Department of Computer Science

Computer Architecture Group

Diploma Thesis

Optimizing Applications and Message-Passing Libraries for the QPACE Architecture

Author:
Simon Wunderlich
siwu@hrz.tu-chemnitz.de

Chemnitz, March 9, 2009

Supervisor: Prof. Dr.-Ing. Wolfgang Rehm
University Adviser: Dipl-Inf. Torsten Mehlau
Company Advisers: Dipl-Ing. (FH) Hans Böttiger
Dipl-Ing. (FH) Heiko J. Schick (IBM Deutschland Research & Development GmbH)
Acknowledgements

I would like to thank Prof. Wolfgang Rehm for providing the topic of this diploma thesis and for his guidance, ideas and suggestions not only for this thesis but also for my studies at TU Chemnitz. Further I would like to thank my company advisers Heiko J. Schick and Hans Böttiger from IBM Deutschland Research & Development GmbH who introduced me to IBM and the QPACE team and supported me with many detailed information and insightful discussions. My university adviser Torsten Mehlan supported me with answers to many technical and administrative questions I had while writing this thesis. I would also like to thank the other colleagues of our computer architecture group Andreas Heinig, Frank Mietke, René Oertel, Timo Schneider and Jochen Strunk for their suggestions and help.

Also I am very grateful to the whole QPACE team for their support and answers to many small but important questions, especially to Dr. Dirk Pleiter for the interesting discussions and his helpful critical comments on my drafts. Special thanks go to Willi Homberg of the Jülich Supercomputing Centre who kindly gave me access and support on the JUICE next QS22 cluster.

Maybe to the greatest part, I am deeply indebted to my girlfriend Christiane Weidauer and my family who unconditionally supported me over all these years.
Abstract

The goal of the QPACE project is to build a novel cost-efficient massive parallel supercomputer optimized for LQCD (Lattice Quantum Chromodynamics) applications. Unlike previous projects which use custom ASICs, this is accomplished by using the general purpose multi-core CPU PowerXCell 8i processor tightly coupled with a custom network processor implemented on a modern FPGA. The heterogeneous architecture of the PowerXCell 8i processor and its core-independent OS-bypassing access to the custom network hardware and application-oriented 3D torus topology pose interesting challenges for the implementation of the applications. This work will describe and evaluate the implementation possibilities of message passing APIs: the more general MPI, and the more QCD-oriented QMP, and their performance in PPE centric or SPE centric scenarios. These results will then be employed to optimize HPL for the QPACE architecture. Finally, the developed approaches and concepts will be briefly discussed regarding their applicability to heterogeneous node/network architectures as is the case in the "High-speed Network Interface with Collective Operation Support for Cell BE (NICOLL)" project.
Theses

1. The QCD oriented special purpose computer QPACE can be opened up for more general applications.

2. The Communication patterns of HPL can be mapped on the 3D torus of QPACE.

3. Collective operations can be implemented efficiently on the QPACE torus.

4. The QS22 HPL version provides a good foundation for an efficient implementation on QPACE.

5. OpenMPI components provide a portable integration of new networks.
# Contents

1 Introduction 1  
1.1 Motivation 1  
1.2 QPACE 2  
1.3 QCD Machines 3  
1.4 Message Passing Libraries 5  
1.5 NICOLL 5  
1.6 Organization 6  

2 High Performance LINPACK on QPACE 7  
2.1 Introduction to the HPL Benchmark 7  
2.1.1 Parameter 8  
2.1.2 Benchmark Algorithm 10  
2.1.3 Look-ahead 11  
2.2 The QS22 Patch for HPL 11  
2.2.1 Computation and Reorganization Specialists 12  
2.2.2 PPE and SPE Load Balancing 12  
2.2.3 Hugepage Support 13  
2.2.4 MPI Collectives 13  
2.2.5 Parameter Limitations 13  
2.3 MPI Binding 13  
2.3.1 Required MPI Features 14  
2.3.2 MPI Versus QMP 15  
2.4 HPL Communication Patterns 15  
2.5 Process Mapping 17  
2.6 Profiling 18  
2.7 Communication Sizes and Network Requirements 20  
2.7.1 Example Setup 20  
2.7.2 Message Passing and Network Requirements 21  

3 QPACE Architecture and Torus Communication 23  
3.1 Introduction to the QPACE Architecture 23  
3.1.1 QPACE Rack Configuration 24  
3.2 Torus Network Hardware 27  
3.2.1 Low-Level Communication Rules 28
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example HPL.dat optimized for a single node run on QS22</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Data distribution scheme over the process grid P x Q</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>High level visualization of the LU process</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Look-ahead illustration: top without look-ahead, bottom with look-ahead</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>SPE function offload architecture</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Embedding 2D and 1D tori in a 3D torus</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>QPACE node card schematic diagram</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of the QPACE architecture</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Torus configuration alternatives within one backplane</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Standard (top) and alternative cabling (bottom) of 4 racks</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Communication Primitives</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparison of memcpy performance</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Sending packets to the NWP</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Dimension ordered spanning tree in a $3 \times 3 \times 3$ mesh</td>
<td>45</td>
</tr>
<tr>
<td>3.9</td>
<td>Illustration of the BlueGene/L adapted broadcast on a $4 \times 4 \times 4$ mesh</td>
<td>46</td>
</tr>
<tr>
<td>3.10</td>
<td>Illustration of the 3D-EDF algorithm for a $3 \times 3 \times 3$ torus</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>MPI on QMP block diagram for a possible implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>QMP on MPI block diagram</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>List of used MPI calls in the QMP MPICH implementation</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>OpenMPI Layer Model</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>OpenMPI BTL Component Interface (OpenMPI version 1.2.8)</td>
<td>67</td>
</tr>
<tr>
<td>4.6</td>
<td>OpenMPI COLL Component Interface (as of OpenMPI 1.2.8)</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>The NICOLL Architecture</td>
<td>74</td>
</tr>
</tbody>
</table>
List of Tables

2.1 MPI Calls used in HPL .............................................. 13
2.2 Communication algorithms in HPL .......................... 16
2.4 Example Profiling output from a run on 2 QS22 .......... 19
2.5 General message sizes and call counts ..................... 20
2.6 Message sizes and call counts for example setup ........ 21

3.1 Memory Access times ............................................. 31
3.2 Edge assignment to the Spanning Trees in a 3D Torus for spanning trees 1-3 ............................................. 49
3.3 Edge assignment to the Spanning Trees in a 3D Torus for spanning trees 4-6 ............................................. 50

4.2 OpenMPI Layers (partial list) (from [1, 2]) .............. 54
4.3 Code size of the HPL library (without MPI and BLAS, and test code), compiled with spu-gcc -Os ......................... 64
List of Acronyms

ACCFS ............ Accelerator File System, page 73
APE ............... Array Processor Experiment, page 4
ASIC .............. Application-Specific Integrated Circuit, page 3
BLAS .............. Basic Linear Algebra Subroutines, page 7
BML ............... BTL Management Layer, page 54
BTL ............... Byte Transfer Layer, page 65
Cell/B.E.™ ........ Cell Broadband Engine™, page 2
COLL .............. OpenMPI Collectives Component, page 71
CP-PACS ............ Computational Physics by Parallel Array Computer System, page 4
DCR ............... Device Control Register, page 24
DDR ............... Double Data Rate, page 4
DMA ............... Direct Memory Access, page 2
DSP ............... Digital Signal Processor, page 4
EIB ............... Element Interconnect Bus, page 2
FLOP .............. Floating Point Operation, page 3
FPGA .............. Field Programmable Gate Array, page 3
FPF ............... Floating Point Unit, page 4
GBIF .............. Global Bus Infrastructure, page 28
GPGPU ............. General Purpose computing on Graphic Processing Units, page 74
HPL ............... High Performance Linpack, page 1
IO-MMU ............ IO Memory Management Unit, page 39
IWC ............... Inbound Write Controller, page 24
LQCD .............. Lattice Quantum Chromodynamics, page 2
MFC ............... Memory Flow Controller, page 2
MMIO .............. Memory-mapped Input/Output, page 29
MPI ............... Message Passing Interface, page 5
NICOLL ............. High-speed Network Interface with Collective Operation Support for Cell BE, page 5
NWP ............... NetWork Processor, page 24
OWC ............... Outbound Write Controller, page 24
PCIe® ......... PCI Express®, page 73
PML ............... Point to Point Management Layer, page 66
PPC64 ............ PowerPC®64, page 2
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPE</td>
<td>Power Processing Element</td>
<td>2</td>
</tr>
<tr>
<td>QCD</td>
<td>Quantum Chromodynamics</td>
<td>2</td>
</tr>
<tr>
<td>QMP</td>
<td>Lattice QCD Message Passing</td>
<td>14</td>
</tr>
<tr>
<td>QPACE</td>
<td>QCD Parallel computing on Cell Broadband Engine™</td>
<td>2</td>
</tr>
<tr>
<td>QS21</td>
<td>IBM BladeCenter® QS21</td>
<td>3</td>
</tr>
<tr>
<td>QS22</td>
<td>IBM BladeCenter® QS22</td>
<td>3</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
<td>4</td>
</tr>
<tr>
<td>RSPUFS</td>
<td>Remote SPU File System</td>
<td>73</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
<td>2</td>
</tr>
<tr>
<td>SPE</td>
<td>Synergistic Processing Element</td>
<td>2</td>
</tr>
<tr>
<td>SXU</td>
<td>Synergistic Execution Unit</td>
<td>2</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very-Large-Scale Integration</td>
<td>4</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter gives an introduction to the topics tackled in this diploma thesis. Section 1.1 explains the motivation of special purpose supercomputers and the High Performance LINPACK benchmark to evaluate these systems. The QPACE machine and its main components are briefly described in section 1.2. Other QCD machines are described and compared to QPACE in section 1.3. The message passing libraries which can be considered for the QPACE machine are introduced in section 1.4. The NICOLL project and its objective is introduced in section 1.5. Finally the organization of the rest of this thesis is described in section 1.6.

1.1 Motivation

Supercomputers are one of the most important research vehicles for modern science. The applications demanding extraordinary high computing power range from climate research, weather forecasting, quantitative finance, and quantum mechanical physics simulation to molecular modeling and cryptanalysis, to name only a few. General purpose clusters which are built from commodity parts are used to support a wide range of these applications, but also special purpose machines like the QPACE supercomputer are built to support one specific application efficiently. The hardware architecture of these special purpose machines is designed according to the requirements of the target problem and can reach higher performance for these problems at a lower cost or power consumption.

To compare the different supercomputers, the High Performance LINPACK [3] benchmark (“HPL”), a portable version of LINPACK [4] for distributed memory systems, measures the floating point performance by solving a linear system $Ax=b$. This problem is one of the most time critical kernels within many applications, and therefore this benchmark is traditionally used to compare supercomputers on the basis of the sustained double precision floating point operations per second (“FLOP/s”) in this benchmark. Other Benchmarks like the HPC Challenge [5] benchmark which evaluate a wider range of properties...
CHAPTER 1. INTRODUCTION

of the supercomputers have been proposed, but HPL is still the most popular benchmark for comparison.

A public ranking of the fastest supercomputers based on their HPL performance has been maintained since 1993 in the top 500 list [6] which is updated twice a year. This list has been established to detect trends in the high-performance computing market. An early implementation of HPL on new machines is therefore not only interesting for performance evaluation but also important for promotional reasons.

The green500 list [7] is a new ranking established in 2007 which ranks the most energy efficient supercomputers from the top 500 list based on their HPL performance per Watt ratio. In this list the focus moves from raw computing power to energy efficiency of the machines which becomes increasingly important for modern data centers. The machines on the first 7 ranks of the current green500 list (November 2008) are based on IBM® PowerXCell® 8i processors, which are one of the most energy efficient commodity processors available today. The QPACE machine which also uses this processor along with a very power efficient architecture could be a strong competitor on this green500 list.

1.2 QPACE

The QPACE machine ("QCD Parallel computing on Cell Broadband Engine™") [8, 9] is a supercomputer designed for Lattice Quantum Chromodynamics (LQCD) applications. Quantum Chromodynamics (QCD) is a well established theory describing interactions between quarks and gluons, the building blocks of particles such as the neutron and the proton. To perform numerical simulation which are necessary to tackle certain QCD problems, Lattice QCD uses a discretized formulation on a space-time lattice which is suitable for massively parallel computers. QCD on the Cell/B.E.™architecture was already explored by many different groups [10, 11, 12, 13, 14]. One of the most time consuming kernels in LQCD codes is the Wilson-Dirac Operator [15], and it has been shown that an implementation on the PowerXCell 8i processor can reach an efficiency of 25% of the peak performance [10]. Hence, The Cell/B.E. architecture appears to be a very promising platform.

The Cell Broadband Engine processor™(Cell/B.E.) [16] is a heterogeneous multicore microprocessor jointly developed by IBM, Sony and Toshiba. It features one general purpose Power PC compatible core called PPE ("Power Processing Element") and 8 vector processing units called SPEs ("Synergistic Processing Element"), which are interconnected on the chip with the high performance ring bus EIB ("Element Interconnect Bus"). While the PPE is dedicated to the operating system and PPC64 compatible legacy applications, the SPEs provide the bulk of potential of the architecture. These SIMD processors don't access the main memory directly, but operate on a 256 KiB local memory called “Local Store” and transfer data from and to memory using an integrated DMA engine called MFC ("Memory Flow Controller"). The processing core SXU ("Synergistic Execution Unit") of a SPE is decoupled from the memory
CHAPTER 1. INTRODUCTION

transfers, which allows an overlap of computation and communication. Commercially available machines which use the Cell/B.E. architecture are the Sony PlayStation 3 which uses a Cell/B.E. processor with 7 enabled SPEs, and the IBM BladeCenter® QS20 and QS21 which are SMP systems with 2 Cell/B.E. processors. The IBM BladeCenter QS22 is the successor of these systems and is equipped with 2 PowerXCell 8i processors, an updated version of the Cell/B.E. with highly improved double precision floating point performance of the SPEs.

The QPACE networks are implemented on an FPGA ("Field Programmable Gate Array"). The advantage of using an FPGA over a custom ASIC ("Application-Specific Integrated Circuit") are the lower risk and development costs of the system, and the possibility to reconfigure the chips even after the system is deployed. 3 networks are implemented: the standard Gigabit Ethernet for storage access and maintenance, a global signal tree for application communication and exception handling, and a custom high performance 3D torus network with nearest neighbor connections which is designed for the QCD applications.

The QPACE architecture is designed to be a low power architecture with very high performance. The PowerXCell 8i processor is a suitable choice for this, as it delivers the highest performance per watt of the currently available commodity processors. Implementing only the required south bridge functionality in the FPGA allows to further reduce the number of power consuming standard components. By using special metal enclosures for the node cards and a cost-efficient cooling system, a very high package density can be reached.

At the time of writing this thesis, the QPACE machine is still being actively developed with several academic institutions together with IBM at the IBM Research and Development Lab in Böblingen (Germany). The QPACE torus network is not yet finished to allow library implementations or a final performance evaluation. The contribution of this work is for this reason not a finished presentation of an implemented application or message passing library, but presents concepts, requirements, and performance extrapolations and proposes enhancements.

1.3 QCD Machines

Specialized QCD machines have a long history [17]. One of the first QCD machines were the PACS/PAX [18, 19] series installed at the Kyoto University and the University of Tsukuba. The first PACS machine PACS-9, which was built in 1978, used a 3 × 3 two dimensional grid of 8 bit microprocessors. The processors were equipped with 1 KiB of memory, reaching an aggregate performance of 0.01 MFLOP/s (Million Floating Point Operations per Second) on solving Poisson equation. Several upgraded machines were built, and the latest PACS machine installed in 2005 is the PACS-CS [20], employing low-power Intel® Xeon® processors interconnected in a Gigabit Ethernet based 3 dimensional hyper crossbar of the size 16 × 16 × 10. The whole system reaches a peak performance of 14.3 TFLOP/s, and is the first to break with the tradi-
CHAPTER 1. INTRODUCTION

tion of custom processors and network interfaces which were still used for its predecessor CP-PACS [21] in 1997.

The QCDSP [22, 23] machines installed in 1998 at Columbia University and RIKEN-BNL research center employ Texas Instruments DSPs (“Digital Signal Processor”) for the calculation and a custom ASIC for accessing the 4D mesh network and memory prefetching. The two machines reach a combined peak performance of 1 TFLOP/s, and a maximum sustained performance of 30% of the peak performance for the QCD code. The successor QCDOC [23, 24, 25] (“QCD on a Chip”) instead used a IBM System-on-a-Chip design. A PPC440 core with an attached 64 bit IEEE FPU, 4 MiB1 memory, 2 Ethernet Controllers, a DDR RAM controller, and a communication controller supporting 24 communication links were combined on a custom ASIC. With this ASIC, all components of the QCDSP node card are combined on a single chip. A 6D mesh network, standard Ethernet and a global interrupt network were employed to interconnect the node cards. The machines installed in 2004 and 2005 at the Brookhaven National Laboratory and the University of Edinburgh both reach a peak performance greater than 10 TFLOP/s.

Another family of QCD computers with a long history are the APE machines (“Array Processor Experiment”) which were designed and manufactured by different research groups of the Italy based INFN (National Institute of Nuclear Physics), DESY (Deutsches Elektronen Synchrotron), and Université Paris-Sud 11 in Orsay. After the first APE machine [26] which was built starting 1984 and featured a 16 node linear array system, the APE 100 [27], the APEmille [28] and APEnext [29] machines were built and deployed in this chronological order. The APEnext machine features a custom VLIW processor (“Very Long Instruction Word”) implemented in a VLSI chip (“Very-large-scale integration”) for computation and 32 MiB RAM on each node card. 16 node cards are combined on one processing board. They are backed up by Linux® based standard PC host systems which handle the start up and disk I/O. A backplane crate holds 16 of these processing boards. The nodes are interconnected with a custom 3D torus network which is integrated in the processing boards and backplane crates. The largest installation can be found in Rome with 6656 nodes arranged in 13 racks, reaching a theoretical peak performance of 10.6 TFLOP/s.

Different factors motivated these groups to use custom hardware instead of commodity hardware. The standard processors available did not offer a reasonable computing power to implement the QCD kernels efficiently. This argument became weaker with the SIMD extensions in conventional microprocessor like SSE. Another reason are the high power consumption of the standard processors. The custom microprocessors work with a fraction amount of power which standard processors would need, delivering the same performance for the specific application. This allows a much tighter packaging and better cooling. The optimal network for QCD applications is a low latency multi dimensional mesh network, and the coupling of the network and the processing elements can be

1The IEC binary prefixes for storage units are used in this document. 1 GiB = 1024 MiB, 1 MiB = 1024 KiB, 1 KiB = 1024 Byte
built much tighter with a custom processor.

