FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

JUICE - Jülich Initiative Cell Cluster Report 2007

Matthias Bolten, Andreas Dolfen, Norbert Eicker,*
Inge Gutheil, Willi Homberg, Erik Koch,*
Annika Schiller, Godehard Sutmann, Liang Yang

FZJ-JSC-IB-2007-13

December 2007

(last change: 22.12.2007)

(*) Institut für Festkörperforschung

Contents

Introduction

In the near future computer platforms will be based on multicore processors which will concentrate dozens or even hundreds of cores on a chip. Hence the number of cores in a high performance cluster will increase to hundreds of thousands. As a result, sequential and parallel programming methods have to be adapted to an appropriate combination of node-local and distributed parallel programming techniques and existing software has at least to be extended or partly new software approaches have to be investigated to use such clusters effectively.

While most processor manufacturers still follow conventional approaches of homogeneous, symmetric multicores with transparent cache hierarchy, other vendors start to explore alternative designs. One first example available to a broader audience is the Cell Broadband Engine developed by STI, a consortium formed by Sony, Toshiba, and IBM. The architectural characteristics of this processor include multiple heterogeneous execution units, SIMD processing engines, limited local store and a software managed cache. Applications can achieve near theoretical-maximum performance if Cell-specific features are respected.

IBM's first Cell-based system shipped was the QS20 blade, equipped with two Cell processors and 512 MB memory each and external connectivity provided by a Gigabit Ethernet and an optional InfiniBand network interface.

In order to explore the capabilities of this innovative architecture, in early 2007, project JUICE (JUelich Initiative CEll cluster) was established at JSC. A cluster of 12 IBM QS20 blades was procured to examine the potential of cell processors as a building block for future high-end computing systems. Being lucky to have very early access to Cell hardware and software the deployment phase however was a bit experimental and e.g. the spidernet ethernet driver had to be adapted to get it running in line with our batch system Torque.

Example codes provided with the installation of the Software Development Kit (SDK) produced high performance results on a single blade right from the beginning. With farming codes more than 90 % of the peak performance could be achieved. The implementation of the CellSs, a programming model developed at Barcelona Supercomputing Center (BSC), facilitated the porting of sequential programs to the Cell platform. Subprojects evolved dealing with Cell-specific implementations of software codes, e.g. based on the Lanczos eigenvalue algorithm and the multigrid method.

The first implementation of the Cell processor is significantly more efficient at computing single-precision floating results, a drawback which will vanish in the next enhanced double precision version Cell eDP. Hence, at present, it is profitable to consider mixed-precision software implementations in order to balance accuracy and performance.

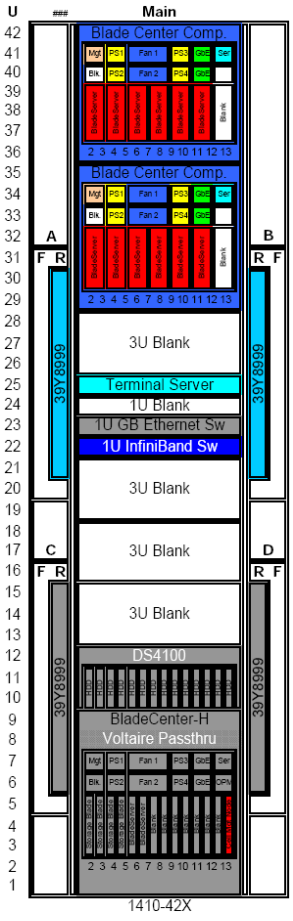
Installation of ParTec's MPI stack allowed for promising tests with distributed parallel jobs. Midyear, the JUICE cluster was equipped with an InfiniBand network and MPI communication became more important. In the end we succeeded in getting at a TFLOPS performance on our 12-nodes cluster.

Chapter 1

Cluster Overview

1.1 Configuration

The JUICE cluster consists of two BladeCenter-1 chassis each equipped with six QS20 Cell blades. Each blade includes two Cell BE sockets with one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs) in each case, 1 GB Rambus XDR memory, a 40 GB hard disk, and a 4x Mellanox InfiniBand card.
The Gigabit Ethernet network is based on the onboard ethernet adapters and the Nortel switches embedded in the BladeCenter. The InfiniBand network is established by an InfiniBand daughter card and a 24 port Voltaire ISR9024 switch.
A x3550 system including dual XEON 5140 processors, 5 GB memory, 300 GB hard disk, and two Gigabit ethernet ports serves as frontend machine.
A terminal server is included to assure access to the serial console of each cluster node. The JUICE hardware is housed in the upper part of a 42 inch rack. Featuring a single-precision floating point peak performance of nearly 205 GFLOPS per Cell processor the overall performance of the JUICE cluster is about 5 TFLOPS. A fully equipped rack with six BladeCenter and seven double wide blades would have an aggregated peak performance of more than 17 TFLOPS thus presenting an outstanding ratio between performance and floor space as well as performance and power consumption.



Rack hosting the JUICE cluster

1.2 Installation and management

The BladeCenter Management Module provides an overview of the system hardware status and basic features like power restart and the setting of the boot sequence. The frontend

serves for user login and as an NFS file server. Operating system installation of the cluster nodes is done in two steps. A Golden Client is set up first by starting a netboot with VNC support from the serial console loading the PPC Fedora Core image from the DHCP/TFTP service on the frontend [13]. After completion the resulting root file system is transferred by means of the System Installation Suite[8] tools to the images repository on the frontend which serves as a basis for an automated installation of the cluster nodes.

1.2.1 ParaStation

The implementation of ParTec's ParaStation furnished JUICE with a cluster middleware for serial and parallel applications comprising an MPI stack, process management, and a collection of administration tools providing a "Single System View" of the cluster.

The GridMonitor is a versatile system monitor for Linux-based compute clusters. A multiplicity of information from different devices and services from a cluster may be read, evaluated and stored. The GridMonitor provides the administrator with various aspects of the available information, from an overall status of all configured clusters to in-depth details of nodes and devices. Data can be grouped with respect to different aspects and are visualized via a web browser. Furthermore, parameters may constantly be monitored and the adminstrator may be informed, if required[9].

1.2.2 Torque batch system

Torque is Open Source and is being used on other Linux clusters at FZJ already. Torque is based on PBS (Portable Batch System), an Open Source batch and resource management system, and is available as RPM package for Linux.

In general, compilation is performed on the compute nodes. The user gets access to the nodes by queueing an interactive job. A typical usage of the qsub command:

```
qsub -q sdk21 -l nodes=N:perfctr -I -X
mpirun -np N myprog
```

If there is more than one SDK version currently in use there is a seperate queue to access the proper nodes (e.g. sdk21). N specifies the number of required nodes for parallel codes. Additional options like perfctr and hugetlb are supported providing access to the Cell performance counters (App.A) or making the hugetlb filesystem available. The option -X is required for graphic support, e.g. when using parallel debugger like DDT.
Torque allows for prologue and epilogue scripts being executed when jobs are started and terminated, respectively, thus granting optional resources if required, e.g. for using the Cell performance counters:

```
if [ ${PBS_RESOURCE_NODES} != ${PBS_RESOURCE_NODES/hugetlb/} ]; then
    cpc -E > /dev/null 2>&1
    chown ${2}:${3} /dev/cellperfctr
fi
```

Here (using SDK-2.1) the performance monitor is enabled and the ownership of the corresponding device is properly set to give access to the user.

Chapter 2

Hardware Architecture and Software Development

2.1 Cell Broadband Engine Architecture

The Cell Broadband Engine (CBE) processor is a heterogeneous multicore processor - based on a PowerPC core and eight so-called synergistic processing elements - which distinguishes itself by its extremely high performance single-precision arithmetic.

