**FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG**
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

**Lehrstuhl für Informatik 10 (Systemsimulation)**

# Multigrid algorithms on QPACE

Daniel Brinkers

Diploma Thesis

# Multigrid algorithms on QPACE

Daniel Brinkers

Diploma Thesis

| | |
|---|---|
| Aufgabensteller: | Prof. Dr. U. Rüde |
| Betreuer: | Dipl.-Inf. M. Stürmer |
| Bearbeitungszeitraum: | 01.09.2009 – 01.03.2010 |

**Erklärung:**

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 1. März 2010 .....

**Abstract**

QPACE is a cluster computer developed for lattice QCD calculations. The computing nodes consists of IBM PowerXCell 8i processors. The interconnect is a custom three dimensional torus network. The application dependent design goals were a latency smaller then 1$\mu$s and a bandwidth about 1 GiB/s. The cluster is very energy efficient and reached the 1st place on the Green500 list of November 2009. With this work a multigrid solver should be implemented and investigated on this hardware. It was concluded, that it is possible to use this hardware for multigrid calculations. Another important result is the conclusion that the analysis of communication patterns is important to understand the performance of programs on this hardware. Methods and parameters for this analysis are introduced.

**Zusammenfassung**

QPACE ist ein Cluster, der für Gitter QCD Berechnungen entwickelt wurde. Die Clusterknoten bestehen aus IBM PowerXCell 8i Prozessoren. Als Netzwerk wird ein eigens entwickeltes drei dimensionales Torus Netzwerk verwendet. Die anhand der Anwendunganforderungen aufgestellten Designziele waren eine Latenz kleiner als 1 $\mu$s und eine Bandbreite um die 1 GiB/s. Der Cluster ist sehr energieeffizient und erreichte den ersten Platz auf der Green500 Liste von November 2009. In dieser Arbeit soll ein Mehrgitter Löser auf dieser Hardware entwickelt und untersucht werden. Es wird festgestellt, dass es möglich ist die Hardware für Mehrgitterberechnungen verwendet werden kann. Ein weiteres wichtiges Ergebnis ist die Feststellung, dass die Analyse von Kommunikationsmustern auf dieser Hardware sehr wichtig ist um die Performance von Programmen einzuschätzen. Methoden und Parameter für diese Analyse werden vorgestellt.

# Contents


**1 Introduction** **7**

**2 QPACE project** **7**
2.1 QCD . . . 7
2.2 eQPACE . . . 8

**3 QPACE architecture** **8**
3.1 Cell . . . 8
3.1.1 EIB . . . 8
3.1.2 SPE . . . 9
3.1.3 PPE . . . 9
3.2 The cluster structure . . . 9
3.3 Torus network . . . 9
3.3.1 Hardware . . . 9
3.3.2 Measurements . . . 10
3.4 Software environment . . . 14
3.4.1 System . . . 14
3.4.2 TNW library . . . 14

**4 Multigrid on eQPACE** **15**
4.1 Multigrid . . . 15
4.2 Model problem . . . 16
4.3 Implementation . . . 16
4.3.1 Communication library . . . 16
4.3.2 Domain splitting . . . 17
4.3.3 Vectorisation . . . 17
4.3.4 Shuffle operation . . . 17
4.3.5 Loop unrolling . . . 18
4.3.6 Aliasing . . . 19
4.3.7 Data layout . . . 19
4.3.8 V-cycle . . . 20
4.3.9 Jacobi relaxation . . . 20
4.3.10 Residual . . . 21
4.3.11 Restriction . . . 21
4.3.12 Prolongation . . . 21
4.3.13 Communication . . . 21
4.3.14 Coarser grids . . . 22

**5 Results** **23**
5.1 Overall . . . 23
5.1.1 Local storage usage . . . 23
5.1.2 Time usage . . . 23
5.2 Kernel . . . 24
5.2.1 Jacobi . . . 24
5.2.2 Residual . . . 24
5.2.3 Restriction . . . 24
5.2.4 Prolongation . . . 25
5.2.5 Overall kernels . . . 25
5.3 Communication . . . 25

**6 A possible improvement** **27**

**7 Conclusion** **27**

# List of Figures


1 The elements of the Cell processor . . . . . 8
2 Communication steps to send a message through the TNW network . . . . . 11
3 Round trip time for different message sizes with one SPU communicating via one link 12
4 Round trip time for different number of messages sent in parallel . . . . . 12
5 Round trip time for more parallel messages . . . . . 13
6 Round trip time for different sizes with four SPUs communicating on four links . . . 13
7 Restriction and prolongation operation in 2D . . . . . 16
8 Domain splitting . . . . . 18
9 The shuffle operations . . . . . 19
10 Restriction and Prolongation operation in 2D on four SPU cores . . . . . 22
11 Restriction on coarse grids . . . . . 22

# 1 Introduction

The development of a processor is very expensive and with the complexity of modern processors the costs are increasing. The costs for developing a processor for mass market refinance through high margins. Processors designed for supercomputers do not have this margins. So the development costs per produced processor are much higher. That makes supercomputers with custom processors rare. Even if a single special designed processor has a greater performance, many general purpose processors for the same money performs most likely better. That is way special processors are vanishing from the supercomputer market and processors originally designed for desktop computers form the majority in the Top500 list. Even processors designed not only for the fastest supercomputers, but for wider market of high performance computing like the Intel Itanium have a hard stand against them.

A bit different is the market for custom interconnect solutions. For many applications on clusters the bottleneck is the interconnect. Especially latencies are a problem. They cannot be reduced by just using more hardware. That is why Infiniband is used in one third of the clusters in the top500 list, although the price for this network is a multiple of the price of a Gigabit Ethernet network. However the fraction of systems with Gigabit Ethernet is still higher than one half. But there are also 37 systems with completely proprietary interconnects.

An approach to make custom integrated circuits development and small series production cheaper are the use of Field Programmable Gate Arrays (FPGA). FPGAs are reconfigurable hardware. It is possible to program and reprogram the logic of this chips. So the same chip can be used for various application areas. That increases the margins and lowers the prices. FPGAs have also a faster and cheaper development cycle than Application-Specific Integrated Circuit (ASIC) development, because after fixing errors the chip can be reprogrammed without producing a hole new chip. For the QPACE cluster this FPGAs were used to realize a custom interconnect, which should satisfy special application requirements. This requirements leads to the design of a 3D torus network.

In this work it should be investigated, if this cluster can also be used for other applications than it was designed for. This is done on the example of a multigrid solver. Multigrid algorithms can be used in a wide area of applications.

In chapter 2 and 3 an introduction in the QPACE hardware is given. A brief overview about the motivation for building this cluster reason the special hardware design. An important part of this chepter is section 3.3.2, where the performance of the network is analysed. It is shown, that the network bounds for this network cannot be described by just latency and bandwidth. A better description and bounds are introduced.

Chapter 4 starts with a short introduction to multigrid and introduces the model problem. The main part of this chapter describes the implementation of the multigrid algorithm. Performance optimization techniques are introduced and explained in example of the multigrid algorithm. This optimizations techniques works on many other hardware, especially on other Cell systems this optimizations work in the same way.

Chapter 5 presents the results of the optimization. The optimized kernels are compared with naive versions and measured runtime are compared with calculations for this times, which were done with the help of the assembler code. It is shown that optimizations on the Cell processor have a high impact on the performance and that runtime on the Cell is very predictable. Another aspect, which is analysed in this chapter is the network performance. It can be shown, that estimates of communication time, done with help of the bounds introduced earlier, are very precise.

Chapter 6 presents a way in wich the network performance could be improved by keeping the network properties striktly in mind.

The conclusion in chapter 7 summarizes the results of this work and gives advices for further software development on the QPACE.

# 2 QPACE project

## 2.1 QCD

Quantum chromodynamics (QCD) is a theory of strong interaction in quantum physics. One approach to QCD is lattice QCD, a discretization of QCD on lattice points. The aim of QPACE

(QCD Parallel Computing on the Cell Broadband Engine) is to develop a hardware, which is suitable to calculate lattice QCD problems. The structure of the lattice QCD problems results in some requirements on the hardware.

Beside the requirement for high computational power, some properties of the network are important. This properties are discused in [Ple07]. Most important is a low latency for small messages. A high bandwidth for large messages is not that important. Nearest neighbour communication in four dimensions is sufficient.

### 2.2 eQPACE

The Partnership for Advanced Computing in Europe (PRACE) is an initiative to boost HPC computing in the EU. One field, in which PRACE is active, is testing and development of new hardware architectures. The eQPACE (extended QPACE) is a PRACE project, which addresses the evaluation of the QPACE hardware for more general or other problems than lattice QCD.

One part of this project was the implementation of the linpack benchmark. The result of this implementation was a 110th place in the 34th top500 list. Because of great energy efficiency this also means the first place of the green500 list[1].

This work, in which the implementation of a multigrid algorithm on the QPACE hardware is studied, is also part of the eQPACE project.

## 3 QPACE architecture

### 3.1 Cell

The processor used in QPACE is the PowerXCell 8i from IBM. It's the latest and last generation of processors based on the Cell Broadband Engine. The Cell Broadband Engine was developed by IBM, Toshiba and Sony. Processors with Cell Broadband Engine are multi-core processors with heterogeneous cores. The Cell processors consists of one PPE core, which is a Power based core, and eight SPE cores, which are cores specialized for SIMD floating point operations. The first generation Cell cores were optimized for single precision calculations. With the 8i IBM introduced a variant, which also can handle double precision calculations well. It is clocked at 3.2 GHz and can do up to 102.4 GFLOPS with double precision calculations. One Cell prozessor consists of many elements. A picture of them are shown in figure 1. A description of this elements follows.

