4 Simulation

Simulation studies are applied to the algorithms to decide the best algorithm to implement. Additionally, the simulation results were helpful to better understand the behaviour of the particular algorithms.

A small simulation program is designed to simulate the behaviour of the four different algorithms. The following features are implemented:

- All above described reporting algorithms are included.
- Two distribution functions (negative exponential and uniform) are included to create traces of incoming requests and service times of each request.
- The number of servers is configurable.
- The software can generate traces of the length of the queues and credit reports for detailed analysis. It also can print summaries for statistical analysis.
- Homogeneous and inhomogeneous setups are possible. This is done by specifying their weights.

There are several counters and statistics available:

- The overall number of requests failed due to complete zero-credit situations. It will be the primary counter to compare the algorithms (dropped requests).

- The overall average length of the queues. This counter is an indicator for the average answer time of requests. Also the average answer times are calculated.
- The number of individual zero-credit situations of each server. Each individual drop represents a deviation from a plain round-robin scheduling.
- The number of individually handled requests of each server as an indicator how the requests were distributed.
- The individual average length of the queues including standard deviation. The length of individual queues should correlate with the capabilities of the servers to achieve similar answer times for each request.

At the initialisation of the simulation, a random trace of requests is generated. Each request will have a specified normalised service time[5]. The random trace depends on a seed that is taken from the command line. This is important to compare different algorithms based on the same trace of requests.

The simulation is implemented as an event-based simulation with two types of events: incoming requests (*request event*) and *service events*. Each event contains a timestamp to trigger the corresponding action. Furthermore, it contains a process or server ID to collect individual statistics. Request events trigger the dispatcher to schedule the request. Service events trigger the server to pick up the next request from the queue. As long as there are requests in the queue of a server, service events are generated after the current request is processed.

The scheduling of requests to a queue is done at a request event. The number of credits is reduced and the length of the queue is increased for the selected server. Each request event sets the timer for the next request according to the interval given in the trace.

The service times t_i of a server i is calculated according to Equation 6.9 on the basis of the time of slowest server t_s. If t_s is calculated according to Equation 6.8, the system will run under 100 % load (λ is the average arrival rate of requests). Since the slowest service time is a command line parameter of the simulator, t_s has to be calculated externally.

$$t_s = \frac{1}{\lambda} * \sum_{k=1}^{n} \alpha_k \tag{6.8}$$

$$t_i = \frac{t_s}{\alpha_i} \tag{6.9}$$

[5] The service time of the server with the speed factor 1 is used.

6.5.4.1 Statistics

It is important to explain the details of how statistics are collected to make the simulation more understandable. First there is a specified *hot phase* to collect data. The first (x) and the last (y) requests are ignored to have a warm-up and a cool-down phase. The calculation of x and y is shown in Equation 6.10. The values are retrieved from observation of detailed traces of queues.

$$x = \frac{requests}{4}; \qquad y = \frac{requests}{8} \tag{6.10}$$

- Dropped requests are counted during the *hot phase*. A drop happens if a request can not be scheduled due to a complete out-of-credit situation at the dispatcher.
- The calculation of the average length of the queue is done at every time tick. This makes the average queue independent from the duration of requests and their processing time.
- The average answer time of a request is calculated when the request is scheduled to one of the servers. Since the load balancing system has no influence on the clients network interconnect and the path through the network, only the time to wait in the queue and processing the requests are considered in the simulations. This value influences the global and the individual statistics.
- Another counter is the number of requests handled by individual servers. This offers a way to evaluate the distribution of requests according to given server weights.

6.5.4.2 Simulated Testbed

The simulated testbed is specified by the secondary factors (see Section 6.5.3). Statistics are calculated from 50 different but fixed random traces, generated from 50 stored random seeds. The distribution of arrival intervals and service times is set to negative exponential.

For load dependent analyses, the mean request interval is varied from 9000 to 11000 to simulate high (9000) and low (11000) loaded situations.

The idea of the burst length test is as follows: Starting from an idle system, a particular load of incoming requests is put into the system. In this case, the same 50 traces as for the other tests are used. Until the first drop, the number of requests handled is counted and taken to compare the algorithms. The higher the number of requests the better is the ability of the algorithm to handle bursts.

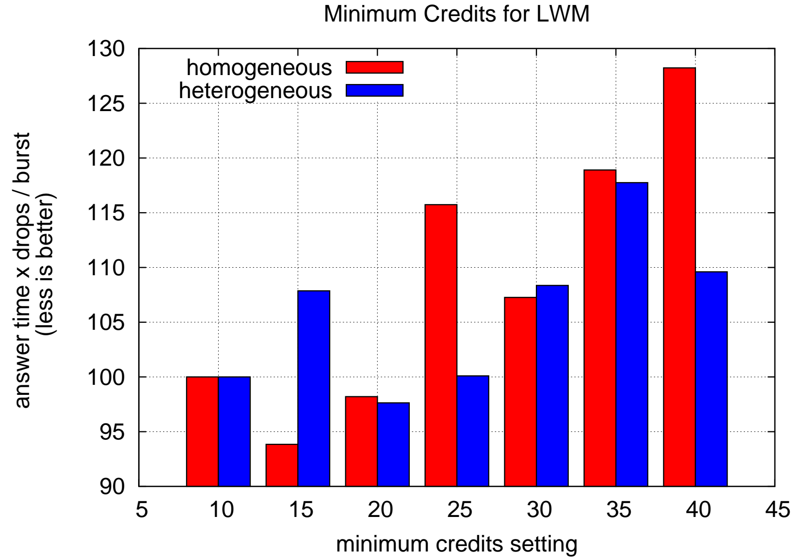

Figure 6.3: Impact of the minimum credits on Low Watermark Reporting.

6.5.5 Minimum Number of Credits

The Low Watermark Reporting requires an additional parameter. This parameter will be determined by experiments instead of guessing. The minimum number of credits is required to trigger a report after a lookup. The value is called c_{min}. The determination is done by simulating two homogeneous servers (1 : 1), heterogeneous servers (1 : 2), and varying the minimum value. Starting from 10 (equals the low watermark setting and the static lookup interval) up to 40. A higher value is expected to further reduce the overall quality.

The results of the tests with the Low Watermark Reporting are shown in Figure 6.3. The evaluation is done by normalising the dropped requests (*drops*), the answer time (*answertime*), and the burst length (*burst*) to the case where the minimum number of credits equals the low watermark (10). By combining these three normalised measurements, each of the parameters has an equal impact on the quality measure (see Equation 6.11). The detailed measurements can be found in Table C.1 in Appendix C.1.

$$f(drops, answertime, burst) = \frac{drops * answertime}{burst} \tag{6.11}$$

It can be seen that the results are better (lower values) if the minimum number is a multiple of the low watermark. The reason is that the low watermark is also used as the lookup interval.

$f(drops, answertime, burst)$ is best when $c_{min} = 20$. It has the best results

	Plain	LWM	SH	DL-SH	DPR	WRR
dropped	285.9	318.9	285.9	385.3	289.84	263.16
answer time	234440	226653	234440	187729	234055	250920
reports	2971	989.5	2971	1940.5	3315.7	0
drop-time	1.0151	1.0946	1.0151	1.0954	1.0275	1.0

Table 6.2: Simulation results for a single server setup under 100 % load.

combined for the homogeneous and the heterogeneous experiment. Therefore, 20 is chosen for further experiments. The chosen value is not completely independent from the other factors like the server speed, but since this algorithm is expected to be suboptimal, it is not fully optimised.

6.5.6 Single Server

Using a single server, the credit algorithms are compared to round-robin (i. e. just forwarding the requests to the server). There will be a different behaviour between the credit algorithms due to different lookup and reporting intervals. Also, a deviation from forwarding will occur. Depending on the report interval, the server may not have reported available credits in time, while the plain forwarding will always consume every free slot in the queue. Therefore, the credit algorithms are expected to perform a little worse than the forwarding.

To evaluate this, a simulation study is done according to the above evaluation testbed but with a single server (Section 6.5.4.2). Table 6.2 shows the results of the simulations. Round-robin performs best in terms of the number of dropped requests. LWM produces the lowest number of reports among the credit-based algorithms.

The evaluation of the answer time is a little tricky since the number of requests in the queue depends on the number of dropped requests. Therefore, the answer time is correlated to the number of dropped requests ($droptime_{algo}$ in Equation 6.12). This is done by normalising both measurements to the result of weighted round-robin ($answertime_{wrr}$, $drops_{wrr}$). The normalised values are multiplied afterwards. The lower the product, the better the algorithm performs.

$$droptime_{algo} = \frac{answertime_{algo}}{answertime_{wrr}} * \frac{drops_{algo}}{drops_{wrr}} \qquad (6.12)$$

The results for a single server are shown in the last row of Table 6.2. This correlation identifies round-robin[6] as the best algorithm in terms of drops and answer time on a single server even though it does not have the best answer time.

[6] which is just forwarding

6.5.7 Two Servers

Using two servers, the load has to be distributed among the available servers. Thus, the advantage of round-robin (or forwarding) will disappear.

The Figures 6.4 to 6.7 show the results of the simulations of homogeneous and heterogeneous servers. They compare the credit algorithms and weighted round-robin with exact and non-exact weights. The exact numbers are moved to Appendix C.1.

The algorithms are compared by the parameters that determine the quality of the load balancing specified in Section 6.1.3. These parameters are the answer time, the number of dropped requests, the length of bursts without a dropped request, and the number of reports.

The abbreviations of the algorithms are as follows:

Plain Plain Full Credits Reporting
LWM Low Watermark Reporting
SH Soft Credits Reporting
DL-SH Dynamic Lookup Soft Credits Reporting
DPR Dynamic Pressure Relieve Algorithm
WRR Weighted Round-Robin with weights set to the server's speed factors (which means exact weights concerning the processing capability of the servers)
WRR 5 Weighted Round-Robin with the weights of the second server differing 5 % from the speed factor of server 2
WRR 10 Weighted Round-Robin with the weights of the second server differing 10 % from the speed factor of server 2

6.5.7.1 Dropped Requests

Figure 6.4 compares the algorithms by the number of dropped requests during the *hot phase* of the simulation.

From the results of homogeneous environments, it is obvious that the credit-based algorithms perform better or equal to round-robin. The improvement is possible because of the self-adaptivity of the credit-based scheduling. The credit-base algorithms respect inhomogeneous requests.

Because of the approach to maximise the number of credits at the dispatcher, the Plain Full Credits Reporting and Low Watermark Reporting perform worse in the heterogeneous environment. The dispatcher runs out of credits for a particular server very late. The result is an imbalance because of the underlying round-robin scheduling among servers with available credits. This issue is addressed by the other credit-based algorithms. Significant improvements are achieved by Soft Credits Reporting and Dynamic Pressure Relieve Algorithm.