- One Power-based PPE, with VMX
  - 32/32kB I/D L1, and 512kB L2
  - dual issue, in order PPU, 2 HW threads
- Eight SPEs, 2-way SMT
  - up to 16x SIMD
  - dual issue, in order SPU
  - 128 registers (128b wide)
  - 256kB local store (LS)
  - 16+16B/cycle DMA, 25.6 GB/s, 16 outstanding requests
- Element Interconnect Bus (EIB)
  - 4 rings, 16B wide at 1/2 clock
  - 96B/cycle peak, 16B/cycle to memory
  - 2x16B/cycle BIF and I/O
- External communication
  - Dual XDR memory controller (MIC)
  - Two configurable bus interfaces (BIC)
    - Classical I/O interface
    - SMP coherent interface

The PPE runs the operating system and manages system resources. It is intended to deal with the coordination of the eight other cores (see fig.2.1). An SPE core comprises a very large register file of 128 registers each 128-bit wide for floating-point as well as integer operations and owns a powerful SIMD engine operating on all 128-bit registers. It supports 16 byte, 8 halfword, 4 word (integer or float), or 2 double word operations in a single clock cycle. Taking into account the fused add-multiply operation which delivers two results at a time the theoretical single-precision peak performance is 25.6 (4 x 3.2 GHz x 2) GFLOPS per SPE at 3.2 GHz and 8 x 25.6 = 204.6 GFLOPS for the CBE.

Due to the large performance difference between single-precision and double-precision arithmetic (with a ratio of 14 as double precision operations are executed with a delay of 7 cycles and only two instead of four operands can be processed in parallel) it is essential to exploit 32-bit floating point arithmetic whenever feasible.

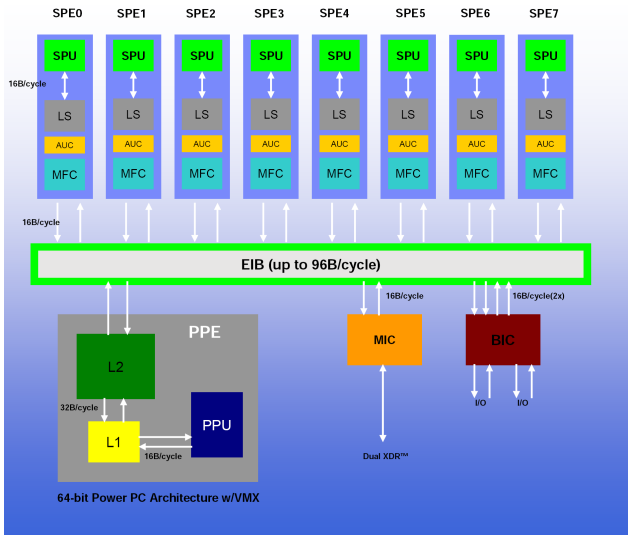


Figure 2.1: Cell processor architecture

2.2 QS20 Cell blades

A QS20 Cell blade consists of two CBE clocked at 3.2GHz. Each Cell processor has access to a 512 MB XDR memory via its MIC. It is possible to coherently access the memory attached to the other Cell chip's MIC by crossing the so called IO interface (via BIC/BIF). However performance is degraded. To achieve good performance it is important to ensure that each thread accesses only the XDR DRAM device of the Cell chip it is running on. There are two ways to control the NUMA policy of programs. From within the code one can include the libnuma and from the command line one can make use of the numactl command (App.A).

2.3 Software development

The Software Development Kit (SDK) is a complete package of tools for the creation of applications meant to run on Cell Broadband Engine Architecture. It is composed of run-time tools such as the Linux kernel, development tools, software libraries and frameworks, performance tools, a Full System Simulator, and example source files[13].

The programming model for the Cell blades is a master slave model. We can distinguish different ways of distributing work to the SPEs. The most simple case is the function offload model where a single function is executed on one SPE. If more than one SPE is used there is the choice between a data parallel and a task parallel model.

In the data parallel case all SPEs do the same computations on different local data. This is very close to the familiar SPMD model and thus it is preferred by most applications up to now. In the task parallel approach each SPE does a different part of the whole task and data is pipelined from one SPE to the other thus reducing access to main memory at the expense of a more complicated load balance.

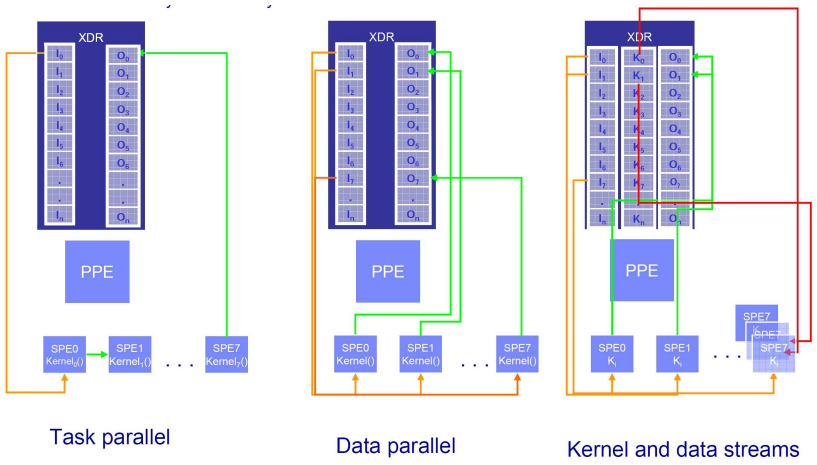


Figure 2.2: Cell programming model

PPE program skeleton (SDK-2.0)

```
extern spe_program_handle_t my_spe_program;
[...]
/* allocate the SPE tasks */
for (i = 0; i < num_spes; i++)
  speid[i] = spe_create_thread (gid, &my_spe_program, NULL,
             NULL, -1, 0);
[...]
/* send messages to SPE mailboxes */
for (i=0; i<num_spes; i++) spe_write_in_mbox(speid[i], 1);
[...]
/* receive messages from SPEs via PPE mailboxes*/
for (i = 0; i < num_spes; i++) {
   while (spe_stat_out_mbox(speid[i]) < 1);
   result[i] = spe_read_out_mbox(speid[i]);
}
[...]
/* wait for the SPEs to complete */
for (i = 0; i < num_spes; i++) spe_wait(speid[i], &status, 0);
```

SPE program skeleton

```
control_block cb __attribute__ ((aligned (128)));

int main(unsigned long long speid, addr64 argp, addr64 envp) {
   unsigned int data0[SPE_BUFFER_ENTRIES] __attribute__ ((aligned (128)));
    mfc_get(&cb, argp.ui[1], sizeof(cb), 0, 0, 0);
   mfc_write_tag_mask(1<<0); mfc_read_tag_status_all();

    while (spu_stat_in_mbox() < 1); spu_read_in_mbox();

    mfc_get(data0, cb.PPE_data_Ptr, cb.size, 0, 0, 0);
   mfc_write_tag_mask(1<<0); mfc_read_tag_status_all();

    compute(data0);
    mfc_put(data0, cb.PPE_data_Ptr, cb.size, 0, 0, 0);
   mfc_write_tag_mask(1<<0); mfc_read_tag_status_all();
    return 0;
}
```

The main program starts on the PPE and creates the required SPE threads which run immediately and acquire their control block information from a dedicated area in main memory. Then, all SPE threads wait for a message from the PPE notifying that input data is in place and subsequently transfer data to the local store. Thereafter, computation is performed and resulting data transferred back to main memory. While all SPE threads perform computations the main program on the PPE waits for a final message from each SPE indicating that all work is done.