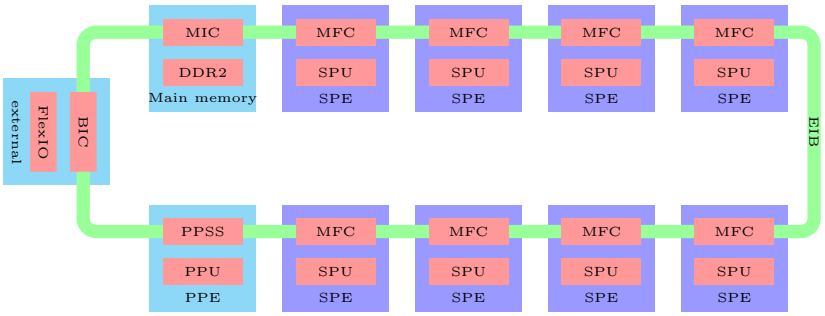


Figure 1: The elements of the Cell processor

#### 3.1.1 EIB

The central Bus, which connects all elements of the Cell processor is the Element Interconnect Bus (EIB). It is organised as four rings with unidirectional connections. With every bus clock cycle one connection can transport three 16 Byte packets. The bus clock runs at half of the system clock

[1] November 2009 list

rate. The component connecting the main memory with the EIB bus is called Memory Interface Controler (MIC).

External hardware can be connected to the EIB bus by FlexIO. The connection between EIB and FlexIO is called Broadband Interface Controler (BIC). FlexIO is a processorbus designed by Rambus.

#### 3.1.2 SPE

One part of the Synergistic Processor Elements (SPEs) is the Synergistic Processing Unit (SPU). The SPU is a SIMD processor with 256 KiB Local Storage. The SPUs are designed for handling the computational tasks. They have two completely static pipelines and no transparent caches. There is no direct access to main memory, only to the local storage. This memory is shared for data and program code.

The two pipelines can start the execution of instructions in parallel. The instructions set is devided in two halves. One half can be executed on one pipeline, the other half on the other pipeline. One pipeline is called even pipeline. On this pipeline arithmetic instructions are executed. On the odd pipeline load, store, shuffle and branch instructions are executed. Both piplines can start the execution of one instruction every processor cycle. The latencies until the result of the instruction is valid in the destination register vary by different instructions.

The speed of the access to the local storage is comparable to cache access on other platforms (six clock cycles latency for load/store). As no cache misses can occur, this time does not vary. The instruction set of the SPUs consists mainly of SIMD instructions. The latency for double precision instructions is nine cycles.

To access main memory the SPEs have a DMA engine. This is called Memory Flow Controller (MFC) and is connected to the EIB. With this engine data can be copied from main memory to local storage and vice versa. SPUs also communicate with each other over this channel.

#### 3.1.3 PPE

The Power Processor Elements (PPE) consists of a Power core, 64 KiB L1 and 512 KiB L2 cache. The Power core and the L1 cache forms the Power Processor Unit (PPU). The Power Processor Storage Subsystem (PPSS) is composed of the L2 cache and the connector to the EIB.

The code for the SPUs is copied from memory to the local storage of the SPUs by the PPE. The PPE also starts the SPEs. The main function of the PPE is to manage the SPEs. Another task for the PPE is to run the operating system. All system calls must be executed from the PPE, because the SPUs lag of functionality to run an operating system like privilege mode.

### 3.2 The cluster structure

A full overview of the hardware is given at [Ple09]. The Cluster has a hierarchical structure. The smallest part is the node card. One node card consists of one cell processor and a network processor. There are also 4 GiB main memory available on every node card.

32 node cards are connected to one backplane. A rack consists of eight backplanes.

There are also service cards. These service cards handle a global clock signal and service functions for the node cards. They can start and stop node cards and write FPGA images. This network of service cards consists of two root cards per backplane and two super root cards per rack.

Both installations of the QPACE hardware in Jülich and in Wuppertal consist of four racks.

### 3.3 Torus network

#### 3.3.1 Hardware

A full description of the torus network is given in [MP09]. To achieve the network requirements a new proprietary network was developed. The network structure only needs to ensure communication between neighbouring nodes. The topology exactly fitting this requirement is a torus network. It was decided that a proprietary network processor (NWP) had been designed.

As the SPUs cannot execute system functions, on conventional Cell clusters the PPEs have to handle communication. This leads to a PPE centric programming model, in which the SPUs are

used as accelerators. So there is an extra overhead when copying data from one SPU to a SPU on another Cell processor. When one wants to program SPU centric on this clusters, he has to make callbacks on the PPE to execute the system calls. This context change from the SPU to the PPE and back is expensive. Both variants mean a too high overhead, when only small messages are sent. To avoid this a SPU centric programming model, with the SPU being able to send and receive data without interacting with the PPE, would be preferable. To make this possible, the SPUs communicate with the NWP over DMA transfers.

This network processor (NWP) is realized with a FPGA, which lowers the development cost compared to custom ASIC hardware. Another benefit of the FPGA is the possibility to extent the functionality, if needed. For example there exists an FPGA image, which makes communication between PPUs possible. The FPGA used is the Xilinx Virtex-5 LX110T. The FPGA is connected to the EIB bus of the Cell processor through FlexIO. The theoretical the bandwidth of this channel is 48 GBit/s. For the link between the FPGAs 10 GBit/s PHYs are used.

When receiving a message from a remote host, the NWP must be able to route messages to the right SPU. To be able to do this virtual channels are introduced. There are eight virtual channels per link availible. The virtual channel number is attached to every message to make the routing possible.

The way of a message goes from one SPU over the EIB to the NWP then over the link to the remote NWP and again over DMA to the destination SPU. For the different steps different communication protocols are used. On the DMA part a credit based communication is used on the link between the NWPs a feedback communication.

Both protocols handle the flow control of the communication. With feedback communication the sender sends a data package and the receiver sends back a feedback. Either that he was able to receive the data (ACK) or that he was not able to receive the data (NACK) e.g. when the receiver has no free memory availible. When receiving a NACK or not receiving a ACK in a given timeout the sender resends the same massage. This can be done until a ACK for this packet is received or stopped after a defined number of retries.

The credit based communication works different: Here the receiver starts the communication. He informs the sender that he is ready to receive some amount of data. He gives a credit of a defined to the sender. Then the sender sends the data. After completion of the message transfer the sender notifies the receiver about that. So the receiver must know when he will receive data and how much data he will receive.

Figure 2 is a timeline chart of communicating one message over one TNW link. The message is split in several data packages.

When a message arrived at the destination NWP before a credit was given by one receiving SPU, the NWP caches the message to delivere it later. One problem that can accure is that the NWP runs out of memory. Then it rejects all data packets. Also the data packets containing the credit. So the NWP is deadlocked in this situation.

One can be sure to avoid this situation, when one send only messages smaller than 2 KiB or by ensuring that the receiving SPU has given credit before sending bigger messages. Due to limitations of the Cell DMA engine the biggest possible message to send is 16 KiB. For the same reason the message size must be a multible of 128 Byte.

Some PHY have alternative links that are used to configure the cluster in smaller partitions. The alternative link closes the circle of the torus in one dimension instead of connecting the neighbouring one. In that way the cluster can be configured into [4, 8, 16]x[4, 8]x[4, 8] partitions. It is also possible to have one or two nodes in one dimension, but then the redundant link is not availible.

Applications with a requirement of a four dimensional torus use the eight SPUs in one Cell processor to form another dimension.

The goals of the network development for the parameters of the network were to use the bandwidth of the PHYs and reach a latency around 1 $\mu$s.

### 3.3.2 Measurements

During this work it turned out, that the network performance must be described more accurately than with latency and bandwidth. To find other parameters the round trip time of a ping pong with various parameters were measured. To make this measurements a central Cell processor is chosen. On this node a program that sends a message to neighbouring nodes and measures the time until

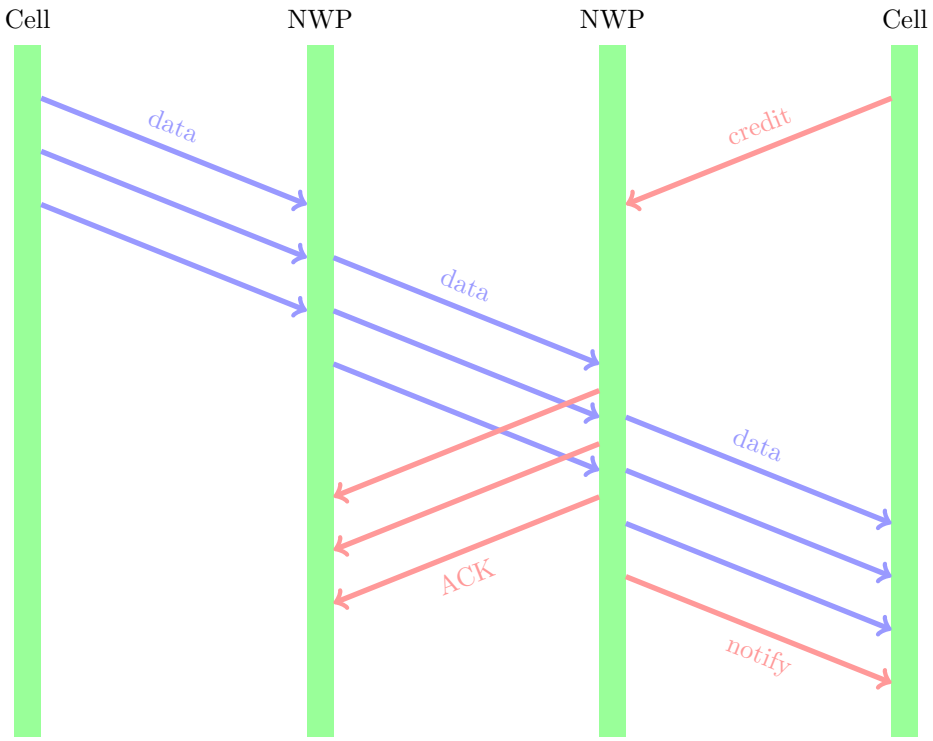


Figure 2: Communication steps to send a message through the TNW network

the message is answered. The neighbouring processors run a program that answers every incoming message with an outgoing one with same size.