The decision for QPACE to use a PowerXCell 8i processor was motivated by the fact that it can deliver high peak performance with a very good performance per Watt ratio, and the needed QCD kernels could be implemented quite efficiently on this architecture [11, 10]. The tight coupling is accomplished by directly connecting the PowerXCell 8i processor with the FPGA without using a south bridge between. The costs and complexity to develop a special purpose VLSI in modern chip technologies are so high that they are no longer feasible for an academic project of this scale.

1.4 Message Passing Libraries

The Message Passing Interface Standard ("MPI") [30, 31] is the de facto standard for message passing and communication on general purpose distributed memory systems. It is implemented in a library with Interfaces to C, C++ and FORTRAN. Optimized libraries are available for many different processor and network architectures. MPI provides various send and receive mechanisms between pairs of processes, so called “Point to Point Communication”, as well as communication within a group of processes, so called “Collective Communication”. These groups of processes are organized in so called “Communicators”. A communicator is a subset of processes from the complete process set, and can be configured according to the distribution requirements of the application. There are also special communicators like the cartesian communicator or graph communicators available which allow to align the application to special logical topologies. Furthermore the messages may use arbitrary datatypes, and MPI takes care of the correct packing and machine datatypes. This feature allows to use heterogeneous clusters where the machine words may have a different format. The MPI 2 standard extends these features with one sided communication, dynamic process management and I/O functionality.

Another message passing standard is the QCD Message Passing Interface (QMP). It is designed for QCD applications which usually communicate in lattice topologies. QMP provides point to point communication with a special focus on repetitive communication patterns. The communication is done between the neighbors of a logical multi dimensional torus or between arbitrary pairs of processes. QMP also provides a set of collective algorithms which QCD applications use, but not as many and general ones as MPI does. These collectives only work on the complete set of processes. QMP can therefore be considered as a functional subset of MPI.

1.5 NICOLL

The NICOLL project (High-speed Network Interface with Collective Operation Support for Cell BE) [32], a project of the Computer Architecture Group at Chemnitz University of Technology in collaboration with the Center of Ad-
CHAPTER 1. INTRODUCTION

Advanced Study IBM Böblingen, Development and Research, is a case study of a heterogeneous system with an AMD Opteron™ microprocessor and a Cell/B.E. microprocessor for acceleration. The microprocessors are tightly coupled using an FPGA as bridge between the AMD HyperTransport™ protocol and the Cell/B.E. interface. The objective of this project is to develop a research prototype and to explore the potential of this tightly coupled hybrid system in an HPC environment using InfiniBand as interconnect. One main focus of this project is to research the potential of the Cell/B.E. architecture as accelerator for collective operations.

1.6 Organization

This Diploma Thesis is organized into 4 parts: An overview of HPL and a discussion how to implement this application efficiently on QPACE is presented in chapter 2. In chapter 3, approaches to use the QPACE torus network for general purpose, which was originally designed for QCD applications, are presented, and torus-optimized collective algorithms are discussed. Possible general message passing libraries for the QPACE architecture and consequences for the programming models and HPL are discussed in chapter 4. Finally, this work is concluded and the applicability of these results and strategies are discussed for the NICOLL architecture in chapter 5.
Chapter 2

High Performance LINPACK on QPACE

This chapter describes the HPL application and optimization approaches on the QPACE architecture. Section 2.1 and 2.2 give an introduction to the HPL Benchmark and its PowerXCell 8i optimized QS22 version. The communication patterns and implementation on MPI is analyzed in sections 2.3 and 2.4. The process mapping for the QPACE HPL version is discussed in section 2.5. Finally, from profilings of the existing HPL version as described in section 2.6, an extrapolation of the expected message patterns and sizes is given for a large QPACE setup in section 2.7, and requirements for network hardware are formulated.

2.1 Introduction to the HPL Benchmark

HPL [3, 33] is the High Performance LINPACK Benchmark, a portable, scalable implementation of LINPACK [4] used to measure and compare the performance of large supercomputers with shared or distributed memory. The Software package depends on either Basic Linear Algebra Subprograms (BLAS) [34, 35] or Vector Signal Image Processing Library (VSIPL) [36] for the local linear algebra tasks and on an MPI 1.1 [30, 31] compliant implementation for Message Passing.

LINPACK implements a solver for a dense system of linear equations \( Ax = b \) using the right-looking LU factorization [37, 38] as a variant of the Gaussian elimination. To leverage the processing power with its memory hierarchies, blocking algorithms are used which enable the implementation of most performance-critical routines with Level 3 BLAS routines [3, 39], which are known to reach execution times close to peak performance on most architectures.

The following analysis is based on the HPL version 1.0a, freely available on the HPL website [33]. At this time of writing, also HPL version 2.0 is available, which adds a better random number generator [40] and a new correctness test.
CHAPTER 2. HIGH PERFORMANCE LINPACK ON QPACE

As these changes are not critical for the QPACE case and the QS22 patch is written for version 1.0a, the version 2.0 was not used.

2.1.1 Parameter

The Benchmark is controlled by various parameters from the input file which control the algorithm variants implemented in LINPACK. The 4 most important parameters are:

- The problem size $N$ specifies the size of the linear system, where a $N \times N$ matrix is factorized. This parameter is limited by the available RAM of the system.

- The block size $NB$ specifies the sizes of the sub-matrices used for computation. It depends on the employed BLAS implementations and hardware characteristics like cache size and architecture.

- The grid size $P \times Q$ specifies the data distribution of the blocks. It depends mostly on the network interconnect between the processes.

Other parameters select algorithm variants for the local factorization and various thresholds. Optimal values for them have been found by single node runs. Furthermore there are parameters to select different algorithms for the communication. The selection of the algorithm should depend on whether they can be implemented in the high-capacity torus network. These algorithms are further investigated in 2.4. Another interesting parameter is the depth of the lookahead, which is described in 2.1.3. A complete sample configuration file used for a single node is given in Figure 2.1.
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out output file name (if any)
6  device out (6=stdout,7=stderr,file)
1  # of problems sizes (N)
20479 Ns
1  # of NBs
128 NBs
0  PMAP process mapping (0=Row-,1=Column-major)
1  # of process grids (P x Q)
1  Ps
1  Qs
16.0 threshold
1  # of panel fact
0  PFACTs (0=left , 1=Crou, 2=Right)
1  # of recursive stopping criterium
2  NBMINs (>= 1)
1  # of panels in recursion
2  NDIVs
1  # of recursive panel fact.
0  RFACTs (0=left , 1=Crou, 2=Right)
1  # of broadcast
0  BCASTs (0=1rg ,1=1rM,2=2rg ,3=2rM,4=Lng,5=LnM)
1  # of lookahead depth
1  DEPTHS (>=0)
3  SWAP (0=bin-exch,1=long,2=mix,3=MPi-coll)
64  swapping threshold
1  L1 in (0=transposed,1=no-transposed) form
0  U in (0=transposed,1=no-transposed) form
1  Equilibration (0=no,1=yes)
64  memory alignment in double (> 0)

Figure 2.1: Example HPL.dat optimized for a single node run on QS22

The blocks are distributed in a row and block cyclic fashion over the process grid PxQ as illustrated in Figure 2.2. This distribution assures good load balancing over the processes while maintaining local alignment to block sizes within the main memory. Locally, the data is stored in column major order format.

A process only communicates with other processes within its local row and local column depending on its position in the process grid. The ratio of P and Q therefore has a strong influence on the communication pattern and load. A description of the communication pattern and its communication space (rows or columns) is given in section 2.4, and the message sizes and number of calls depending on the grid size is given in section 2.7.
2.1.2 Benchmark Algorithm

The main benchmark can be divided into the following steps for a single run:

1. Init and Setup
2. for all panels in A
   (a) factorize a panel
   (b) broadcast the panel
   (c) update trailing sub-matrix
3. backward substitution
4. check the solution

The time for steps 2 and 3 are measured and used for the time result of the benchmark. Step 2 is visualized in Figure 2.3. The panel factorization step 2a is only performed by the column process group of the grid which “owns” the panel. The other processes receive the factorized panel via the broadcast in step 2b and use it to update the trailing sub-matrix in step 2c. The sub-matrix is shrunk by the block size in both dimension in each iteration, and by the end of step 2 the matrix A is replaced by the lower triangle matrix L and the upper triangle matrix U. In step 3, backward substitution is used to receive the final result vector x. To verify the results, various scaled residuals are computed to check the correctness and precision of the computation.
2.1.3 Look-ahead

A very popular optimization of blocking factorization algorithms is the “look-ahead”. With the basic algorithm as described above, columns which do not factor a panel have to wait for the broadcast and the matrix update before they can start to factorize their own panel. This creates idle times as we can see in Figure 2.4. With look-ahead, the matrix update is postponed and the factorization of the local panel is brought forward, thus creating an overlap of the different steps. The dependencies of the different steps of the LU part are relaxed, and the pipelining effect removes idle times on the processors. Different look-ahead depths can be selected for HPL, where a depth of 1 or 2 is assumed to give the best performance. More detailed information about this technique can be found e.g. in [41, 42].

2.2 The QS22 Patch for HPL

The QS22 patch for HPL optimized by IBM [43] contains multiple optimizations to maximize the performance on QS22 Blade clusters. Because QS22 architecture shares the same processor PowerXCell 8i and have comparable memory
CHAPTER 2. HIGH PERFORMANCE LINPACK ON QPACE

performance as the QPACE node cards, the QS22 Patch for HPL offers a good foundation for a QPACE optimized HPL version.

2.2.1 Computation and Reorganization Specialists

The performance of HPL is dominated by the BLAS routines DGEMM and DTRSM. The QS22 patch contains so called “specialists”, optimized routines for certain kernels which are implemented on the SPEs. The data is reordered into a custom blocked-row format by SPE reorganization specialists before calling the computation specialists to reach maximum performance. The data is reformatted back just before its panel factorization, and by the end of the LU process it is completely formatted in the conventional column major order format.

SPE routines are called by function offloading: the PPE informs the SPEs by mailbox messages about the task to do. The SPEs then fetch the parameters from the main memory, execute the job and notify the PPE about the completion by setting a Byte in a completion array in main memory.

2.2.2 PPE and SPE Load Balancing

![Figure 2.5: SPE function offload architecture](image)

Most of these specialists are called synchronously, i.e. the PPE starts the SPE and waits (idles) for their completion. The only point where the specialists are called asynchronously is within the update of the trailing sub-matrix: the SPEs are called for a (very long) DGEMM matrix multiplication operation while the PPE polls for the panel broadcast. Figure 2.5 illustrates the design of this function offload mechanism.
2.2.3 Hugepage Support

Using the conventional main memory with 4KiB page size to store the matrix produces a lot of TLB misses when working on large data sets, which slows down the performance. This TLB trashing can be avoided by using huge pages with a page size of 16 MiB, as much less page entries are needed to reference the matrix data.

2.2.4 MPI Collectives

The matrix update step algorithm versions which were implemented using point to point communication primitives in the original HPL were extended by a version using MPI collectives. The MPI library might provide algorithms for these tasks which are optimized for the underlying network and perform better than the original point to point versions.

2.2.5 Parameter Limitations

The computation specialists operate on block sizes of 128 elements, thus the parameter NB is fixed to 128. As the matrix A is stored along with the vector b in an \( N + 1 \times N \) matrix, the best performance is reached if \( N \) has the form \( N = 128 \cdot k - 1, k \in \mathbb{N} \). All blocks will then be properly aligned. L1 must not be transposed (1), and U must be transposed (0), as only this parameter constellation is implemented. The memory alignment must be set to multiples of 64 double words.

2.3 MPI Binding

The following list of HPL/LINPACK MPI Functions has been extracted from the HPL binary with the IBM patch.

<table>
<thead>
<tr>
<th>Component</th>
<th>MPI Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>MPI_Init, MPI_Wtime, MPI_Finalize, MPI_Abort</td>
</tr>
<tr>
<td>Message Passing</td>
<td>MPI_Send, MPI_Ssend, MPI_Issend, MPI_Recv,</td>
</tr>
<tr>
<td></td>
<td>MPI_Irecv, MPI_Iprobe, MPI_Wait</td>
</tr>
<tr>
<td>Datatype Management</td>
<td>MPI_Type_vector, (MPI_Type_contiguous), (MPI_Type_struct),</td>
</tr>
<tr>
<td></td>
<td>MPI_Type_commit, MPI_Type_free, MPI_Address</td>
</tr>
<tr>
<td>Communicator Management</td>
<td>MPI_Comm_size, MPI_Comm_free, MPI_Comm_rank, MPI_Comm_split</td>
</tr>
<tr>
<td>Collectives (new from IBM</td>
<td>MPI_Barrier, MPI_Allgatherv, MPI_Scatterv</td>
</tr>
<tr>
<td>patch)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: MPI Calls used in HPL
2.3.1 Required MPI Features

The requirements on the MPI implementation are discussed for each feature set. This is important for a possible early MPI subset for the QPACE torus, and the decision if the QCD Message Passing Library QMP is sufficient for the HPL Benchmark.

Message passing

Panel broadcasts are processed in a nonblocking way. Many functions probe at their beginning or end if a panel is received, and forward it if needed. Instead of probing and then calling MPI_Recv, it is possible to use a nonblocking MPI_Irecv operation and poll for its completion with MPI_Test instead.

Messages do not overtake each other for one pair of sender and receiver in HPL. Panel broadcast communication is overlapped with panel factorization communication, but these communication patterns are performed in different communicators, either row or column communicators.

Probing for a message is not possible in QMP. It would be necessary to replace the probing mechanism with the nonblocking receive counterparts as described above. Nonblocking receive operations are supported by QMP.

Communicators

MPI_Comm_split() is used to build row and column communicators. Most collectives happen in one column (panel factorization) or in one row (panel broadcast etc). The process grid spanned by HPL can be mapped on the 3D-torus as described in section 2.5. The resulting communicators are tori of lower dimension.

QMP does not provide communicators, but mapping row/column communicator ranks to global ranks is trivial, which is sufficient as long as primitive send and receive operations are used.

Datatypes

Most usage of MPI Datatypes can be turned off by setting the define HPL_NO_MPI_DATATYPE. However, MPI_Type_vector datatypes are not disabled. They are used to send non-contiguous pieces of a block in various functions. QMP does provide support for vector and strided datatypes, but not for structured datatypes. By using the HPL_NO_MPI_DATATYPE directive, the QMP datatypes can be employed without further changes.

Collective communications

Most of the Communication in LINPACK is collective communication. In the original HPL it is implemented with only Send/Recv primitives to allow benchmarking of early machine prototypes which only have an MPI subset available. It is possible to replace some of them with MPI collectives, but others like the
panel broadcast overlap computation and communication and call the collectives in a nonblocking way. Replacing these algorithms with blocking MPI collectives would kill the performance. An overview of used communication algorithms is given in section 2.4. Using MPI collectives has the advantage that they could be optimized for the 3D torus topology, while the original implementations use hypercube algorithms or other topologies which can not be mapped on a torus with nearest neighbor communication only.

Besides initialization routines which use the MPI_COMM_WORLD communicator, all time-critical communicators use either row or column communicators. The collectives must therefore support these sub communicators. QMP does not support sub communicators. It provides only a few of the needed collectives with limited functionality (e.g. broadcast only from root = 0) in the global communicator.

2.3.2 MPI Versus QMP

The QMP collectives don’t appear to be useful for HPL. The collective routines have to be implemented from scratch for the torus in both the MPI and the QMP case. There is no aspect that would make collectives on the QMP primitives impossible. However, implementing them directly on the torus hardware instead of QMP has more optimization potential, for example a broadcast could use a cut through approach to forward a packet as soon as it arrives instead of waiting for a whole buffer. A light-weight MPI subset with the required collectives could be implemented directly on the low-level torus API. This could be reused for other applications later, even by a QMP implementation.

The Message Passing library must support the main memory as valid position for message buffers, as the matrix data is stored in main memory. Flow control and buffer management in the collectives algorithms can also take this issue into account for further optimization, e.g. by scheduling when the buffers are kept in the Local Store or written back to main memory.

2.4 HPL Communication Patterns

An overview of the communication algorithms of HPL is given in Table 2.2. We will evaluate whether the functions can be replaced by MPI collectives to optimize them for the 3D torus topology and gain performance advantages.