The compilation of a Cell program consists of the following steps:

```
# compile spu code
spu-gcc -o test_spu.o test_spu.c

# compile ppu code
ppu-gcc -o test_ppu.o test_ppu.c

# embed spu object into ppu object
embedspu test_spu test_spu.o test_spu_embed.o

# link everything together
ppu-gcc -o test test_ppu.o test_spu_embed.o
```

After compilation of the SPE program the created object file must be embedded into an

PowerPC object which at last is linked together with the object code of the PPE main program to get an executable for the Cell processor.[20]

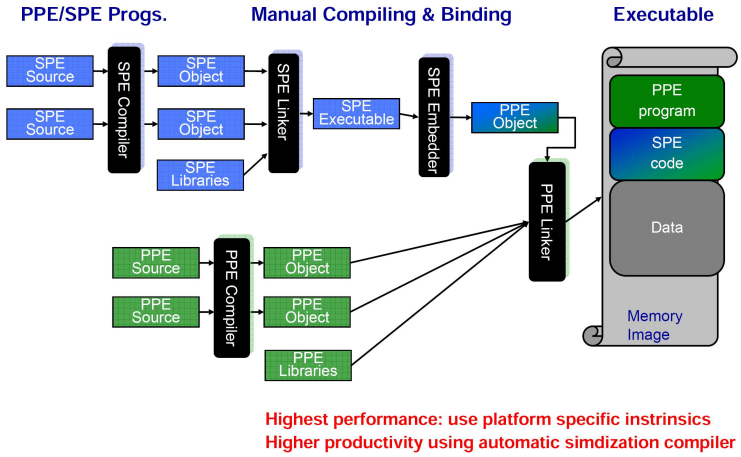


Figure 2.3: Cell compilation scheme

The CBE is no general purpose processor and algorithms only run efficiently if sufficient computation is performed on the 256 KB local store between DMA transfers and if SIMD parallelisation can be properly used. The considerable difference between single- and double-precision perfomance suggests to perform compute intensive operations using single-precision arithmetic and to improve the accuracy of the solution in double-precision. Good performance results have been attained by the use of mixed-precision in the iterative refinement step in solving systems of linear equations [17].

Chapter 3

Projects

3.1 Lanczos implementation for Hubbard model calculations on Cell *(Andreas Dolfen, Erik Koch)*

To design new materials with superior properties we have to accurately solve the many-body Schrödinger equation. The dimension of the Hilbert space grows, however, exponentially. Thus we need to use approximations. A very successful approach, density functional theory, makes realistic calculations for many materials possible. It relies on a single particle picture. New materials of high technological interest show however effects of strong correlations. This gives rise to exciting physics such as magnetoresistance, high-temperature superconductivity or spin-charge separation. In these compounds electrons lose their individuality and we need to go beyond effective single particle theories and perturbative approaches. This is, however, only feasible for model Hamiltonians. For these models to be realistic they must be as large as possible. Thus, the need for supercomputers. The CBE offers top performance on a single chip and is therefore an interesting architecture to explore for these kind of calculations.

3.1.1 Hubbard model

The Hubbard model is the simplest many-body model which cannot be reduced to an effective single-particle system. Although it is a very simple model it contains rich physics. In its real-space representation it reads,

$$H = -\sum_{\sigma,ij,\nu} t_{ij} c^{\dagger}_{i\sigma} c_{j\sigma} + U \sum_{i} n_{i\uparrow} n_{i\downarrow} \ , \tag{3.1}$$

where the first term denotes the kinetic energy and the second one the Coulomb repulsion. In the kinetic energy term an annihilation operator $c^{\dagger}_{i\sigma}$ annihilates an electron with spin σ on site i and the creator operation $c_{j\sigma}$ puts one on site j. Thus, effectively describing the hopping of an electron from site i to j with hopping amplitude t_{ij}.

In the Coulomb term the operator $n_{i\sigma}$ denotes the number of σ electrons on site i. A site can only be occupied ($n_{i\sigma} = 1$) or unoccupied ($n_{i\sigma} = 0$) by electrons of a single spin due to the Pauli principle. Thus, there are only four states a site can have. Either it is empty, singly occupied with a spin-up or down electron, or doubly occupied with electrons of either spin. In the latter case the Coulomb term describes a penalty of U in energy due to the Coulomb repulsion of equally charged particles. In all other cases this term is equal to zero.

3.1.2 Lanczos method

The Hubbard model can be solved numerically using the Lanczos method. It is an iterative method which heavily relies on sparse matrix vector multiplication. In a nutshell it works by starting from a random vector $|\phi_0\rangle$, which must not be orthogonal to the ground-state vector, and applying the matrix to this vector. Afterwards the resulting vector is orthogonalized with respect to the starting vector and then normalized, i.e.

$$\langle\phi_1|H|\phi_0\rangle|\phi_1\rangle = H|\phi_0\rangle - \langle\phi_0|H|\phi_0\rangle|\phi_0\rangle \ . \tag{3.2}$$

Similarly the second iteration is carried out, i.e.

$$\langle\phi_2|H|\phi_1\rangle|\phi_2\rangle = H|\phi_1\rangle - \langle\phi_1|H|\phi_1\rangle|\phi_1\rangle - \langle\phi_1|H|\phi_0\rangle|\phi_0\rangle \ . \tag{3.3}$$

The n-th iteration can be expressed recursively as

$$\beta_{n+1}|\phi_{n+1}\rangle = H|\phi_n\rangle - \alpha_n|\phi_n\rangle - \beta_n|\phi_{n-1}\rangle \ , \tag{3.4}$$

where $n \in 2, \ldots, m$ and

$$\alpha_n = \langle\phi_n|H|\phi_n\rangle, \quad \beta_{n+1} = \|H|\phi_n\rangle - \alpha_n|\phi_n\rangle - \beta_n|\phi_{n-1}\rangle\| \ .$$

Equation (3.4) shows that the Hamiltonian is a tridiagonal matrix in the basis of so-called Lanczos vectors $\{|\phi_n\rangle\}$, where the $\{\alpha_i\}$ are the diagonal whereas the $\{\beta_i\}$ are the off-diagonal elements. This tridiagonal matrix can be diagonalized by standard means. The resulting lowest eigenvalue is a good approximation to the ground-state eigenvalue as long as the number of iterations m is sufficiently large. It shows that only relatively few iterations are needed to get a well converged ground-state energy. This is what makes this method so powerful.

3.1.3 Lanczos and Cell

A supercomputer based on Cell will be a distributed memory system. An implementation of the Lanczos method for Hubbard models has already been developed [11] and runs efficiently on massively parallel systems like BlueGene [10]. It relies on the fact that the Lanczos vectors can be written as a matrix whose indices denote the indices of the up and down spin configurations. This leads to the idea of performing a matrix transpose on these vectors. If the elements for different up spin configurations where stored locally in memory, after transposition the same would be true for the down spin configurations.

As long as correlations are not too strong, i.e. there are no strongly separated energy regimes, single precision suffices at least for test calculations. Hence we can make use of the computational single precision power of the Cell chip. If single precision does not suffice we can still resort to the technique of iterative refinement [18]. This technique was widely used in former times when double precision computations were much more expensive than single precision ones. This situation is enjoying a renaissance with the current implementation of the Cell Broadband Engine Architecture.

General observation

Before actually starting the development of the Lanczos code it is practical to take a closer look at the Cell system itself, since many design decisions strongly depend on the

actual performance of various subsystems and their interplay. For instance is it feasible to perform operations, which are needed often, once and store them or is it better to redo the calculations each time the result is needed? This, of course, depends on the transfer/compute ratio. If the calculations can be performed within the time needed to get the elements from memory, on-the-fly calculation is feasible. We thus assess the actual DMA transfer bandwidth using a norm calculation which employs double buffering to overlap communication and computation. The results are shown in plot 3.1.