The number of SPUs used on the processors can be changed. All SPUs run the same program communicating in all desired directions. The SPUs and the different processors are synchronised before one run but never while the test is running. Another parameters that can be changed is the number of neighbours, to which the central processor should communicate. The message size and the number of runs can also be changed. The number of runs are set to one thousand to smooth the variability of the network. When using multiple links or SPUs the loops for different links/SPUs are only synchronised before starting the measurement. So the influence of synchronisation delay is low.

Measurements for one link like in figure 3 seams to indicate that the design goals were nearly reached. The time needed for one round trip increases approximately linear. The offset of 5.9 $\mu$s equals to the double latency of 2.95 $\mu$s. The slope of the graph is about 2.14 ns/Byte. Considering that the ping pong serializes the communicating and that two messages are sent in one ping-pong, this equals a bandwidth of 891 MiB/s

When increasing the number of messages sent in parallel by increasing the number of links or the number of communicating SPUs slightly the round trip time nearly stays constant. That can be seen in figure 4. But this nice behaviour only occurs with low load.

The increase of time on figure 5 is nearly linear to the number of sent messages. Independent on how much links or SPUs this messages were produced and for small message sizes independend of the size of the messages. This is another bound of the network. The slope of this line is about 5.35 $\mu$s/message. The NWP can only handle 0.75 messages per $\mu$s for some reason.

The increase of the time for the 2 KiB messages can be explained by another bound. The NWP has an overall bandwidth bound. This was investigated with bigger message sizes. The results are shown in figure 6. This is again nearly a linear function. The slope is 13.5 ns/Byte. Considering that 16 messages are sent in parallel this equals to $\frac{16}{13.5\ \text{ns/Byte}} = 1130$ MiB/s

So to analyse the communication of a program and how this would work on the QPACE architecture one has to consider three bounds:

- Latency and bandwidth bound for every link: 2.95 $\mu$s and 891 MiB/s

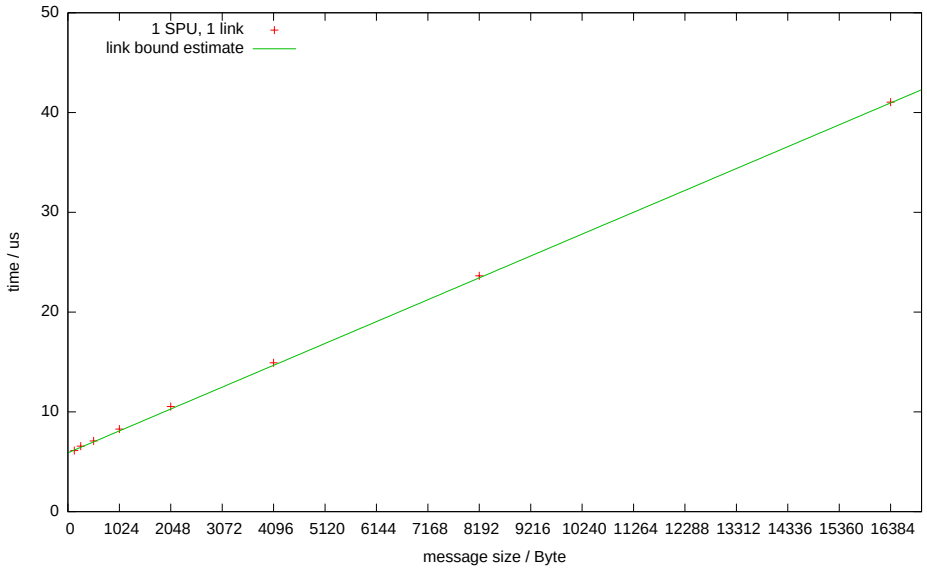


Figure 3: Round trip time for different message sizes with one SPU communicating via one link

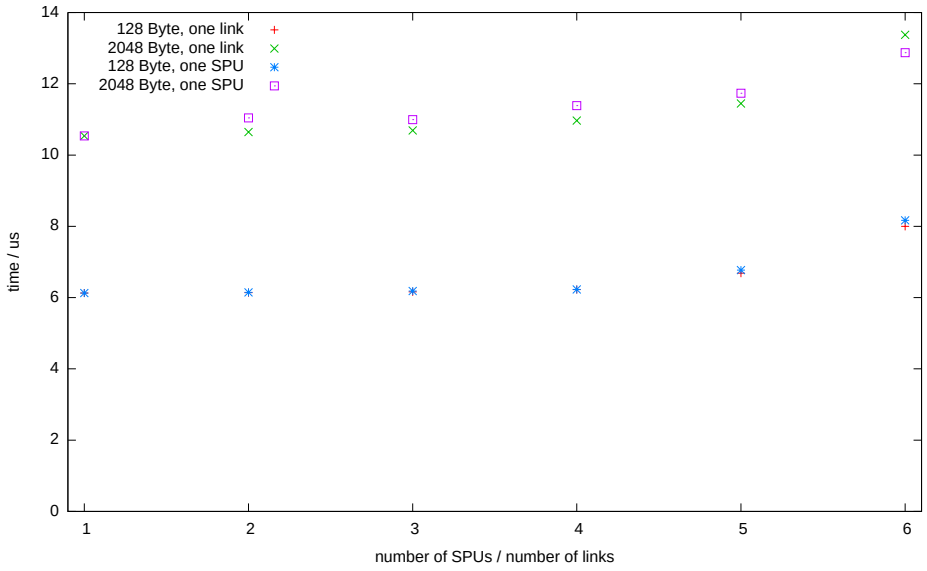


Figure 4: Round trip time for different number of messages sent in parallel

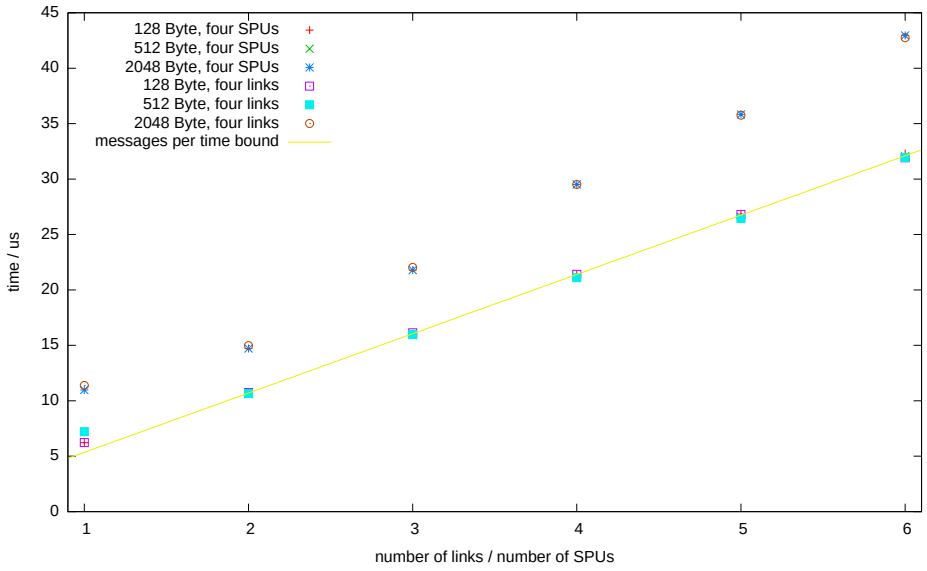


Figure 5: Round trip time for more parallel messages

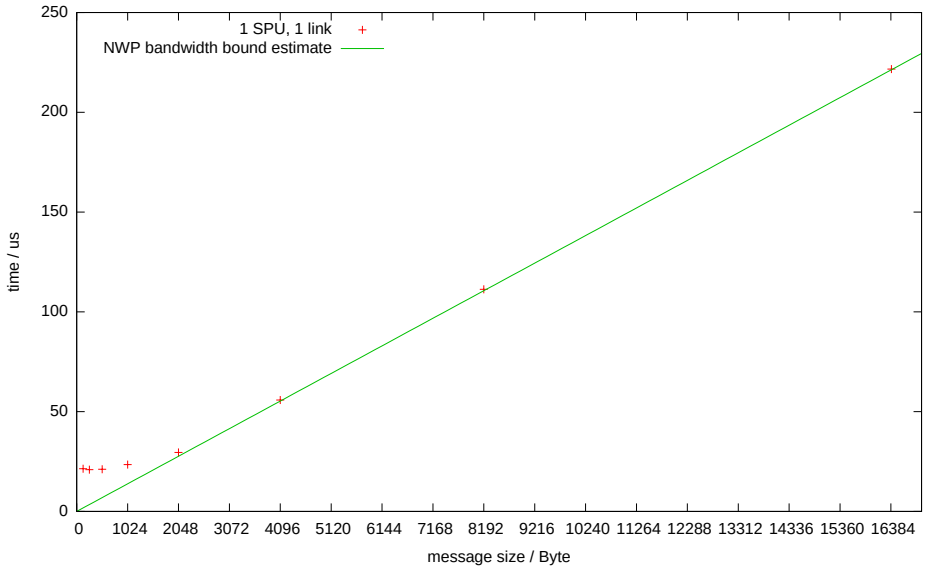


Figure 6: Round trip time for different sizes with four SPUs communicating on four links

- Bound for the sum of the bandwith for all links of one Cell processor: 1130 MiB/s
- Bound for the frequency of sent messages by all SPUs of one Cell processor: 0.75 messages/$\mu$s

This numbers are measurements of the currently used firmware for the NWP. The performance may increase in some aspects in future releases[2].