As we can see, all communication can be optimized to use the 3D torus topology efficiently, given a reasonable process-to-node mapping. Some algorithms use ring topologies which can be embedded into the torus. Other algorithms can be replaced with MPI counterparts which in turn can use optimized trees within the 3D torus. Each of them can be implemented to use nearest neighbor communication only (see sections 2.5 and 3.4).
<table>
<thead>
<tr>
<th>Section</th>
<th>Source code</th>
<th>Communicator</th>
<th>Notes</th>
<th>MPI replaceable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td><code>HPL_pdinfo</code>, <code>HPL_allreduce</code>, <code>HPL_broadcast</code></td>
<td>world</td>
<td>Initialization, exchanging parameters and other information. not time critical</td>
<td>The allreduce and broadcast can be replaced trivially by their MPI counterparts</td>
</tr>
<tr>
<td>Setup &amp; Check</td>
<td><code>HPL_barrier</code>, <code>HPL_reduce</code>, <code>HPL_broadcast</code></td>
<td>world, rows, columns</td>
<td><code>HPL_pdttest</code> calls the main function (<code>HPL_pdgesv</code>) and some not time critical functions like barrier, <code>HPL_pdlange</code> for calculating the norm, etc</td>
<td>The reduce, broadcast and barrier can be replaced trivially by their MPI counterparts</td>
</tr>
<tr>
<td>Panel Broadcast</td>
<td><code>HPL_bcast</code>*</td>
<td>rows</td>
<td>6 different broadcast algorithms (increasing ring, increasing ring modified, increasing 2 ring, increasing 2 ring modified, long, long modified) are provided in the original HPL algorithm. Other functions within the panel factorization are probing for available panels from time to time to forward them while overlapping with the computation.</td>
<td>It is not advisable to replace these nonblocking functions with blocking MPI collectives. However topologies like the increasing ring can be mapped into the torus without performance disadvantages (always nearest neighbor communication).</td>
</tr>
<tr>
<td>Panel Factorization</td>
<td><code>HPL_pdpan</code>, <code>HPL_pdmxswap</code></td>
<td>columns</td>
<td>A hypercube algorithm is used to broadcast the current row and swap (reduce) the maximum row at the same time.</td>
<td>The hypercube algorithm can be replaced with an MPI_Bcast() and MPI_Allreduce() using a custom-defined combination function.</td>
</tr>
<tr>
<td>Update trailing sub-matrix</td>
<td><code>HPL_pdsupdate</code>, <code>HPL_pdlaswp</code>, <code>HPL_roll</code>, <code>HPL_spread</code></td>
<td>columns</td>
<td>In the original HPL, 3 algorithms are provided for the trailing sub-matrix update: (0) binary exchange, (1) long (using <code>HPL_roll</code> and <code>HPL_spread</code>), and (2) a mix of both.</td>
<td>The IBM patch adds another algorithm (3) using the MPI collectives <code>MPI_Allgatherv()</code> and <code>MPI_Scatterv()</code>.</td>
</tr>
<tr>
<td>Backward Substitution</td>
<td><code>HPL_pdttrs</code>, <code>HPL_broadcast</code></td>
<td>columns, rows</td>
<td>Is called after the LU process. The algorithm uses a decreasing ring broadcast with look-ahead depth 1 over the rows. Another decreasing ring broadcast is then done in the column communicator. <code>HPL_broadcast</code> is called for the last solution block.</td>
<td><code>HPL_broadcast</code> is easy to replace. The decreasing ring broadcasts are not so trivial to replace because the computation is embedded within each step of the broadcast. However a ring can be mapped into a torus.</td>
</tr>
</tbody>
</table>

Table 2.2: Communication algorithms in HPL
2.5 Process Mapping

Figure 2.6: Embedding 2D and 1D tori in a 3D torus

A key factor to efficiently use the QPACE torus network in HPL is to map the process grid into the 3D torus such that row and column communication of each process can be performed using nearest neighbor communication. We therefore need to embed the row and column communicators, which are logically utilized as 1D-rings or collective communications, into the torus. One possible mapping of 2 rings, expressed as functions from cartesian coordinates to ranks, is:

\[
\begin{align*}
\text{ring1}(x, y, z) &= x \\
\text{ring2}(x, y, z) &= z \cdot n_y + \text{even}(z) \cdot y + (1 - \text{even}(z)) \cdot (n_y - y)
\end{align*}
\]

\[
\text{even}(z) = \begin{cases} 
1 & \text{z is even} \\
0 & \text{z is odd}
\end{cases}
\]

For tori with even dimensions, this mapping creates rings with length \( n_x \) and \( n_y \cdot n_z \) which suffice the topology requirements of HPL. An illustration is given in Figure 2.6. Note that \text{ring2} can also be used as 2D torus (the additional edges are dotted in the illustration). It can be used for collective communication which will be faster than only using the logical ring because of more links and smaller path lengths.
The QPACE hardware allows different physical configuration of the links, as
described in section 3.1.1. Assuming the maximum configuration of one rack,
the tori sizes for the default cabling are \( n_1 \times n_2 \times n_3 = 8 \times 16 \times 2 \cdot \# \text{Racks} \).
With a logical torus \( n_x = n_2, n_y = n_1, n_z = n_3 \), a “square” symmetric grid of
dimension \( 16 \times 16 \) can be generated with 1 rack. The largest grid \( 16 \times 128 \) can
be generated with 8 racks. Because of the asymmetry in the 3 dimension, this
is the most symmetric 2D mapping which can be generated for 8 racks with the
default cabling. With an alternative cabling (see section 3.1.1), a grid size of
\( 32 \times 64 \) could be used with 8 racks.

Another possible approach is to implement only one of the communicators
in the torus, and to use the other one with GBit Ethernet. This relaxes the
mapping constraints. For example the column communicator, which is latency
sensitive in the panel factorization, can be implemented in \( 2 \times 4 \times 4 \) cube. The
row communicator could use the GBit Ethernet. 3D cubes have lower latencies
because of shorter path lengths than 2D tori, for large process numbers. The
relaxed neighborhood constraint allows more square process mappings. However
the PPE-limited bandwidth of the Gigabit Ethernet device might be a severe
bottleneck.

2.6 Profiling

To find bottlenecks and confirm assumptions about running times of the different
sections, the HPL source code has been instrumented. Every MPI call and every
BLAS routine was wrapped into a respective wrapper routine which measures
the execution time using the high resolution timers of the PPE. The parameters
passed to the functions were evaluated, which allows to calculate the FLOP
count for each BLAS routine and the message sizes of MPI calls. The source
code is divided in the different sections as described above, and the calls are
accounted to their respective section. Based on these evaluations we can compile
aggregate statistics for the routines and see if they perform as expected.

Table 2.4 gives an example of a profiling output. The data has been gathered
on 2 QS22 blades with the modified and instrumented QS22 version of HPL.
The benchmark was run on a problem size \( N = 30847 \) with a grid \( P \times Q = 2 \times 2,\)
with 2 processes per node.
<table>
<thead>
<tr>
<th>Section</th>
<th>Operation</th>
<th>Calls</th>
<th>Time (s)</th>
<th>GFLOP</th>
<th>GFLOP/s</th>
<th>MiB</th>
<th>MiB/s</th>
<th>Message Size range (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trailing Matrix</strong></td>
<td>dgemm</td>
<td>358</td>
<td>47.41</td>
<td>4832.006</td>
<td>101.927</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>dtrsm</td>
<td>358</td>
<td>0.32</td>
<td>30.199</td>
<td>94.409</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Allgatherv</td>
<td>358</td>
<td>5.26</td>
<td>1800.000</td>
<td>342.238</td>
<td>128.00, 15232.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Scatterv</td>
<td>358</td>
<td>2.85</td>
<td>1800.000</td>
<td>342.238</td>
<td>128.00, 15232.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Panel</strong></td>
<td>dgemm</td>
<td>15120</td>
<td>2.45</td>
<td>14.864</td>
<td>6.069</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Factorization</strong></td>
<td>dtrsm</td>
<td>15120</td>
<td>0.08</td>
<td>0.042</td>
<td>0.520</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dtrsv</td>
<td>7680</td>
<td>0.02</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dscal</td>
<td>15360</td>
<td>0.24</td>
<td>0.118</td>
<td>0.498</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dgemv</td>
<td>7680</td>
<td>0.94</td>
<td>0.236</td>
<td>0.252</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dcopy</td>
<td>46191</td>
<td>1.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>idamax</td>
<td>15360</td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Bcast</td>
<td>15360</td>
<td>0.37</td>
<td></td>
<td></td>
<td>15.000</td>
<td>40.575</td>
<td>1.00, 1.00</td>
</tr>
<tr>
<td></td>
<td>MPI_Allreduce</td>
<td>15360</td>
<td>0.72</td>
<td></td>
<td></td>
<td>15.469</td>
<td>21.414</td>
<td>1.03, 1.03</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>dtrsv</td>
<td>120</td>
<td>0.02</td>
<td>0.002</td>
<td>0.100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Substitution</strong></td>
<td>dgemv</td>
<td>119</td>
<td>1.30</td>
<td>0.236</td>
<td>0.181</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dcopy</td>
<td>120</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>daxpy</td>
<td>120</td>
<td>0.00</td>
<td>0.000</td>
<td>0.083</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Send</td>
<td>120</td>
<td>0.01</td>
<td>0.117</td>
<td>17.601</td>
<td>1.00, 1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>120</td>
<td>0.01</td>
<td>0.117</td>
<td>14.565</td>
<td>1.00, 1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Panel Broadcast</strong></td>
<td>MPI_Send</td>
<td>120</td>
<td>2.54</td>
<td>907.618</td>
<td>362.276</td>
<td>129.01, 15361.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>121</td>
<td>2.57</td>
<td>922.742</td>
<td>359.513</td>
<td>127.01, 15489.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Iprobe</td>
<td>3590019</td>
<td>7.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Setup</strong></td>
<td>MPI_Send</td>
<td>3</td>
<td>0.00</td>
<td>0.000</td>
<td>0.280</td>
<td>0.00, 0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>7</td>
<td>4.04</td>
<td>0.000</td>
<td>0.000</td>
<td>0.00, 0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Check</strong></td>
<td>dgemv</td>
<td>1</td>
<td>2.68</td>
<td>0.472</td>
<td>0.176</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Send</td>
<td>6</td>
<td>1.75</td>
<td>0.586</td>
<td>0.334</td>
<td>0.01, 120.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>6</td>
<td>2.52</td>
<td>0.000</td>
<td>0.000</td>
<td>0.01, 0.01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Example Profiling output from a run on 2 QS22
We can see that the dgemm and dtrsm operations in the trailing matrix update section, which is the most time consuming one, are performing nearly at the peak performance of the 8 SPEs [44] which is 102.4 GFLOP/s. This is due to the fact that mostly big data sets are used in this section. On the other hand, the same operations in the panel factorization section perform much slower because mostly small or thin matrices are used in this section.

### 2.7 Communication Sizes and Network Requirements

Based on the profiling results and analysis of the source code it is possible to estimate the sizes of the messages and call counts of the individual routines. Table 2.5 shows an overview of the message sizes and calls depending on the main parameters. The other parameters are set according to the sample configuration file in Figure 2.1. The message sizes for the collective operations are given as the combined size of all parts of the message, e.g. for a scatter the original vector (and not its parts) are displayed. A factor for the message sizes accounts for the part of the message size sent or received. For example in the MPI_Allgatherv it is assumed that equally sized parts of size $m_p$ are sent from each node, and the whole message $m$ is received. Further communication needed to forward or accumulate the data is not considered. Note that the call number and message sizes are rough estimates which may depend on to position in the grid, the random input data and other algorithmic details. The formulas have been derived from the results of benchmark runs on many different small systems while varying the N, NB, P, and Q parameters.

<table>
<thead>
<tr>
<th>Section</th>
<th>Operation</th>
<th>Calls</th>
<th>Message size (double)</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trailing Matrix Update</td>
<td>MPI_Allgatherv</td>
<td>$\frac{N}{NB} + \frac{N}{NB\cdot Q}$</td>
<td>$NB, \frac{N\cdot NB}{Q}$</td>
<td>$1+1/P$</td>
</tr>
<tr>
<td></td>
<td>MPI_Scatterv</td>
<td>$\frac{N}{NB} + \frac{N}{NB\cdot Q}$</td>
<td>$NB, \frac{N\cdot NB}{Q}$</td>
<td>$1/P$</td>
</tr>
<tr>
<td>Panel Factorization</td>
<td>MPI_Bcast</td>
<td>$\frac{2}{Q}$</td>
<td>$NB$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>MPI_Allreduce</td>
<td>$\frac{2}{Q}$</td>
<td>$NB$</td>
<td>$2$</td>
</tr>
<tr>
<td>Backward Substitution</td>
<td>MPI_Send/</td>
<td>$\leq \frac{N}{NB\cdot Q}$</td>
<td>$(Q-1)\cdot NB$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Panel Broadcast</td>
<td>MPI_Send/</td>
<td>$\frac{N\cdot (Q-1)}{NB\cdot Q}$</td>
<td>$\left[NB^2, \frac{N\cdot NB}{P}\right]$</td>
<td>$2$</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: General message sizes and call counts

### 2.7.1 Example Setup

As from previous experience with HPL and the HPL FAQ [45] we know that ratio of P:Q in the range of 1:k with small k (say 2) reaches the best performance on
convetional networks. With the dimension-based process mapping (see section 2.5) and the various torus configurations (see section 3.1.1) the largest grid with a “good” P:Q ratio would be a 2 rack configuration with \( P \times Q = 16 \times 32 \) for the default cabling. More racks would allow larger grids with a larger Q value, but a more asymmetric grid. Therefore we can choose 2 racks for the “best” per-node performance for a big system. These 2 racks would have a peak performance of \( 16 \cdot 32 \cdot 108.8 \text{ GFLOP/s} = 55.7 \text{ TFLOP/s} \). Assuming an efficiency of 70% which is a reasonable ratio from the experience with QS22 systems, a LINPACK performance of \( 0.7 \cdot 55.7 \text{ TFLOP/s} = 39.0 \text{ TFLOP/s} \) can be expected for this system.

The block size NB is limited to 128 by the SPE specialists. The problem size depends on the available main memory. Assuming that 240 hugepages of 16 MiB are available, each node can hold a part of \( m = 3840 \text{ MiB} \) of the Matrix. The maximum problem size can then be calculated:

\[
N = \sqrt{\frac{m \cdot P \cdot Q \cdot \text{sizeof(double)}}{8\text{Byte}}} \approx 507640
\]

Aligning this value to the next lower multiple of 128 minus 1 gives a final value of \( N = 507519 \). Furthermore we set the broadcast algorithm to “1ring”, which is the only one usable for nearest neighbor communication in a ring. We use MPI collectives wherever possible, assuming the message passing subsystem executes them optimally on the given communicator. Other parameters affect local computations, which can be adopted from the single node configuration (see Figure 2.1).

### 2.7.2 Message Passing and Network Requirements

<table>
<thead>
<tr>
<th>Section</th>
<th>Operation</th>
<th>Calls</th>
<th>Message Size (KiB)</th>
<th>Total Transfer (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trailing Matrix Update</td>
<td>MPI_Allgather</td>
<td>4089</td>
<td>[1, 15860]</td>
<td>33628.163</td>
</tr>
<tr>
<td></td>
<td>MPI_Scatter</td>
<td>4089</td>
<td>[1, 15860]</td>
<td>7916.234</td>
</tr>
<tr>
<td>Panel Factorization</td>
<td>broadcast</td>
<td>15860</td>
<td>1</td>
<td>15.488</td>
</tr>
<tr>
<td></td>
<td>allreduce</td>
<td>15860</td>
<td>1</td>
<td>30.976</td>
</tr>
<tr>
<td>Backward Substitution</td>
<td>send/recv</td>
<td>( \leq 495 )</td>
<td>31</td>
<td>15.004</td>
</tr>
<tr>
<td>Panel Broadcast</td>
<td>send/recv</td>
<td>3841</td>
<td>[128, 31720]</td>
<td>118983.404</td>
</tr>
<tr>
<td>Total Transfer</td>
<td></td>
<td>( \leq 44234 )</td>
<td></td>
<td>160589.269</td>
</tr>
</tbody>
</table>

Table 2.6: Message sizes and call counts for example setup

From the formulas for the message sizes given in section 2.7, we can extrapolate the expected message sizes for the example setup, which is given in Table 2.6.
On one hand we have large messages in the range of multiple MiB like the panel broadcast, where the run-time is mostly bound by the available bandwidth. By changing the grid dimensions the trailing matrix update messages or the broadcast messages will change its respective message sizes and weight in the overall benchmark, but it’s not possible to “configure” all messages small. On the other hand we have many small messages of size 1 KiB in the panel factorization step which have a latency bound run time. These message sizes are invariant to the grid and are only determined by the fixed block size. Therefore the network and the message passing system must handle both long and short messages efficiently.

In our example, approximately 160.6Gib are transferred within 44234 operations. Assuming that all communication operations have a linear running time like \( t_{op} = \alpha_{op} + m_{op} \cdot \beta_{op} \), where \( m \) is the message size in Bytes, \( \beta \) is the bandwidth and \( \alpha \) the latency and call overhead, we can estimate the total running time for the communication. Assuming an optimistic average bandwidth of \( \beta = 1\,\text{GB/s} \) and an average operation latency of \( \alpha = 100\,\mu\text{s} \), which is not so unrealistic as most operations are collectives which have to pass multiple nodes, the running time would be:

\[
\begin{align*}
    t_{\text{total}} &\geq \sum_{op=1}^{44234} (\alpha_{op} + m_{op} \cdot \beta_{op}) \\
    &= 44234 \cdot \alpha + m_{\text{total}} \cdot \beta \\
    &= 4.42s + 160.5s \\
    &= 164.92s
\end{align*}
\]

This approximation is a lower bound for the used communication size because more time may be consumed by protocols or synchronization effects, e.g. waiting for messages to arrive. We can see in the calculation that the share of time attributed to the bandwidth is much higher than the time for the latency. If a tradeoff would have to be made between bandwidth and latency, a bandwidth increase would improve the communication time a lot while an increased latency could be tolerated.
Chapter 3

QPACE Architecture and Torus Communication

In this section, the QPACE Architecture and the possibilities of communication on the QPACE torus are explored. The section 3.1 gives an introduction to the QPACE architecture and the physical interconnection of the torus network. Section 3.2 analyzes the low level access to the torus network. Alternatives to open the QCD-specialized torus network for a wider range of applications are proposed, and limitations with a special focus on PPE access are discussed. A model for the QPACE torus network is introduced in section 3.3, which allows to formulate communication algorithms. Example algorithms optimized for the QPACE architecture based on this model are presented in section 3.4.

3.1 Introduction to the QPACE Architecture

![QPACE node card schematic diagram](image)

Figure 3.1: QPACE node card schematic diagram
The QPACE machine employs custom node cards as main computing components. Each QPACE node card (see Figure 1) contains an IBM PowerXCell 8i processor clocked at 3.2 GHz with a maximum IEEE-compliant double precision peak performance of 108.8 GFLOP/s [44] for computing, and a Xilinx Virtex-5 FPGA for I/O and communication where the NetWork Processor (NWP) is implemented. The PowerXCell 8i processor and the Virtex-5 FPGA are coupled with 2 FlexIO™ links with an aggregate bandwidth of 6 GiB/s using RocketIO transceivers. The node cards are interconnected with a custom three-dimensional torus network with nearest neighbor connections. Unlike other Cell/B.E.-based parallel machines, it is possible to transmit data directly from the Local Store of one SPE to the Local Store of a SPE from a neighbor node. The design goal is to reach a latency of 1 $\mu$s for SPE-to-SPE transfers and a bandwidth of 1 GiB/s. The node cards in the machine are also interconnected with Gbit Ethernet and a global signal tree network.

The NWP address space is mapped into the physical address space of the PowerXCell 8i processor. Various controllers handle the memory transfers between the PowerXCell 8i processor and the NWP: The DCR bus (‘Device Control Register’) allows to control the low-speed devices, provides status registers and is used for the torus network to provide credits. 32 bit reads and writes are supported. The IWC (‘Inbound Write Controller’) is used to send the packets for the torus network or Gigabit Ethernet. It supports DMA write transfers with packet sizes of 128 Bytes from the Cell/B.E. SPE MFC, which allows very high speed. The OWC (‘Outbound Write Controller’) writes the received packets from the torus network or the Gigabit Ethernet into the main memory or the SPE Local Stores.

3.1.1 QPACE Rack Configuration

![Figure 3.2: Overview of the QPACE architecture](image)

The QPACE node cards as described above are plugged into a custom QPACE backplane which provides power and network access. Each backplane can host up to 32 nodecards and 2 rootcards. One rootcard manages 16 nodecards: It controls the boot up, provides the global tree network, generates and/or
distributes the clock signal, and monitors status signals of the node cards at run time.

One dimension of the torus network is completely routed within the backplane, providing a $8 \times 4 \times 1$ torus. Each group of 8 node cards together provide the first dimension (“red links”). Using redundant links, the network can also be configured as multiple smaller tori with sizes $\{1, 2, 4, 8\} \times \{1, 2, 4\} \times 1$. The possible configurations are illustrated in Figure 3.3.