For a single server, a dynamic or longer lookup interval was a problem. This has a negative impact on the number of drops when using two servers, too. In both

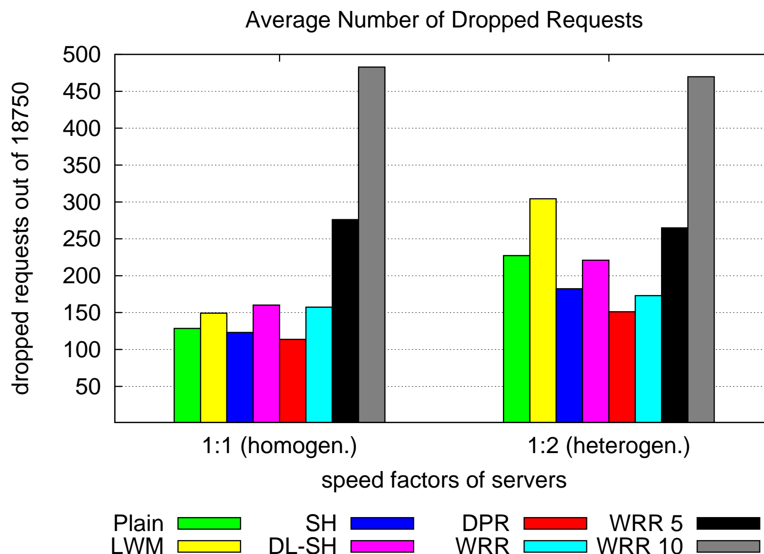

Figure 6.4: Average number of dropped requests out of 18750

the homogeneous and the heterogeneous setups, the Dynamic Lookup Soft Credits Reporting performs worse than Soft Credits Reporting because of the longer interval. Due to Equation 6.6 at Page 147, the average interval is longer than the fixed interval of the Soft Credits Reporting.

If the weights for round-robin differ by 5 % or 10 % from the server speeds, the number of drops significantly raises. Even in heterogeneous environments, the credit-based algorithms outperform weighted round-robin with inexact weights.

6.5.7.2 Answer Time

The second important result is shown in Figure 6.5. It shows the average answer time in homogeneous and heterogeneous environments.

Comparing the answer time by their absolute value is not possible because the number of handled requests is different for each algorithm. Therefore, the same evaluation steps as for the single server is done.

While the Plain Full Credits Reporting and Low Watermark Reporting had a low number of dropped requests, the Dynamic Lookup Soft Credits Reporting drops as many requests as the round-robin (see Figure 6.4). But the answer time of round-robin is 22 % longer (see Figure 6.5). This indicates that a dynamic interval improves the distribution of requests.

The main issue to achieve a good distribution is to differ from round-robin as early as possible, but, only if there's an imbalance of load. This can be achieved only if the dispatcher is out of credits of servers under higher load. Thus, every

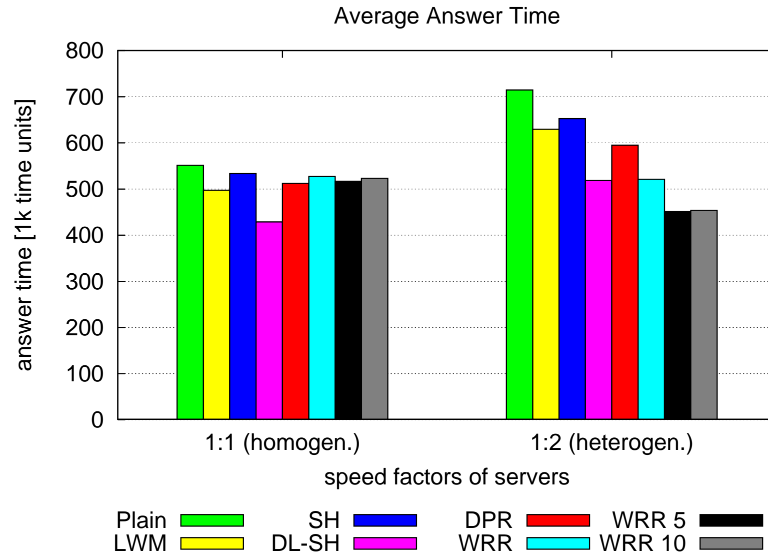

Figure 6.5: Average answer time.

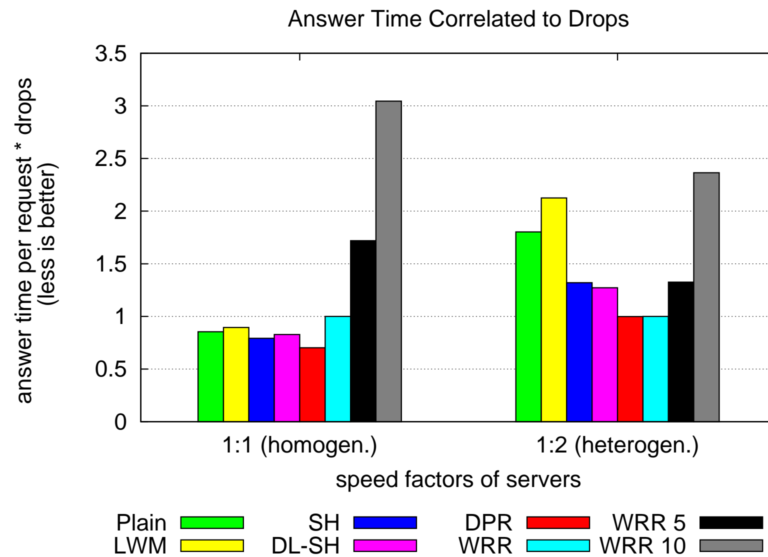

Figure 6.6: Answer time correlated to number of drops normalised to WRR.

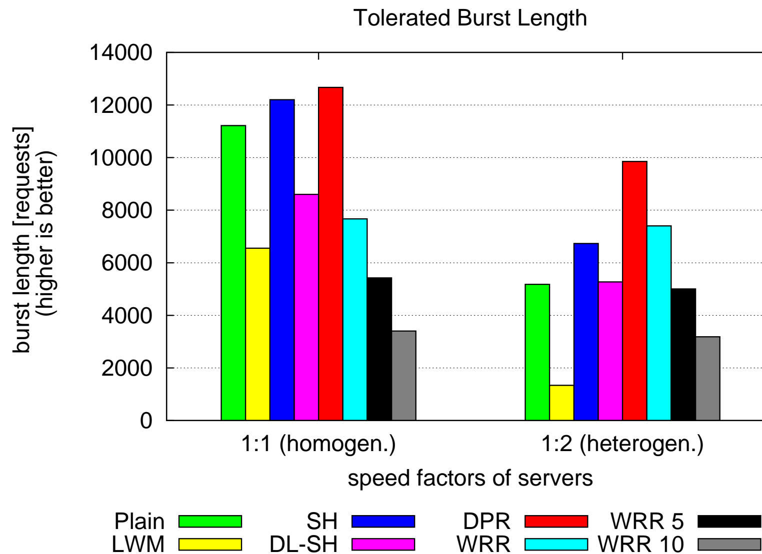

Figure 6.7: Average tolerated burst length

mechanism to avoid out-of-credit situations has a negative impact on the distribution. But, a global out-of-credit situation has to be avoided since it increases the number of drops.

According to the drop-time-correlation in Figure 6.6, the Soft Credits Reporting and the Dynamic Lookup Soft Credits Reporting perform very similar. The former has a low number of dropped requests. The latter has a better answer time. The performance of the Dynamic Pressure Relieve Algorithm algorithm is best among the simulated algorithms.

The answer time of weighted round-robin with inexact weights is improved because the servers have to handle fewer requests. The Dynamic Pressure Relieve Algorithm performs similar to the weighted round-robin algorithm with exact weights in the heterogeneous setup. This is a good result, because only the availability of reported credits at the dispatcher adapts a simple round-robin scheduling to heterogeneous server speeds.

6.5.7.3 Burst Length

Figure 6.7 shows that bursts can be handled better with a short and fixed lookup interval. This is derived from the comparison between Soft Credits Reporting and Dynamic Lookup Soft Credits Reporting.

If the load is distributed better, the length of tolerated bursts also increases. This can be seen from the improvements of Soft Credits Reporting over Plain Full Credits Reporting. The former achieves a better distribution by using soft credits.

	DPR	WRR	DPR	WRR
failed	44.7	1315	0	1241
queue	40.8	47.6	37.5	37.3
answer time	4 825 350	4 894 690	3 851 840	3 790 920
drop-time	0.0335	1.0	0.0	1.0

Table 6.3: Comparison between credit-based and WRR using 19 servers.

Further, the Dynamic Pressure Relieve Algorithm significantly outperforms all other simulated algorithms in homogeneous as well as in heterogeneous environments. This is because it combines frequent reports with the best load balancing among the compared algorithms.

6.5.7.4 More Servers

A a comparison between the Dynamic Pressure Relieve Algorithm and weighted round-robin is performed for 19 servers to test the credit-based algorithms with more than two servers. Because of the larger number of servers, the overall number of incoming requests is increased to $300,000$. A homogeneous and a heterogeneous setup is tested using exact weights for weighted round-robin. The heterogeneous environment uses the server speeds $1 : 1.5 : 2 : 2.5 : \cdots : 9.5 : 10$ (hence the 19 servers). Each server of the homogeneous setup has an average service time of 190 000 units of time. The slowest server of the heterogeneous setup processes requests in 1 045 000 units of time (average)[7].

The results in Table 6.3 show that Dynamic Pressure Relieve Algorithm is able to outperform weighted round-robin in both environments. With two servers, weighted round-robin performed similar to the credit-based algorithm. But, with 19 servers the Dynamic Pressure Relieve Algorithm wins.

This makes the credit-based load balancing even more advantageous since determining weights for a large number of servers requires a lot of effort.

6.5.8 Summary of Important Impacts

The previous sections were to compare the credit-based algorithms as a whole. This section is to find important factors and their influence on the quality of the distribution of requests. The factors to observe are the lookup interval, the reporting interval, and the number of credits to report since these factors can be extracted from the five algorithms.

[7]The service times of the other servers are calculated according to their speed factors.

6.5.8.1 Lookup and Reporting Interval

The only algorithm with differences in lookup and reporting is the Low Watermark Reporting. Since it has shown to be not the best algorithm and the report of credits is omitted only under special conditions, report and lookup intervals are taken as synonyms.

The influence of different lookup intervals can be derived from a comparison between algorithms with fixed and with dynamic lookup intervals. The dynamic lookup interval algorithms use a larger interval under heavy load, which results in a higher number of zero-credit situations and potentially a higher number of dropped requests. This can be seen from Figure 6.4. Comparing Soft Credits Reporting and Dynamic Lookup Soft Credits Reporting, the number of drops is significantly higher with the latter algorithm.

A longer lookup interval increases the number of zero-credit situations for a particular server. This improves the distribution of requests because the scheduling differs from the underlying round-robin among available servers.

The report interval has the same impact on the length of possible bursts. Short intervals increase the number of requests before the first request has to be dropped.

The Dynamic Pressure Relieve Algorithm is the result of combining good burst tolerance and few drops by a short interval with longer (dynamic) report-intervals to improve the distribution.

There is another effect of the reporting interval that is not explained yet. The server itself consumes a number of requests from the current queue until the next report. The queue will be filled to 100 %. But, since no further requests will be forwarded, the length of the queue will be reduced by the credit report interval. The length of the queue will more or less alter between full and full minus credit report interval. The average will be at about a full queue minus half the credit report interval. Thus, the average answer time will also be reduced..