## 3.4 Software environment

### 3.4.1 System

The operation system used on the nodes is a Linux system. All special drivers are included in the vanilla kernel. The distribution used is Fedora. So many standard tools are available. The system can run code compiled for the Cell processor.

### 3.4.2 TNW library

To access the network a special library was written. This library and the standard FPGA firmware is designed for a SPU centric programming. So the communication functions are called on the SPUs. No context change to the PPU is necessary.

The TNW library provides the SPU functions to use the TNW network. The functions implements the hardware interface of the NWP. The full documentation of the API is [NP10]. The most important communication functions should be introduced here.

`int tnw_init()` is an initialization function, which must be called by the PPE.

`void tnw_init(id)` is the initialisation function called on the SPUs. The parameter `id` is the thread id, which is handed over to the program as `main` parameter.

`void tnw_finalize()` is a SPU function, which must be called for some cleanups.

`void tnw_put(phy, channel, txbuf, offset, size, dmatag)` sends a message to remote host. `phy` is the physical link and `channel` the virtual channel, which should be used. `txbuf` is a pointer to the data to be send. `offset` is an offset, which is added on the receiver side to the credit base address. This offset can be used to make several `tnw_put` calls to serve one credit. `size` is the size of the message. `dmatag` is a DMA tag, which is used for the DMA transfer to the NWP. The application can use it for waiting until the send buffer is available again.

`void tnw_wait_put(dmatag)` This function waits until the DMA transfer to the NWP of the `tnw_put` with the `dmatag` is finished.

`unsigned int tnw_test_put(dmatag)` returns one, when the DMA transfer of the according `tnw_put` is finished.

`tnw_rw_t` is a data type, which is used to handle connections for receiving data. A variable of this type has alignment requirements. To fulfill them the variable must be defined globally, or the memory must be reserved with `tnw_rx_alloc`.

`tnw_rx_t * tnw_rx_alloc()` reserves memory space for a `tnw_rx_t` variable with attention to memory alignment.

`void tnw_rx_free(tnw_rx_free()` is the function to free the memory previously reserved with `tnw_rx_alloc`.

`void tnw_credit_base(phy, channel, addr)` is the function that assigns the receiving buffer at address `addr` to the virtual channel `channel` of the link `phy`.

`void tnw_init_rx(rx, phy, channel)` assigns the `rx` handle to the virtual channel `channel` of link `phy`

[2]FPGA version v00000523-200, TNW lib version 0.8.2

`void tnw_credit(rx, addr, size, msgtag)` is the function, wich gives a credit for the channel associated with `rx` of size `size`. When receiving data the data is written to the address given by the `tnw_credit_base` call plus `addr` plus the offset used in the corresponding `tnw_put` call. The `msgtag` is used to be able to differ between different credits, when waiting for the notification.

**void tnw_wait_notify(rx, msgtag)** waits until the notification for the credit associated to the `msgtag` of the channel assigned to `rx` is written.

**unsigned int tnw_test_notify(rx, msgtag)** returns 1, if the notification for the credit associated to the `msgtag` of the channel assigned to `rx` is written, otherwise 0.

# 4 Multigrid on eQPACE

## 4.1 Multigrid

Multigrid is a very interesting field of research and has many aspects, which should not be discussed here. A good introduction to this is given in [BHM00]. Here only a few aspects, which are necessary to understand the implementation, should be mentioned. Multigrid methods work optimal for linear systems rising from discretization of elliptic partial differential equations. They are an improvement of iterative methods like the Jacaobi method for solving systems of linear equations. An observation on Jacobi iterations is that the methods can reduce error components of high frequency in few iterations, but needs many iterations for error components of low frequency components. Whith a discretization of the problem on a coarser grid, the Jacobi method cannot improve on the high frequent error components, because of the error of the discretization, but the convergence of the low frequent errors is better. Multigrid methods try to utilize this.

The idea is to iterate on a fine grid to reduce the high frequent error component and switch to a coarser grid to reduce the lower frequent error components. This is not done by iterating directly on the solution, but by solving the residual equation. The linear equation looks like:

$$Ax = b$$

The residual for an estimated solution $x'$ is:

$$r = b - Ax'$$

For the error $e$

$$e = x - x'$$

the equation

$$Ae = r$$

and

$$A(x' + e) = b$$

holds.

So with a solution for $e$, the whole system can be solved by adding $e$ on the estimate $x'$. This solution is calculated on a coarser grid. To be able to do this the Matrix $A$ and the vectors must be transfered between the finer grid and the coarser grid. The size of the grid in every dimension is halved. When solving PDEs the Matrix $A$ is a discretization of the differential operator. There also exists a discretization for the coarser grid. The residual is transfered from the finer to the coarser grid with an operation called restriction. This operation maps the vector to a lower dimensional space. The simplest way is just using the data of every second grid point, this is called injection. Often a more complicated operation, which involves the neighbouring data values is used. The operation, which interpolates the solution from the coarser grid to the finer one, is called prolongation. Summing up this interpolated vector to the existing solution on the finer grid is called correction. In most cases the restriction operator used is the transpose of the chosen interpolation of the prolongation.

## 4.2 Model problem

The problem to be solved for this implementation is the Poisson problem with periodic boundary conditions. The problem is solved on a regular three dimensional grid. The Poisson problem:

$$\Delta u = f$$

The discretization of the Laplace operator results in a seven point stencil:

$$\frac{1}{6h^2}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -6 & 1 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

For the restriction and the interpolation a cell based approach is chosen. That means that every grid point is seen as a cell. The transition to a coarser grid is the fusion of eight ($2^3$) cells to one. The restriction operator calculates the mean of this eight points. The prolongation just copies the value of the one coarse grid to the eight fine grid cells. This can be expressed by the following stencil:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

The figure 7 visualizes the restriction and the prolongation in 2D.

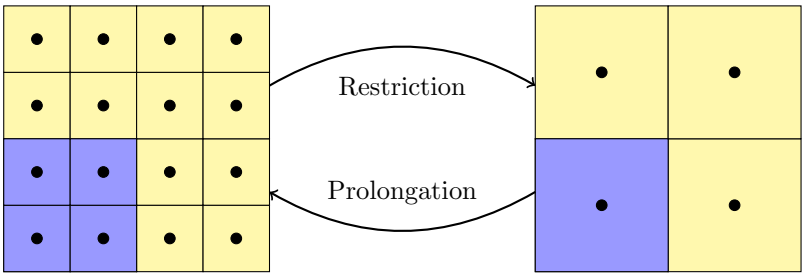


Figure 7: Restriction and prolongation operation in 2D

A Jacobi smoother is used. When using Jacobi as smoother in a multigrid, the damped version must be used. A factor of $\frac{2}{3}$ is optimal for the Poisson problem in two dimensions.

The multigrid cycle used for solving this problem is the so-called V-cycle. The procedure of the V-cycle is the following:

- Apply smoothing iterations ("pre-smoothing")
- Residual calculation and restriction
- Recursive call of this procedure for coarser grid
- Prolongation and correction
- Apply smoothing iterations ("post-smoothing")

The recursive calls are done until the grid size is $2^3$. This grid is smoothed with some Jacobi iterations. In this implementation two pre-smoothing iterations and two post-smoothing iterations are chosen.

The number of grid points is limited by the local store of the SPUs. No main memory should be used. This results in a domain size of $16^3$ for every SPU. The standard cluster size for measurements in this work is $4^3$ node cards. This means an overall grid size of $128^3$.

## 4.3 Implementation

### 4.3.1 Communication library

The TNW library has only functions for communication between SPUs of different Cell processors. So the communication between SPUs on the same processors has to use DMA transfers directly. To

have a single interface for both communication ways a new communication library was written. With this library the torus build up by the network infrastructure between the processors is extended to a virtual torus with links between SPUs. To do this the topology of the eight SPUs of one processor must be chosen. Any four dimensional rectangular grid with exactly eight points would work. This grid is embedded in the cluster's torus grid. After defining the topology one can use functions for sending and receiving in the two directions of each of the four dimensions.

The communication between SPUs on the same processor is also implemented as credit based communication, so that the same interface can be used. A communication starts with the receiver giving a credit. The receiver saves the information of the address of the receiving buffer and the address for the notify on the remote SPU. Then the receiver increases a pointer to the last valid credit for this connection on the remote SPU. The sender waits until he has a new valid credit and sends his message to the given address. After the message was written, he writes the notify to the given address. The order of the writes of the message and the notification can be ensured by the DMA engine. As no buffers are available messages cannot be cached. The sender is blocked while waiting for a credit. The receiver can check if a message arrived by reading the notify buffer.