![Figure 3.3: Torus configuration alternatives within one backplane](image)

One QPACE rack hosts up to 8 backplanes, 4 in the front and 4 in the back. The 4 backplanes at one side are physically interconnected by cables to provide the second dimension (“green links”), along with the links embedded in the backplane. Redundant links allow software reconfiguration for the green links. One rack can therefore offer partitions of size $\{1, 2, 4, 8\} \times \{1, 2, 4, 8, 12, 16\} \times \{1, 2\}$.

Multiple QPACE racks finally form the complete QPACE system. The third dimension (“blue links”) is used to connect the different racks and the front and back sides within the rack. The configuration of the third dimension can only be changed by recabling. The final installed QPACE system with $n$ racks provides torus sizes $\{1, 2, 4, 8\} \times \{1, 2, 4, 8, 12, 16\} \times \{1, 2 \cdot n\}$. The first 2 systems which will be delivered contain 4 racks, and 8 racks would be possible with the current configuration.

Other torus sizes are possible if other (longer) cables can be used: The front and the back sides of one rack could be connected at the top and the bottom using the green links to form a long ring of 32 nodes in the second dimension. The third dimension (blue links) would be decreased accordingly and would only form a ring of size $n$ instead of $2 \cdot n$ along the front planes or the back planes. The maximum size of this configuration is $8 \times 32 \times n$ which is more asymmetric in 3 dimensions, but allows very symmetric 2D torus mappings: With the mapping algorithm described in section 2.5 and $n = 4$ racks, a square 2D torus with the dimension $32 \times 32$ can be embedded. An application like HPL which benefits from square 2D tori can then gain a better performance. An illustration of the two explained cabling alternatives is presented in Figure 3.4.

25
Figure 3.4: Standard (top) and alternative cabling (bottom) of 4 racks
3.2 Torus Network Hardware

As a three-dimensional torus, each NWP contains six links to its neighbors, two for each direction (X+, X-, Y+, Y-, Z+, Z-). To allow all 8 SPEs to communicate independently, there are 8 virtual channels for each link. Packets have a size of 128 Bytes, whereas up to 8 KiB (according to the configuration at the time of writing) can be sent in one step. The maximum message size is limited by the provided window in the mapped address space and by the buffer size of the transmit FIFO in the NWP. Given a larger window, longer messages could be sent, but would possibly block the interface until the message is sent completely. This behavior could lead to deadlocks (see section 3.2.1). Messages are transmitted in order reliably to the nearest neighbor, practically offering an interface with channel semantics. The interface between the PowerXCell 8i processor and the NWP however does not transfer the 128 Byte packets in-order for performance reasons.

It is not possible to send to other nodes than the nearest neighbors, as no hardware routing for messages is implemented. Data and credits are transmitted by writing to addresses of the NWP address space which depend on the peer link and channel. By using this implicit addressing, no headers or control information have to be sent from SPE to SPE.

Data is sent to the NWP by performing a DMA transfer into a memory window of the NWP address space. This can be accomplished by using the SPEs Memory Flow Controller (MFC), executing a simple DMA PUT to the address specified by link and channel of the target peer. On the receiver side, a credit has to be granted for this data to allow the transfer to the target NWP. A credit is a pair of an address offset and length (up to 512 KiB in multiples of 128 Bytes), allowing the NWP to write the data asynchronously to the base + offset. The base usually points to the Local Store of the granting SPE. The offset can currently address the memory within a window of 1 MiB and alignment of 16 Byte, but this may be changed to a window of 8 MiB and alignment of 128 Bytes. After transmit, the NWP writes to the notify location in the Local Store of the SPE, informing that the transfer has finished. 16 pending credits per link and channel may be active at one time. Note that no interrupt mechanism is used to inform about the completion of transfers, and all transfers happen concurrently to the code flow on the SPE. The basic communication primitives are illustrated in Figure 3.5.
There are 8 KiB packet buffer space at the sender side (TX FIFO) and at least 8 KiB packet buffer space at the receiver side on the NWP for each link. These buffer are shared among the virtual channels for each link.

### 3.2.1 Low-Level Communication Rules

Because the Communication is reliable but the buffer space on the NWP is limited, communication rules have to be followed to avoid back pressure and deadlocks. In the worst case, pending DMA PUTs to the NWP may end up locking the interface to the GBIF (“Global Bus Infrastructure”) when no buffer space is available and the remote side has not (yet) issued credits.

The rules defined for the QCD applications are:

1. The order of send and receive operations must match for each pair of nodes connected by a link and channel.

2. Credits should be provided before sending the data to the NWP.

Rule 1 can be followed for arbitrary communication types by employing protocols. While rule 2 is acceptable for synchronous applications like QCD where all communication operations match, it is challenging to apply this rule to general asynchronous message passing as used in MPI. In these applications, the assumption is made that packets can always be inserted into the network, or it can be checked for congestion without blocking the interface. An MPI subsystem can postpone the transfer of a message if the network is congested.
An alternative to rule 2 is to monitor the filling level of the NWP transmit-FIFO and only send new packets if the FIFO is “almost empty”. For considerations on this filling level see 3.2.6.

When applications like HPL only employ one process per PowerXCell Si processor, only one virtual channel is necessary. Therefore no synchronization between multiple SPEs is needed to monitor the filling level. Applications which use multiple virtual channels need to synchronize the filling level before sending data, as the TX FIFO is shared among the virtual channels. This overhead can make the proposed alternative expensive to use for these applications.

3.2.2 SPE Access

The QPACE torus hardware is designed with SPE access in mind. The SPE can send by programming the MFC for a DMA PUT with channel instructions. The filling level of the FIFO can be checked by reading a DCR register, which can be done with a 32-bit DMA GET using the MFC. Receiving is done by writing a 32 bit credit into a DCR register of the NWP. The completion of the DMA PUT can be verified by polling or waiting on the channels, and the completion of the Receive operation can be checked by polling the configured notification area in the Local Store.

3.2.3 PPE Access

The PPE alone can not write messages directly to the NWP because the PPE can not write 128 Byte packets which is the only packet size the NWP accepts for messages (greater data blocks are internally fragmented into 128 Byte packets). A possible alternative is to use the SPEs Proxy Command Queue. The SPE offers its MFC service to other devices through the MMIO register interface in the problem state area [46]. This proxy interface provides a separate queue and is not shared with the SPEs local command queue. The provided MMIO register interface uses the same semantics as the SPE channel interface, with the limitation that no DMA lists transfers are possible and only 8 transfers can be queued up at one time (the local limit is 16 on the SPEs). PUT and GET commands to transfer data to and from the SPEs Local Store with Fence and Barrier modifiers are supported. The completion of the transfers can be checked with an MMIO register in the MFC command area.

The NWP can also be used from the PPE when a SPEs Local Store and the MFC may be used. The DCR registers for checking the filling level and providing credits are accessible by simple read and write instructions to the NWP address space. The receive operation can be executed by writing the credit directly to the DCR register in the NWP and polling the notification area of the Local Store until the transfers is complete.

An alternative is to use the MFC to provide the credits or generally access the DCR. This is cheaper in terms of used cycles on the PPE (one MFC instruction and one LS read), but the overall latency for MFC assisted DCR reads is higher:
the MFC access to DCR has a similar latency as from the PPE, but the MFC
instruction and LS read add more latency until the data is available on the PPE.

### 3.2.4 Proof of Concept: Memcp y with MFC

For a proof of concept of the MFC access and performance evaluation, a simple
memcp y routine has been implemented. It uses double buffering on the SPE
and interleaved PUT and GET instructions to copy the data to the Local Store
and back to its new position. Barriers in the GET instructions are used to
guarantee the correct order of the message chunks. The benchmark results
in Figure 3.6 illustrate and compare the results with the same access pattern
implemented on the SPE, the PPE glibc memcp y version and a hand-optimized
PPE replacement. The optimized PPE memcp y performs cache prefetching and
writes complete cache lines without loading the destination buffer into the cache,
which allows significant speedup. The buffer sizes used on the Local Store (and
consequently the data size per DMA transfer) were 2 KiB, 4 KiB and 16 KiB.
The transferred data was aligned at 128 Byte boundaries in the main memory,
and the benchmark was performed on a QS22 with processor 0 and its respective
memory.

![Figure 3.6: Comparison of memcp y performance](image)

From the performance measurements we can see that the PPE can sustain
the bandwidth at least for large DMA transfer sizes. The sustainable bandwidth

30
is limited by the memory bandwidth on one hand and by the latency to instruct the MFC from the PPE on the other hand. For the $m = 2$ KiB buffer size case, this effect is visible in the experiment results. An upper bound to the bandwidth can be given by considering the latencies (for measurements see section 3.2.5):

$$\text{Bandwidth} \leq \frac{m \cdot 3.2 \cdot 10^9 \cdot \text{cycles}}{t_{MFC}}$$

$$= \frac{2048 \text{Byte} \cdot 3.2 \cdot 10^9 \cdot \text{cycles}}{795 \text{cycles}}$$

$$= 7.677 \text{GiB/s}$$

### 3.2.5 Access Time Benchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Memory Area</th>
<th>PPE Time (cycles)</th>
<th>SPE Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read 32 bit</td>
<td>NWP DCR</td>
<td>$t_{DR} = 2775.960$</td>
<td>2767.764</td>
</tr>
<tr>
<td>Read 32 bit, 6 times</td>
<td>NWP DCR</td>
<td>16640.508</td>
<td>4149.420</td>
</tr>
<tr>
<td>Write 32 bit</td>
<td>NWP DCR</td>
<td>$t_{DW} = 2585.580$</td>
<td>2587.884</td>
</tr>
<tr>
<td>Write 32 bit, 6 times</td>
<td>NWP DCR</td>
<td>15584.484</td>
<td>15560.088</td>
</tr>
<tr>
<td>Instruct MFC PUT/GET</td>
<td>SPE MMIO/Channels</td>
<td>$t_{MFC} = 795$</td>
<td>7.5</td>
</tr>
<tr>
<td>Read MFC Queue Status</td>
<td>SPE MMIO/Channels</td>
<td>$t_{Q} = 232$</td>
<td>54</td>
</tr>
<tr>
<td>Read 32 bit</td>
<td>SPE Local Store</td>
<td>$t_{LR} = 227$</td>
<td>7</td>
</tr>
<tr>
<td>Write 32 bit</td>
<td>SPE Local Store</td>
<td>$t_{LW} = 127$</td>
<td>12</td>
</tr>
<tr>
<td>Write 128 byte</td>
<td>NWP IWC</td>
<td>$t_{IW}$(not measurable)</td>
<td>(not measurable)</td>
</tr>
<tr>
<td>Read 128 byte</td>
<td>NWP OWC</td>
<td>$t_{OR}$(not measurable)</td>
<td>(not measurable)</td>
</tr>
</tbody>
</table>

Table 3.1: Memory Access times

In Table 3.1, the results of the memory benchmarks are listed. Each operation is executed multiple times (100000 for DCR reads/writes, and 8 or 16 for MFC instructions, which is the limit of the queue) and measured with the high resolution clocks of the respective core, which was available with a resolution of 120 clock cycles (timebase of 26.666 MHz at a clockrate of 3.2 GHz) on the QPACE node card. Loop overhead was not removed. Each operation is started and being waited until completion for the timing measurement. This approach works for the measurements listed in the table, as the operation are acknowledged or answered by their respective receivers. The IWC latencies however
can not be measured with this method, as not the IWC but the I/O controller acknowledges the requests. The OWC latencies can not be measured as well because not the PowerXCell 8i processor, but the NWP initiates these memory requests.

On the SPE, all MFC transfers can be executed in an overlapping fashion, and the initiation or polling can always be done in <60 cycles, therefore no bandwidth impact because of high latency times is to be expected.

This is unlike the PPE, where DCR access or MFC instructions are very expensive and can not be overlapped, thus the PPE will wait until one operation is completed. The impact of this fact is analyzed in section 3.2.8.
3.2.6 TX FIFO Filling Level Considerations

To prevent a buffer overrun of the TX FIFO, it is possible to first check if buffer space is available and then send data accordingly. However if we wait until the buffer is completely empty and send data only after this event, the link will be idle from the empty signal until the next data arrives. To prevent this idle time, we can use a signal if the buffer is “almost empty”. This signal should be triggered if the buffer is nearly empty, but enough data should remain in the buffer to keep the link busy until the next data arrives. This mechanism prevents idle time of the NWP link and avoids overflow of the send buffers, and should allow to reach the maximum sustainable bandwidth.

To calculate the optimal threshold for this signal, we consider three times from the NWP view:

- $t_0$: the “almost empty” level is reached and the signal is set to 1.
- $t_1$: the DCR access request of the PPE arrives and the “almost empty” bit is returned as 1.
- $t_2$: the first data from a DMA PUT data is incoming.

The TX FIFO should keep enough (or more) data which can be sent in the maximum possible time span $t_2 - t_0$ to prevent NWP idle time. We further assume the worst case for the event $t_0$: the “almost empty” signal is raised immediately after the last PPE status request was returned as not “almost empty”. Under optimal circumstances,

$$t_1 - t_0 \leq t_{DR}$$

holds when the sending core is constantly polling the NWP. The time until the DMA data arrives at the NWP can be split into the following components:

$$t_2 - t_1 = t_{DR\text{-back}} + t_{MFC} + t_{DMA\text{-startup}} + t_{IW\text{-forward}}$$

The DCR and IWC access times have been split into the forward direction from the Cell/B.E. core to the NWP and the backward direction from NWP to the Cell/B.E. core. As we can only measure round trip timings, we assume
\[ t_{DR} := t_{DR-forward} + t_{DR-back} \]
\[ t_{DW} := t_{DW-forward} + t_{DW-back} \]
\[ t_{IW} := t_{IW-forward} + t_{IW-back} \]
\[ t_{DR-back} := \frac{t_{DR}}{2} \]
\[ t_{IW-forward} := \frac{t_{IW}}{2} \leq \frac{t_{DW}}{2} \]

As mentioned before the IWC access times can not be measured, but it’s safe to assume that the IWC access time is less or equal the DCR access time. The time \( t_{DMA-startup} \) can not be measured at all, but we can assume that the MFC request is dispatched fast, so we set
\[ t_{DMA-startup} := 0 \]

The specific times depend whether the PPE or the SPE issue the request:

\[ t_2 - t_0 = (1 + \frac{1}{2}) \cdot t_{DR} + t_{MFC} + t_{DMA-startup} + t_{IW-forward} \]
\[ = \begin{cases} 6563 & \text{cycles on the PPE} \\ 5781 & \text{cycles on the SPE} \end{cases} \]

If the assumed link bandwidth of 1 GiB/s is reached, the minimum filling level of the TX FIFO is

\[ \frac{(t_2 - t_0) \cdot 2^{30} \text{byte}}{3.2 \cdot 10^9 \text{cycles}} = \begin{cases} 2202 & \text{byte for PPE access} \\ 1940 & \text{byte for SPE access} \end{cases} \]

For very tight timing and SPE only access, 2 KiB may be enough. For a more conservative timing and general access, an almost empty limit of 3 KiB is more advisable.

### 3.2.7 PPE Overhead Times

From the latency times we can calculate the send and receive overhead times, by considering a single short message which is stored in memory. On the sender side, the steps are:

1. DMA GET from Main Memory to the Local Store (with Barrier)
2. check via DCR read if the TX FIFO link is empty
3. DMA PUT the message into the TX FIFO
We can assume that the message dispatched in step 1 is transferred to Local Store after step 2 because the DCR read takes sufficiently long.

The time from the beginning of step 1 until the data arrives at the network processor is therefore:

\[ t_{send-ovhd} = t_{MFC} + t_{DR} + t_{MFC} + t_{IW-forward} \]
\[ = 5894.61 \text{cycles} \triangleq 1.842 \mu s \]

Assuming that the credit has been granted before the data arrives, the remaining steps for the receiver are:

1. Check via a Local Store read if the data arrived
2. DMA PUT the message into the Main Memory
3. Check Queue Status of the MFC

Assuming the OWC access time is similar to the DCR access time, we set \( t_{OR} = t_{DR} \). The receive overhead can then be expressed as:

\[ t_{recv-ovhd} = t_{OR-back} + t_{LR} + t_{MFC} + t_Q \]
\[ = 2717.58 \text{cycles} \triangleq 0.85 \mu s \]

The minimum transfer overhead for Main Memory to Main Memory without the network latency (NWP to NWP) is \( t_{send-ovhd} + t_{recv-ovhd} = 2.69 \mu s \).

### 3.2.8 Latency Limitations on the PPE

Compared to the SPE, issuing a transfer on the PPE takes very long time because the MFC is not integrated in the PPE, and the transfers to the MFC or the NWP can not be overlapped. The PPE stalls until its request is complete. For the following examples we consider messages from main memory and an optimistic “almost empty” limit of 2 KiB. The messages are sent in chunks of 6 KiB, because this is the limit which can be safely sent to the NWP without risking a buffer overrun.

As we will see, only from latencies it is possible that the PPE is not fast enough to request the transfers to sustain the links, thus the latencies become a bottleneck. For the following examples we will only consider latencies to issue the commands, without regarding dependencies, stall times or bus contention. MFC assisted DCR reads are used to minimize the PPE latency. The achievable bandwidths can therefore be seen as upper bound:

#### Sending /Receiving long messages in one direction

We consider sending and receiving a long message. It is assumed that double buffering is used, therefore \( 2 \cdot 6 \text{KiB} = 12 \text{KiB} \) of buffer space in the Local Store has to be provided. On the sender side, for each 6 KiB chunk we have to:
CHAPTER 3. QPACE ARCHITECTURE AND TORUS COMMUNICATION

- 1 DMA GET of 6 KiB from main memory to Local Store into the first buffer
- 1 MFC assisted DCR read to check if the TX FIFO of the link is empty
- 1 DMA PUT of 6 KiB data from the other buffer

The sustainable send bandwidth is therefore bounded by:

$$\frac{6144\text{Byte} \cdot 3.2 \cdot 10^9 \text{cycle}}{3 \cdot t_{MFC} + t_{LR}} = 7.010 \text{GiB/s}$$

On the receiver side, we have to issue:

- 1 MFC assisted DCR write for the credit of the first buffer
- 1 Local Store read to check if the previous credit has been received successfully (in best case).
- 1 DMA PUT of 6 KiB to transfer the other Local Store buffer into Main Memory

The upper bound for the receive bandwidth is therefore:

$$\frac{6144\text{Byte} \cdot 3.2 \cdot 10^9 \text{cycle}}{(t_{LR} + 2 \cdot t_{MFC})} = 10.077 \text{GiB/s}$$

We can see that on both sides the PPE is fast enough to issue the commands to sustain the link which is limited to 1 GiB/s.