At least one important impact is not included in the simulation. It is the implied overhead of reporting the credits. It usually consists of network communication, reading the length of the queue, and all its impacts on the involved machines. CPU usage, network traffic, and interrupt handling are some of the overheads not observed here. Since it is overhead, the number of reports should be as low as possible. The number of reports is reduced by two mechanisms. First, to not report credits at every lookup and, second, to extend the lookup interval. Both have their advantages and drawbacks seen in the simulation results. Unfortunately, the best performing algorithm (Dynamic Pressure Relieve Algorithm) has the highest number of reports.

6.5.8.2 Number of Credits

The number of credits that are reported has the most important impact on the quality of the distribution. It was noted above that zero-credit situations of single servers are essential to improve the balancing of requests. Thus, reporting only a part of the available credits is useful. The quality is improved if the reported value depends on the current growth rate of the queue. This is indicated by the simulation result of Soft Credits Reporting. This algorithm uses a two-step strategy by reporting soft and hard credits, because the reduced number of soft credits would result in an intolerable higher number of dropped requests.

6.5.9 Limitations of Simulation Results

Each simulation is a model of the real problem and this model does not include all aspects of reality. The following aspects are not included in the simulation:

6.5.9.1 Unimplemented Aspects

- The time between credit reduction at the dispatcher and the queue entry at the server is ignored. This delay can result in reports with to much credits. Some credits are not recognised at the server but are already consumed at the dispatcher.
- The costs of a lookup are not included. Therefore, the performance of the backend servers could be reduced in comparison to round-robin.
- The time to report credits is not respected. If this time is long, zero-credit situations can happen more often, since it would have the same effect as a long lookup interval.
- The scheduling time is not included. Thus, the influence of the scheduling algorithm (credit-based or plain round robin) has no impact on the simulation results. This is a expected to be a minor impact since the underlying scheduling algorithm has a complexity of $O(1)$.

6.5.9.2 Multi-Process Applications and Wait Time

The simulation does not take into account the time that a connection is kept established. If a server can handle a limited number of concurrent connections and established connections are kept open but idle for some time, may be the queue is processed slower than the server can process the requests itself. This also holds for the time, an open socket is in the state `TIME WAIT` in practise.

This particular behaviour was observed during measurements with the Apache2 web-server application (see Section 6.6). In those cases, the queue is processed at

the speed that is determined by the behaviour of other clients (disconnect/continue).

6.5.10 Summary

Five credit-based algorithms were presented and evaluated by simulations in this section.

The conducted simulations were helpful to better understand the behaviour of the particular credit reporting algorithms and their influence on the distribution of requests. The simulations have shown that it is worth to rely on two separate values (soft and hard credits) because the most important factors of credit reporting (number of credits and reporting interval) have contrary effects.

Reporting fewer credits improves the distribution to reduce the answer time, especially in heterogeneous environments. This is because each time the dispatcher has no credits from a particular server, the scheduling starts to differ from round-robin. The simulation results show that algorithms with earlier zero-credit situations for particular servers produce a more balanced distribution of load. But, a few available credits at the dispatcher increase the risk to completely drop requests.

Using a long reporting interval improves the distribution because of the same reason as the fewer reported credits (by early out-of-credit situations of single servers). Longer intervals help to reduce the number of reports.

The best simulation results were achieved with a two-level scheduling. Primary scheduling based on a few and dynamically reported (soft) credits improves the distribution and helps the system to better adopt to heterogeneous environments. A secondary (fallback) scheduling based on a frequently reported maximum number of credits makes the service more robust to tolerate bursts and reduces the overall rate of failed requests.

This section determined a good and promising credit-based algorithm that can be used together with one-sided communication.

6.6 Implementations of SlibNet

Chapter 4 introduced the theoretic concepts of one-sided communication. A credit-based scheduling scheme was developed in the previous sections. This section answers the question if the concepts of one-sided communication and credit-based scheduling are implementable.

This work was done within the context of student theses [Fri06, Ryl07, Zin07]. However, since they show that the above theoretic analysis are implementable, it is presented here. Some *historic* steps towards the current state are also explained.

6.6.1 SlibNet: InfiniBand-Based Credit SLB

The first implementation of SlibNet did not focus on one-sided communication. It showed the applicability of credit-based scheduling. The number of free resources in InfiniBand networks was determined from the number of unused queue pairs. Since the communication manager (CM) as a central component on each server is able to calculate this value, a credit-based load balancing system was designed to study the behaviour in comparison to round-robin scheduling. This was the diploma thesis of Sven Friedrich [Fri06, FSS05].

The implemented approach was restricted to InfiniBand networks. But, the thesis proofed the concept of the Low Watermark Reporting.

6.6.2 SlibNet: Credit-Based Scheduling

The thesis of Olaf Ryll [Ryl07] took the next step towards a credit-based load balancing system with one-sided communication. An implementation of automatic registration of backend servers and the corresponding credit-based round-robin scheduling was designed and implemented.

This thesis made the first use of RDMA write to report credits. It shows that reporting credits and scheduling can be done with very few synchronisations.

- First, the data has to be written exclusively since two processes access the data as explained in Section 6.2.2. Since credits can be represented by a very small amount of data, the system can rely on atomic access to the memory when writing single bytes or words.

- If servers report new credits but were taken out of the schedule because of no credits, there has to be some notification. Otherwise, the server would stay outside the active ring (see Section 6.3.3 and Figure 6.2 on Page 144).

  In the current concepts and implementations, this notification is omitted. The dispatcher keeps track of the reported credits. The implementation in [Ryl07] uses a separate thread to keep track of the rings according to the number of reported credits. This is improved in the follow up work of Jörg Zinke [Zin07] by using a *look ahead pointer* (see below).

Another outcome of these two works is that establishing a reliable connection over InfiniBand verbs takes a lot of time (about 300 μs measured with OFED 1.1). This is not acceptable if connections have to be established frequently or on a critical path of data. Establishing connections over InfiniBand is only required when a new backend server is registered. Further details are published in [SS07b].

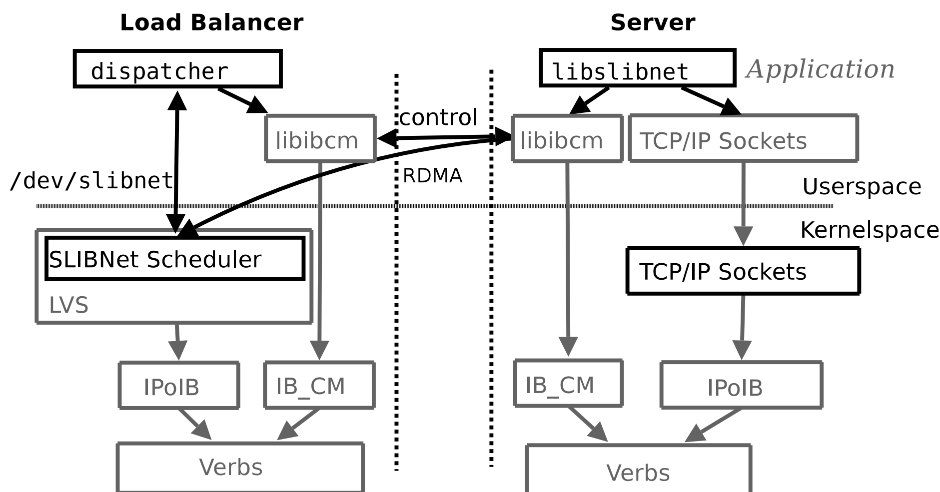

Figure 6.8: Architecture of SlibNet for Socket-based applications [Zin07].

6.6.3 SlibNet: Socket-Based Credit SLB

Jörg Zinke performed the next step in his masters thesis [Zin07]. He designed and implemented a credit-based load balancing module for the Linux Virtual Server (LVS) [Zha00, Lin07b]. LVS is a popular kernel level load balancer that is integrated into the Linux kernel. Performance measurements in [FKSS06] show that LVS can handle load balancing nearly at the speed of forwarding for small numbers of servers. In conjunction with IP over InfiniBand, this software is able to distribute the load of TCP/IP socket-based services.

Figure 6.8 shows the architecture of the implementation[8]. Two components are implemented at the dispatcher (Load Balancer): the kernel-level scheduling module for LVS and a user-level process to handle incoming InfiniBand connections from new servers. The memory area for credits is mapped from kernel to user space via the device `/dev/slibnet`. Therefore, the server-side lib `libslibnet` directly writes the credit updates to the kernel memory of the dispatcher.

The current server implementation requires a small kernel patch in order to read the length of the socket's backlog queue via a `getsockopt`-call. The implementation transparently works for TCP/IP-Socket applications. It modifies the socket API call of `listen`, `bind`, and `accept`. The modification of `bind` is required to prepare the library to report credits and count accepts.

If the server starts listening on the port, it establishes an InfiniBand reliable connection to the dispatcher and performs the first report of credits. After this

[8]The original picture is taken from [Zin07]. The only modification is the translation of the German word *Anwendung* to the word *Application*.

registration, the dispatcher can distribute incoming requests from clients to the server.

The reports are triggered by counting the number of accepts. After a specified number of accepts (see Section 6.4), the credits are calculated and reported to the dispatcher's memory.

The update of the rings of servers at the dispatcher is done by the scheduler itself. It uses a kind of look ahead strategy. If a server is picked up, a check is performed for a fixed number of registered servers. A drawback of this scheme is that some servers are considered for scheduling a little late. However, it introduces less overhead compared to a thread and keeps the scheduler's complexity of $O(1)$.

This work shows that the credit-based scheduling works together with credits reported via one-sided communication (that is, RDMA in case of the InfiniBand implementation). The presented approach works for the large class of TCP/IP socket-based client-server applications.

6.6.4 Problems with InfiniBand

The concepts of InfiniBand RDMA fit well to the concept of reporting credits without synchronisation. However, during the implementation, there were several problems that are introduced by the OFED stack implementation.

An important issue is implied by the nature of common client-server applications. For example, the server process of Apache creates child processes to process several clients in parallel. Each of these processes can accept new connections from the same socket backlog queue. Since each process calls its own accept, accepts have to be counted globally. By using a small chunk of shared memory, this issue can easily be solved. However, it requires mutual exclusion for the access of the counter.

Another issue arises from the fact that any of the child processes can be the one to report the current amount of credits. Each of the processes must be able to write to the dispatcher's memory. During the initialisation of the service socket, a reliable connection is established. The implementation of InfiniBand is not able to inherit the open connection to the child processes. Therefore, every child process has to establish a new connection to the dispatcher. This is a performance issue for the current implementation since processes are dynamically created and stopped.

One result of this issue is a limited number of concurrent processes. Each process requires a connection. Each connection requires a Queue Pair (according to the OFED Mailing list[Ope] about 1 MB per Queue Pair). And each Queue Pair consumes resources. Especially, at the dispatcher this implementation will not scale with a large number of servers.