For the communication between two SPUs, which are on different Cell processors, a mapping from direction to the right virtual channel is done. The configuration used for the multigrid implementation is shown in figure 8.

#### 4.3.2 Domain splitting

An important question in cluster computing is always how to distribute the data and the calculation workload. With this problem it it easy to answer this question, as the network has a similar topology to the problem domain. So there is a natural relation between the domain and the computational nodes. To make things even simpler, the problem is limited to domain sizes of $(2^n)^3$ and the partitions of the cluster to sizes of $(2^m)^3$. The SPUs of one cluster node are configured as $2^3$ cube. So that the virtual cluster has a size of $(2^{(m+1)})^3$. A part of the cluster with network links and the data distribution is shown in figure 8.

#### 4.3.3 Vectorisation

The SPU is designed for SIMD instructions. The arithmetic instructions all operate on two element vectors for double precision instructions and on four element vectors for single precision instruction. So calculating on one value in double precision would be punished at least by a factor of two. In fact the drawback is even higher, because the memory position of the single double value is interpreted as a double vector. To ensure that the second value is not modified by a floating point operation, the vector must be copied and combined with the result before it can be stored again. So one optimization goal on SIMD hardware is to use SIMD operations whenever possible.

#### 4.3.4 Shuffle operation

One problem of handling SIMD operations on the Cell is that the SIMD vectors, which consists of the two double values can only be loaded from addresses, which are multiples of the vector size (16 Byte). So every double value in memory are naturally a first or a second entry of a vector. A special SIMD operation frequently used to manage this problem is the shuffle operation. This operation is used to load two values from two different vectors into one vector. The shuffle operation takes three 16 Byte operands. All operands are interpreted as Byte arrays with 16 entries. Every entry of the third operand holds an index. This index indicates the value, which should appear in the result array at this position. Indices smaller than 16 refer to the first operand, indices bigger than 16 to the second operand (index minus 16). In this implementation three different operands are used to align data from two double vectors:

- `sh10` A shuffle operation with this operand loads the second value of the first operand into the first position and the first value of the second operand into the second position of the result vector. `c = spu_shuffle(a, b, sh10)` does `c[0] = a[1]; c[1] = b[0];`
- `sh00` `c = spu_shuffle(a, b, sh00)` does `c[0] = a[0]; c[1] = b[0];`
- `sh11` `c = spu_shuffle(a, b, sh11)` does `c[0] = a[1]; c[1] = b[1];`

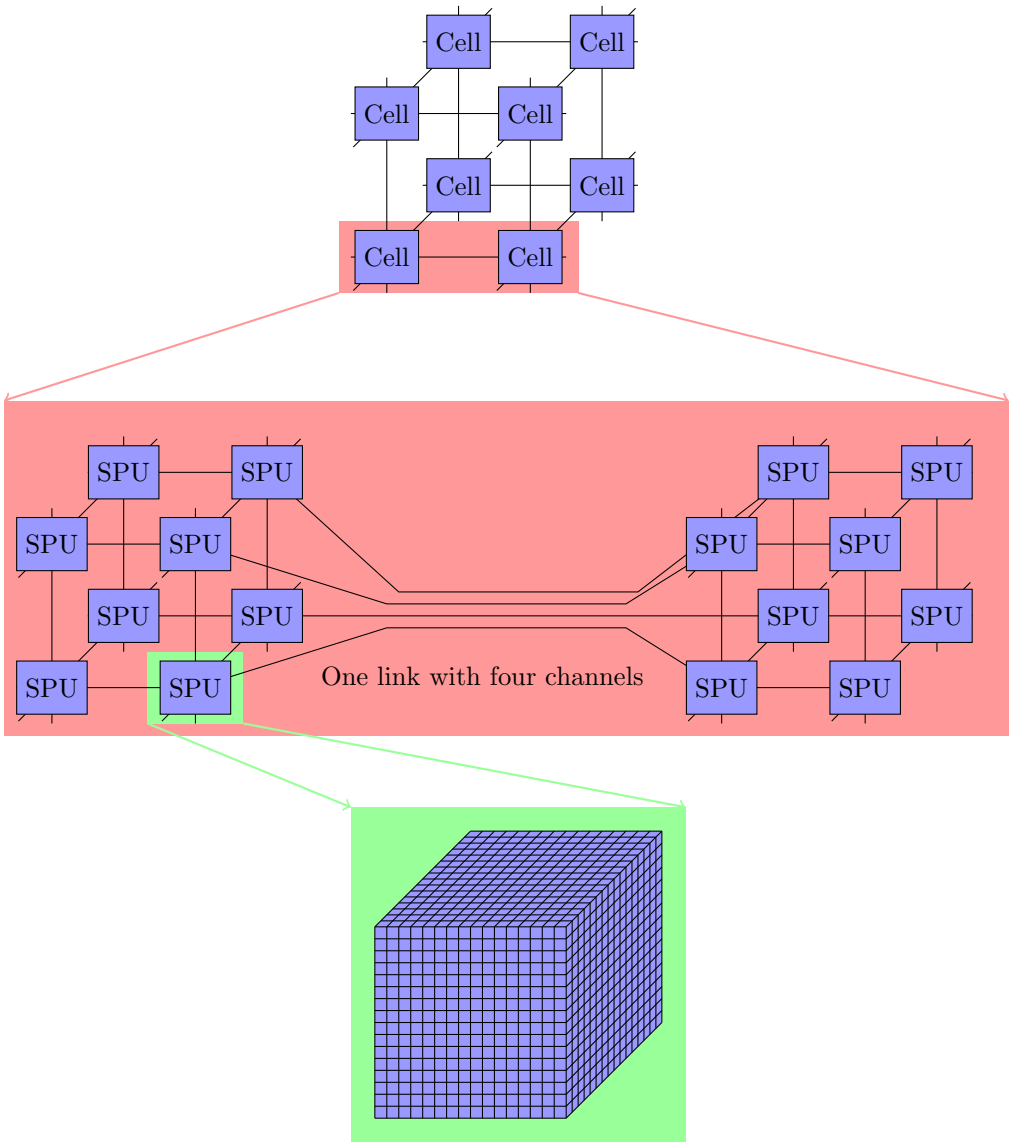


Figure 8: Domain splitting

Figure 9 visualizes this operations.

### 4.3.5 Loop unrolling

Loop unrolling is an optimization technique. Multiple loop passes are written consecutive in one loop. The number of loop passes is adapted. Running some iterations with the unoptimized loop to gain the exact same number of loop iterations might be necessary. On a first look this saves some increments of the increment variable, some comparisons of this variable and some branches.

But on modern processors the benefit can be much higher. To use the pipelining of the processors, it is important to do independent calculations in following instructions. When single loop passes are independent, a simple and good way to achieve this is to calculate multiple loop passes in parallel. Loop unrolling makes this possible. After one instruction of one loop is executed, the same instruction of the next loop pass can be executed. This is not so important on out-order capable processors, because there the hardware tries to solve the data stall problem. But on in-order processors like the Cell this gets very important, because the hardware cannot go forth in the execution pipeline, until all instructions the next instruction depends on are executed.

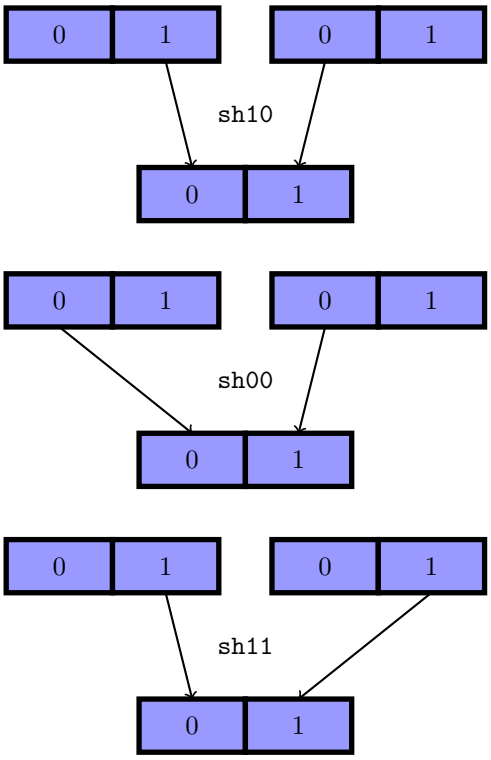


Figure 9: The shuffle operations

One drawback of this technique is bigger binary code, because the operations for one loop exist multiple times. Another problem can be the higher register usage, because of the need of storing more temporary results. With the large register file of 128 entries, this is often tolerable on the SPUs.

Loop unrolling can be done manually or by the compiler.

#### 4.3.6 Aliasing

It is important that the compiler is able to reorder the operations of the different loop runs to benefit from loop unrolling. To do this the compiler must be sure that there is no dependency between loop runs. One problem to foreclose such an dependency are load and store operations. In general every load operation may depend on any previous store operation, because the loaded value might be the stored one. So the compiler needs to use further informations from the source code to ensure that this do not happen, when he wants to exclude a dependency. This results in a problem on normal C code, because data stored in arrays is handled by pointers. When the compiler sees a function with multiple pointer arguments, he cannot foreclose that the arrays indicated by this pointers overlap at some point. The fact that one may have more variable names for one memory address is called aliasing. When aliasing is possible the compiler must assume a dependency and cannot reorder loads before stores, in which pointers are involved. To give the compiler the information of arrays being independent the `restrict` keyword was introduced. For a pointer with this keyword it must be ensured, that the whole array indicated by this pointer is addressed only by this variable.