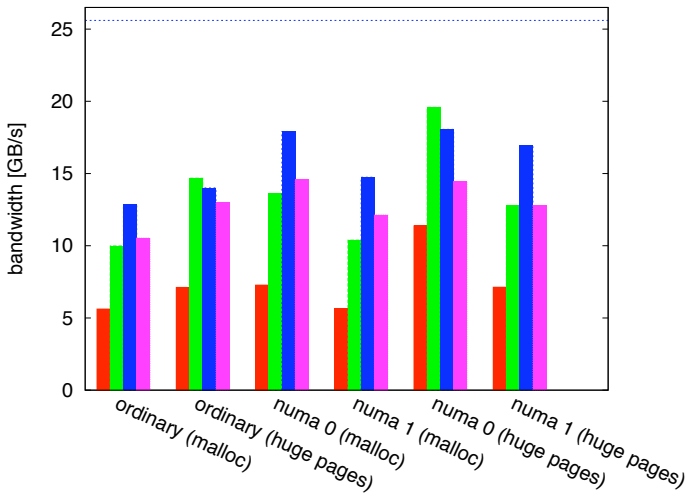


Figure 3.1: Bandwidth for norm calculation on one(red), two (green), four (blue) and eight (magenta) SPEs. The size of the vector is 128 MB which is equivalent to 32 k pages of size 4 kB. Timings measured from PPE. Code employs double-buffering. The plot shows the timings for different methods of memory allocation and NUMA affinities. The blue dashed line denotes the theoretical peak performance of 25.6 GB/s. It is advisable to use more than one SPE to exploit the DMA bandwidth. Moreover, binding threads to NUMA nodes also is crucial to achieve good performance. SDK2.0 with libsdk1 is used. Shown are average values for five iterations.

We observe that the Cell boards are NUMA architectures, which is important to have in mind when coding. It is interesting to note that it does matter which NUMA node we bind the memory and the computation to. NUMA node 0 always yields better performance (7.2 GB/s vs. 5.6 GB/s). We are not sure how this comes about. A reason could be the address concentrator, which is only located at numa node 0. But this should only have an impact on the latencies.

The second effect one has to take care of are TLB misses. Norm calculations of large vectors obviously access large ranges of memory and thus suffer from many TLB misses, if the pages are small. We thus use 16 MB pages, effectively eliminating TLB misses and thus making address translation efficient. And indeed, using NUMA to bind the computation and the memory allocations to node 0 as well as using huge pages yields the best results for a single SPE with 11.4 GB/s. This is still far from the theoretical peak bandwidth. The situation, however, improves when using more SPEs. With four SPEs, NUMA policies set to node 0 and huge pages the norm code gives a performance of about 18.2 GB/s. Strangely, however, the best performance (19.6 GB/s) in this test was achieved having two SPEs using NUMA node 0 and hugetlbs.

Coulomb term

First we discuss the calculation of the diagonal elements of the Hamiltonian, i.e. the Coulomb term. In our implementation a single configuration is stored in two integers, one for the up-spin configuration and one for the down-spin configuration. The bit representation of the integers shows which sites are occupied and which are not. For the local Hubbard term we need to calculate the number of doubly occupied sites, i.e.

$$\sum_i n_{i\uparrow} n_{i\downarrow} \ .$$

This can be achieved by using a bitwise `and` operation on the two integers and then counting the number of set bits. This number obviously is equal to the number of doubly occupied sites.

Shall we use on-the-fly calculation or precalculated diagonal elements? Let us try both. The on-the-fly implementation exploits the SIMD capabilities of the SPEs performing these operations on four 32 bit integers per SPE in a single clock cycle. In the local store of each SPE participating in the calculation we need to store the lookup tables which map the integer labels for up and down spin configurations to the actual configurations encoded again in integers. Alternatively we can precalculate the diagonal elements on the PPE and

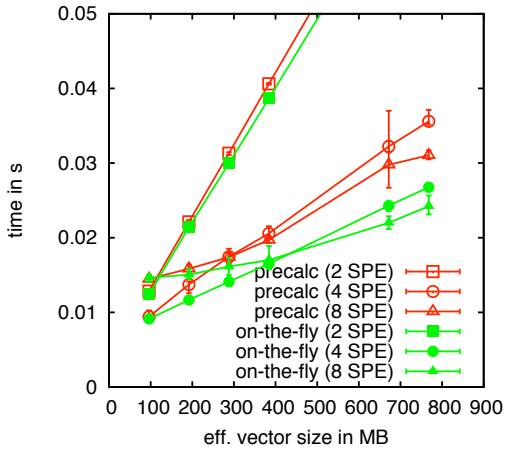


Figure 3.2: Comparison of the runtime for on-the-fly and pre-calculation of the Hubbard term for different numbers of SPEs. For two SPEs both methods are almost equally fast. Starting from four SPEs on, the on-the-fly calculation becomes significantly faster. This is because the bandwidth is bounded and the precalulation-code is transferwise more expensive. For four SPEs it is almost fully exploited and thus on-the-fly calculation becomes preferable.

transfer it to the SPEs. Aside from the two (get and put) DMA transfers of the wave vector elements we then need a third transfer of the same size. Depending on the DMA to computation performance ratio either on-the-fly calculation or precalculation is faster. Figure 3.2 shows the runtime measured from the PPE for on-the-fly and pre-calculation codes using two, four and eight SPEs, respectively. First we observe the expected linear increase of runtime with respect to vector size. For a single SPE (not shown) and two SPEs

on-the-fly calculation and pre-calculation are equally fast. Starting from four SPEs on the one-the-fly calculation is significantly faster.

Outlook

The Cell chip is a very promising architecture for compute-intensive tasks like the simulation of strongly correlated materials. The current cell blades have, however, some significant drawbacks. While the Cell chip has a highly efficient but simplified architecture which gives much responsibility and control to the programmer, the boards come with a complicated NUMA architecture which makes it difficult to reliably achieve high performance. We feel that it would be completely sufficient for our purposes to have a single chip per board/node. Then there would be no overhead to preserve coherence, no strange memory access time differences for different NUMA nodes (see section 3.1.3). All in all a less complex and more deterministic system.

Also the operating system does not quite fit the philosophy of the Cell chip. It is a conventional full-featured Linux system, where instead a lightweight kernel on the compute nodes (CNK) similar to the one found on BlueGene super computers would be more appropriate. High performance applications in general have no need for sophisticated virtual memory solutions and their computational overhead. On the BlueGene/L Run Time Supervisor (BLRTS) the kernel and the userspace share the same address space and only a single process can run on the CNs. To avoid being overwritten by the user program the kernel protects itself by reprogramming the PowerPC MMU [1].

Finally, we think that an important consideration for the development of Cell architectures is the balance of floating point performance, memory access, and network. Currently the small size of the local store is a severe limitation to efficiency (see on-the fly vs. memory access) and we would like to see the local store increased.

3.2 A multigrid method for the solution of the Poisson equation *(Mathias Bolten)*

Multigrid methods are known to be optimal methods for the solution of linear systems arising from the discretization of elliptic PDEs, although they are not limited to these problems. So they are of interest for a broad range of applications originating from different fields of research. As multigrid methods are useful for a number of applications and as they are an important representative of the class of methods that deals with structured grid problems and the emerging matrices we decided to study the performance of a multigrid method on the Cell broadband engine architecture.