A second issue arises from the mapping of memory between the user and kernel at the dispatcher. For the handling of incoming InfiniBand connections

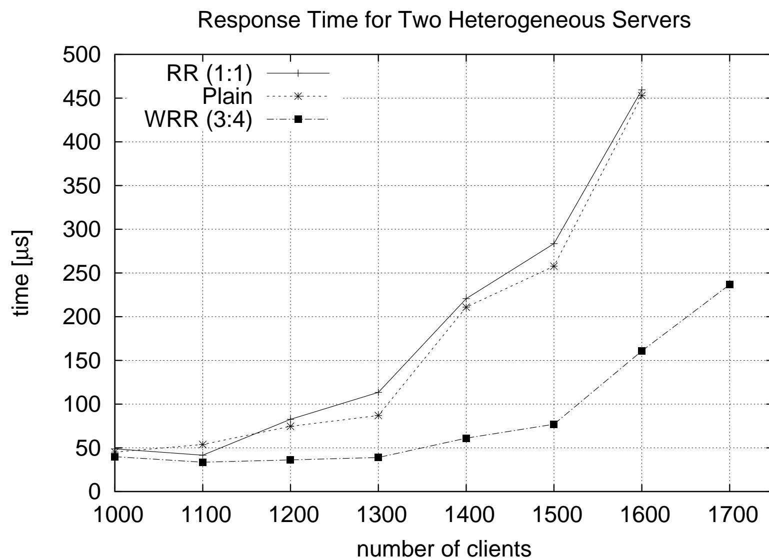

Figure 6.9: Response time of Round-Robin with and without weights compared to SLIBNet.

it is easier to map the user-space memory into the kernel. In this case the state of the mapped memory collides with requirements of the OFED implementation. Memory, intended for RDMA operations, must not be previously mapped, since the kernel part of the OFED stack needs to map and lock it for its own purposes.

6.6.5 Evaluation

In the context of a case study by Janette Lehmann, several scheduling algorithms of LVS were analysed and compared. The results of the comparison between LVS and the implementation of the Plain Full Credits Reporting have confirmed the simulation results of the Plain Full Credits Reporting algorithm in Section 6.5. This is presented in [SSZL08]. Currently, only an implementation of the Plain Full Credits Reporting algorithm exists. In homogeneous environments, the credit approach is able to outperform round-robin. In heterogeneous environments, weighted round-robin performs better. This confirms the simulation results (see Figure 6.9).

A comparison to more sophisticated algorithms is subject of future work. It was deferred because algorithms like least connections or shortest expected delay have a complexity of $O(n)$. The credit-based scheduling uses a strict $O(1)$ approach.

The conclusions of the case study are:

- The time to establish a connection does not depend on the load of the server.

The kernel handles the SYN and ACK packets by interrupts. This has consequences for the Least Connections (LC) algorithm, since it counts the established connections at the dispatcher. The dispatcher has no information whether the connection is accepted by the service or not. This indicates an advantage for the credit-based algorithm, because it counts the number of pending connections in the backlog queue of the service.

- Round-robin, LC, and Shortest-Expected-Delay (SED) perform very similar if the service has a very low load, even in heterogeneous environments. Thus, the underlying unweighted round-robin of the credit-based load balancing is sufficient.

- The self-adaptivity of LC and SED in heterogeneous setups works since the faster servers close handled connections faster. But, with inexact weights, they perform less efficient because they cannot estimate the number of connections that a particular server can handle in the future.

- If the service is configured to accept long lasting connections, the impact of the scheduling algorithm disappears. The main reason is that the scheduling algorithm is responsible only for the very first packet to establish a new TCP connection. All further packets have to be forwarded to the same server regardless of the load of the chosen backend server. If the number of concurrent connections is limited, the machine can become partially idle without being able accept further connections. Nevertheless, this does not contradict the assumption, since the maximum number of concurrent connections is a limited resource as well as CPU or RAM. If the service is well configured, the consumption of resources (CPU, RAM, number of connections, etc.) should be balanced. This balance is out of the control and scope of the load balancing algorithm.

The Experiments with the Apache2 server show that the InfiniBand-based reporting of credits suffers from the fact that established connections between Queue Pairs are not inherited to child processes by `fork` (see Section 6.6.4). This has to be taken into account when comparing the credit-based implementation to the other LVS scheduling algorithms.

Figure 6.9 shows the average response time of the above experiments. It shows that the Plain Full Credits Reporting is able to slightly outperform Round-Robin in heterogeneous environments if Round-Robin is used with inexact weights. In general, the measurements confirm the results of the simulations in Section 6.5.

6.7 Conclusion

Monitoring is a good case for one-sided communication because it is possible to work without synchronisation. The issues of parallel applications with MPI and NEON are unimportant for monitoring since no *producer/consumer* interaction occurs.

Buffer announcements (similar to `MPI_Win_Post` or `NEON_Post`) have to be done only once when a server is registered. Notification for resource monitoring is only required in case of a status change of a server from unavailable to available. This can be omitted if dispatcher-based lookup strategies are implemented. In this way, the dispatcher has control of the load introduced by reports. This makes the system more scalable.

The credit-based load balancing has shown to be self-adapting to heterogeneous servers and heterogeneous workload. This is achieved by a backend-based metric that uses the available connection endpoints of a service to tell the dispatcher the maximum number of requests that can be forwarded to a particular machine. This prevents servers from overloading. It also allows the dispatcher to early drop requests if there are no more resources left. This avoids forwarding of requests that would be dropped by the overloaded server.

A simulation study of five credit-based algorithms was conducted to understand the behaviour of reporting credits. It is essential to report two separate values within two different intervals. A soft credits value tells the dispatcher how many requests a server recommends to forward. The hard credits report all available resources to the dispatcher.

The dispatcher works best with a two-level scheduling based on the two values. The primary scheduling should rely on soft credits that are reported within a dynamic interval. This improves the distribution of load. A secondary scheduling based on frequently reported hard credits helps to keep the service available under heavy load.

The current state of implementations is a scheduling module for the Linux Virtual Server and a small library for TCP/IP-based socket applications. The approach uses the backlog of a socket to calculate the number of credits to report. The report is done via InfiniBand RDMA.

An evaluation has confirmed the simulations of the Plain Full Credits Reporting algorithm. Therefore, it is expected that the good results of the Dynamic Pressure Relieve Algorithm will further improve the results. The experiments have shown that the approach has the potential to improve server load balancing in heterogeneous environments without specifying weights.

Chapter 7

Conclusions

This work proofs that the benefits of one-sided communication rise and fall with the nature of synchronisation required by the application. Applications that require less synchronisation can benefit from one-sided communication. If applications cannot benefit from one-sided communication, even RDMA-capable hardware cannot improve the performance compared to send/receive based communication.

7.1 Conclusions from the Communication Model

In Chapter 4, the *Virtual Representation Model* was proposed to abstract interprocess communication. It includes the application into the overall process of communication (like the ISO/OSI model). However, it abstracts the layers 1 to 6 of the ISO/OSI model into a communication system to allow a focus on the efficiency of the API and its implementation. Furthermore, this abstraction combines the synchronisation according to the *producer/consumer* synchronisation with the concept of data transfer over a pipeline.

The general model can be applied to several transports by changing some of the virtual representations into physical representations. The application to a specific environment enables the retrieval of essential hints for the design of a communication interface and its implementation. For example:

- the model indicates whether the buffer announcement has to traverse the network or not.
- if the characteristics of the external and internal communication steps are known, the model helps to decide whether internal buffers are helpful or not and if the message transfer can be deferred or not in case of an *early sender problem*.
- a physical representation does not require API calls. Therefore, it helps to decide which functionality has to be exposed to the application and which can be handled transparently.

The basic layout of an efficient one-sided communication API is another result of combining the *producer/consumer* synchronisation and the pipeline model. The main property of this API is the separation of notification and completion for non-blocking communication.

7.2 Results from the NEON API

The result of Chapter 5 is the design and implementation of a NEw ONe-sided communication API (NEON). The API proves the applicability of the communi-

cation model for parallel applications. NEON is implemented on top of Sockets and on top of InfiniBand. The results show that the performance of applications can be improved by the separation of notification and completion.

If a communication pipeline imposes a bottleneck step, the notification has to happen as early as possible to enable overlap of the synchronisation message. Completion has to happen as late as possible to increase process skew tolerance and the potential to overlap communication and computation.

Applications with a bidirectional synchronisation will suffer from an implicit barrier if the notification and the completion are combined in a single API call. In case of the Cellular Automaton, this reduces efficiency in two ways. First, it reduces the process skew tolerance of the application and therefore the runtime variance in one of the processes will affect other processes (at least the communication partners). Second, a slight process skew allows at least one communication partner to announce its buffer. This implies a lower risk of deferred data transfers. The data transfer has not to be deferred and communication can be overlapped with computation.

An API is efficiently implementable on various networks if the notification can be embedded into the communication API calls. A network, like InfiniBand, that is most efficient by transferring the notification messages as a separate message can easily and efficiently split the API call into a data message and a notification message. However, the opposite way does not work efficiently. If the underlying transport, like Ethernet, performs better with the notification included in the data message, the data transfer has to be deferred until the notification API is called. This would violate the pipeline model.

One-sided communication can synchronise multiple communication operations within a single *producer/consumer* synchronisation. This is where applications may benefit from one-sided communication.

Synchronisation becomes a problem with one-sided communication if multiple networks are available. Independently of using implicit or explicit synchronisation, the synchronisation has to be the last message to be delivered to the destination process.

7.3 Results from Server Load Balancing

One-sided communication is beneficial for resource monitoring since this application requires only sparse synchronisation. Resource monitoring does not require the *producer/consumer* synchronisation. Therefore, efficient implementations can be done especially on top of RDMA-capable hardware.

The proposed credit-based server load balancing scheme relies on resource monitoring and reporting. It benefits from the usage of one-sided communication.

Credits can be directly written into the memory of the dispatcher.

The credits are a simple and powerful metric to represent the future service capabilities of a server. However, this only works if the metric taken to determine the number of credits has an upper limit. The number of connection endpoints has shown to be such a metric for client-server applications. The result of the research presented in Chapter 6 is a self-adapting server load balancing technique that makes efficient use of one-sided communication.

Client-server applications put some limits on the self-adaptability of the credit mechanism. If the load balancer relies on a connection counter (established or available connection slots) the quality of the load distribution will depend on how the application works with connections. If connections are short-living, the resulting schedule will quickly self-adapt to the current load of the server. If connections are long term connections with much idle time, the availability of a service depends on the overall number of simultaneous connections the cluster can handle. For connection oriented services, connections are the main factor to monitor, since connections are the limiting factor of the service.

7.4 Future Work

The analyses in this work have pointed out some future directions that could not be part of the research of this work.

Separate notification messages or messages with and without notification to the same buffer require special treatment in networks without message ordering. For example on multi-rail networks (e. g. multi-port InfiniBand) or mesh networks, messages can easily arrive without ordering. In those networks, it has to be assured that the notification is delivered as the last message.

The Dynamic Pressure Relieve algorithm is worth to investigate. The simulations showed promising results of this algorithm. Another interesting study is the impact of additional information on the number of reported (soft) credits.

Appendix A

Benchmark Testbeds

In this section, the main testbeds are described. The hardware and the software is used in several experiments. Therefore, this is the central place to describe the environments.