#### 4.3.7 Data layout

A Jacobi relaxation needs three data arrays. One for the right hand side. One for the next solution and one for the current solution. The arrays for the solutions swap function in each iteration, so one does not need to copy them. In the multigrid situation one of the solution arrays and the right hand side array must be persistent, while iterating the coarser grids, because the data is needed for the post-smoothing.

To save space the data of the coarser grids is saved in the temporal area of the finer grids.

This makes the access to the memory more complicated than normal arrays. To handle this access functions are written. This functions have to do some calculations, which are to expensive to be done for every data access. To avoid this expensive calculations addresses in the kernels are calculated incremental. The kernels work in simple loops on the data, so this calculations are straight forward. The addresses for the stencil have to be calculated relative to the center point.

An important issue with the data layout is the alignment of the data for the SIMD operations. Every line of data should start with a new double vector. To assure this the array is increased by one in x dimension, when the size in x direction would otherwise be odd.

#### 4.3.8 V-cycle

The rough structure of the program is like described in section 4.2:

- Apply smoothing iterations ("pre-smoothing")
- Residual calculation and restriction
- Recursive call of this procedure for coarser grid
- Prolongation and correction
- Apply smoothing iterations ("post-smoothing")

Additionally functions for communication must be inserted. Functions that needs values of grid points, which lies in the local storage of neighbouring SPUs, are the Jacobi iteration and the residual calculation. They need for every grid point the values of the direct neighbouring points. So before calling them the boundary data must be exchanged. That is not necessary for the first pre-smoothing step on every grid level, because every grid level starts with 0 as starting solution. With two pre-smoothing, two post-smoothing and one residual calculation this makes four boundary exchanges for every grid level.

#### 4.3.9 Jacobi relaxation

The most important operation to optimize is the Jacobi relaxation, because it needs the most operation and it is called four times on every level. It calculates the following for every grid point:

$$u'(x,y,z) = \tfrac{1}{3}u(x,y,z) + \tfrac{2}{3}\tfrac{1}{6}(h^2 f(x,y,z) \\ +u(x,y,z-1) + u(x,y,z+1) + u(x,y-1,z) + u(x,y+1,z) + u(x-1,y,z) + u(x+1,y,z))$$

$h$ is the grid spacing, $u'$ the new solution, $u$ the current one and $f$ the right hand side. This operation locally needs the data of the grid point, its right hand side and the two neighbouring grid points in each dimension. The data of the grid points, which belongs to neighbouring points in y or z direction, is saved properly aligned so it can just be loaded. The data neighbouring in x direction is not naturally aligned. To get this data in the vector registers shuffle operations are used. The left side is a `sh01` shuffle of the previous data vector and the current one, the right side a `sh01` shuffle with the current and the next data vector.

Single stencil operation are independent of the previous ones. So a loop unrolling may help to fill the pipeline better. The stencil operates on arrays, so aliasing is a problem for the compiler. In this operation the two solution arrays can be seen as six independent arrays. The five input arrays are two for the stencil points in y, two for stencil points in z direction and one for the center points and stencil points in x direction. The output is written to another array.

As described previously in section 4.3.7 the array structure is so complicated that addresses for array elements have to be calculated by an extra function. So the `restrict` keyword cannot be used to inform the compiler about the independence of the arrays. This could be fixed by calculating the start of each line by this functions and utilize pointers for the relative addressing within the lines. But the SPU version of the gcc 4.3.2 is not able to use the information given by `restrict` and doesn't reorder any load before a store.

So a manual unrolling with a reordering by hand is used. This makes the source code hard to read, debug and maintain. A working automatic optimization as intended would be helpful at this point. To make the code a bit better readable the operation of one loop is extracted into one macro.

To avoid extra functions, which handle the grid points next to boundary points, the boundary points are copied into the solution array. This costs extra memory and some performance. But extra functions to handle the near boundary points would cost more memory and the memory on local storage is very limited.

In x direction the boundary points are saved in a vector together with an interior point. The Jacobi iteration must be calculated for the interior point. Because it's too difficult to separate the two values of one vector, the Jacobi is also evaluated on the boundary point. For the data points, which are not available on the SPU, some arbitrary data is used. So the saved value is invalid. But it is never read before overwritten in the next boundary exchange.

#### 4.3.10 Residual

This operation is almost the same as the Jacobi iteration, so the same optimizations are done here.

#### 4.3.11 Restriction

In this operation data of $2^3$ points have to be summed. The vectorisation is done in a way that the result is a vector that can be saved to the destination array without needing further alignment operation. So the sum for two values on the coarser grid has to be calculated at once. In the first step the sums in z direction are calculated, in the second step the sums in the y direction. The sum instruction can directly use the loaded values, because the data layout ensures right alignment of the vectors. The two sums are calculated for two vectors, which are consecutive in x direction. The last sum in x direction is more difficult as we have to calculated the "horizontal sum" of a vector. To do this sum we have to use shuffle operations again. Two temporal registers are used. The first one is set to the result of a `sh00` shuffle operation, the second one to the result of a `sh11` operation. For both operations the first and the second operant is the same, the sum of the both other dimensions. The final result is the sum of the two vectors.

Similar unrolling and reordering of the store operations as in the Jacobi operator is used to fill the pipeline denser.

#### 4.3.12 Prolongation

Prolongation and correction is done in one function. Here the vectorisation is done by means of the data layout of the source array. For the vectorisation it must be ensured, that all operations, which depend on one source vector are done at the same loop pass. The data of one source value has to be summed to $2^3$ destination registers. The splitting of the source data to two registers holding the same value in both entries is done by two shuffling operations. The first value is extracted by a `sh00` shuffle of the same source register. The second value with a `sh11` operation. These results have to be added to the old solution on the finer grid.

With the vectorisation data of $2^3$ registers are calculated in one loop pass. Additional loop unrolling is not necessary, because the calculation for this eight registers is independent enough.

#### 4.3.13 Communication

For some of the operations data of the neighbouring nodes is needed. That are the Jacobi iteration and the residual calculation. So before this calculations one has to exchange data with the neighbouring nodes. The maximum size of one message on the finest grid is 2 KiB ($16^2 \cdot 8$ Byte). So there is no problem with the message size limit.

There is the possibility to calculate the inner points, while exchanging the boundary points. But this leads to extra functions for the calculation of the near boundary points. As the memory layouts of the boundary points differs for every dimensions, this means extra functions for the boundary layer, the boundary edges and the boundary corners. With extra versions for the various grid sizes, this increases the memory usage of the functions exceeds the local storage. To be able to use the same loop for every grid point, the exchanged data is copied to the right memory locations. This increases the need of memory, because in the other version, the calculation can directly access the communication buffers. But this extra memory use is smaller than the extra use caused by the bigger functions would be. The drawback is that the processors cannot do any calculations while waiting for data.

On coarse enough grids there are less grid points on the global grid than computing SPUs. At this situation the communication gets more complicated. There are three different types of nodes involved in this situation. Nodes holding data, nodes between nodes holding data and nodes not involved at all. Only the nodes holding data have to calculate and the data, they need, are on the next nodes holding data. To get this data from one computing node to another, the nodes between the computing nodes have to forward data. The other nodes have nothing to do, but to wait that the V-cycle gets to a finer grid level again.

Figure 10 shows the data used for restriction/prolongation between a $8^2$ and a $4^2$ grid on $2^2$ nodes. The thicker lines shows the boarders of the domain splitting. For one SPU all cells are colored. One color stands for the data used by one operation. As long as one SPU holds two values in one dimension on the finer grid, no data of neighbouring SPUs is needed.

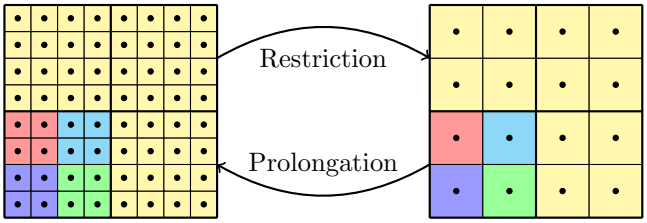


Figure 10: Restriction and Prolongation operation in 2D on four SPU cores

When the grids are so coarse that on the coarse grid one SPU holds data of one grid point, data has to be communicated. For the restriction all data have to be communicated to one of the $2^3$ SPUs. Figure 11 shows the communication steps needed to do that. First one value is communicated along the x direction from nodes with higher x coordinate to the neighbour with smaller x coordinate (blue arrows). Then nodes, which now holds two values and have a higher y coordinate communicate their values in y direction (red arrows). In the last step the node holding four values and having the higher z coordinate communicate its value to the node, holding the other half of the values.

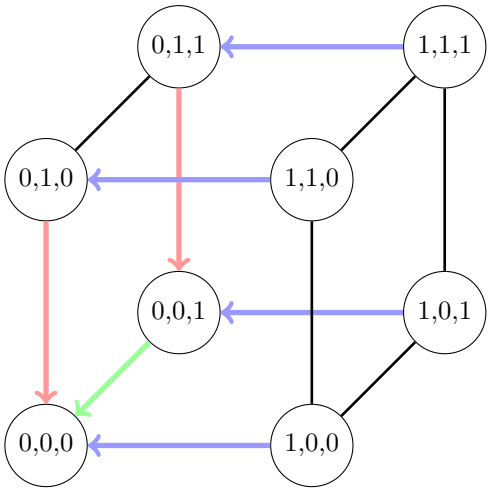


Figure 11: Restriction on coarse grids

The communication for the according restriction does the communication in reverse order and communicating just one value.