3.2.1 A short introduction to multigrid methods

Multigrid methods are solvers for linear systems of algebraic equations of the form

$$A\mathbf{x} = \mathbf{b}. \tag{3.5}$$

Instead of presenting a detailed introduction to multigrid methods we refer to the introduction by Briggs, Henson and McCormick [4] and to the excellent textbook by Trottenberg, Oosterlee and Schüller [24]. Nevertheless we like to mention the fundamental observation that is exploited in geometric multigrid methods. When we apply an iterative solver like Jacobi or SOR to the system (3.5) and observe the error it turns out that the error is not reduced very efficiently, but it is smooth after only a few iteration steps, c.f. figure 3.3 for an example of the standard 5-point discretization of the Laplacian. Analyzing the iteration

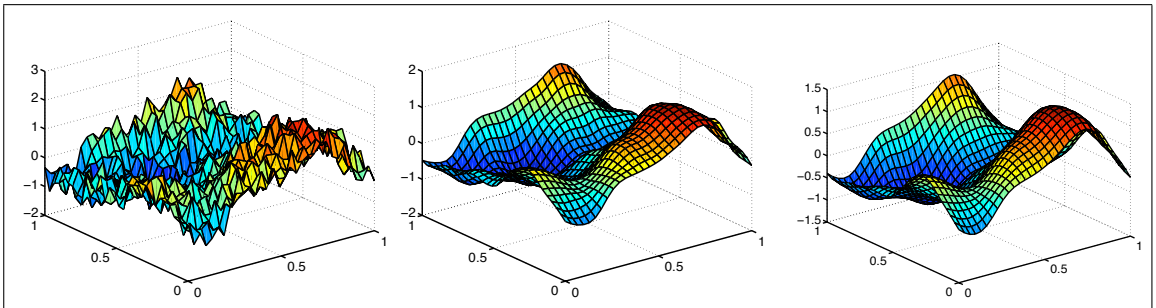


Figure 3.3: Error after 0, 5 and 10 iterations of damped Jacobi iteration for standard 5-point discretization of Laplacian

matrix of the Jacobi iteration affirms this observation, as the error components belonging to the eigenvectors of the matrix A that are highly oscillating are damped very fast, whereas the low-frequency modes are hardly affected at all. Similar observations can be made for other iteration methods like SOR. A two grid method exploits this fact in the following way: After carrying out a few iterations of a such a smoother the resulting residual

$$\mathbf{r} = \mathbf{b} - A\mathbf{x}$$

is computed and transferred to a coarser grid. On that level the defect equation

$$A_C\mathbf{d} = \mathbf{r}$$

is solved using an appropriate representation A_C of A on that level, i.e. a rediscretization of the underlying partial differential equation using a larger grid width, and the resulting

defect is transferred back to the fine grid. Using that approximation of the defect the current approximate solution is corrected. The process of transferring the residual vector to a coarser grid, solving the defect equation on that level and correcting the current approximate solution is called *coarse grid correction*. After the coarse grid correction a few additional iterations of our smoother are applied, as it introduces highly-oscillating error-components, again. Using that technique the computational requirements can be reduced significantly, as the problem on the coarse grid is typically much smaller than the original problem. Due to the fact that smooth error modes can be well-represented on the coarse grid, the method is very efficient. If this technique is applied recursively to solve the defect equation on the coarse grid the resulting method is a multigrid method. As seen in figure 3.4 the resulting method has a linear convergence rate, a huge benefit compared to the damped Jacobi iteration, which is its basis.

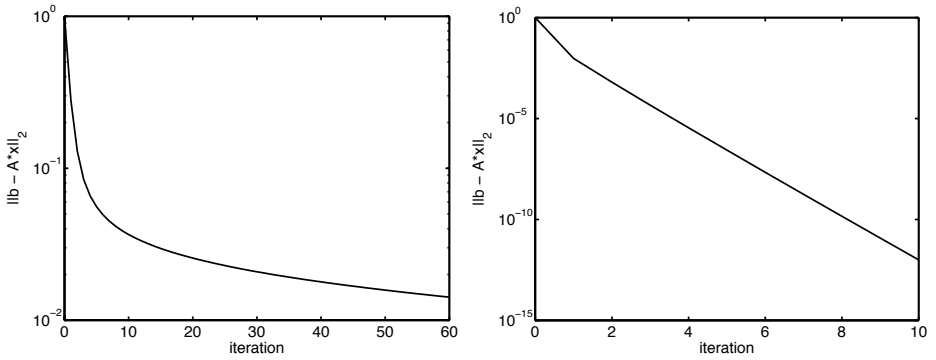


Figure 3.4: Convergence of the damped Jacobi iteration and multigrid method using 3 presmoothing steps and 3 postsmoothing steps of the damped Jacobi method

3.2.2 Multigrid method on the CBEA

The Cell broadband engine architecture with its limited local storage and the availability of DMA transfers for load/store operations to/from the main memory is well-suited for the solution of linear systems arising from the discretization of partial differential equations on structured grids. As seen in the previous section the smoothing iteration plays an important role in the multigrid method and as it is computationally the most expensive part we started with the implementation of the damped Jacobi method for a 7-point discretization of the Laplacian in three dimensions, where the computation is offloaded to the SPUs.

Implementation of the Jacobi smoother

The implementation of the Jacobi solver works as follows: The PPU program allocates the memory for the right hand side and the solution and it supervises the work of the SPUs, which poll their mailboxes for messages from the PPU. The sizes are chosen such that the boundaries are included in the volume for ease of implementation and to avoid conditional branches which are expensive to do for the SPU. First the PPU tells the SPU programs to initialize, i.e. read in the addresses of the solution and the right hand side and the appropriate sizes. Furthermore, memory for their part of eight slices of the data is allocated in the local storage. The data distribution amongst the SPUs is depicted in figure 3.5 and explained as follows: The SPUs work on one xy-slice of the data at once.

This slice is distributed amongst the SPUs in the y-direction and packed into the vectors in x-direction. After initialisation the SPUs are ready to do work. To do an iteration of the

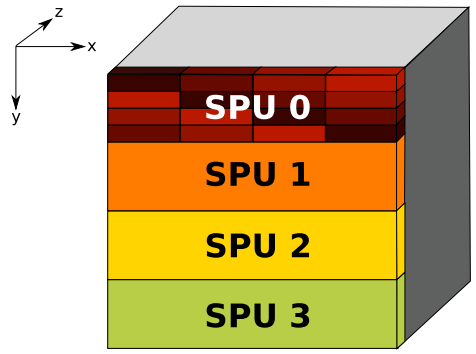


Figure 3.5: Distribution of the data to the SPUs

Jacobi method the PPU sends an iterate command to each of the SPUs. This causes the SPUs to do an iteration of the Jacobi iteration. To use multibuffering two slices of the right hand side and four slices of the current solution are kept in the local store, additionally two slices of temporary storage are needed that carry the new iteration of the solution. When the SPUs have finished the work for one iteration they put a message in their outbox. The outbox is polled by the SPU. The arithmetic operations of the Jacobi iteration are vectorized using the vector intrinsics of the SPU.

Some performance results

We measured the different aspects of the performance of our implementation. The performance for 64^3 grid points using up to 8 SPUs can be found in Table 3.1 and in figure 3.6. It can be seen that the time per iteration can be reduced by a factor of 4.5 if 8 SPUs are used. The available bandwidth is also utilized very well, although there is still some potential. The next test regarded the dependence of the time per iteration on the number

#SPUs	bandwith/SPU [GB/s]	total bandwith [GB/s]	time per iteration [s]
1	3.36	3.36	$1.04 \cdot 10^{-3}$
2	3.43	6.86	$5.24 \cdot 10^{-4}$
3	2.72	8.16	$4.53 \cdot 10^{-4}$
4	3.19	12.76	$2.98 \cdot 10^{-4}$
5	2.32	11.60	$3.37 \cdot 10^{-4}$
6	2.15	12.90	$3.10 \cdot 10^{-4}$
7	2.04	14.28	$2.89 \cdot 10^{-4}$
8	2.33	18.64	$2.27 \cdot 10^{-4}$

Table 3.1: Bandwith and time per iteration for 64^3 grid points for various numbers of SPUs

of grid points of the original grid. For that purpose we tested grids up to a size of 128^3 grid points on the 8 SPUs of one chip and grids up to 256^3 grid points on the 16 SPUs of one blade, respectively. The results can be found in Tables 3.2 and 3.3, additionally a plot of the respective times per iteration can be found in figure 3.7. As the 8 SPUs were forced to use memory local to the Cell chip only, the performance was slightly better compared to the 16 SPU variant, although in the latter case the mere floating point performance is twice as large. One also notices that while the algorithm scales linearly using 8 SPUs