A.1 Uranus

Uranus is a cluster of 8 Double-CPU nodes plus a master. The CPUs are 1 GHz Pentium III machines equipped with 1 GB of SDR-RAM. The SuperMicro 370DE6 mainboard contains a NetGear GA-621 Gigabit Ethernet network card attached to a 64 bit/66 MHz PCI-Bus. This network is intended for high speed communication of parallel applications. The switch is a BATM-Titan-0530. The Gigabit Ethernet is fiber-based. An administrative network is available via a EtherExpress PRO/100 Fast Ethernet network interface.

The software environment of this cluster is based on Debian 3.1 *sarge* using Linux-Kernel version 2.6.8. The network driver for the GA-621 is the ns87415 driver (out of the box without further tuning).

The MPI library is the MPICH2 implementation (mpich2-1.0.4p1). It uses the `ssm`-module for communication. This module implements socket-based communication together with optimisations for shared memory.

A.2 Einstein

Einstein consists of 16 compute nodes and 2 masters. Each node is equipped with Intel XEON 2.6 GHz processors and 1 GB of ECC-DDR RAM.

The nodes are interconnected via Intel 82540 Ethernet network interfaces. This network link is used for administration as well as communication between applications.

The Einstein cluster runs a Debian 3.1 *sarge* GNU/Linux with Kernel 2.6.8. The network driver is the e1000 driver.

A.3 InfiniBand Cluster

This cluster is heterogeneous. It consists of 3 pairs of machines that have been added step by step. Table A.1 shows the setup of the 3 pairs. The first pair and the second pair differ only in the version of the host channel adapter (HCA). The last pair is equipped with more modern hardware and PCI-Express instead of PCI-X.

IB1 and IB2	
CPU	Intel XEON 2.66 GHz
RAM	1 GB
HCA	2-Port Mellanox MT23108 MHX-CE128-T
Firmware	3.5.0
PCI	PCI-X
IB3 and IB4	
CPU	Intel XEON 2.66 GHz
RAM	1 GB
HCA	2-Port Mellanox MT23108 MHXL-CF128-T
Firmware	3.5.0
PCI	PCI-X
IB5 and IB6	
CPU	Intel Pentium 4 2.80 GHz
RAM	1 GB
HCA	2-Port Mellanox MT25208 MHEL-CF128-T
Firmware	4.7.6
PCI	PCI-Express

Table A.1: Machines of the InfiniBand cluster.

Each machine runs a Debian 3.1 *sarge* Linux system. The Kernel 2.6.18 is patched with the OFED extensions to provide InfiniBand HCA drivers. The OFED release is version 1.1.

Appendix B

Benchmarks

A detailed description of benchmarks is required in order to compare the results of an experiment to others. This is done in this section. First, the micro-benchmarks are explained (ping-pong, Cellular Automaton). Afterwards, some application benchmarks are presented.

B.1 Micro-Benchmarks

Micro-benchmarks are mainly used to evaluate one or a few parameters of a system. In case of networks, the ping-pong test is very common. A special version of this test is implemented at the institute for computer science called *Eins*.

It is important to describe how the measurement is done. Otherwise, the results can be misinterpreted or cannot be compared to other measurements.

B.1.1 MPI Ping-Pong

This MPI-based ping-pong test was derived from the *pingpong2* example in [SW97]. It starts a parallel application. The processes with an even rank perform the measurement and start to send data to a single odd rank ($own_rank + 1$) using `MPI_Send`. After sending data, `MPI_Recv` is called to receive the reply. The timestamps are retrieved by `MPI_Wtime` before `MPI_Send` and after `MPI_Recv` (see Listing B.1).

```
1 for (i = 0;  i < tries;  i++) {
2   /* start time */
3   lastTime = MPI_Wtime();
4 
5   /* Ping-Pong */
6   MPI_Send(buffer, length, MPI_DOUBLE, myId+1, PING,
```

```
7                                           MPI_COMM_WORLD);
8
9    MPI_Recv(buffer, length, MPI_DOUBLE, myId+1, PONG,
10                                  MPI_COMM_WORLD, &status);
11
12   /* end time */
13   nowTime = MPI_Wtime();
14
15   timing[i] = nowTime - lastTime;
16   lastTime = nowTime;
17 }
```

Listing B.1: Measurement loop of the MPI ping-pong

B.1.2 Eins

Eins is a recursive acronym and stands for *Eins is not sockping*. It was derived from a TCP/IP ping-pong micro-benchmark developed in [Sch03]. Eins is designed and written by Hynek Schlawack in [Sch06]. He extended the socket ping-pong to make it more flexible and usable to measure other network protocols.

The first extension was IPv6 support. A UDP module is included. Eins allows user level fragmentation and measurement sequences. Jörg Zinke included a new module to measure the time to establish a TCP/IP connection.

This tool has become the defacto standard pingpong micro-benchmark at the professorship for Operating Systems and Distributed Systems of Prof. Dr. Bettina Schnor.

```
1  // Main measure-loop
2  for (size_t try = 0; try < ma.tries; try++) {
3     alltime[try] = (nm->measure() - measuredelta) / 2;
4  }
```

Listing B.2: Measurement loop of Eins

The central measurement routine works as follows (see Listing B.2). It runs the module-specific routine. The requirement for this routine is to return the duration of a single ping-pong sequence. The overhead of a measurement is subtracted and the one-way latency is calculated (also called *half round-trip-time*). The values are collected in an array that is evaluated afterwards. The median and the standard deviation are calculated.

The timestamps are based on fetching the TSC (time stamp clock) register of modern CPUs of IA32 family (i586 and above). Initially, Eins calculates the

number of ticks per second to calculate a duration from the TSC values. This makes the current version of Eins stick to CPUs which provide the TSC

The current version of Eins supports the measurement of:

- TCP
- UDP
- BMI (buffered message interface of PVFS2)
- TCP connection establishment
- malloc() memory allocation
- SCTP (Stream Control Transmission Protocol)
- NEON (only original Socket-based API)

B.2 Application Benchmarks

Micro-benchmarks are a common way to discover performance issues or bottlenecks. However, it is not sufficient to evaluate a software using micro-benchmarks only [SS07a]. Applications make use of communication patterns that are different from micro-benchmarks. Furthermore if an application is running, the main task of the machine should be to process the implemented algorithm instead of processing communication. Often this aspect is not recognised and also not intended by miro-benchmarks.

B.2.1 The Cellular Automaton

The *Cellular Automaton* is an example for the large class of *bulk-synchronous* applications. The algorithm allows the overlapping of communication and computation of all calculations except the topmost and undermost line of cells. Listing B.3 shows the pseudocode of the Cellular Automaton.

```
1  for (all iterations) {
2    Buffer_announcement( neighbours );
3
4    /* calculate first and last row */
5    simulate(first_row);
6    simulate(last_row);
7
8    /* include Notification */
9    Initiate_Transmission(first_row, previousPE);
10
11   /* include Notification */
```

```
12      Initiate_Transmission(last_row, nextPE);
13
14      /* calculate remaining cells */
15      simulate(2 - last_row -1);
16
17      /* synchronise */
18      Complete_All();
19  }
```

Listing B.3: Pseudocode of the Cellular Automaton.

The pseudocode shows a kind of ideal solution. The buffer announcement is initiated as early as possible. Then the minimum of cells to communicate is simulated before the transfer of these cells is initiated. Next, all remaining cells are updated. Finally, all non-blocking communication operations have to be completed.

B.2.2 httperf

httperf simulates clients accessing a web server by firing html requests at the server. It can be configured to simulate a number of clients starting at a certain rate. *httperf* can also inject the requests by following recorded traces. By including idle times into these traces, *httperf* can simulate the behaviour of *real* surfers following a click/read sequence.

This feature of *httperf* was used to create the workload for the load balancing tests. *httperf* executed traces that were recorded using RUBiS.

B.2.3 RUBiS

RUBiS is the abbreviation of Rice University Bidding System. It is intended to simulate a bidding, browsing, and selling web site similar to ebay®. It simulates a number clients navigating through the service. According to given probabilities a client accesses different documents and services starting from an entry page.

Only the browsing component of RUBiS is active for the measurements in the context of this thesis. This only employs the web server (Apache in this case). It avoids interference with other services like databases or application servers that are required for the bidding and selling components.

Appendix C

Measurement Data

C.1 SlibNet: Simulation of Credit Algorithms

min	drops	answer time	burst length	product
homogeneous				
10	133.06	541538	6249	1
15	149.64	494667	6841	0.938372
20	149.22	497269	6553	0.982002
25	167.42	456509	5727	1.15735
30	164.64	449158	5979	1.07261
35	183.6	401697	5379	1.18906
40	189.18	403385	5161	1.28232
heterogeneous				
10	278.44	683909	1300	1
15	316.28	630916	1263	1.07858
20	304.26	629395	1339	0.97634
25	334.58	573616	1309	1.00091
30	332.34	566432	1186	1.08358
35	374.96	512891	1115	1.17747
40	364.44	511429	1161	1.09595

Table C.1: LWM minimum number of credits.

	Plain	LWM	SH	DL-SH	DPR	WRR
dropped	162	164	153	210	145	203
queue	52.0	51.9	49.9	40.6	48.2	49.0
answer time	551539	549688	532874	428104	513361	525986
queue ratio	1.009	0.968	1.019	1.012	0.996	1.054
drop-time	0.8368	0.8443	0.7636	0.8420	0.699	1.0

Table C.2: Simulation results for homogeneous setup with two servers under 100 % load.

	Plain	LWM	SH	DL-SH	DPR	WRR
dropped	162	164	153	210	145	645
queue	52.0	51.9	49.9	40.6	48.2	50.7
answer time	551539	549688	532874	428104	513361	524385
queue ratio	1.009	0.968	1.019	1.012	0.997	5.290
drop-time	0.2642	0.2665	0.2410	0.266	0.221	1.0

Table C.3: Simulation results for non-exact-homogeneous weights.

	Plain	LWM	SH	DL-SH	DPR	WRR
failed	314	332	245	309	196	219
queue	69.2	68.4	63.1	49.7	56.4	48.2
answer time	714531	706006	652111	518468	593099	520337
queue ratio	1.941	2.016	1.451	1.422	1.387	0.900
drop-time	1.963	2.0546	1.3973	1.4031	1.0183	1.0

Table C.4: Simulation results for heterogeneous setup with two servers under 100 % load.

	Plain	DPR	WRR	WRR5	WRR10
failed	314	196	219	355	627
queue	69.2	56.4	48.2	42.8	43.2
answer time	714531	593099	520337	451034	453568
queue ratio	1.941	1.387	0.900	0.3086	0.1289
drop-time	1.963	1.0183	1.0	1.4011	2.4897

Table C.5: Simulation results for heterogeneous setup with two servers using non-exact weights (10 %) under 100 % load.

Appendix D

Specification of Units

Unit	Measure	Explanation
1 kB	1 000 Bytes	size of messages or buffers
1 MB	1 000 kB	size of messages or buffers
1 GB	1 000 MB	size of messages or buffers
1 KiB	1 024 Bytes	size of messages or buffers
1 MiB	1 024 KiB	size of messages or buffers
1 GiB	1 024 MiB	size of messages or buffers
1 kbit/s	1 000 bit/s	transfer rate, throughput
1 Mbit/s	1 000 kbit/s	transfer rate, throughput
1 Gbit/s	1 000 Mbit/s	transfer rate, throughput
1 kB/s	1 000 B/s	transfer rate, throughput
1 MB/s	1 000 kB/s	transfer rate, throughput
1 GB/s	1 000 MB/s	transfer rate, throughput
1 KiB/s	1 024 B/s	transfer rate, throughput
1 MiB/s	1 024 KiB/s	transfer rate, throughput
1 GiB/s	1 024 MiB/s	transfer rate, throughput

Table D.1: Specification of Units.