Broadcast example

Consider a double buffered broadcast, where the example node broadcasts a message from main memory to all 6 links. Again, 12 KiB of buffer space in the Local Store has to be provided. For each 6 KiB part of data, we have to do:

- 1 DMA GET of 6 KiB from Main Memory to Local Store into the first buffer
- 6 MFC assisted DCR reads to check each links TX FIFO if its empty (we assume that the checks always return positive)
- 6 DMA PUTs of 6 KiB data from the other buffer to the NWP, one for each link

Only from latencies to instruct the commands (without considering the packet transfer yet), the sustainable bandwidth on each link for the broadcasting node is limited to:

$$\frac{6144\text{Byte} \cdot 3.2 \cdot 10^9 \text{cycle}}{6 \cdot t_{LR} + (1 + 6 + 6) \cdot t_{MFC}} = 1.565 \text{GiB/s}$$
This is enough to sustain the links. However it is possible that the MFC assisted DCR read cannot be used in the scheduling because the time between the received filling level and the actual transmission of the data. If only direct DCR access can be employed, the sustainable bandwidth bound per link drops to:

\[
\frac{6144 \text{Byte} \cdot 3.2 \cdot 10^9 \text{cycle/s}}{6 \cdot t_{DR} + (1 + 6) \cdot t_{MFC}} = 0.791 \text{GiB/s}
\]

which is less than the peak bandwidth of 1GiB/s.

**All-to-all example**

The worst case is the all-to-all pattern, where the node sends and receives from/to all links simultaneously. Again assume a chunk size of 6KiB with double buffering. The buffer space needed on the Local Store is \(2 \cdot 2 \cdot 6 \cdot 6 \text{KiB} = 144 \text{KiB}\), which is more than half of one SPE Local Store size. If an application can not allow that much buffer space for the communication, the buffers could be distributed over multiple Local Stores. Another argument to employ multiple SPEs is that the needed memory bandwidth of \(12 \text{GiB/s} (6 \text{GiB/s for each direction})\) can not be achieved with one SPE as seen in the MFC micro benchmark in section 3.2.4. Multiple SPEs allow to reach a higher aggregate bandwidth, while the number of MFC instructions from the PPE is the same.

In this example, each node has to do:

- 6 DMA GETs of 6 KiB from Main Memory to Local Store into the first buffer for the respective link
- 6 MFC assisted DCR reads to check each links TX FIFO if its empty (we assume that the checks always return positive)
- 6 DMA PUTs of 6 KiB data from the other buffer to the NWP, one for each link
- 6 MFC assisted DCR writes to grant credits for the next incoming data, one for each link.
- 6 Local Store reads to check that the received data is completely transferred
- 6 DMA PUTs of 6 KiB data from the Local Store receive buffer to the Main Memory

The sustainable bandwidth which is used for each link in each direction is limited by the instruction latencies to:

\[
\frac{6144 \text{Byte} \cdot 3.2 \cdot 10^9 \text{cycle/s}}{(2 \cdot 6) \cdot t_{LR} + (5 \cdot 6) \cdot t_{MFC}} = 0.862 \text{GiB/s}
\]
If for scheduling reasons the DCR access must be done directly from the PPE, the maximum sustainable bandwidth bound drops to:

\[
\frac{6144 \text{ Byte} \cdot 3.2 \cdot 10^9 \text{ cycle}}{6 \cdot t_{DR} + 6 \cdot t_{DW} + 6 \cdot t_{LR} + (3 \cdot 6) \cdot t_{MFC}} = 0.368 \text{ GiB/s}
\]

This example shows again that the PPE latencies may become a bottleneck in the communication.

### 3.2.9 Suggestions to Circumvent the PPE Latency Limitations

As we have seen in the previous examples, the PPE latency limits can limit the sustainable bandwidth. Scheduling, congestion and other synchronization effects will further decrease the bandwidth which can be seen in practice. Therefore suggestions to circumvent the PPE limitations are presented.

**TX FIFO filling levels in one DCR register**

Instead of using expensive DCR operations to read the TX FIFO filling levels for each link separately, the filling level information could be combined in one DCR register. Special care has to be taken because the information might already be outdated when many links are served in an interleaved fashion, e.g. the FIFO is already empty while the information indicates that its still full, or the link runs out of data because other links are served with DMAs first. This technique would decrease the needed DCR access on the sender side from up to 6 DCR reads to 1 DCR read.

Using this method, the upper bound of the send bandwidth for the broadcast example could be increased by 78% for the MFC assisted DCR access, and by 172% for the direct DCR access.

**Use a faster controller to read/write credits and status information**

Instead of using the slow DCR bus to control status information or provide credits, an additional controller could be integrated in the NWP to provide the time critical functions. This controller should be directly coupled with the FlexIO interface or integrated into another high speed controller like the IWC. Special care has to be taken that the credits are provided in order, e.g. by using fenced MFC operations on the SPE.

**NWP direct write to main memory**

The NWP credits are granted for buffer spaces defined by the offset within the credit and a base address set when initializing the torus interface. This base address usually points to the physical address of the respective Local Store address to allow direct SPE communication. For large messages, an alternative
would be to let the base address point to the receive buffer within the main memory. The NWP would then transfer the received data directly to the main memory. Larger credits could be issued, as they are not limited by the size of the Local Store buffers. Larger credits also decrease the number of needed DCR writes per block.

The direct write to main memory relieves the SPE in two ways: First, the MFC is not involved in the receive direction, which is a benefit if an application uses the SPEs while communicating. The second advantage is that no buffer space has to be reserved in the Local Store for receiving data.

The NWP can not handle virtual addresses. One strategy would be to allocate a contiguous buffer in the physical address space using a special device driver. The address has to be translated from the virtual to the physical address, and the physical address would then be passed to the NWP. Another possible strategy is to program the IO-MMU of the PowerXCell 8i processor to handle the address mapping.

When the base address is set, the offset in the credit can only address a memory window of 1 MiB. If a larger message is to be transmitted, the base address has to be moved multiple times and new credits have to be provided for the moved window. While the base address is reset, no credits may be in flight to avoid inconsistencies. This implies that no data can be received while moving the base address, which will lead to minor regressions in the bandwidth.

Note that this mechanism can already be used without modification to the current FPGA design, given that the software layer provides support for the address handling.

Use a DMA engine to read directly from main memory

The send direction is implemented by actively sending buffers to the NWP. If the NWP had a DMA engine, the PPE would only instruct the DMA engine to perform transfers to the NWP. The NWP could decide on its own when TX FIFO space is available, thus the filling level checks could be skipped. Another advantage is that the SPE is relieved in terms of MFC access and Local Store usage, similar to the NWP direct write case.

The same memory restrictions as from the NWP direct write case must be applied to the DMA engine: a simple DMA engine would not be able to read from virtual addresses, and special buffers or the IO-MMU must be used.

For a system with a DMA engine combined with the direct memory write on the receiver side, we can review the all-to-all example. We assume that message sizes of 512 KiB can be sent by the DMA engine and received with one credit. We assume 2 DCR writes for the sending side (one word for the address, one word for the length). 1 DCR read is used to check whether the send has completed. 1 DCR write and 1 Local Store read is used for the credit. Furthermore 1 DCR write is used to move the credit base address after each received packet:
512KI$n \cdot 3.2 \cdot 10^9 \frac{cycle}{s}

\frac{6 \cdot (4 \cdot t_{DW} + t_{DR} + t_{LR})}{18.367GiB/s}

As we can see, the bound to the bandwidth inflicted by the PPE latency is no longer a bottleneck for the link bandwidth. Another advantage is that the SPE and its MFC are completely decoupled from the message transfer. No buffer space must be allocated in the SPE Local Store. Performance regression of parallel running SPE code can still occur because the interface to main memory on the PowerXCell 8i processor must still be shared between the SPEs and the NWP.

**Change blocking communication model to a feedback model**

Instead of using a blocking write which requires the filling level check to avoid deadlocks, a nonblocking feedback model has been proposed [47] which could be used instead. A message can then be sent without blocking regardless of the buffer filling, and the NWP must then provide feedback e.g. by writing a notification in the Local Store. A message would only be accepted if enough TX FIFO space is available. The impact on the PPE latencies is relaxed as the DCR reads for the filling level can be skipped and LS reads to check for success have to be added instead. However the scheduling is simplified as timing dependencies between the DCR reads and the MFC sends are removed. From the optimistic sending we can also expect an improvement in the memory to memory latency, as no filling level check has to be done before sending a message.

**Larger buffers for the TX FIFO**

If the buffer space on the FPGA is available, an obvious improvement would be to enlarge the buffers of the TX FIFO. As the upper bounds in the presented examples all linearly depend on this buffer size, increasing the buffer size would increase the sustainable bandwidth by the same factor. The largest buffer size is 16 KiB, this is the maximum transfer size possible for the MFC. Larger buffers (e.g. 16 KiB) combined with the feedback model would allow an upper bound in the all-to-all example of:

\[
\frac{16384Byte \cdot 3.2 \cdot 10^9 \frac{cycle}{s}}{6 \cdot t_{DW} + 2 \cdot 6 \cdot t_{LR} + (3 \cdot 6) \cdot t_{MFC}} = 1.454GiB/s
\]

**3.3 Torus Network Model**

For theoretical examination of communication algorithms we make the following assumptions:

1. The processing nodes are organized in a 3D torus of dimension $n_x \times n_y \times n_z$, with $n = n_x \cdot n_y \cdot n_z$ nodes in total.
2. Each node can only communicate with its nearest neighbors.

3. It is possible to send and receive from all links simultaneously, 6 send and 6 receive operations may be active at the same time with full bandwidth.

4. The time to transmit a message located in SPE Local Stores from one node to its neighbor is $\alpha + \beta \cdot m$, where $m$ is the message size in Bytes, $\alpha$ is the latency (design goal: 1 $\mu$second) and $\beta$ is the sustainable bandwidth (design goal: 1 GiB/s).

Note for assumption 3 that the messages are practically not sent simultaneously, but sent individually by the MFC. If one certain message is sent to multiple links, one DMA PUT per link has to be issued separately. The bandwidth of the link between the PowerXCell 8i processor and the FPGA however is 6 times higher than the actual torus link bandwidth.

From these assumptions, we can find the following conclusions:

1. Due to assumption 3, it is also possible to “multicast” one message to several peers with full bandwidth.

2. When routing a message from arbitrary nodes of the torus, the minimum sustainable latency is $n_{hops} \cdot \alpha$, where $n_{hops}$ is the number of links used on the path. This follows from the fact that the latency for one hop is $\alpha$, which is the time to complete the transfer from one node to its neighbor.

3. When routing a message from arbitrary nodes of the torus, the maximum sustainable bandwidth is $\beta$. This follows from assumption 3 and 4 and can be achieved using double buffering.

### 3.3.1 Memory Locations

For the whole model, it is always assumed that data is transferred from and to the Local Stores of the respective SPEs. Transferring messages from and to the main memory is also possible, but the latency will be slightly higher: The packets have first to be transferred to the Local Store, sent via the network as usual, and moved back from Local Store to the main memory on the receiver side. Assuming that the latency between LS and main memory is $\gamma$, the expected time is $\alpha \cdot n_{hops} + \beta \cdot m + 2 \cdot \gamma$ for a path of length $n_{hops}$. The bandwidth will still be limited by the network bandwidth, as the SPE memory bandwidth is much larger than the expected torus link bandwidth of 1GiB/s. We can also assume that the memory latency $\gamma$ will be much lower than the network latency $\alpha$. For simplicity, this additional memory latency is ignored, but can trivially be added to the presented run time formulas.

### 3.3.2 Communicators

In MPI, communicators can be used to create subspaces of other communicators or the original set of processes. The communicators can be used to organize
the processes according to the pattern used in the application, and use collective algorithms only in these communicators. For example, in HPL separate communicators are used for row and column communication.

When using the QPACE torus network, an obvious limit is that the processes of one communicator must be connected, because only nearest neighbor communication is supported. For arbitrary connected communicators, we can implement most of the algorithms by using spanning trees. There also have been publications [48, 49, 50] for many different architectures [51, 52, 23] about efficient collective algorithms in torus and mesh networks, which mostly operate on different assumptions: network devices are only single ported, or the torus is not limited to nearest neighbor communication. Unlike arbitrary communicators, it is often easier to find the spanning trees in torus and meshes, usually they don’t even have to be constructed explicitly (peers can be determined by the cartesian coordinates of the local node), and multiple edge-disjoint spanning trees may also be employed to increase throughput. 4 types of communicators for a 3D torus network will be considered here, from the general to the specific:

1. **General Communicators**: an arbitrary set of the nodes.

2. **Connected Communicators**: A general communicator, where each pair of nodes of the communicator is connected by a path of nearest neighbors which are part of the communicator.

3. **Mesh**: A connected communicator where the communicator are numbered with ids \((x, y, z)\) which hold \(0 \leq x \leq n_x - 1, 0 \leq y \leq n_y - 1; 0 \leq z \leq n_z - 1\). \(n_x, n_y, n_z\) are the dimensions of this communicator. The nodes are connected to their neighbors
   \((x + 1, y, z),\)
   \((x - 1, y, z),\)
   \((x, y + 1, z),\)
   \((x, y - 1, z),\)
   \((x, y, z + 1),\)
   \((x, y, z - 1),\)
   if these neighbors are within the defined boundaries.

4. **Torus**: A mesh with the difference that each node \((x,y,z)\) is connected to the neighbors
   \(f(x + 1, y, z),\)
   \(f(x - 1, y, z),\)
   \(f(x, y + 1, z),\)
   \(f(x, y - 1, z),\)
   \(f(x, y, z + 1),\)
   \(f(x, y, z - 1),\)
where \( f(x, y, z) = (x \mod n_x, y \mod n_y, z \mod n_z) \). Unlike the mesh there are no “boundary points” or corners, each node has exactly 6 neighbors.

### 3.3.3 Topology Graphs

We will consider the topology graph as the directed graph \( G = (V, E) \) to formulate propositions, where the vertices \( V \) are the set of nodes in the network, and the edges \( E \) are the (bidirectional) links of the torus network, where

\[
E = \left\{ (x_1, y_1, z_1), (x_2, y_2, z_2) \right\} \in \binom{V}{2} \mid (x_1, y_1, z_1) \neq (x_2, y_2, z_2) \land (x_1, y_1, z_1) \text{ is neighbor of } (x_2, y_2, z_2) \text{ in the torus}
\]

Communicators can then be considered as subgraphs \( G' = (V', E') \) with \( V' \subseteq V \) and \( E' = E \cap \binom{V}{2} \).

### 3.4 Communication Algorithms

We will consider an example set of collective communication algorithms in different type of communicators (connected, mesh, torus), and will give performance estimations. The estimation assume that all nodes enter the communication at the same time, and the network has no contention. All algorithms are defined analogous to the MPI standard [30, 31].

We define the **latency** as the time starting with the entry of the communication function and ending when the last node has finished the communication, while sending the smallest possible message (\( n = 0 \)). This time is the minimum time required for all nodes to complete. One node might finish its participation of the communication faster, and pipelining effects [49] when calling many broadcasts in a row may give faster results than to be assumed from the latency alone.

The **bandwidth** is defined as the ratio of data processed in the time from start to completion of the algorithm, for sufficiently large data sets (theoretically unlimited). The “processed data” depends on the algorithm.

### 3.4.1 Broadcast

In a broadcast, a message is sent from one node called “root” to all other nodes in the network. The data processed is the message sent by the root.

**Spanning trees**

For connected communicators, a broadcast can be implemented by constructing a rooted spanning tree on the topology graph \( G \) using the root node as the tree root. The message can then be sent in a pipelined fashion: The root nodes sends a chunk of the message to all of its neighbors within the tree simultaneously.
CHAPTER 3. QPACE ARCHITECTURE AND TORUS COMMUNICATION

Each other node forwards this chunk of message to all other neighbors within the tree (except the original sending neighbor) when receiving a chunk [49].

Bandwidth

The sustainable bandwidth for a single spanning tree is limited by the bandwidth of one link $\beta$. The messages at each node are received by exactly one link with bandwidth $\beta$, and can be forwarded to the outgoing links with full speed because of conclusion 1.

Latency bounds

The latency of any spanning tree based collective algorithm has a lower bound by the maximum path length $l$ from the root node to any other node of the communicator. For a torus, the maximum path length is obviously $l_{\text{torus}} = \left\lfloor \frac{n_x}{2} \right\rfloor + \left\lfloor \frac{n_y}{2} \right\rfloor + \left\lfloor \frac{n_z}{2} \right\rfloor$ for any root node.

In meshes, the maximum path length depends on the position of the root node. If the root node is one of the corners, the farthest node is the node at the opposite corner of the mesh. The maximum path length for this case is therefore $(n_x - 1) + (n_y - 1) + (n_z - 1)$. On the other hand, when the root node is in the center of the mesh, the maximum path length is $\left\lfloor \frac{n_x}{2} \right\rfloor + \left\lfloor \frac{n_y}{2} \right\rfloor + \left\lfloor \frac{n_z}{2} \right\rfloor$, just as the torus. Therefore $\left\lfloor \frac{n_x}{2} \right\rfloor + \left\lfloor \frac{n_y}{2} \right\rfloor + \left\lfloor \frac{n_z}{2} \right\rfloor \leq l_{\text{mesh}} \leq (n_x - 1) + (n_y - 1) + (n_z - 1)$ holds.

For arbitrary connected communicators, the path length $l_{\text{connected}}$ may be very long depending on the topology, but we can say for sure that $\left\lfloor \frac{n_x}{2} \right\rfloor + \left\lfloor \frac{n_y}{2} \right\rfloor + \left\lfloor \frac{n_z}{2} \right\rfloor \leq l_{\text{connected}} \leq n$ holds.

The lower bound for the latency is $l \cdot \alpha$, because this is the minimum time the information needs to travel from the root node to the farthest node (or back). Depending on the spanning tree, the maximum path length can be much longer than the introduced lower bound, e.g. a Hamiltonian path through a torus is also a tree with the maximum path length of $n$.

Constructing spanning trees in meshes and tori

A simple, yet optimal spanning tree in terms of latency is the dimension ordered spanning tree, which can be constructed in meshes and tori. The construction is similar to the dimension order routing [53]. The root node sends in all possible directions. Each other node forwards in all possible directions which are the same or behind the receiving direction in the dimension order X,Y,Z. In a torus, a link is not used if the neighbor has a lower distance to the root node$^1$, to prevent loops. The constructed tree hits the minimum bounds for the path lengths, and the latency is therefore optimal with $l_{\text{min}} \cdot \alpha$. As expected for a single spanning tree, the bandwidth is $\beta$. The running time of this algorithm is $t = l_{\text{min}} \cdot \alpha + m \cdot \beta$ with the message size $m$ in Bytes.