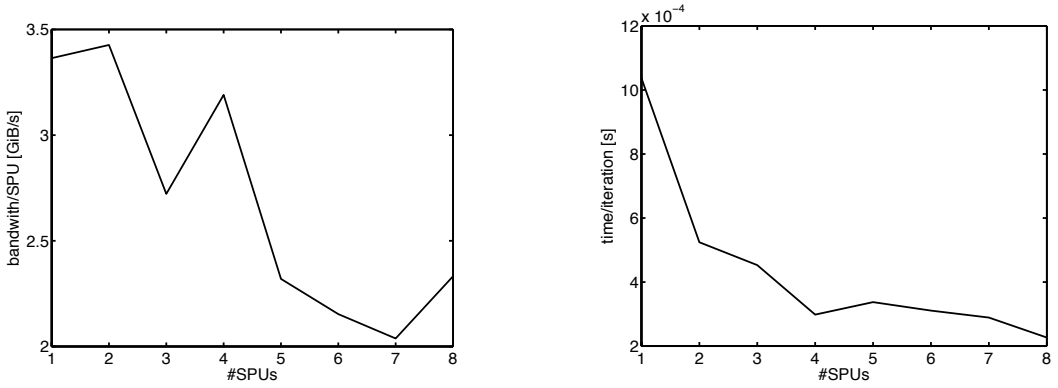


Figure 3.6: Bandwith and time per iteration for 64^3 grid points for various numbers of SPUs

for problems from 32^3 grid points on, the scaling behavior is not so clear in the 16 SPU case, which seems to be founded in the NUMA architecture. We like to note the achieved floating point rate is far below the hardware maximum, which is clear, as the algorithm is dominated by load/store operations.

$\sqrt[3]{\#\text{grid points}}$	time per iteration [s]	GFLOPS [1/s]
8	$7.736690 \cdot 10^{-5}$	0.05
16	$7.748190 \cdot 10^{-5}$	0.42
32	$7.806350 \cdot 10^{-5}$	3.36
64	$2.265704 \cdot 10^{-4}$	9.26
128	$1.557704 \cdot 10^{-3}$	10.77

Table 3.2: Time per iteration and GFLOPS for 8 SPUs for various grid sizes

$\sqrt[3]{\#\text{grid points}}$	time per iteration [s]	GFLOPS [1/s]
16	$1.760862 \cdot 10^{-4}$	0.17
32	$1.758365 \cdot 10^{-4}$	1.49
64	$2.752577 \cdot 10^{-4}$	7.62
128	$1.746457 \cdot 10^{-3}$	9.61
256	$1.246028 \cdot 10^{-2}$	10.77

Table 3.3: Time per iteration for 16 SPUs for various grid sizes

3.2.3 Conclusion and outlook

The implemented Jacobi smoother allowed us to get more familiar with the Cell broadband engine architecture. The results were as expected: The performance benefits a lot from the high available memory bandwith, although we only get about 5 percent of the possible floating point performance. The enormous performance drop for small grid sizes was not unexpected, although future tests have to show how multigrid performance is harmed by this fact. We currently plan to expand our code to use a pipelining mechanism to transfer slices from one SPU to another to benefit more from each load/store operation. The most important step is to run a whole multigrid cycle on the Cell's SPUs, not only the smoother.

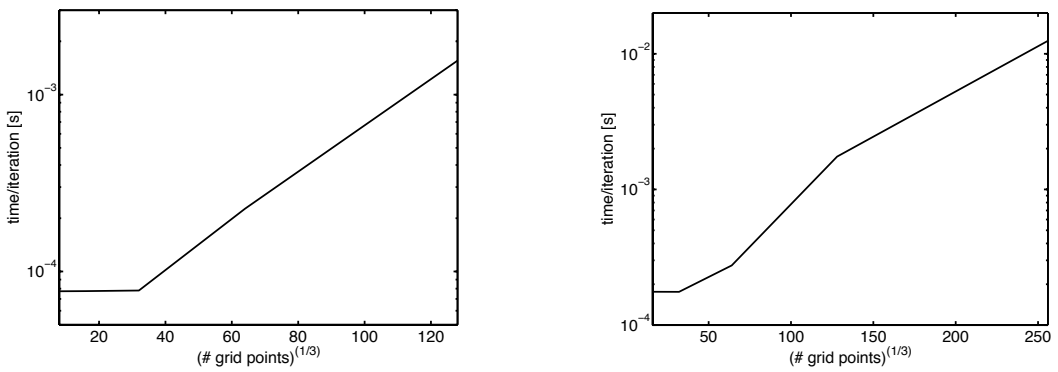


Figure 3.7: Time per iteration for 8 SPUs (left) and for 16 SPUs (right) for various grid sizes

3.3 A fast Wavelet based implementation to calculate Coulomb potentials *(Annika Schiller, Godehard Sutmann)*

Long range interactions play an important role in complex molecular systems. The calculation of long range interactions is computationally very demanding. Since all interactions between particle pairs have to be considered, the calculation scales like $O(N^2)$. To overcome the quadratic scaling, methods were developed to reduce the complexity to $O(NlogN)$ or even $O(N)$. In general those methods can be classified into mesh-free and mesh-based algorithms. Mesh-free algorithms like the Fast Multipole Method (FMM) yield better results, since the discretization error is avoided, but are demanding to implement. Alternatives may be found in mesh-based algorithms, where fast methods like multigrid techniques can be applied.

In the present work we consider a mesh-based method, which uses a fast Wavelet transform technique. This method reduces the computational complexity to $O(N)$. To make this method even faster, an implementation was developed, which uses the cell processors of the IBM Cell BladeCenter QS20 in Jülich (JUICE).

3.3.1 Fast Wavelet based method

The particles interact via Coulomb interactions, so the potential at a charges position with index i is given by

$$\phi(r_i) = \sum_{i \neq j}^{N} \frac{q_j}{|r_i - r_j|} \tag{3.6}$$

where N is the number of charges in the system, q_j the charge of particle j and r_j its position. The interaction energy of particle i with all other particles is then given by

$$U_i(r_i) = q_i \cdot \phi(r_i) \tag{3.7}$$

Equation 3.6 can also be written in short hand notation as matrix-vector product

$$\Phi(\{R\}) = A(\{R\}) \cdot Q \tag{3.8}$$

where $\Phi = \{\phi_1, ..., \phi_N\}$ is the potential for every particle and $Q = \{q_1, ..., q_N\}$ is a constant charge vector. The matrix elements are given by $A_{ij} = 1/|r_i - r_j|$. The interaction energy could then simply be written as $U = diag(Q\Phi^T)$.

The charge vector Q is constant, but the matrix elements change from step to step because the particles move along their trajectories. Therefore, the computational complexity is $O(N^2)$. To shift the time dependence from the matrix to the vector Q we introduce a mesh, onto which the particle charges are distributed. This mesh has a constant grid spacing throughout the time evolution of the system. Nevertheless, with this grid based summation, the complexity is still $O(N^2)$ due to the dense matrix-vector product. To obtain an efficiency gain, it will be necessary to transform the matrix into a sparse representation. This transformation is realized via Wavelets.