List of Figures

2.1 General Ethernet data transmission scheme using DMA. 25
2.2 Ratio of transmission times (network vs. memory). 27
2.3 InfiniBand network overview [Inf02a]. 28
2.4 InfiniBand Programming Interfaces and Protocols [RnSH05]. . . . 29
2.5 Data transmission scheme of InfiniBand (non-RDMA). 30

3.1 VI Architectural Model [CCC97] 38
3.2 Direct Access Transport Framework [DAT07b] 40
3.3 Fence synchronisation mechanism of MPI-2 [MPI97]. 45
3.4 Post-Start-Complete-Wait synchronisation of MPI-2 [MPI97]. . . 45
3.5 Architecture of MPICH and MPICH2 47
3.6 Abstraction Layers of Open MPI. 48
3.7 Basic rendezvous transmission scheme. 57
3.8 Communication pipeline example. 58
3.9 Effect of sending fragments through a pipeline. 59
3.10 Comparing eager (upper) and rendezvous (lower) mode of MPI 1.2.7. 62

4.1 Producer/Consumer Example. 70
4.2 Communication model of virtually provided remote process. . . . 71
4.3 Pipeline steps of the communication model. 71
4.4 Physical mapping of process B into the communication system. . . 77
4.5 Physical mapping in case of shared memory as transport. 78
4.6 Physical mapping in case of shared address space. 78
4.7 Implicit barrier with bi-directional synchronisation. 86
4.8 Bi-directional synchronisation in the proposed API. 92

5.1 One-dimensional domain decomposition of the Cellular Automaton. 96
5.2 Jumpshot visualisation of the CA with MPI_Sendrecv. 98
5.3 Time spent in the barrier at the end. 100
5.4 Communication between two processes. 104

5.5 Architecture of the NEON implementation (derived from [Dav08]) 113
5.6 Impact of early notification and implicit barrier in MPI-2. 124
5.7 Application of the Communication Model to Shared Memory. . . 126

6.1 Architecture of SlibNet. 137
6.2 Rings of registered and available servers. 144
6.3 Impact of the minimum credits on Low Watermark Reporting. . . 154
6.4 Average number of dropped requests out of 18750 157
6.5 Average answer time. 158
6.6 Answer time correlated to number of drops normalised to WRR. . 158
6.7 Average tolerated burst length 159
6.8 Architecture of SlibNet for Socket-based applications [Zin07]. . . 165
6.9 Response time of Round-Robin with and without weights compared to SLIBNet. 167

List of Tables

5.1 Runtime and of communication time of the Cellular Automaton. . 97
5.2 Latency and Bandwith of NEON and MPICH2. 120
5.3 Runtime comparison (in seconds) of the cellular automaton using MPICH2 one-sided and two-sided communication and NEON. . . 121
5.4 Time per iteration of Cellular Automaton. 124

6.1 Parameter overview of simulations. 151
6.2 Simulation results for a single server setup under 100 % load. . . . 155
6.3 Comparison between credit-based and WRR using 19 servers. . . 160

A.1 Machines of the InfiniBand cluster. 176

C.1 LWM minimum number of credits. 181
C.2 Simulation results for homogeneous setup with two servers under 100 % load. 182
C.3 Simulation results for non-exact-homogeneous weights. 182
C.4 Simulation results for heterogeneous setup with two servers under 100 % load. 182
C.5 Simulation results for heterogeneous setup with two servers using non-exact weights (10 %) under 100 % load. 182

D.1 Specification of Units. 183

Bibliography

[AAC+04] George Almási, Charles Archer, José G. Castaños, C. Chris Erway, Philip Heidelberger, Xavier Martorell, José E. Moreira, Kurt Pinnow, Joe Ratterman, Nils Smeds, Burkhard Steinmacher-burow, William Gropp, and Brian Toonen. Implementing MPI on the BlueGene/L Supercomputer. In Danelutto et al. [DVL04], pages 833–845.

[ACH+08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., March 2008.

[ADD+04] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of LA-MPI, A Network-Fault-Tolerant MPI. *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 15b, 2004.

[AGSZ06] Michael Andraschek, Stephan Gensch, Matthias Schulz, and Jörg Zinke. NEON: A New Efficient One-sided communications iNterface. Seminar Paper, Universität Potsdam, Institut für Informatik, April 2006.

[AS06] Michael Andraschek and Matthias Schulz. Evaluierung von MPI-2 One-Sided Communications auf TCP/IP-Netzwerken. Student Research Project, Universität Potsdam, Institut für Informatik, 2006.

[Ass98] VMEbus International Trade Association. Myrinet on VME Protocol Specification. http://www.myri.com/open-specs, August 1998.

[Ast08] Astrophysikalisches Institut Potsdam. AMIGA. `http://www.aip.de/People/AKnebe/AMIGA/`, 2008. accessed 02/2009.

[BB03] Christian Bell and Dan Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *IPDPS '03:*

Proceedings of the 17th International Symposium on Parallel and Distributed Processing, page 198.1, Washington, DC, USA, 2003. IEEE Computer Society.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[BBNY06] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3 edition, November 2005. covers version 2.6.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[BD03] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *In Proc. Support for High Performance Scientific and Engineering Computing (SHPSEC-03*, pages 91–99, 2003.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[BJL+06] Tim Brecht, G. (John) Janakiraman, Brian Lynn, Vikram Saletore, and Yoshio Turner. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. *SIGOPS Operating Systems Review*, 40(4):265–278, 2006.

[BMG06a] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 521–530, Washington, DC, USA, 2006. IEEE Computer Society.

[BMG06b] Darius Buntinas, Guillaume Mercier, and William D. Gropp. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In Mohr et al. [MTWD06], pages 86–95.

[BMM+05] Christine Böckmann, Irina Mironova, Detlef Müller, Lars Schneidenbach, and Remo Nessler. Microphysical Aerosol Parameters from Multiwavelength Lidar. *Optical Society of America*, 22(3), March 2005.

[Bou01] Tony Bourke. *Server Load Balancing*. O'Reilly, 2001.

[Bru99] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proceedings of IEEE Infocom*, pages pp. 534–542, 1999.

[BRU05] Ron Brightwell, Rolf Riesen, and Keith D. Underwood. Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications. *International Journal of High Performance Computing Applications*, 19(2):103–117, 2005.

[BSL07] Brian Barrett, Galen M. Shipman, and Andrew Lumsdaine. Analysis of Implementation Options for MPI-2 One-Sided. In Cappello et al. [CHD07], pages 242–250.

[BSWP02] Pavan Balaji, Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *CLUSTER*, pages 179–186. IEEE Computer Society, 2002.

[BU03] Ron Brightwell and Keith D. Underwood. Evaluation of an Eager Protocol Optimization for MPI. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *PVM/MPI*, volume 2840 of *Lecture Notes in Computer Science*, pages 327–334. Springer, 2003.

[BU04] Ron Brightwell and Keith D. Underwood. An Analysis of NIC Resource Usage for Offloading MPI. *ipdps*, 09:183a, 2004.

[CC99] G. Chiola and G. Ciaccio. Porting MPICH ADI on GAMMA with Flow Control. In *Proceedings of IEEE - ACM 1999 Midwest Workshop on Parallel Processing (MWPP 1999)*, Kent, OH, August 1999.

[CCC97] Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture Specification Version 1.0. `ftp://download.intel.com/design/servers/vi/VI_Arch_Specification10.pdf`, December 1997. accessed 12/2007.

[CCE+03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[CES02] G. Ciaccio, M. Ehlert, and B. Schnor. Exploiting gigabit ethernet capacity for cluster applications. In *LCN '02: Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, page 0669, Tampa, Florida, USA, 2002. IEEE Computer Society.

[CHD07] Franck Cappello, Thomas Hérault, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*. Springer, 2007.

[Chu96] H.K. Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996. San Diego, CA.

[Cia] G. Ciaccio. MPI/GAMMA home page. `http://www.disi.unige.it/project/gamma/mpigamma/`.

[Cia99] Giuseppe Ciaccio. *Efficient Protocols for Cluster Computing*. PhD thesis, University of Genoa, 1999.

[Clu03] Cluster File Systems, Inc. Lustre File System. `http://www.clusterfs.com/`, 2003. accessed 01/2008.

[CYZEG04] Francois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity Analysis of the UPC Language. *ipdps*, 15:254a, 2004.

[DAT07a] DAT collaborative. Kernel Direct Access Programming Library. `http://www.datcollaborative.org/kdapl.html`, January 2007. accessed 12/2007.

[DAT07b] DAT collaborative. User Direct Access Programming Library. `http://www.datcollaborative.org/udapl.html`, January 2007. accessed 12/2007.

[Dav08] David Böhme. Porting and Analysis of NEON over InfiniBand. Diploma thesis, Universität Potsdam, January 2008.

[Day95] John Day. The (un)revised OSI reference model. *SIGCOMM Comput. Commun. Rev.*, 25(5):39–55, 1995.

[DB06] Douglas Doerfler and Ron Brightwell. Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces. In Mohr et al. [MTWD06], pages 331–338.

[DBP+08] Anthony Danalis, Aaron Brown, Lori L. Pollock, D. Martin Swany, and John Cavazos. Gravel: A Communication Library to Fast Path MPI. In Alexey L. Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, pages 111–119. Springer, 2008.

[DVL04] 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*, volume 3149 of *Lecture Notes in Computer Science*. Springer, 2004.

[DW08] Dennis Dalessandro and Pete Wyckoff. RDMA Enabled Apache. `http://www.osc.edu/research/network_file/projects/rdma/overview.shtml`, accessed 12/2008.

[DZ83] J. D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[EA07] Enrico Ellguth and Michael Augustin. Vergleich der MPI-OSC Implementationen MVAPICH, Open MPI und HP-MPI über InfiniBand. Student Research Project, Universität Potsdam, Institut für Informatik, 2007.

[EAR08] EARLINET. the European Aerosol Research LIdar NETwork. `http://www.earlinet.org/`, 2008. accessed 09/2008.

[EGC01] Tarek El-Ghazawi and Sebastien Chauvin. UPC Benchmarking Issues. In *ICPP '01: Proceedings of the International Conference on Parallel Processing*, pages 365–372, Washington, DC, USA, 2001. IEEE Computer Society.

[EGC02] Tarek El-Ghazawi and Francois Cantonnet. UPC performance and potential: a NPB experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[Ehl03] Marko Ehlert. Portierung der Genoa Active Message MAchine (GAMMA) auf die GigabitEthernet Netzwerkkarte Netgear GA621. Diploma thesis, Universität Potsdam, February 2003.

[EL06] Norbert Eicker and Thomas Lippert. Scalable Ethernet Clos-Switches. In *Euro-Par*, pages 874–883, 2006.

[Ell08] Enrico Ellguth. Parallelisierung von Clasp. Diploma thesis, Universität Potsdam, October 2008.