$^1$If the distance is the same, the message is only sent if the direction is positive, to support even dimensions
Figure 3.8: Dimension ordered spanning tree in a $3 \times 3 \times 3$ mesh
Figure 3.9: Illustration of the BlueGene/L adapted broadcast on a $4 \times 4 \times 4$ mesh

**Edge disjoint spanning trees**

In dense graphs, multiple edge-disjoint spanning trees may be constructed. In a 3D torus, the number of edge disjoint spanning trees $k$ can not be larger than 6:

Each spanning tree has $n - 1$ edges. Because of assumption 3, each node can have 6 ingoing and 6 outgoing edges. This limits the number of edges to $\frac{12n}{3} = 6 \cdot n$, because each edge is counted once as outgoing and once as ingoing at each node. We can then see that $k \cdot (n - 1) \leq 6 \cdot n$ is feasible for $k \leq 6$ (assuming $n \geq 8$).

With multiple spanning trees, we can interleave the chunks of messages on these $k$ spanning trees. Each spanning tree transfers the chunks independently as described above for single trees. With this mechanism we can achieve a bandwidth of $k \cdot \beta$ at maximum. The latency is still bound by the maximum path length of the spanning trees used. Constructing $k$ spanning trees in a general graph with $k \geq 2$ is known to be a NP-hard problem [54], but efficient algorithms for regular topologies like meshes and tori can be formulated.

**Constructing 3 edge disjoint spanning trees in meshes and tori**

Algorithms to construct these spanning trees for Meshes have been implemented e.g. for the IBM BlueGene/L<sup>®</sup>/L™[50]. The broadcast algorithm for the BlueGene/L machine is not optimal in terms that it doesn’t construct the maximum number of spanning trees possible in Tori, but it offers higher bandwidth for mesh communicators than a single tree.

We assume a mesh communicator. The 3 spanning trees are practically rotated copies of the first spanning tree. Figure 3.9 presents an illustration of the constructed trees.

The spanning trees are constructed in 3 steps:

1. The message is distributed from the root node along the X-axis to the outside boundary of the mesh.
2. The message is then broadcast using a dimension ordered routing along
the Y and Z axis.

3. All nodes on the boundary planes which don’t have the same Y and Z
coordinate as the root node forward back to the center, and the inner
nodes forward it in the same direction they received it from until the X
coordinate of the root is reached.

The 2 other spanning trees are constructed using the same algorithm but shifted
dimensions: Spanning Tree 2 replaces the dimensions X,Y,Z in the algorithm
with Y,Z,X, and Spanning Tree 3 replaces the dimensions X,Y,Z with Z,X,Y.

We can verify that all links are mutually exclusive: The inner links of the
mesh are only used by the first dimension of the respective spanning tree. The
links within the boundary planes are used only for the outside direction in step
2 in the respective spanning tree and for the inside direction in step 3 in the
remaining spanning trees.

The maximum path length \( l \) from the root to a leaf in the constructed
spanning tree is bound by

\[
 l \leq \max(2 \cdot n_x + n_y + n_z, \quad n_x + 2 \cdot n_y + n_z, \quad 2 \cdot n_x + n_y + 2 \cdot n_z)
\]

and reaches its maximum if the root node is a corner node.

The running time of this broadcast algorithm is \( t = l \cdot \alpha + 3 \cdot m \cdot \beta \) with the
message size \( m \) in Bytes, as 3 disjoint spanning trees were constructed.

**Constructing 6 edge disjoint spanning trees in tori**

Based on the EDF (edge-disjoint fences) algorithm from Barnett et al [48] for
2D Tori, we can construct 6 edge disjoint spanning trees for 3D tori. Without
loss of generality we can assume that the root is \((0,0,0)\), as any other root \((x,y,z)\)
can be reduced to this case by translating all nodes by \((-x,-y,-z)\).
Figure 3.10: Illustration of the 3D-EDF algorithm for a $3 \times 3 \times 3$ torus

On each node, the links in positive directions $X^+, Y^+, Z^+$ are sending for spanning tree 1-3 and receiving for spanning tree 4-6, and the links in negative direction $X^-, Y^-, Z^-$ are receiving on spanning tree 1-3 and sending for spanning tree 4-6. Each link is assigned to two of the 6 spanning trees, according to the assignment in Tables 3.2 and 3.3, once for receiving and once for sending for a spanning tree.

Proof of correctness

Connectivity can be checked by verifying in the tables that for any node (except the root), for each spanning tree, the sending links only exist when there is exactly one receiving link. Furthermore, to ensure connectivity, we need to check that the constructed spanning trees are indeed loop free: All nodes are only forwarding in positive direction for spanning tree 1-3, and negative direction for spanning tree 4-6. We can verify that for each spanning tree that in each dimension a plane exists which does not forward packets of this spanning tree except along this plane, acting like a barrier for the routing. Thus the routing in this spanning tree is equivalent to positive (negative) routing in a mesh, which is known to be loop free:

- Spanning tree 1 is not further forwarded in the 3 planes $x = 0; y = y_n - 1; z = z_n - 1$
- Spanning tree 2 is not further forwarded in the 3 planes $x = x_n - 1; y = 0; z = z_n - 1$
- Spanning tree 3 is not further forwarded in the 3 planes $x = x_n - 1; y = y_n - 1; z = 0$
Table 3.2: Edge assignment to the Spanning Trees in a 3D Torus for spanning trees 1-3
### Table 3.3: Edge assignment to the Spanning Trees in a 3D Torus for spanning trees 4-6

<table>
<thead>
<tr>
<th>X Range</th>
<th>Y Range</th>
<th>Z Range</th>
<th>X+</th>
<th>Y+</th>
<th>Z+</th>
<th>X-</th>
<th>Y-</th>
<th>Z-</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0</td>
<td>y = 0</td>
<td>z = 0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>root</td>
</tr>
<tr>
<td>x = 0</td>
<td>y = 0</td>
<td>z = 1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>corner points</td>
</tr>
<tr>
<td>x = 0</td>
<td>y = 1</td>
<td>z = 0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>corner points</td>
</tr>
<tr>
<td>x = 0</td>
<td>y = 1</td>
<td>z = 1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>corner points</td>
</tr>
<tr>
<td>x = 1</td>
<td>y = 0</td>
<td>z = 0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 1</td>
<td>y = 0</td>
<td>z = 1</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 1</td>
<td>y = 1</td>
<td>z = 0</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 1</td>
<td>y = 1</td>
<td>z = 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>edge points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>y = 0</td>
<td>z = 0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>edge points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>y = 0</td>
<td>z = 1</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>edge points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>y = 1</td>
<td>z = 0</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>edge points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>y = 1</td>
<td>z = 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 0</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>z = 0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 0</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>z = 1</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 1</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>z = 0</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 1</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>z = 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>edge points</td>
</tr>
<tr>
<td>x = 0</td>
<td>y = 0</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>face points</td>
</tr>
<tr>
<td>x = 0</td>
<td>y = 1</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>face points</td>
</tr>
<tr>
<td>x = 1</td>
<td>y = 0</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>face points</td>
</tr>
<tr>
<td>x = 1</td>
<td>y = 1</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>face points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>z = 0</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>face points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>z = 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>face points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>y = 0</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>face points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>y = 1</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>face points</td>
</tr>
<tr>
<td>x = 0</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>inner points</td>
</tr>
<tr>
<td>x = 1</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>inner points</td>
</tr>
<tr>
<td>1 &lt; x ≤ x_n - 1</td>
<td>1 &lt; y ≤ y_n - 1</td>
<td>1 &lt; z ≤ z_n - 1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>inner points</td>
</tr>
</tbody>
</table>
• Spanning tree 4 is not further forwarded in the 3 planes $x = 0; y = 1; z = 1$
• Spanning tree 5 is not further forwarded in the 3 planes $x = 1; y = 0; z = 1$
• Spanning tree 6 is not further forwarded in the 3 planes $x = 1; y = 1; z = 0$

As each spanning tree is connected and loop free, the constructed spanning trees are proper trees.

Properties

The maximum path length used in this algorithm is $(x_n - 1) + (y_n - 1) + (z_n + 1) + 1$ in any of the constructed spanning trees. Compared to the dimension ordered spanning tree, this is more than twice of the path length used there. However, the sustainable bandwidth is $6 \cdot \beta$, which is optimal for our model, as all links are used (except the receiving links at the root node) and no part of the message is sent redundantly.

The running time of this broadcast algorithm is $t = l \cdot \alpha + 6 \cdot m \cdot \beta$ with the message size $m$ in Bytes, as 6 disjoint spanning trees were constructed.

Using Gigabit Ethernet for broadcasts

The Ethernet protocol supports unreliable native broadcasts which can be used for example with the UDP protocol. Based on the unreliable multicast or broadcast, it is possible to build a practically constant time broadcast [55] for MPI_COMM_WORLD communicator. Depending on the final latency performance in the torus network, this method might be superior in the latency time, as all nodes can be reached through the switched Ethernet interconnect without intermediate nodes. The bandwidth however is limited to 1 GBit/s, which is only a fraction of the bandwidth available in torus network.

3.4.2 Reduce

A reduce operation reduces a value (or a vector of values) from all processes to one value, which is returned at a distinct node called the root node. Usually rooted trees are employed where each node combines the incoming messages with its own and forwards the result to its father in the tree. The communication pattern is the same as in the broadcast operation, but in reverse direction, and we can use the same (multiple) spanning trees as for the broadcast operation.

A potential bottleneck is the processing power and memory bandwidth for the combination of the input data. Unlike the broadcast, where the message parts can be buffered in the Local Store and sent out in the next step, the message parts from all incoming links must be collected and reduced before the result can be forwarded. This requires more synchronization with the neighbors than in the broadcast. The reduction must be actively performed by the PPE or the SPE. The PPE might be a bottleneck in this reduction step: For example when a node has 5 incoming links and 1 outgoing link like in the spanning tree,
it requires a memory bandwidth of 6 GiB/s if all links are used, which is more than the PPE can handle. It is therefore advisable to outsource the reduction operation to the SPE by using function offloading (see section 4.2.2) if a PPE centric approach is used. The SPE does not have this limitation if the message parts are kept in the Local Store.

Assuming that there is no limitation by the processing speed to perform the reduction, the run time of the Reduce operation is similar to the broadcast operation for the same spanning trees. The same bandwidth can be sustained using double buffering, but the latency will be higher because of the reduction. Assuming a time $\tau$ which is needed to perform a reduction of the message parts on one node, the running time for the reduce operation is

$$t_{\text{reduce}} = t_{\text{broadcast}} + \tau \cdot l$$

where $l$ is the maximum path length of the spanning tree used in the original broadcast tree.
Chapter 4

Message Passing Libraries on QPACE

In this chapter the message passing libraries and consequences for the programming models are discussed. The Cell/B.E. architecture as a heterogeneous multi core microprocessor offers different alternatives to be programmed. Section 4.1 presents possible SPE centric message passing designs and embeddings in each other. In section 4.2 the PPE centric approaches are discussed. The proposed programming models are then discussed for HPI in section 4.3. Finally, an interface description for the integration into the MPI framework OpenMPI is presented in section 4.4.

4.1 SPE Centric Approaches

The SPE centric implementation options of the communication libraries MPI and QMP on QPACE are discussed. Various alternatives are possible: Implement QMP or MPI on the low level interface which is provided by the torus, or map one of the libraries on top of the other. Parts of the libraries can also be outsourced to PPE assisted callbacks. In a SPE centric programming model, also the constraints of the SPEs like the Local Store size must be considered.

From the introduction to QMP [56]:

"Depending upon demand, a subset of MPI could be implemented above this new API so that legacy codes which use MPI could function on the new architectures which implement only the new API (albeit at somewhat reduced efficiency). Further, the new API has been implemented atop MPI so that new applications using this new API can still be run on older machines for which only MPI is available, with negligible overhead."

From the functionality point of few, QMP is only a subset of MPI specialized for QCD, designed for efficient (zero copy) nearest neighbor communication.
CHAPTER 4. MESSAGE PASSING LIBRARIES ON QPACE

Therefore implementing QMP directly or on top of MPI or directly on the network hardware is the natural choice, and implementations exist at least for MPI (MPICH [57]), QCDOC [23] and M-VIA [58].

4.1.1 MPI on QMP

There seems to be no publicly available implementation for an MPI mapping on QMP to the time of writing. The imposed limits or efforts needed to overcome them for a (complete) MPI implementation on top of QMP are discussed.

A high level approach

<table>
<thead>
<tr>
<th>OpenMPI Layer</th>
<th>Offered by the Layer</th>
<th>Offered by QMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI - User Interface</td>
<td>user interface</td>
<td>none</td>
</tr>
<tr>
<td>DDT - Derived Datatype Engine</td>
<td>derived, strided, vector and basic datatypes, and conversion</td>
<td>strided, vector, basic datatypes</td>
</tr>
<tr>
<td>TOPO - Process Topology</td>
<td>global communicator, graph mapping, cartesian mapping, arbitrary sub communicators</td>
<td>global communicator, nearest neighbor torus communication (subset of the cartesian communicator), no sub communicators</td>
</tr>
<tr>
<td>COLL - Collective Communications</td>
<td>collective operations for intra- and intercommunicators</td>
<td>some collectives with limitations, only in global communicator</td>
</tr>
<tr>
<td>PML - Point to Point Messaging Layer</td>
<td>point to point communication, message scheduling</td>
<td>most point to point communication, no message scheduling</td>
</tr>
<tr>
<td>BML - BTL Management Layer</td>
<td>resource discovery, management</td>
<td>supported (assuming QMP exclusive use)</td>
</tr>
<tr>
<td>BTL - Byte Transport Layer</td>
<td>move raw bytes inorder</td>
<td>supported</td>
</tr>
</tbody>
</table>

Table 4.2: OpenMPI Layers (partial list) (from [1, 2])

A blunt approach might be to embed QMP in an existing MPI framework like OpenMPI. Features which QMP provides can be used, unsupported features could be used from the frameworks codebase. A compliant MPI could be produced like this, but some limitations make it difficult to port a huge library to the SPE:

- the small Local Store size of 256 KiB which has to keep the library code, application code and run-time data
• missing direct hardware access (only partly true for QPACE)
• the unconventional programming model which does not give direct access to the main memory
• slow path to the operating system (via PPE assisted callbacks)

There are various approaches to challenge these problems, like software managed caches or code overlays. These techniques still require rather large porting efforts and also add not negligible overhead. Previous implementations [59, 60] therefore choose to implement only a subset of the library to keep the footprint small.

However, from the organization of the OpenMPI Layers as illustrated in Table 4.2 we can estimate what QMP can offer for MPI, and can compare the layers and their features.

A low level approach

Instead of using a complete MPI framework like OpenMPI, we can try to build the MPI primitives directly on the QMP primitives. This is an approach for a scenario where a reduced subset of MPI is acceptable, as there are some limitations in function and performance as shown below. The needed features must be implemented from scratch. When a more complete MPI is targeted, QMP would only be used as pure Byte Transport Layer. Complicated Functions could be outsourced to the PPE and an existing MPI implementation. The design would be similar to the direct MPI on QPACE torus approach as described in 4.1.4, but with the (redundant) QMP layer between.
Send/Recv implementation illustration

The most common primitives, MPI_Send() and MPI_Recv(), can be easily implemented directly with the QMP primitives:

```c
int MPI_Send(void *buf, const int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
{
    int nbytes;
    void *packed_buf;
    int phys_dest;
    QMP_msghandle_t msghandle;

    nbytes = pack(buf, count, datatype, &packed_buf);
    phys_dest = get_phys_pos(dest, comm);
    msgmem = QMP_declare_msgmem(packed_buf, nbytes);
    msg_handle = QMP_declare_send_to(phys_dest, msgmem, nbytes);

    QMP_start(msghandle);
    QMP_wait(msghandle);
    QMP_free_msgmem(msgmem);

    free(packed_buf);
}

int MPI_Recv(void *buf, const int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Status *status)
{
    int nbytes;
    void *packed_buf;
    int phys_src;
    QMP_msghandle_t msghandle;

    nbytes = get_size(buf, count, datatype);
    packed_buf = malloc(nbytes);
    phys_src = get_phys_pos(src, comm);
    msgmem = QMP_declare_msgmem(packed_buf, nbytes);
    msg_handle = QMP_declare_receive_from(phys_src, msgmem, nbytes);

    QMP_start(msghandle);
    QMP_wait(msghandle);
    QMP_free_msgmem(msgmem);

    unpack(buf, count, datatype, packed_buf, nbytes);
    free(packed_buf);
}
```

Redundant declaration and deallocation of buffers

QMP is designed for repetitive send and receive of similar data. In the implementation above the memory is declared and freed for each send/recv. This is probably slow, from [56]:

---

56
"QMP messaging is meant to be highly repetitive and high performance, and uses a gated message channel paradigm. In this case messaging is done by first declaring the source and destination buffers and node ID (expensive part), then executing the pre-computed I/O operation on demand as rapidly as possible. Destinations are always known & pre-allocated buffers are used (no queuing and so no extra copy for all but very short messages)."

**Datatype engine**

Datatypes must be converted from MPI datatypes to QMP datatypes. As aggregated datatypes are not supported, they must be converted manually. This is illustrated with the pack()/unpack() functions above.

However it is possible to use QMPs support for strided datatypes or vectors, which is not illustrated above for simplicity.

**Groups, communicators**

QMP only uses the global communicator and does not support sub-communicators, so all communicator housekeeping must be handled alone. This is illustrated with the get_phys_pos() functions. Communicator ranks have to be computed relative to the physical ranks defined by QMP.

The communicator among identifiers must be stored in some control information for the message if nonblocking communication is to be supported (see below).

**Collectives/global operations**

There are a few MPI collectives that can be represented with QMP collectives: MPI_barrier() maps to QMP_barrier(), MPI_Allreduce() maps to QMP_sum*(), MPI_broadcast() maps to QMP_broadcast() if the root is 0. The main limitation is that only the main communicator (MPI_COMM_WORLD) is supported here, sub-communicators cannot use these functions. Some other collectives can probably be mapped on the provided QMP collectives (with performance overhead), e.g. MPI_Reduce() could be mapped on MPI_Allreduce(). The datatypes might also limit the usefulness of the QMP collectives.

For all uncovered cases mentioned above it’s possible to implement the collectives using the Send/Recv primitives as fallback routines.