When the grids gets even coarser forwarding of messages gets necessary, too.

#### 4.3.14 Coarser grids

Because of the completely unrolled loops the optimized kernels must be compiled for every grid size. This produces more binary code. As local storage is limited, there is not enough space for

having optimized versions for all grid sizes. The grid size reduces with every step by a factor of eight. So the optimizations on finer grids has a much higher impact on the overall runtime. As a consequence the optimized versions of the kernels are only compiled for the two finest grid levels.

# 5 Results

## 5.1 Overall

### 5.1.1 Local storage usage

The local storage of the SPUs is with 256 KiB very limited. The three arrays of size $18^3$ need $3 \cdot 18^3 \cdot 8$Byte = 137 KiB. The send and receive buffers have a size of $16^2$. That are $2 \cdot 6 \cdot 16^2 \cdot 8$ Byte = 24 KiB. The code, libraries and other static data needs another 71 KiB. So there are just 24 KiB left for stack and heap.

The Jacobi iteration and the residual operation have no extra functions to handle the near boundary points. To make this possible the array holding the data is bigger. The memory needed to make this possible is $(18^3 - 16^3) \cdot 8$ Byte = 14 KiB. The space saved by simpler functions is bigger. The Jacobi loop for the finest grid needs 12 KiB. To handle the boundary points with different memory layout several specialized versions of this loop were needed. So the overall memory usage would be bigger and would exceed the 256 KiB available.

### 5.1.2 Time usage

To evaluate the performance time measurements were done. To measure real times the SPU decrementer were used. That is a timer, which counts down with a specific frequency. This timer can be started and read on the SPUs. On QPACE this timer is clocked with 26.666666 MHz.

Static code analysis were done with the `spu_timing` tool of the Cell SDK [IBM08]. This tool calculates the static execution pipeline for assembler code. So one can count how many cycles some part of a code would take. As the behaviour of the SPUs is very predictable this analysis is very precise.

One V-cycle run needs 1050 $\mu$s in the optimized version. One SPU can do about 3.8 million instructions on each pipeline in this time.

To compare this number, the theoretical needed floating point instructions are counted. One Jacobi iteration needs to sum up the neighbouring points (5 adds). Multiplying the right hand side with the square of the grid spacing and adding to the sum can be done in one multiply-add instruction. The sum must be divided by 6 and multiplied by two third of the damping, resulting in a single multiplication with a factor of $\frac{1}{9}$. To get the result one third of the current solution must be added. This is done by one multiply-add again. The whole procedure sums up to eight floating point instruction for one grid point in one iteration.

For the residual instruction again the sum of neighbouring points must be calculated (5 adds). This sum must be subtracted from six times the central point (multiply-add). This result must be multiplied by the multiplicative inverse of the grid spacing and subtracted from the right hand side (multiply-add). So there are seven floating point instructions for every grid point.

The restriction is just a sum of eight grid points. From the view of the finer grid this are $\frac{7}{8}$ floating point instructions per grid point.

The prolongate and correct operation sums up every grid point and the corresponding solution of the coarser grid. That makes one floating point instruction per grid point.

For two pre-smoothing and two post-smoothing steps in each level of the V-cycle this sums up to $40\frac{7}{8}$ instructions per grid point. The grid points per SPU for the whole V-cycle are $16^3 8^3 4^3 2^3 1^3 \frac{1}{2}^3 \frac{1}{4}^3 \frac{1}{8}^3$. In the last level there are no residual, restriction or prolongation calculations, so this level has just 16 floating point instructions per grid point.

All this sums up to 191342 floating point instructions per V-cycle per SPU. That are instructions on single double values. So it could be calculated in 95671 SIMD instructions on a SPU. So using 3.8 mullion instructions for that is about 2.5% of peak performance. As multigrid is often the perfect algorithm, this may still be a satisfying result, but using only such an small part of the possible performance needs further investigations.

## 5.2 Kernel

When counting the grid points, on which calculations are done 98% of the calculations are done on the finest two grids. That is why they are optimized. The further investigations have a focus on this kernels.

### 5.2.1 Jacobi

A simple version of this operation, without any optimizations needs 139 cycles for one update. Much of the time is wasted for the calculation of the address in the array, for the calculation and for applying the shuffle operation, which makes single value calculations with the SIMD instructions possible. The first step of optimization uses a vectorized loop and a more efficient calculation of the addresses. This leads to a loop run calculating two values in 67 cycles. The unrolling of nine loop runs, which is a full line on the finest grid, leads to a 594 cycles loop. This is not a big improvement. Only with the reordering of the stores this loop gets significant smaller to 120 cycles. Table 1 shows the number of cycles for one line of the finest grid. The vectorized versions calculate 18 values. The naive version only need 16 loop runs, because it does not calculate the result for the boundary values.

| | naive | SIMD | loop unroll | reordering |
|---|---|---|---|---|
| cycles | 2502 | 603 | 594 | 91 |

Table 1: cycles needed to calculate one line of iterations

To estimate a theoretical bound for the cycles needed for this calculation the cycles needed on each of the two pipelines must be considered. The floating point instructions done in this loop are executed on the even pipeline. The load, store and shuffle instructions are executed on the odd pipeline. For the even pipeline the numbers for one loop are already calculated. That sums up to $8 \cdot 9 = 72$ instructions. To calculate this loop the five input arrays must be loaded and one array must be stored. That are $6 \cdot 9 = 45$ load and store operations. To get the left and the right vectors shuffle instructions has to be executed. The right value can be reused as left value for the next calculation. So for the first calculation two shuffle instructions has to be executed, for the other ones only one. That are 10 shuffle operations. So the sum for the odd pipeline is 55 cycles. The limit for the even pipeline is bigger.

The loop for ten values needs 91 cycles (40 theoretically).

### 5.2.2 Residual

Due to the same structure of the residual calculation to the Jacobi iteration this loop performs in a similar way at the different optimization steps. The number of cycles for the not optimized version for one update is 92 cycles. The optimized version needs 115 cycles for 18 value calculations and 84 cycles for 10 values. The theoretical bounds are 63 and 35 cycles. So the calculation of the residual is comparable efficient.

### 5.2.3 Restriction

The loop for the restriction operates on the coarser grid size. So one loop run operates on eight grid points on the finest grid. In one run the sum of eight points is calculated. The naive version needs 103 cycles to do this. The optimized version needs 70 cycles to do the same calculation on ten times more points and 58 for six runs. The theoretical best loops here needs to do seven additions in one loop run. That are $5 \cdot 7 = 35$ and $3 \cdot 7 = 21$ floating point instructions. But the odd pipeline has to execute 16 load, two shuffle and two store instructions for the calculation of two vectors. For the last vector eight loads, two shuffles and one store are executed. That are in sum $2 \cdot 20 + 11 = 51$. The loop unrolled to six runs needs $20 + 11 = 31$. So the bottleneck is not the calculation but how to get data fast enough and right aligned from the local store in the registers and back. And the efficiency of this loop is comparable with the other functions.

### 5.2.4 Prolongation

This loops also loops on the coarser grids and makes eight additions. The unoptimized version needs 213 cycles for that. The optimized version needs 32 cycles for calculating on the double number of values. For this operation there do not exist different versions for different grid sizes. This 32 cycles are used to calculate just eight floating point operations. But on the odd pipeline one load, two shuffles and eight store are executed. So the theoretical limit for this calculation is 11 cycles. So the efficiency of this function on the finest grid is slightly worse than the other functions. One reason for this is the less unrolling.

### 5.2.5 Overall kernels

Summing up cycles for the finest level of the V-cycle:

$$4 \cdot 16^2 \cdot 120 + 16^2 \cdot 115 + 8^2 \cdot 70 + 8^2 \cdot 5 \cdot 32 = 167040$$

This should take about 52 $\mu$s. Real measured times are about 53 $\mu$s. The difference is caused by not counted instructions around the innermost loop. But the small difference shows how predictable the runtime on the SPUs is. The same sum for the 2nd finest grid:

$$4 \cdot 8^2 \cdot 91 + 8^2 \cdot 84 + 4^2 \cdot 58 + 4^2 \cdot 3 \cdot 32 = 31136$$

That would be theoretically 9.7 $\mu$s. This is near to the measured time of 10.5 $\mu$s. The reason for the bigger relative difference between the real and the theoretical time is that more time is spent outside the innermost loop. The absolute difference is approximately the same.

The calculations on the other six grids are done with functions completely lagging optimizations. On grids, which only hold data on some SPUs the case that the SPU under investigation has to calculate an update are presumed. This leads to this number of cycles:

$$(4^3 + 2^3 + 1 + 1 + 1 + 1) \cdot (4 \cdot 139 + 92 + 103/8 + 213/8) = 52250$$

That would be 16 $\mu$s but the measured time is 31 $\mu$s. Here the divergence is much bigger, because the innermost loop, which calculates just one value in this case, is a smaller fraction of the overall runtime.

The efficiency for the coarser grids is much smaller than for the finer ones, but the sum of all calculation is just 95 $\mu$s. Which is only a tenth of the real runtime. The difference is the communication, which is not measured here.