[FD00] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[FHL+05] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server network scalability and TCP offload. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[FKSS06] Sven Friedrich, Sebastian Krahmer, Lars Schneidenbach, and Bettina Schnor. Loaded: Server Load Balancing for IPv6. In *ICNS*, page 8. IEEE Computer Society, 2006.

[Fri06] Sven Friedrich. Lastverteilung in InfiniBand-Netzwerken. Diploma thesis, Universität Potsdam, January 2006.

[FSS05] Sven Friedrich, Lars Schneidenbach, and Bettina Schnor. SLIBNet: Server Load Balancing for InfiniBand Networks. Technical Report ISSN 0946-7580, TR-2005-12, Institute for Computer Science, University of Potsdam, December 2005.

[GAS06] GASNet Specification. `http://gasnet.cs.berkeley.edu/`, November 2006. Release 1.8.

[Geo06] Patrick Geoffray. A Critique of RDMA. `http://www.hpcwire.com/hpc/815242.html`, 2006.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI:

Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GJM+05] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A platform for distributed answer set solving. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *LNAI*, pages 227–239. Springer, 2005.

[GJM+06] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy. On Probing and Multi-Threading in Platypus. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the European Conference on Artificial Intelligence (ECAI'06)*, pages 392–396. IOS Press, 2006.

[GKNS07a] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A Conflict-Driven Answer Set Solver. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 260–265. Springer-Verlag, 2007.

[GKNS07b] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-Driven Answer Set Solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http://www.ijcai.org/papers07/contents.php.

[GL94] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical report, Argonne National Laboratory, 1994. Preprint MCS-P342-1193.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Standard. Technical report, Argonne National Laboratory and Mississippi State University, 1996.

[GRBK98] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting*

on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 180–187, London, UK, 1998. Springer-Verlag.

[GT05] William D. Gropp and Rajeev Thakur. An Evaluation of Implementation Options for MPI One-Sided Communication. In Martino et al. [MKD05], pages 415–424.

[GT07] William D. Gropp and Rajeev Thakur. Revealing the Performance of MPI RMA Implementations. In Cappello et al. [CHD07], pages 272–280.

[HC07] Nor Asilah Wati Abdul Hamid and Paul Coddington. Averages, Distributions and Scalability of MPI Communication Times for Ethernet and Myrinet Networks. In *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*, pages 269–276, Anaheim, CA, USA, 2007. ACTA Press.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2 edition, 1996.

[HSJP05a] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, and Dhabaleswar K. Panda. Design Alternatives and Performance Trade-Offs for Implementing MPI-2 over InfiniBand. In Martino et al. [MKD05], pages 191–199.

[HSJP05b] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, and Dhabaleswar K. Panda. Scheduling of MPI-2 One Sided Operations over InfiniBand. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, page 215.1, Washington, DC, USA, 2005. IEEE Computer Society.

[HX97] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing, Technology, Architecture, Programming*. WCB/McGraw-Hill, 1997.

[Inc08] Cray Inc. Chapel Language Specification 0.775. Technical report, Cray Inc., 2008.

[Inf02a] InfiniBand Trade Association. InfiniBand Architecture Specification Volumes 1 and 2 Release 1.1. `http://www.infinibandta.org/specs`, November 2002.

[Inf02b] InfiniBand Trade Association. Sockets Direct Protocol v1.0. `http://www.infinibandta.org/specs`, November 2002.

[Ins85] Institute of Electrical and Electronics Engineers. IEEE 802.3 Standard, 1985.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis. Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[JCB96] P. Steenkiste J. C. Brustoloni. Effects of buffering semantics on I/O performance. In *Proceedings OSDI'96*, pages pp. 277–291. USENIX, October 1996. Seattle, WA.

[JD05] Hao Jiang and Constantinos Dovrolis. Why is the Internet Traffic Bursty in Short Time Scales? *SIGMETRICS Perform. Eval. Rev.*, 33(1):241–252, 2005.

[JTC94] JTC 1. ISO 7498-1 , textgleich mit DIN ISO 7498, hat den Titel: Information technology - Open Systems Interconnection - Basic Reference Model: The basic model, 1994.

[Jül08] Forschungszentrum Jülich. Scalasca. `http://www.fz-juelich.de/jsc/scalasca/`, 2008. accessed 09/2008.

[Ker03] Jeremy Kerr. Using Dynamic Feedback to Optimise Load Balancing Decisions. Australian Linux Conference - linux.conf.au, 2003.

[Ker08] Jeremy Kerr. Feedbackd Project Homepage. `http://ozlabs.org/~jk/projects/feedbackd/`, 2008.

[KK07] Christoph Kessler and Jörg Keller. Models for Parallel Computing: Review and Perspectives. *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen PARS*, 24:13–29, 2007.

[KKF+08] Adrian Knoth, Christian Kauhaus, Dietmar Fey, Lars Schneidenbach, and Bettina Schnor. Challenges of MPI over IPv6. In *Proceedings of the 4th IARIA International Conference on Networks and Systems (ICNS), IPv6 Deploying Future Internet Workshop*, pages xx–yy, Gossier, Guadeloupe, 2008. IEEE Computer Society.

[KKPF07] Christian Kauhaus, Adrian Knoth, Thomas Peiselt, and Dietmar Fey. Efficient Message Passing on Multi-Clusters: An IPv6 Extension

to Open MPI. In *Proceedings of KiCC'07, Chemnitzer Informatik Berichte*, February 2007.

[KKZL03] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling Molecular Dynamics to 3000 Processors with Projections: A Performance Analysis Case Study. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev, editors, *International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 23–32. Springer, 2003.

[KRS01] Christian Kurmann, Felix Rauch, and Thomas Stricker. Speculative Defragmentation - Leading Gigabit Ethernet to True Zero-Copy Communication. *Cluster Computing*, 4(1):7–18, 2001.

[KTF02] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, November 2002.

[Kun91] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing System. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[Lin07a] Linux VS Group. Job Scheduling Algorithms in LVS. `http://www.linuxvirtualserver.org/docs/scheduling.html`, 2007.

[Lin07b] Linux VS Group. Linux Virtual Server. `http://www.linuxvirtualserver.org`, 2007.

[LWP04] 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.

[MBKQ00] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, March 2000.

[MJ98] David Mosberger and Tai Jin. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(3):31–37, December 1998. (Originally appeared in Proceedings of the 1998 Internet Server Performance Workshop, June 1998, 59-67.).

[MKD05] Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors. *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*. Springer, 2005.

[Mog03] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 25–30, Berkeley, CA, USA, 2003. USENIX Association.

[MPI95] MPI: A Message Passing Interface Standard, June 1995. Message Passing Interface Forum.

[MPI97] MPI-2: Extensions to the Message Passing Interface, July 1997. Message Passing Interface Forum.

[MPI07] The MPICH2 Project. `http://www.mcs.anl.gov/research/projects/mpich2/index.php`, 2007. Argonne National Laboratory.

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15:217–252, 1997.

[MRB+06] Frank Mietke, Rober Rex, Rober Baumgartl, Torsten Mehlan, Torsten Höfler, and Wolfgang Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par*, volume 4128 of *Lecture Notes in Computer Science*, pages 125–133. Springer, 2006.

[MS05] Arthur A. Mirin and William B. Sawyer. A Scalable Implementation of a Finite-Volume Dynamical Core in the Community Atmosphere Model. *Int. J. High Perform. Comput. Appl.*, 19(3):203–212, 2005.

[MTWD06] 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*. Springer, 2006.

[MVA07] MVAPICH: MPI over InfiniBand and iWARP. `http://mvapich.cse.ohio-state.edu/`, 2007.

[Myr05] Myricom, Inc. Myrinet Exprress (MX): A High-performacne, Low-Level, Message-Passing Interface for Myrinet. Technical Report 1.0, Myricom, Inc., January 2005.

[NAS07] NAS Parallel Benchmark Suite. `http://www.nas.nasa.gov/Resources/Software/npb.html`, 2007.

[NC99] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Libray for Distributed Array Libraries and Compiler Run-Time Systems. In José D. P. Rolim, Frank Mueller, Albert Y. Zomaya, Fikret Erçal, Stephan Olariu, Binoy Ravindran, Jan Gustafsson, Hiroaki Takada, Ronald A. Olsson, Laxmikant V. Kalé, Peter H. Beckman, Matthew Haines, Hossam A. ElGindy, Denis Caromel, Serge Chaumette, Geoffrey Fox, Yi Pan, Keqin Li, Tao Yang, G. Ghiola, Gianni Conte, Luigi V. Mancini, Dominique Méry, Beverly A. Sanders, Devesh Bhatt, and Viktor K. Prasanna, editors, *IPPS/SPDP Workshops*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer, 1999.

[NTKP06] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda. High Performance Remote Memory Access Comunications: The ARMCI Approach. *International Journal of High Performance Computing and Applications*, 20(2):233–253, 2006.

[OMP07] The Open MPI Project. `http://www.open-mpi.org/`, 2007.

[Ope] Open Fabrics Alliance. `http://www.openfabrics.org/`. accessed 12/2007.

[Ope05] OpenMP Architecture Review Board. OpenMP Application Program Interface. Technical report, OpenMP Architecture Review Board, 2005.

[PBW+05] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with namd. *J Comput Chem*, 26(16):1781–1802, December 2005.

[PcFH+02] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January/February 2002. ISSN 0272-1732.

[PKP03] 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 *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[Pot04] Potsdam Institute for Climate Impact Research. Modular Ocean Model 3. `http://www.pik-potsdam.de/research/research-domains/earth-system-analysis/climber3/ocean.html`, 2004. accessed 09/2008.

[PVF03] PVFS-Development-Team. Parallel Virtual File System, Version 2. `http://www.pvfs.org/`, sep 2003. accessed 01/2008.

[Raj05] Rajeev Thakur and William Gropp and Brian Toonen. Optimizing the Synchronization Operations in MPI One-Sided Communication. *High Performance Computing Applications*, 19(2):119–128, 2005.

[RnSH05] Hal Rosenstock and Roland Dreier nad Sean Hefty. OpenIB Core Software: Architectural Overview. `http://openfabrics.org/docs/oib_wkshp_022005/openib_core_SW1.pdf`, 2005. Slides from OpenIB Developers Workshop, accessed 01/2008.

[Ryl07] Olaf Ryll. Native Lastverteilung in InfiniBand-Netzwerken. Diploma thesis, Universität Potsdam, March 2007.

[SBB$^+$07] Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. In Cappello et al. [CHD07], pages 178–186.

[SBS08] Lars Schneidenbach, David Böhme, and Bettina Schnor. Performance Issues of Synchronisation in the MPI-2 One-Sided Communiction API. In *15th European PVM/MPI Users' Group Meeting*, pages 177 – 184, Dublin, Ireland, September 2008. Spinger, Lecture Notes in Computer Science 5205.

[SC03] Piyush Shivam and Jeffrey S. Chase. On the elusive benefits of protocol offload. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 179–184, New York, NY, USA, 2003. ACM.

[Sch03] Lars Schneidenbach. Design und Implementierung einer Socket-Schnittstelle für das Leichtgewichtprotokoll GAMMA. Diploma thesis, Universität Potsdam, March 2003.