In principle, Wavelets are a tool to analyse data on different time and length scales. A Wavelet transform can be represented by an orthogonal matrix $\mathcal{W}$. In this case equation 3.8 can be written as

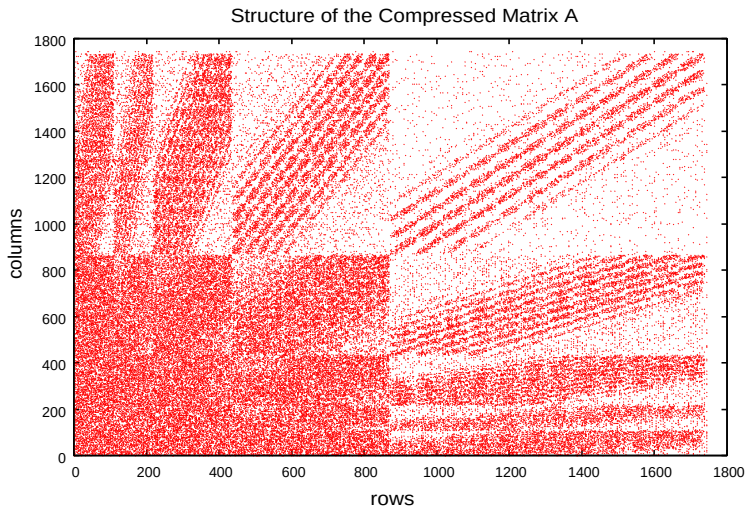


Figure 3.8: Matrix A after Wavelet transform

$$\mathcal{W}\,\Phi = \mathcal{W}\,A\,\mathcal{W}^T\,\mathcal{W}\,Q \tag{3.9}$$

After this transformation the matrix A is still dense. To transform A into a sparse matrix we introduce a threshold value, below which the absolute value of coefficients is set to zero. The structure of the compressed matrix A is shown in figure 3.8. A back-transformation using the sparse matrix representation still contains the essential information. The Wavelet transform of A has to be calculated only once at the beginning of the simulation because the distance between grid points is kept constant during time. The time dependence is completely shifted to the vector $\mathcal{W}Q$. Using this approach the computational complexity is reduced to $O(N)$ [23].

3.3.2 Calculation on the cell processor

Although the Wavelet transform of A has to be calculated only once, it is nevertheless the most time-consuming part of the whole calculation. So the question is, if we can use the cell processor to calculate the following triple matrix multiplication as efficiently as possible:

$$\tilde{A} \;= W\,A\,W^T \;\;\Rightarrow\;\; \tilde{A}_{ij} = \sum_{k,l} W_{ik}\cdot A_{kl}\cdot W_{jl} \tag{3.10}$$

To answer this question we first of all have to look at the structure of the matrices. The inverse-distance matrix A is dense and symmetric with dimension $N \times N$ where $N = np\cdot np\cdot np$ is the number of grid points and np is the number of grid points in one dimension. The elements of matrix A can be calculated as follows:

$$A_{ij} = \begin{cases} \frac{1}{|r_i - r_j|} = \frac{1}{a\cdot\sqrt{(ix-jx)^2+(iy-jy)^2+(iz-jz)^2}} & ,\; i \neq j \\ 0 & ,\; i = j \end{cases} \tag{3.11}$$

where

$$\begin{aligned} ix &= i \,\%\, np \\ iy &= (i \,\%(np \cdot np))/np \\ iz &= i/(np \cdot np) \end{aligned} \tag{3.12}$$

In this equation a is the width of one grid cell, i and j are the indices of the elements of matrix A, ix, iy, iz are the corresponding coordinates of matrix index i in the three-dimensional $np{\cdot}np{\cdot}np$ grid. Note, that the matrix A contains a lot of redundant information. More precisely many entries are the same. That is because the three-dimensional grid of the simulated system has to be mapped onto the two-dimensional matrix A as shown in equation 3.12. In this three-dimensional grid many particles have the same distances, so that there are only $O(N)$ different distances. That means that it is enough to store $O(N)$ values instead of $O(N^2)$ and get the right distance by mapping the two-dimensional addressing of the A matrix back onto the three-dimensional addressing of the systems grid.

The Wavelet matrix $\mathcal{W}$ is sparse and has a kind of band structure (see figure 3.9). Therefore it is well suited to store it via Compressed Sparse Row (CSR) format.

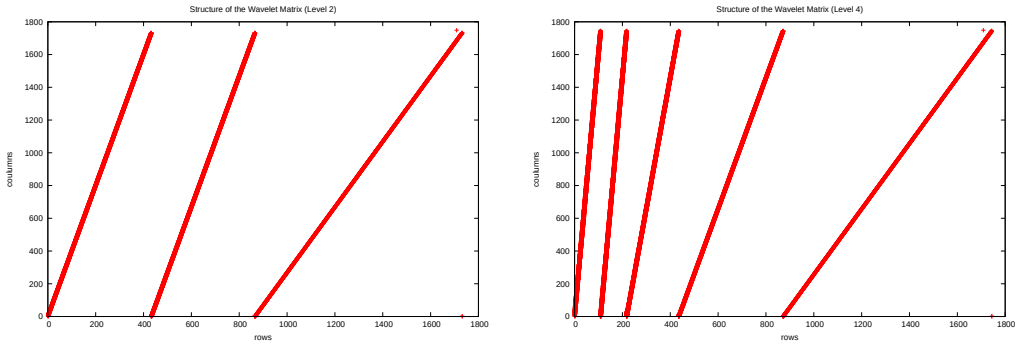


Figure 3.9: Structure of the wavelet matrix $\mathcal{W}$ (Haar Wavelet)

Due to the Wavelet matrix $\mathcal{W}$ we have an algorithm which depends on sparse linear algebra operations. Studies have shown that the cell processor is not best suited for sparse matrix operations [7]. Sparse linear algebra operations have a very irregular memory access pattern, make heavy use of indirect addressing and are inherently memory bound. So the operation's performance is limited by the speed of the bus. It have been shown that with sparse matrix operations an efficiency of at most 6% of the peak performance is possible to achieve. That corresponds to 12.8 Gflops [7]. But nevertheless, 12.8 Gflops is a remarkable speed which is higher than the speed of conventional processors.

3.3.3 Memory efficient algorithm vs. calculation efficient algorithm

In this work two different algorithms are implemented, one algorithm that is more efficient on memory usage and another one with a more efficient calculation. The first approach was to calculate the elements of A on the SPU to minimize storage space and avoid DMA transfers. Each element of the result matrix $\tilde{A}$ is calculated in one step as shown in equation 3.10 where A_{kl} is calculated as shown in equation 3.11. This approach considers that for larger problem sizes the matrix A becomes very large and contains more redundant information. Regarding the fact that the memory of the PPU is limited to 1 GB and the local store of the SPU is limited to 256 KB the amount of data is an important aspect. This approach is efficient on the usage of memory space but it turned out to be not very

efficient on the speed. Measurements showed that the number of DMA transfers causes no problem but the time for the calculation of the triple matrix product was surprisingly high. A closer look at the operations needed for the calculation of an element of matrix A yields a first explanation. To calculate the inverse distance between particles division and square root operations are necessary which need several cycles on the Cell. Another reason is that e. g. for $N = 1744$ and a Daub4 basis, every element of A is calculated about 83 times on average during the whole matrix multiplication. Measurements of the time for DMA and calculation yield that the DMA transfer is about 10^4 times faster than the calculation of the triple matrix multiplication on the SPU. In this case double buffering would not have much impact on the performance.

To avoid the redundant calculation of the elements of matrix A, another approach was to calculate the complete matrix A only once on the PPU and load it line by line onto the SPU. The result matrix $\tilde{A}$ is calculated column by column in two steps as follows:

$$\tilde{A} \;= W\, A\, W^T \;\;\Rightarrow\;\; \tilde{a}_q = W \times (A \times w_q^T) \tag{3.13}$$

In this equation $\tilde{a}_q$ is the q-th column of the result matrix $\tilde{A}$ and w_q^T is the q-th column of the transposed Wavelet matrix W^T. In the first step the intermediate result $b_q = A \times w_q^T$ is calculated. In the second step the q-th column of $\tilde{A}$ is calculated via $\tilde{a}_q = W \times b_q$. These steps have to be repeated for each of the N columns. In this approach the calculation was reduced but the number of DMA transfers increased. For every calculation of b_q the whole dense matrix A has to be loaded. Nevertheless, it is faster than the first approach.