## 5.3 Communication

When running just the communication one V-cycle run needs 848 $\mu$s. The difference between the sum of the kernel runtime and the communication time and the overall runtime is caused by additional functions like copying the exchanged data between the buffers.

The analysis should start with the four finest grids. For comparison an estimate based on a latency of 2.95$\mu$s and a bandwidth of 891 MiB/s in each direction is calculated. The number of communication steps for each grid sizes is four. So there are 16 communications for the four finest grids. The message sizes are $16^2 \cdot 8$ Byte = 2 KiB, $8^2 \cdot 8$ Byte = 512 Byte, $4^2 \cdot 8$ Byte = 128 Byte and 128 Byte. The last one would be smaller, but the smallest possible message size is 128 Byte. That sums up to 11 KiB for each SPU in each direction. With four SPUs communicating through one link this are 44 KiB for each direction. Assuming that the latencies for one link do not sum up the time for communicating for the four finest grids should be:

$$16 \cdot 2.95\ \mu\text{s} + \frac{44\ \text{KiB}}{891\ \text{MiB/s}} = 95\ \mu\text{s}$$

But the measured time is with 559 $\mu$s much bigger.

To explain this difference the results of the measurements of section 3.3.2 must be used. First for every grid one must be determine by which bound the communication is limited. On every grid every Cell processor sends four messages in every direction. That are 24 messages for the NWP. With a throughput of .75 messages/$\mu$s that would take 32 $\mu$s.

With a message size of 2KiB on the finest grid the bandwidth use would be:

$$\frac{24 \cdot 2 \text{ KiB}}{(32-2)\ \mu\text{s}} = 1563 \text{ MiB/s}$$

This is higher when the 1130 MiB/s bound.

With the bandwidth limits of one link the communication of the four messages would take:

$$2.95\ \mu\text{s} + \frac{4 \cdot 2048\ Byte}{891 \text{ MiB/s}} = 12\mu\text{s}$$

That is the smaller than the message bound and will stay smaller for smaller messages. For the finest grid the NWP bandwidth is the limit.

On the next finest grid the bandwidth use with the message bound is:

$$\frac{24 \cdot 512}{(32-2)\ \mu\text{s}} = 391 \text{ MiB/s}$$

So here the messages per second is the higher bound. This is true for smaller messages, too.

The calculation of the time for the communication on the finest grid is similar to the previous calculation. But the sum of all outgoing messages of a Cell processor and the overall bandwidth must be taken into account. The sum according the more precise bounds is:

$$4 \cdot 2.95\ \mu\text{s} + \frac{192 \text{ KiB}}{1130 \text{ MiB/s}} + 3 \cdot 4 \cdot 32\ \mu\text{s} = 562\ \mu\text{s}$$

This is a very good estimate of the measured time.

For the four coarsest girds only the precise analysis is done. The communication of the this grids is more complex. On the next grid level every SPU holds one grid point. The exchange for the Jacobi iteration and residual calculation is the same as on the previous grids and is still limited by the message bound. The additional communication steps for the restriction and prolongation has to be considered, too. But the grid points are chosen in a way that all $2^3$ grid points of the finer grid involved in one restriction/prolongation operation are associated to SPUs, which are on the same Cell processor. So there is DMA communication done, which can be ignored in this estimation. The complete sum for that grid is:

$$4 \cdot 32\ \mu\text{s} = 128\ \mu\text{s}$$

On the next coarser grid, only one of eight SPUs holds data. So one SPU of every Cell is used for calculation. The forwarding on the Jacobi/residual data exchange does not hurt much, because the internal DMA transfer is much faster than the external communication. So the communication pattern for this setting is one 128 Byte message exchange in every direction for every node card. That are six messages, so the message per time bound gives 8 $\mu$s. The link bound would be $2.95\ \mu\text{s} + 128\ Byte/891 \text{ MiB/s} = 3\ \mu\text{s}$ This is the first grid level on which the communication for restriction/prolongation get relevant.

For that communication messages with one to four values are sent. So all messages have the size of 128 Byte. The communication is serialized in three steps like illustrated in figure 11.

The pattern is one 128 Byte message for every Cell processor in every serialized step. That is bounded by the link, which gives 3 $\mu$s. Overall for this grid size:

$$4 \cdot 8\ \mu\text{s} + 2 \cdot 3 \cdot 3\ \mu\text{s} = 50\ \mu\text{s}$$

On the next coarsest grid there exist Cell processors, which do no calculation. Only every 8th Cell does calculations. The calculating Cell processors have the same communication pattern as on the previous grid. But in the forwarding three SPUs are involved. One link between them is an external link. When messages are just send in one direction, every forwarding processor must send one message. When messages are exchanged every forwarding processor has to send two messages. One in each direction of the dimension the processor is forwarding. The processors have to handle just one message at a time, but the middle processor of a forwarding route, when exchanging data, has to do two. The message bound for sending this two messages would be 2.7$\mu$s. The link bound for one message is 3.1 $\mu$s. For two messages 3.2 $\mu$s. So this is the bound limiting

the speed here. The Jacobi/residual communication is an exchange, the restriction/prolongation one direction communication. With one forwarding step this one processor is in the middle of the route. This sums up for this grid to:

$$50\ \mu\text{s} + 4 \cdot 3.2\ \mu\text{s} + 2 \cdot 3.1\ \mu\text{s} = 69\ \mu\text{s}$$

On the finest grid no residual, restriction or prolongation calculation are done. Now the forwarding is done across three cell processors. One of them in the middle. That sums up to

$$3 \cdot (8\ \mu\text{s} + 3.2\ \mu\text{s} + 2 \cdot 3.1\ \mu\text{s}) = 52\ \mu\text{s}$$

The sum for the four coarsest grids is 299 $\mu$s, which is near the measured value of 289 $\mu$s.

So it is possible to predict the network performance of the QPACE torus network. But one has to use the three bounds, developed in section 3.3.2

# 6 A possible improvement

The analysis of the network shows that sending small messages is too expensive compared to sending big messages. Especially when the communication is bounded by the message per time bound. So one approach to improve the performance is reducing the number of messages. One approach to do this are bigger message sizes, when the bound is the message throughput.

As a example the communication of the $2^3$ grid should be examine. For this code an idea is to start exchanging larger boundary layers on coarse grids. The data on the SPU after a data exchange should be enough to calculate two Jacobi iterations or one Jacobi iteration and one residual iteration. To do this calculations the right hand side must be exchanged, too. This must be done before the first iteration, so this cannot be included in a later data exchange. This data must be available only for the points, which are the boundary points on the standard scheme, so communication to direct neighbours can be used. The message size is 128 Byte. The communication is similar to the normal Jacobi/residual data exchange on this grid level. So this lasts 32 $\mu$s. The boundary points needed for doing two iterations are the boundary points of the standard scheme plus the neighbouring points of this points. That are a $2^2$ area of points in the direction of the boundary and additional two values in each edge of the cube. The communication is serialized in three steps like the restriction/prolongation communication. In the first step $2^3$ data points are exchanged in x direction. The second communication step exchanges $2^3$ data points of the sending SPU's grid and from the boundary values just received two edges. This are overall $2^3 + 2 \cdot 2 = 12$ values. In the third step data of four edges are communicated that sums up to $2^3 + 4 \cdot 2 = 16$. 16 double values need 128 Byte of memory, so all messages are 128 Byte long. So the messages have the same size as in the standard scheme. With sending from four SPUs in two directions on each dimension the NWP has to handle eight messages at a time. That is enough to reach the message per time bound. So the serialized data exchange can be done in the same time as the full parallel. With this trick just two exchanges on the current solution data are needed, but for the right hand side an additional communication step is needed. But the overall communication time reduces on this grid from $4 \cdot 32\ \mu s = 128\ \mu s$ to $3 \cdot 32\ \mu s = 96\ \mu s$. The extra calculations, which must be done due the increased number of grid points are cheap compared to the communication time.

# 7 Conclusion

It is possible to use the QPACE cluster to calculate multigrid algorithms. The algorithm can be adapted to the SIMD instruction set of the Cell processor well. The limiting factor for the performance for this algorithm is the network. To be able to understand this limitation in detail the communication of the implementation has to be analysed. The bounds which has to be considered in this analysis are devised in section 3.3.2. With this characterization of the network in mind the behavior of communication is very predictable

The SPU centric programming model simplifies the programming of the Cell processor in a cluster. The PPE just loads the SPU programs, start them and wait for them to finish. There are no callbacks for communication or other system calls. This can be setup as a framework, so that one has to program only the SPU part in future projects. A further simplification is the extended API,

which is introduced in section 4.3.1. This abstracts the hierarchical network of the torus network between the Cell processors and the DMA interconnect between the SPUs on one Cell to a torus network between SPUs.

It is worthy to consider the use of QPACE for more problems, but some attention has to be payed on the fit of the network properties to the application requirements.

# References


[BHM00] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, Philadelphia, 2000.

[IBM08] IBM Corp. *Software Development Kit for Multicore Acceleration, Version 3.1, Programmer's guide*, 2008.

[MP09] H. Simma M. Pivanti, F. Schifano. *Notes on the torus link logic*, February 2009. version 0.15.

[NP10] A. Nobile and D. Pleiter. *QPACE system software: torus*, February 2010.

[Ple07] D. Pleiter. Lattice qcd on the cell broadband engine. talk at Power Architecture Developer Conference, 2007.

[Ple09] D. Pleiter. Architecture of the qpace machine. talk at eQPACE Workshop, FZ Jülich, February 2009.