[Sch06] Hynek Schlawack. Analyse und Optimierung der Netwerkschnittstellen BMI und NEON. Diploma thesis, Universität Potsdam, September 2006.

[Shm01] Quadrics Supercomputers World Ltd. *Shmem Programming Manual*, 3 edition, June 2001. Quadrics Shmem Manual.

[Sil08] Silicon Graphics, Inc. SHMEM API for Parallel Programming. `http://www.shmem.org/`, 2008. accessed 09/2008.

[SL03] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[SLP94] B. Schnor, H. Langendörfer, and S. Petri. Einsatz neuronaler Netze zur Lastbalancierung in Workstationclustern. In H. Langendörfer, editor, *Praxisorientierte Parallelverarbeitung*, pages 154–165, München, October 1994. Hanser Verlag.

[SN08] Vijay Saraswat and Nathaniel Nystrom. Report on the Experimental Language X10. Technical Report Version 1.7, IBM Corporation, 2008.

[SNM+98] Gautam Shah, Jarek Nieplocha, Jamshed Mirza, Chulho Kim, Robert Harrison, Rama K. Govindaraju, Kevin Gildea, Paul Dinicola, and Carl Bender. Performance and Experience with LAPI - a New High-Performance. In *Communication Library for the IBM RS/6000 SP. In Proceedings of the International Parallel Processing Symposium*, pages 260–266, 1998.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetzov. Beyond Softnet. In *Proceedings of the USENIX 2001 Annual Technical Conference*, 2001.

[SP04] Taher Saif and Manish Parashar. Understanding the Behavior and Performance of Non-blocking Communications in MPI. In Danelutto et al. [DVL04], pages 173–182.

[SPL96] B. Schnor, S. Petri, and H. Langendörfer. Load Management for Load Balancing on Heterogeneous Platforms: A Comparison of Traditional and Neural Network Based Approaches. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Parallel Processing, Volume II of the Proceedings of the Second International Euro-Par Conference (Euro-Par'96)*, volume 1124 of *Lecture Notes in Computer Science*, pages 615–620. ENS Lyon, Springer, August 1996.

[Spr05] Volker Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[SS05] Lars Schneidenbach and Bettina Schnor. Migration of MPI Applications to IPv6 Networks. In *Proceedings of the 23rd Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 172–176, Innsbruck, Austria, February 2005.

[SS07a] Lars Schneidenbach and Bettina Schnor. Design issues in the implementation of MPI2 one sided communication in Ethernet based networks. In *Proceedings of the 25th IASTED Parallel and Distributed Computing and Networks (PDCN)*, pages 277–284. ACTA Press, February 2007.

[SS07b] Lars Schneidenbach and Bettina Schnor. Self-Adapting Server Load Balancing in InfiniBand Networks. Technical Report ISSN 0946-7580, TR-2007-2, Institute for Computer Science, University of Potsdam, 2007.

[SSO+95] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 1–10. ACM, 1995.

[SSP03] Lars Schneidenbach, Bettina Schnor, and Stefan Petri. Architecture and Implementation of the Socket Interface on Top of GAMMA. In *LCN '03: Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks*, pages 528–536, Washington, DC, USA, 2003. IEEE Computer Society.

[SSZL08] Lars Schneidenbach, Bettina Schnor, Jörg Zinke, and Janette Lehmann. Self-Adapting Credit-based Server Load Balancing. In *Proceedings of the 26th International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages xx–yy, Innsbruck, Austria, February 2008. Acta Press.

[Sun88] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol specification: Version 2. Sun Microsystems. http://www.ietf.org/rfc/rfc1057.txt, June 1988. Status: Informational.

[SW97] Peter Sanders and Thomas Worsch. *Parallele Programmierung mit MPI: ein Praktikum*. Logos Verlag Berlin, 1997.

[SYW01] V. Springel, N. Yoshida, and S. D. M. White. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6:79–117, April 2001.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 2 edition, 2001.

[TOHI98] H. Tezuka, F. O'Carrol, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. *International Parallel Processing Symposium (IPPS)*, page 0308, 1998.

[TOP08] Top 500 Suptercomputer Sites. http://www.top500.org/, November 2008. accessed 01/2009.

[TRH00] Jasper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The Implementation of MPI–2 One-Sided Communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.

[UPC05] UPC Consortium. UPC Language Specifications v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, May 2005.

[VBR+04] Kees Verstoep, RAOUL A.F. Bhoedjang, Tim Rühl, Henri E. Bal, and Rutger F.H. Hofman. Cluster Communication Protocols for Parallel-Programming Systems. *ACM*, 22(3), Aug 2004.

[WA99] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., 1999.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[WKM+98] R. Wang, A. Krishnamurthy, R. Martin, T. Anderson, and D. Culler. Modeling and Optimizing Communication Pipelines. In *Proceedings of the 1998 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1998. Madison.

[WW05] P. Wyckoff and J. Wu. Memory Registration Caching Correctness. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 1008–1015, Washington, DC, USA, 2005. IEEE Computer Society.

[YGP06] Weikuan Yu, Qi Gao, and D.K. Panda. Adaptive connection management for scalable MPI over InfiniBand. *Parallel and Distributed Processing Symposium, International*, 0:81, 2006.

[Zha00] Wensong Zhang. Linux Virtual Server for Scalable Network Services. In *Proceedings of Ottawa Linux Symposium*, 2000.

[Zin07] Jörg Zinke. Server Load Balancing für TCP/IP-Anwendungen in InfiniBand-Netzen. Master's thesis, Universität Potsdam, September 2007.

[Zit95] Martina Zitterbart. *Flexible und effiziente Kommunikationssysteme für Hochleistungsnetze*. International Thomson Publishing, 1995.

Index

Abstract Device Interface, 46
access, 72
active ring, 144
active target synchronisation, 46, 75, 97
ADI, 46, 47
aggregated handles, 51
all ring, 144
ARMCI, 50
asynchronous, 17

backend server, 133
backend-based monitoring, 139
blocking, 17, 18
bridge-path, 134
buffer, 17
buffer announced, 75
buffer announcement, 73, 74, 76, 77, 79, 83, 85, 89, 91, 124, 139
buffer management, 53
bulk-synchronous, 22, 86, 96, 112, 179

cache pollution, 54, 81
Cellular Automaton, 86
Cellular Automaton, 179
channel device, 46
clusters, 16
communication endpoint, 142
communication model, 71
communication progress, 50
communication system, 53, 55, 71
completion, 17, 18, 73–76, 79, 85, 88, 91–93, 139
completion queue, 28
computer, 17
credit-based scheduling, 140

DAPL, 39
DAT, 39
direct access, 18, 42, 72, 90
Direct Access Programming Library, 39
Direct Access Transport, 39
direct memory access, 64
direct server return, 134
Direct User Sockets Interface, 36
dispatcher, 133
dispatcher-based monitoring, 138
DMA, 64
Doorbell, 37
double buffering, 102
dropped requests, 134
DUSI, 36
Dynamic Pressure Relieve Algorithm, 147
dynamic process management, 45

eager, 77, 119
eager buffer, 55, 56
eager mode, 56
early sender problem, 55, 62, 81, 84, 115, 122, 124, 125, 129, 172
efficiency, 34, 35
Eins, 30, 177, 178
Ethernet, 24
expected, 55
expected credits, 145, 146
explicit notification, 90
external interaction, 76

fast path, 55

feedbackd, 139
flat-based, 134
fragmentation, 63
frontend server, 133

GAMMA, 37
GASNet, 52
get, 104
global address space language, 52, 53

half round-trip-time, 178
hard credits, 146
HCA, 106
host, 17
Host Channel Adapter, 27, 106
httperf, 137, 180
hybrid, 43, 49, 84, 115, 117

implicit barrier, 87, 124
implicit notification, 90
independent progress, 61
indirect access, 72, 90
InfiniBand, 27, 47
interface, 17
internal interaction, 76
interrupt service routines, 37
ISO/OSI Model, 71

job, 117
Jumbo-frames, 63
jumpshot, 61

language complexity, 34
large message transfer, 57
late receiver problem, 55
late sender problem, 82
latency, 58
latency hiding, 59
LC, 138
least connections, 138, 148
lightweight protocol, 37
Linux AIO, 36
Linux Virtual Server, 135
LMT, 57
lookup, 145
lookup interval, 145
low watermark, 145
LVS, 135

machine, 17
massively parallel processor, 17
master-worker, 23
Maximum Transfer Unit, 56, 63
memory constraints, 81
memory pooling, 56
memory region, 17
memory registration, 54
memory window, 105
message passing, 42
Message Passing Interface, 10, 43
minimal copy, 54
MPI, 10, 43
MPI Forum, 129
MPI-2, 44, 47
MPI-3, 129
MPICH2, 46, 47
MPP, 17
MTU, 56, 63, 64
Mutual exclusion, 42
MVAPICH, 47
Myrinet, 26, 64

NAPI, 65
NAT-based, 134
Nemesis, 46
NEON, 96
node, 17
non-blocking, 18
notification, 17, 18, 73, 74, 76, 79, 83, 85, 88, 93, 139

offloading, 53, 60
one-sided, 10, 43, 45, 70
Open Portable Access Layer, 47

Open Run-Time Environment, 47
Open MPI, 47
overlap, 50, 58
overlapping, 53

packet size, 63
parallel I/O, 45
Parallel Virtual Machine, 43
partially synchronous, 99
passive target synchronisation, 46
PIO, 64
pipeline, 57
Point-to-point, 42
post-start-complete-wait, 86
process group, 43
process skew tolerance, 124
producer/consumer, 18, 42, 70, 71, 73, 76, 79, 90, 92, 93, 105, 112, 139, 169, 172, 173
program complexity, 34
progress engine, 117
put, 104

Queue Pair, 28, 39, 143

rank, 43
RDMA, 10, 13, 18, 39, 47, 54, 70
remote direct memory access, 10, 13, 54
remote memory access, 18, 42, 45
Remote Procedure Calls, 37
rendezvous, 77, 119
rendezvous mode, 56, 57, 61
rendezvous protocol, 57
report interval, 145
resource monitoring, 138
RMA, 18, 42, 45
RMA-capable, 18
round trip time, 135
round-robin, 138
route-path, 134
RR, 138
RTT, 135
RUBiS, 137

segregated free lists, 56
server load balancer, 133
Server load balancing, 132
server load balancing, 14, 24
shortest expected delay, 148
Socket, 35, 143
socket backlog queue, 143
soft credits, 146
speed factor, 136, 142
stencil, 96
synchronisation, 73
synchronisation point, 18, 55, 90, 103
synchronous, 18
system area network, 27

tag, 105
Target Channel Adapters, 27
two-armed, 134
two-sided, 42, 70

unexpected message, 55, 62, 115, 119
Unified Parallel C, 34, 52
UPC, 52, 53

Verbs, 29, 39, 47
VIP, 133
Virtual Interface, 37
Virtual Interface Architecture, 37
virtual IP, 133
Virtual Representation Model, 10, 13, 18, 70, 71, 76, 79, 93, 132, 140, 172

Weighted Round-Robin, 137
window, 105
work queue, 37
work queue entry, 28
work requests, 28

zero-copy, 54