**Nonblocking communication, tags, protocols**

It is possible to split the MPI_Send (MPI_Recv) call from above into non-blocking MPI_Isend (MPI_Irecv) + MPI_wait() calls. However, unlike MPI it is not expected for QMP that messages can overtake each other, so deadlocks might occur. From the QMP introduction [56] (in Communication Operations, Restrictions):
"Starting a second send (separate handle) to the same adjacent node before the first completes, or a second receive before the first completes, is allowed. The second send/receive start function is allowed to block. That is, the implementation is not required to implement an I/O queue to support this behavior."

The same applies for different tags. Tags are completely unsupported by QMP, so they must be embedded in some control information for the sent data. Furthermore deadlocks might be possible if different nonblocking messages from the same node can not overtake each other.

To support the concurrent messages, message matching (identified by count, datatype, tag, communicator), message scheduling and flow control must be implemented. This can be done by a layer of protocols and message queues between the MPI calls and the corresponding QMP calls.

4.1.2 QMP on MPI

There is already an implementation for QMP on MPI, written by Jie Chen and Robert Edwards of the Jefferson Lab HPC group. The code is rather compact with about 5000 lines of code and about 22 KiB of code size (.text segment compiled with spugcc) in QMP-2.3.1. This version worked nearly out of the box on SPE-MPI, with only a one-line patch to make MPI 1.X compliant. It offers a complete implementation of QMP and is intended for single node use and clusters where direct hardware support is (not yet) available. As QMP is mostly a functional subset of MPI, many functions like collectives can be translated to the MPI variants without limitations.

Only a subset of MPI is used, so QMP on MPI only needs a few MPI functions actually (efficiently) implemented. A reduced subset of MPI or a version where only special cases are implemented efficiently would be sufficient. The performance is expected to be lower than a directly implemented QMP on the QPACE torus (see next section).

Figure 4.2: QMP on MPI block diagram

\footnote{Available at: http://usqcd.jlab.org/usqcd-software/qmp http://usqcd.jlab.org/usqcd-software/qmp}
CHAPTER 4. MESSAGE PASSING LIBRARIES ON QPACE

<table>
<thead>
<tr>
<th>Component</th>
<th>MPI Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>MPI_Init, MPI_Finalize, MPI_Wtime,</td>
</tr>
<tr>
<td></td>
<td>MPI_Get_processor_name, MPI_Error_string, MPI_Abort</td>
</tr>
<tr>
<td>Message Passing</td>
<td>MPI_Send_init, MPI_Recv_init, MPI_Start,</td>
</tr>
<tr>
<td></td>
<td>MPI_Startall, MPI_Wait, MPI_Waitall, MPI_Test,</td>
</tr>
<tr>
<td></td>
<td>MPI_Testall, MPI_Request_free</td>
</tr>
<tr>
<td>Datatype Management</td>
<td>MPI_Type_vector, MPI_Type_struct, MPI_Type_commit,</td>
</tr>
<tr>
<td></td>
<td>MPI_Type_free</td>
</tr>
<tr>
<td>Communicator</td>
<td>MPI_Comm_size, MPI_Comm_rank, MPI_Comm_dup</td>
</tr>
<tr>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>Collectives</td>
<td>MPI_Barrier, MPI_Bcast, MPI_Allreduce, MPI_Op_create</td>
</tr>
</tbody>
</table>

Figure 4.3: List of used MPI calls in the QMP MPICH implementation

because of software overhead like the message scheduling in MPI, which is not needed for the QMP operation. The impact of this kind of overhead has yet to be evaluated. A list of the employed MPI calls is given in Figure 4.3.

4.1.3 QMP on QPACE Torus

The torus low level library offers nonblocking nearest neighbor communication primitives which can be used directly by the corresponding QMP calls. There is no need for flow control if it can be expected that the program is running synchronous, posting receives at the same time as the neighbor is posting sends. Collective calls can be implemented on top of the basic communication primitives.

However, QMP has more capability requirements (as listed in [56]) which are not so trivial to be satisfied. Sending a contiguous message to a given node which is not a nearest neighbor requires routing capabilities from the torus network. This is not implemented in hardware to this time of writing. Software routing is an option, but requires protocols to handle unexpected incoming messages which have to be forwarded, and hence the performance advantage over MPI by not using protocols but only having synchronous communication is ruined.

Another option for the routing problem is to use PPE assisted callbacks for these primitives. This could be solved for example as in SPE-MPI [61], with a QMP instance running on the PPE for each SPE node. This option would slow down the non-nearest-neighbor communication latency by an order of magnitude, but would keep the nearest neighbor communication fast and synchronous. This would be a fair option if non-nearest-neighbor communication is not time critical.

4.1.4 MPI on QPACE Torus

There are different approaches to implement MPI on the SPE. Reduced subsets of MPI have been introduced in [59, 60]. A complete MPI implementation can
be provided with an approach like SPE-MPI [61], which can also be combined with the reduced implementations. The idea is to implement only time critical calls and features on the SPE, and falling back to PPE assisted callbacks if this is not possible. For example on QPACE, an MPI_Send could check if it is an easy datatype (basic, maybe also vector or strided) and if the destination is a neighbor. If the requirements are met, the data can be sent directly over the torus, otherwise the PPE must send it via its own MPI implementation, e.g. OpenMPI using Ethernet. It is important to note that this mechanism only works if the requirement decision can also be made a-priori on the receiver side. Otherwise there could be conflicts, for example the data is transmitted directly to the SPE, but the PPE is waiting for the reception and can not be notified. 2 This also makes the MPI_ANY_SOURCE or MPI_ANY_TAG wildcard operators unusable, they must be forbidden if direct torus communication should be used. This is not a very strong limitation in favor of the gained performance, but ruins the complete MPI compliance.

There are features which have to be implemented for the direct torus communication to work, and some which are optional and can be applied if the application needs it. The considerations for optional features are the gained performance and the bloat of the library size.

Mandatory features

Communication protocol on the QPACE torus

The QPACE torus allows messages to be delivered in-order to the nearest neighbor, where the message is size is limited by the transmit buffer. Large messages have to be split into smaller packets. Each neighbor link FIFO is limited to 8 KiB, which may not be exceeded to avoid deadlock problems, hence a flow control engine with congestion control must be used. MPI semantics support concurrent messages which might have to be delivered out of order, for example when nonblocking or buffered sends with different tags are used and the corresponding receive calls are posted in a different order.

There are various approaches [60, 62] to implement MPI message passing on zero copy RDMA or channel semantic networks, for example with eager and rendezvous protocols, and the different options must be evaluated on the QPACE hardware.

Optional features

Communicators

The MPI_COMM_WORLD communicator is mandatory. If subcommunicators are to be supported, the book keeping information has to be stored on the SPE and synchronized with the PPE, e.g. the map of neighbors to the physical devices within the communicator or a flag if the communicator

\footnote{Employing MPI_Irecv, polling on both PPE and SPE, and calling MPI_Cancel when the other Core has received the data would be an option, but is too expensive because PPE assisted callbacks would have to be used in every case.}
is connected. More complicated communicators like graph communicators can be left unsupported and handled by the PPE.

Collectives
Collectives can be easily build on top of the nearest neighbor primitives when the MPI_COMM_WORLD communicator is used. If this is not the case, it depends on the communicator topology if collectives are efficiently implementable. For example a communicator in a subspace of the torus can use the same algorithms as in MPI_COMM_WORLD, but if the communicator is scattered over the torus in not connected parts and routing capability is not available, one needs to resort to PPE assisted callbacks and it might even be more efficient to just call the corresponding collective call on the PPE.

An overview of some collective communication algorithms is given section 3.4.

Datatypes
The datatype engine for QPACE can be more simple than in a general MPI like OpenMPI which has to support heterogeneous environments. Derived datatypes still pose a greater challenge and require a packing/unpacking engine or special support to build scatter/gather Lists for non contiguous datatypes. A first implementation might only support the basic datatypes (MPI_FLOAT, MPI_DOUBLE, ...) and can then be extended to support strided datatypes or vector datatypes.

Software routing
The torus does not support routing in hardware. Software Routing is needed if torus communication beyond neighbors should be possible on the SPE, without falling back to the PPE. The link protocol between the SPEs has to be extended to support this, and interrupts on the SPEs should be used to allow fast forwarding of packets and thus reasonable latencies.

4.1.5 Conclusion
Mappings of the different message passing protocols on each other have been discussed. Each mapping comes with its own overheads. Figure 4.2 and Figure 4.1 show possible designs with the most important features described in this section.

For the MPI on QMP mapping, QMP lacks several features and semantics which have to be implemented (again) in the MPI Layer. QMP can only provide limited datatype support and collective support, and this functionality is redundant if these functions must be implemented in MPI again for full support. However for an MPI implementation which should only offer a limited subset this is a reasonable option. For a complete implementation, most of the MPI library has to be implemented on the SPE which is not feasible. Using PPE for
assisted callbacks is an option, but in this case it would also be possible to skip
the QMP layer at all and use the torus low level library directly.

The QMP on MPI mapping is more favorable from a practical point of view,
as the QMP on MPI implementation is already available and only needs to be
adopted to the SPE architecture. QMP only uses a subset of MPI, therefore it’s
enough for QMP if only this subset of MPI is implemented efficiently, e.g. with-
out PPE assisted callbacks. It has to be evaluated how the protocol overhead in
MPI affects the performance in QMP compared with a direct implementation.
If the performance appears not to be satisfying, this approach can also be used
as base for further optimizations where critical calls are implemented directly
on the torus library.

4.2 PPE Centric Approaches

Applications written for general purpose architectures are sometimes challenging
to port to the SPE architecture. A common approach which also has been
applied to HPL is to keep the original main code on the general purpose PPE
and only port performance (computation) critical routines for the SPE. Even if
the QPACE torus network is designed with SPE centric applications in mind,
the torus can be used from PPE centric applications as described in section 3.2.

4.2.1 PPE with direct MFC access to the NWP

The memcpy method described in section 3.2.3 can be used to send and receive
data to and from main memory. For the send, one chunk from the main memory
can be transferred to the SPE Local Store and then from the Local Store to the
IWC of the NWP. Receiving a message would require a supplied credit which
can be sent directly from the PPE or with the MFC. The packet received in the
Local Store then can be transferred with another MFC operation back to the
main memory.

The only requirements for this method are that some buffer space on the
SPE is available. The usual SPE code could be executed independently on the
SPE during transfer, but performance regression are to be expected as the MFC
is shared between PPE and SPE. The PPE only needs access to the problem
state area on the SPE, which can be requested in a Linux system with the
sufficient privileges.

If all links are used for sending and receiving at the same time at full speed,
the MFC has to process at least $6 \cdot 2 \cdot 1 \text{GiB/s} = 12 \text{GiB/s}$ for the send direction
(2 memory transfers per packet) and at least $6 \cdot 1 \text{GiB/s} = 6 \text{GiB/s}$ for the
receiving direction. This totals in $18 \text{GiB/s}$ required bandwidth, which can
only be reached with packets sizes of at least 5KiB according to the bandwidth
calculation given in section 3.2.4 because of the latency to program the MFC
from the PPE. Distributing the transfers onto multiple SPEs would not change
this as the number of requests is constant over the used SPEs; however this is a
valid option if it turns out that one MFC is not sufficient to sustain the memory bandwidth.

4.2.2 Function Offloading

If the SPEs in an application are not busy with tasks while communicating, they can also be used for communication functions. This is mostly the case for HPL with its synchronous calls to the SPEs (see section 2.2.2). Collective operations can be outsourced as functions on the SPEs. This requires that the SPE code is prepared for these functions: it should not only contain the application-specific code, but also the communication routines. These could be implemented as a SPE library, and the application specific dispatcher function needs to recognize the commands for these routines. If the application does not use the SPEs at all or keep one SPE for communication only, the message passing library has to assure that the communication code is loaded.

Besides the function offloading, the SPEs could also asynchronously send and receive messages for the PPE: The PPE only gives the command for a send or receive with the respective data position and length in the main memory, and the SPE executes the transfer asynchronously. This is a possible solution especially for nonblocking communication. The SPE can then poll for progress in the NWP without disturbing the PPE.

4.2.3 Integration into MPI or QMP

The previous approaches can be integrated into existing OpenMPI frameworks or as QMP implementation to support legacy applications. This method is convenient as the porting effort of the applications can be minimized. Only the process mapping should be set according to the applications communication patterns to allow nearest neighbor communication wherever possible. As an example, an integration into the OpenMPI framework is described in section 4.4.

4.3 Programming Model Considerations for HPL

Different programming models are possible on a heterogeneous processor like the PowerXCell 8i processor. The IBM version of HPL uses the PPE centric model, handing over the most critical BLAS tasks like matrix multiplication to the SPEs, but keeping the communication library and other BLAS routines on the PPE. Task parameters are pushed via processor internal mailbox transfers. The main considerations in this model are the task granularity versus the overhead to hand over the task, and the library size on the SPE with its limited Local Store size.
Another alternative programming model for HPL would be the SPE centric model, where the complete application code is located on the SPE. The PPE is only called for Operating System tasks via PPE assisted callbacks. As we can see in Table 4.3, a code size optimized HPL where MPI and BLAS have yet to be included consumes already 75% of the SPE Local Store which can store 256 KiB data. The SPE Local Store also needs to keep the stack and run-time data, which makes code overlays and other data size saving techniques (and resulting overhead) mandatory. The currently optimized specialists almost consume the complete Local Store size too. This makes techniques like SPE overlays unavoidable, which will decrease the performance.

For a SPE centric model, there is the question of how many SPEs should form one process on each node. When each SPE has its own LINPACK process, each process will only have 512 MiB of RAM available for data. The SPE computation specialists however only reach a good performance on very large data sets, a small process to SPE ratio is therefore recommendable. Using only one SPE as “master” SPE which needs to synchronize with the other SPEs brings us to a similar model as the PPE centric accelerator model, with similar overheads.

An advantage of the SPE centric model in the case of QPACE is that the network devices are designed for SPE access and should provide lower latencies. However the cost of SPE overlays will probably ruin this benefit, and this possible small advantage alone is no justification for the porting efforts needed to port the code for the SPE.

As the advantages of a PPE centric model strongly prevail, the proposal is to use this model for the QPACE HPL implementation. A pragmatic approach is to use and enhance the publicly available QS22 version of HPL.

### 4.3.1 SPE Accelerated Communication Tasks

The communication tasks are various collectives and have been identified above. Like in the IBM version of HPL for the BLAS routines, we can outsource MPI collectives like `MPI_Allreduce()`, `MPI_Broadcast()`, `MPI_Scatterv()` or HPL-specific routines to the SPE. Instead of calling the MPI library, we would call the SPE library instead.

If the granularity of the collectives turns out to be too fine-grained, the calling functions can also be moved into the SPE until the granularity is coarse.
CHAPTER 4. MESSAGE PASSING LIBRARIES ON QPACE

enough. If asynchronous calls are possible the SPE may also run concurrently to the PPE.

There is only one MPI process per PowerXCell 8i processor, so one SPE is sufficient to handle communication, provided that the MFC can handle the memory requests fast enough. Implementing the functions on the SPE instead of the PPE has the advantage that memory accesses do not cause stalls, but can be overlaid, which enables better performance (see section 3.2 for more on this topic).

4.3.2 MPI Network Module

Instead of outsourcing the communication tasks on the application level, these communication tasks could also be implemented and outsourced in the message passing library as described in 4.4 for OpenMPI. However the communication operations of an MPI library are likely to be more general than the specific application requires, and therefore overhead might be associated with the support of this generality.

Unlike the SPE accelerated communication tasks, the granularity is fixed to the MPI calls. Inside of the MPI module, both PPE network access and SPE accelerated communication is possible, provided that the SPE code supports it. A very strong argument for this option is that the original application can be transparently used without modification, and other applications can benefit from this approach as well. Also the MFC assisted access can be used in this module.

4.3.3 Conclusion

As the High Performance LINPACK sends large messages in most of its time critical communication tasks, the fixed granularity to MPI tasks is not a performance problem for this application. For portability reasons the proposal is therefore to implement an MPI module which can then be used by HPL without major modification in the application. A possible solution for this proposal is described in the following section.

4.4 Integration into OpenMPI

An OpenMPI BTL[2, 63] or Channel Device for MPICH [64] or MPICH2 [65, 66] allows unmodified applications to run on the PPE using the torus network while minimizing the efforts to support the new network. Function offloading or direct MFC access can be employed to implement the module. If the direct MFC access method is used, bandwidth-critical applications may however be limited by the PPE latencies for some communication patterns as described in 3.2.8. The application should be started with a process mapping that allows nearest neighbor communication in most of its communication patterns, otherwise the MPI implementation will resort to the slow Gigabit Ethernet.
CHAPTER 4. MESSAGE PASSING LIBRARIES ON QPACE

As an example we will consider the requirements for the OpenMPI modules in more detail:

4.4.1 OpenMPI Modular Component Architecture

Figure 4.4: OpenMPI Layer Model

OpenMPI is organized into layers as illustrated in Figure 4.4. Most of these layers are implemented in so called “components” of the OpenMPI “Modular Component Architecture” (MCA). Each of these components is implemented according to a well defined interface. This modular architecture allows to develop the components independently, or substitute components with other implementation without changing the other components. This is especially useful for the implementation of new network devices: Only a byte transfer layer (BTL) component, and optionally a collective component (COLL) has to be implemented, which are marked green in the Figure 4.4. An efficient implementation of only these components is enough to leverage the hardware potential of the new network interconnect. The runtime system can decide which components are to be enabled, or the user can forbid certain components.

Any component must provide version information and implement the following functions:
/**
 * BTL module interface functions and attributes.
 */
struct mca_btl_base_module_t {
    /* BTL common attributes */
    mca_btl_base_component_t * btl_component; /**< pointer back to 
        ** the BTL component structure */
    size_t btl_eager_limit; /**< maximum size of first fragment 
        ** - eager send */
    size_t btl_min_send_size; /**< threshold below which the 
        ** BTL should not fragment */
    size_t btl_max_send_size; /**< maximum send fragment size 
        ** supported by the BTL */
    size_t btl_min_rdma_size; /**< threshold below which the 
        ** BTL should not fragment */
    size_t btl_max_rdma_size; /**< maximum rdma fragment size 
        ** supported by the BTL */
    uint32_t btl_exclusivity; /**< indicates this BTL should 
        ** be used exclusively */
    uint32_t btl_latency; /**< relative ranking of latency 
        ** used to prioritize btl's */
    uint32_t btl_bandwidth; /**< bandwidth (MB/Second) 
        ** supported by each endpoint */
    uint32_t btl_flags; /**< flags (put/get... ) */
    /* BTL function table */
    mca_btl_base_module_add_procs_fn_t btl_add_procs;
    mca_btl_base_module_del_procs_fn_t btl_del_procs;
    mca_btl_base_module_register_fn_t btl_register;
    mca_btl_base_module_finalize_fn_t btl_finalize;
    mca_btl_base_module_alloc_fn_t btl_alloc;
    mca_btl_base_module_free_fn_t btl_free;
    mca_btl_base_module_prepare_fn_t btl_prepare_src;
    mca_btl_base_module_prepare_fn_t btl_prepare_dst;
    mca_btl_base_module_send_fn_t btl_send;
    mca_btl_base_module_put_fn_t btl_put;
    mca_btl_base_module_get_fn_t btl_get;
    mca_btl_base_module_dump_fn_t btl_dump; /**< diagnostics */
    /**< the mpool associated with this btl (optional) */
    mca_mpool_base_module_t * btl_mpool; /**< register a default error handler */
    mca_btl_base_module_register_error_fn_t btl_register_error;
};
typedef struct mca_btl_base_module_t mca_btl_base_module_t;

Figure 4.5: OpenMPI BTL Component Interface (OpenMPI version 1.2.8)

- `int mca_open_component (void);` loads the component and initializes 
  component internal data structures.
- `int mca_close_component (void);` is called after all modules have been 
  finalized to clean up internal data structures.

4.4.2 OpenMPI Byte Transfer Layer (BTL)

A BTL module offers point to point byte data transfer service from one process 


to another. It therefore implements low level point to point primitives directly 


on the network interface. Furthermore the BTL offers information to the upper 


protocol layers, and the BTL Management Layer (BML) can decide depending 


on these information and attributes which BTL component to use.
Figure 4.5 shows the interface of the BTL component. The scalar values inform the upper layers about thresholds and limits of the BTL as described in the comments. The exclusivity, latency, bandwidth and flags parameters are used to prioritize and select the appropriate BTL for a certain message.

The BTL functions which must be implemented are:

- **int btl_add_procs(struct mca_btl_base_module_t* btl, size_t nprocs, struct ompi_proc_t** procs, struct mca_btl_base_endpoint_t** endpoints, struct ompi_bitmap_t* reachable);** is called by the Point to Point Management Layer (PML) to inform about new processes which have been added to the process list. The BTL can then inform the upper layers within the reachable bitmap which process can be reached. A torus BTL would only mark those processes reachable which are nearest neighbors of this node. The endpoints parameters can be used to store BTL specific information about the neighbor for faster access, e.g. link number.

- **int btl_del_procs (struct mca_btl_base_module_t* btl, size_t nprocs, struct ompi_proc_t** procs, struct mca_btl_base_endpoint_t** );** is called to inform the BTL that processes were deleted from the process list.

- **int btl_register (struct mca_btl_base_module_t* btl, mca_btl_base_tag_t tag, mca_btl_base_module_recv_cb_fn_t cbfunc, void* cbdata );** registers a callback which is called when a packet fragment is received.

- **int btl_finalize(struct mca_btl_base_module_t* btl );** is called before unloading to let the BTL clean up and release its allocated resources.

- **mca_btl_base_descriptor_t* btl_alloc (struct mca_btl_base_module_t* btl, size_t size );** allocates a segment of the specified size which can be used for sending or receiving.

- **int btl_free (struct mca_btl_base_module_t* btl, mca_btl_base_descriptor_t* descriptor );** returns an allocated segment to the BTL.

- **int btl_prepare_src (struct mca_btl_base_module_t* btl, struct mca_btl_base_endpoint_t* endpoint, mca_mpool_base_registration_t* registration, struct ompi_convertor_t* convertor, size_t reserve, size_t* size );** prepares a descriptor for a send. The function must pack the data if the data supplied in the convertor is not contiguous. reserve specifies how many bytes should precede the packed data, and size returns the actual size of the data.
CHAPTER 4. MESSAGE PASSING LIBRARIES ON QPACE

- The `btl_prepare_dst` call has the same prototype as `btl_prepare_src` and is only required if MPI-2 RDMA functionality should be supported.

- `int btl_send (struct mca_btl_base_module_t* btl, struct mca_btl_base_endpoint_t* endpoint, struct mca_btl_base_descriptor_t* descriptor, mca_btl_base_tag_t tag);` initiates an asynchronous send of the prepared data to the peer. If the data can not be sent in the moment of this call, the data may be queued and sent later in a progress function.

- `int btl_put (struct mca_btl_base_module_t* btl, struct mca_btl_base_endpoint_t* endpoint, struct mca_btl_base_descriptor_t* descriptor);` initiates an asynchronous RDMA put. This function is only required if MPI-2 RDMA functionality is implemented, and we may skip this for the QPACE torus.

- `int btl_get (struct mca_btl_base_module_t* btl, struct mca_btl_base_endpoint_t* endpoint, struct mca_btl_base_descriptor_t* descriptor);` initiates an asynchronous RDMA get. This function is only required if MPI-2 RDMA functionality is implemented, and we may skip this for the QPACE torus.

- `int btl_dump (struct mca_btl_base_module_t* btl, struct mca_btl_base_endpoint_t* endpoint, int verbose);` dumps diagnostic information of the BTL state for this endpoint.

The component must furthermore implement the following functions for the component:

- `struct mca_btl_base_module_t** mca_btl_component_init (int* num_btls, bool enable_progress_threads, bool enable_mpi_threads );` initializes one or more modules (instances) of this BTL for each device (we would only create one for the torus device), the number is returned in `num_btls`. The hardware discovery and setup should be done in this function. `enable_progress_threads` and `enable_mpi_threads` inform the BTL whether progress threads or MPI threads are supported.

- `int mca_btl_base_component_progress (void);` defines the progress function. This function should poll for incoming packets and call the appropriate receive handler or send packets which were postponed before. For the QPACE torus we should check the notify area, provide credits, and send buffered fragments which have been queued by `btl_send`.

As we have seen the send and receive functions are called in a nonblocking way. This requires that any send operation must return immediately even if the fragment could not be sent (yet) because of back pressure in the network. Furthermore a tag is used to identify the message on the peer side. A minimal envelope protocol must therefore be employed to identify the messages.
The decision whether the QPACE BTL should be implemented with the direct MFC access or the function offloading model depends on the application: When the SPEs are not utilized by the application, synchronous SPE calls are used or one (or more) SPE(s) can be sacrificed for communication, the function offloading model can be employed. The PPE could then pass new transfer request to the SPE, which checks for progress asynchronously to the PPE. An application which uses the SPEs while communicating may only use the direct MFC access method from the PPE.
4.4.3 Collectives Component (COLL)

A collective component can offer special network optimized collective operation support for a specific network. As in the BTL, the upper layers may have multiple candidates for a collective operation in a given communicator and can choose the best candidate available by priority.

```c
/*
 * Structure for coll v1.0.0 components
 * Chained to MCA v1.0.0
 */
struct mca_coll_base_component_1_0_0_t {
    mca_base_component_t collm_version;
    mca_base_component_data_1_0_0_t collm_data;
    /* Initialization / querying functions */
    mca_coll_base_component_init_query_fn_t collm_init_query;
    mca_coll_base_component_comm_query_1_0_0_fn_t collm_comm_query;
    mca_coll_base_component_comm_unquery_fn_t collm_comm_unquery;
};
typedef struct mca_coll_base_component_1_0_0_t mca_coll_base_component_1_0_0_t;
/*
 * This struct is hung on the communicator by the winning coll component
 * after the negotiation. It has pointers for all the collective
 * functions, as well as a "finalize" function for when the
 * communicator is freed.
 */
struct mca_coll_base_module_1_0_0_t {
    /* Per-communicator initialization and finalization functions */
    mca_coll_base_module_init_1_0_0_fn_t coll_module_init;
    mca_coll_base_module_finalize_fn_t coll_module_finalize;
    /* Collective function pointers */
    mca_coll_base_module_allgather_fn_t coll_allgather;
    mca_coll_base_module_allgatherv_fn_t coll_allgatherv;
    mca_coll_base_module_allreduce_fn_t coll_allreduce;
    mca_coll_base_module_alltoall_fn_t coll_alltoall;
    mca_coll_base_module_alltoallv_fn_t coll_alltoallv;
    mca_coll_base_module_barrier_fn_t coll_barrier;
    mca_coll_base_module_bcast_fn_t coll_bcast;
    mca_coll_base_module_bcastv_fn_t coll_bcasc;
    mca_coll_base_module_exscan_fn_t coll_exscan;
    mca_coll_base_module_gather_fn_t coll_gather;
    mca_coll_base_module_gatherv_fn_t coll_gatherv;
    mca_coll_base_module_reduce_scatter_fn_t coll_reduce;
    mca_coll_base_module_reduce_scatterv_fn_t coll_reducescatter;
    mca_coll_base_module_scan_fn_t coll_scan;
    mca_coll_base_module_scatter_fn_t coll_scatter;
    mca_coll_base_module_scatterv_fn_t coll_scatterv;
};
typedef struct mca_coll_base_module_1_0_0_t mca_coll_base_module_1_0_0_t;

Figure 4.6: OpenMPI COLL Component Interface (as of OpenMPI 1.2.8)

The interface definition of a collective is given in Figure 4.6. `collm_version`
and `collm_data` provide metadata for the component. A collective component implements:

- `int coll_init_query (bool enable_progress_threads, bool enable_mpi_threads);` allows the collective component to back off if it does not support the required threads.

71
• const mca_coll_base_module_1_0_0_t *coll_comm_query
  (struct ompi_communicator_t *comm, int *priority, struct mca_coll_base_comm_t **data); is called when a new communicator is created. The component can then analyze the new communicator comm, for example check if the communicator topology is a torus, a mesh, or even connected. It then sets its priority for this communicator and provides a set of function pointers to the appropriate collective functions with the data pointer if it can support this topology.

• int coll_comm_unquery (struct ompi_communicator_t *comm, struct mca_coll_base_comm_t *data); allows to clean up data which has been allocated in the coll_comm_query function. This function is optional.

When the collective component has the highest priority for this communicator, it is hung to the communicator structure. This structure contains the functions:

• struct mca_coll_base_module_1_0_0_t * coll_module_init
  (struct ompi_communicator_t *comm); is called for the winning module and allows to hang additional data to the communicator comm.

• int coll_module_finalize (struct ompi_communicator_t *comm);
  is called when the communicator is destroyed to let the module clean up its data from the communicator comm.

• the respective collective functions

One or more collective operations can be implemented, and optimized for the different topologies. For example a broadcast can be implemented for a 3D torus and a 2D mesh, and the correct callback function is then passed by the query function. It is also possible to not implement a collective operation - one of the standard algorithms will then be used (which might not use the torus network).

The considerations whether direct MFC access or function offloading should be used are similar to the BTL case, as the COLL component also has direct access to the network hardware.
Chapter 5

Conclusion and Outlook

In this chapter, the applicability of the strategies developed for QPACE are discussed in section 5.1. The thesis is then concluded in section 5.2, and the next steps to provide a final HPL implementation on the QPACE architecture are discussed in section 5.3.

5.1 Application to the NICOLL Project

The NICOLL architecture shares similarities with the QPACE architecture as network access from the Cell/B.E. SPE is intended in both systems. In section 5.1.1 we discuss the steps taken towards a NICOLL prototype. Section 5.1.2 presents assumptions what to be expected from this prototype, and based on these assumption the applicability of QPACE programming models and message passing strategies are finally discussed in section 5.1.3.

5.1.1 Related Work

The NICOLL projects’ objective is to build a hybrid system which couples an AMD Opteron microprocessor and an IBM Cell/B.E. microprocessor by employing an FPGA which bridges between the processor specific protocols HyperTransport and FlexIO. This proposed architecture is illustrated in Figure 5.1. Unfortunately such a prototype is not yet available. As an intermediate step, a PCI Express coupled system using standard hardware was built. An x86(-64) Host system and an IBM BladeCenter QS21 were interconnected with PCIe@host interface boards [67], establishing a PCIe x8 Link between the systems. This method allowed to explore the bring up, system level integration and shared memory approaches of such a hybrid system [68]. For software and operating system integration, a Remote SPU File System (RSPUFS) was proposed which allows to create and manage SPE contexts from other systems than the Cell/B.E. architecture [69]. This approach has evolved to ACCFS [70, 71, 72], a modular Linux file system driver which allows to control various types of accel-
Figure 5.1: The NICOLL Architecture

operators like Cell/B.E. SPEs, FPGAs or GPGPUs ("General Purpose computing on Graphic Processing Units"), providing one consistent interface to manage different types of accelerators. This interface is the preferred interface for the NICOLL prototype because of its portability. It allows not only the acceleration with Cell/B.E. SPEs, but also for example reconfigurable modules in the FPGA [73].

5.1.2 Architecture Assumptions for NICOLL

As the final NICOLL architecture is not yet specified and there still are different alternatives for the architecture which have yet to be evaluated, we need to make assumptions to formulate ideas or apply the QPACE strategies.

The NICOLL machine is assumed to be a shared memory architecture, at least from a process point of view. The SPEs can access at least a specified window in the main memory which can also be accessed from the Opteron. The shared memory is assumed to be cache coherent, or mechanism exist to assert the coherence of the memory system, e.g. by calling explicit synchronization operations.

The AMD Opteron takes the typical place of the PPE, it runs the operating system and legacy application code. The reason for this decision is that the AMD Opteron is superior in terms of processing power, memory bandwidth and IO bandwidth over the PPE, and more legacy codes are available for the x86 architecture. To support an efficient acceleration of collective operations on the SPE, it is required that the communication hardware can physically access the SPE Local Store. The SPE Local Stores must therefore be mapped into the physical address space of the system and accessible from the FPGA.

As the cluster communication device, the InfiniBand interconnect was planned. No proof of concept study of an InfiniBand stack implemented on
a Cell/B.E. SPE core has been published yet. A possible reason for this is that traditional Cell/B.E.-based systems don’t allow physical access to the SPE Local Store from the I/O hardware. However, there is a case study of a 10 Gbps Ethernet Interface on a SPE core [74] using a custom Cell/B.E.-based system, and very good performance could be reached in this approach. One problem however was the limited size of the Local Store, and several techniques have been applied to avoid exhausting this space. From this study it can be concluded that a stripped down InfiniBand stack might also be feasible on the SPE.

Another alternative for NICOLL could be to use a custom network interconnect like the QPACE torus network. The advantage of this approach is that the interface is very simple and a software stack could be implemented with very small amount of Local Store space. Because many interesting alternatives are possible, we further only assume that a network interconnect exists which allows an efficient implementation on an SPE core.

### 5.1.3 QPACE Programming Models and Message Passing Strategies for NICOLL

When the Opteron takes the place of the PPE of the Cell/B.E. with the same functionality, the proposed programming models for the Cell/B.E. can be employed. The PPE centric approaches function offloading, direct MFC transfers or OpenMPI components could be implemented in the same way as Opteron centric versions. The memory position where messages are stored will influence the performance: If the Opteron memory is used, the link between the FPGA and the Cell/B.E. is occupied multiple times for data transfer to Local Store and network transfer. Therefore messages stored in the Cell/B.E. memory should allow better performance, as the link is then exclusively available for the network transfer.

The latencies to the MFC of the SPEs must be revised for Opteron-driven communication, for example as in section 3.2.4. These latencies are expected to be higher than PPE-driven communication, as the MFC is not chip local for the Opteron. These might require larger buffer sizes to sustain the peak network bandwidth. The function offloading mechanism should only suffer from higher call overheads, which are not critical for large messages. This mechanism can be implemented using the ACCFS framework. Finally, these mechanisms can be included in an MPI framework module which would allow legacy code to directly benefit from the accelerated network functionality.

SPE centric message passing libraries are also possible. However, special care has to be taken regarding the different datatypes. Assisted callback mechanisms as used in SPE-MPI [61] build on the fact that the datatypes in 32 bit PPE code and SPE code are very similar. The native Opteron datatypes however differ from the SPE datatypes in both size and byte ordering.
CHAPTER 5. CONCLUSION AND OUTLOOK

5.2 Conclusion

This thesis provides the theoretical background for a successful High Performance LINPACK implementation on QPACE. An implementation could not be provided and evaluated as the QPACE hardware is still under active development.

We have seen that the communication pattern of HPL can be mapped on a 3D torus using nearest neighbor communication, and how to map the processes on the torus. The network requirements indicate that mostly large messages are sent, and the communication performance mostly depends on the available bandwidth. The implementation should be based on the QS22 version of HPL, which already provides well optimized linear algebra subroutines for the most demanding kernels.

The QPACE architecture has been analyzed for these requirements. The original QCD optimized communication model has been evaluated, and an alternative model for general communication has been proposed. We have seen that also the PPE can be used for torus communication. The limits of the current hardware and enhancement proposals have been presented. Based on an abstract model, example collective algorithms for different topologies and their performance estimations have been presented which can be used in QPACE optimized message passing libraries.

Different programming models with a focus on SPE and PPE centric programming have been shown. QMP and MPI were compared from a functional point of view and different alternatives for their implementation have been evaluated. The PPE centric mechanisms, function offloading and direct MFC access were revised for the use in the message passing library, and their integration into an existing MPI framework has been evaluated for the example OpenMPI. The programming model alternatives have been evaluated for the HPL application, concluding that a PPE centric approach with either SPE accelerated communication tasks or a torus optimized MPI implementation should be employed.

Finally we’ve seen that it is possible to apply these programming model strategies to the NICOLL project, with certain limitations posed by the architectural differences and higher latency times.

5.3 Further Work

The natural next steps are to implement the low level interface and the proposed MPI components on the QPACE machine. The scheduling of the message transfers to different links needs to be analyzed along with an evaluation of the practical sustainable NWP capabilities for long messages. The proposed NWP enhancement should be evaluated for their feasibility in the current hardware design, and eventually implemented on the FPGA to allow higher bandwidth.

As soon as a large QPACE partition is available, the High Performance LINPACK benchmark can be evaluated. The impact of the PPE directed MFC transfers on the SPE accelerated functions should be analyzed. If possible, a
recabling of the machine allows new grid sizes for the benchmark. The more symmetric grids should then sustain a higher performance on 4 racks. Interesting enhancements for HPL which can be further evaluated are to convert more of the SPE accelerated synchronous functions to asynchronous functions, which would allow more overlapping between PPE and SPE tasks.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

Source Code

The source code which has been written for this diploma thesis can be obtained on request from the Computer Architecture Group at TU Chemnitz. By an agreement between the author and IBM, this source code belongs to IBM and may only be used with permission of IBM. It contains:

- A patch to the QS22 version of HPL 1.0a which adds function instrumentation and collective support as described in section 2.6.
- A patch to the MPICH version of QMP 2.3.1 to make it MPI-1.1 compliant.
- The MFC memcpy benchmark as described in section 3.2.4.
- The access time benchmark as described in section 3.2.5.
Thesis Declaration

I hereby declare that the whole of this diploma thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography. I declare that it has not been submitted in whole, or in part, for any other degree.

Chemnitz, March 9, 2009

______________________________
Simon Wunderlich