In the left graphic of figure 3.10 the excecution time for the triple matrix multiplication of the two approaches is compared to the execution time of the original Fortran implementation running on the PPU and on the frontend. Obviously, the second approach that reduces the calculation is much faster than the first one which optimizes the usage of memory. The right hand graphic shows the speedup of the two approaches compared to the execution time using one SPU. One can see that the first approach has a better scalability than the second one.

One disadvantage of the second approach is, that we need more memory to store the whole dense matrix A with its redundant information. Another disadvantage is, that the symmetry of the matrix A is not exploited. For future research it would be interesting to optimize this approach in exploiting the symmetry and improve the storage to avoid redundant information. The advantage of this approach in contrast to the first approach is, that it is more general. The Wavelet transform is not specific for the Coulomb interaction but it can be applied to every quadratic matrix A. So this implementation can be applied to a broader spectrum of applications, e. g. it can be used in image processing.

3.3.4 Summary

The efficient implementation of sparse matrix operations on the Cell is difficult. The computational power of the processor cannot be fully exploited. The reasons are well known. The SPUs are very fast arithmetic units which are constructed to handle a range of SIMD instructions. That is a problem with sparse matrices because they often make heavy use of indirect addressing due to the fact that storage formats like CSR are used. So these algorithms are hard to vectorize. Another problem that results from the storage format is the memory alignment. It is very hard to follow a regular pattern to memory on aligned

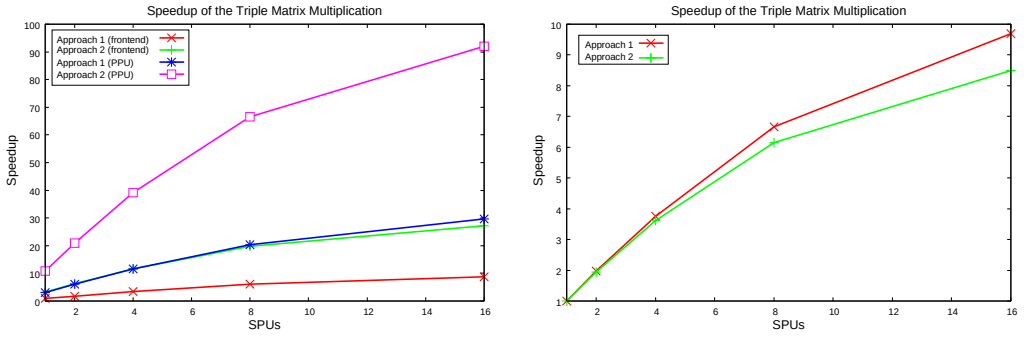


Figure 3.10: Speedup of the two approaches ($N = 1744$, Daub4 basis): compared to the original Fortran implementation executed on the frontend and on the PPU (left) and compared to the same implementation running with one SPU (right)

locations. Because of the indirect addressing it is also difficult to reduce the number of branches.

All these aspects also appear in the implementation of the triple matrix multiplication. Additionally we have to consider, that the matrices can become very large. So the ideal solution will be an algorithm which can be calculated efficiently on the Cell and which optimizes the usage of memory. In this work we introduced two implementations. The first one was efficient in memory usage but the speed was comparatively slow. The second one was faster because of the reduction of calculation but needed more memory. All together the second implementation is more general. It can also be applied for other applications like image processing while the first algorithm is specific for Coulomb interactions.

3.4 Cell Superscalar *(Liang Yang)*

3.4.1 Introduction

CellSs is proposed as a programming model for multicore processors. Based on a source to source compiler and a runtime library the programming model generates the code for both PPE and SPEs from a sequential code with simple annotations. Using CellSs requires that the application is composed of coarse grain functions and that these functions do not have collateral effects (only local variables and parameters are accessed). With CellSs, the annotation preceding a coarse grain function (task) does not indicate the parallel region as OpenMP does but just indicates the direction of parameters of this function. During runtime, CellSs builds a data dependency graph by collecting the information about these parameters and schedules independent tasks to different SPEs concurrently. As well, all data transfers required for the computations on the SPEs are automatically handled by the system.

3.4.2 Examples and results

Dense matrix multiply

For dense matrix multiply, the common block algorithm was used. With the code displayed

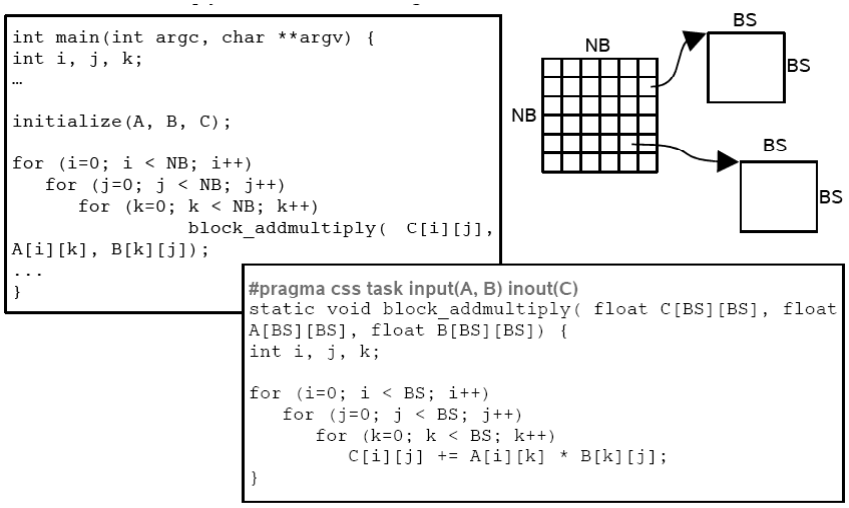


Figure 3.11: Blocked dense matrix multiply

in figure 3.11, we can achieve about 2 GFLOPS when it is running on 8 SPEs. If vector computing is adopted and the calculation on the SPE properly pipelined, up to 33 GFLOPS can be achieved. As presented in figure 3.12, both the speedup for scalar and vector computing increases almost linearly along with the number of SPEs indicating that the communication between PPE and SPEs is matched with the computation on SPEs and is not the bottleneck of the whole operation.

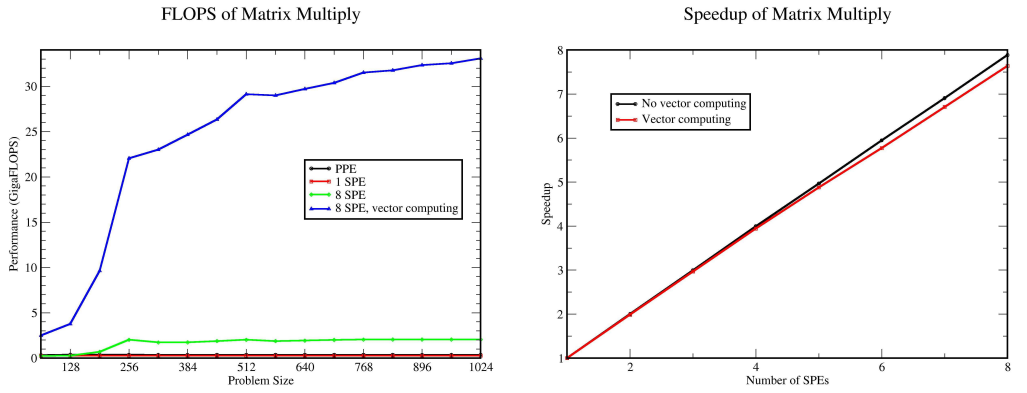


Figure 3.12: Performance and speedup for matrix multiply

3-D Jacobi

In this application, the computation below is executed.

$$A^{n+1}_{i,j,k} = f_{i,j,k} + A^{n}_{i-1,j,k} + A^{n}_{i+1,j,k} + A^{n}_{i,j-1,k} + A^{n}_{i,j+1,k} + A^{n}_{i,j,k-1} + A^{n}_{i,j,k+1} \tag{3.14}$$

In computation, the whole matrix is partitioned into smaller blocks to fit in the local memory space of the SPE. While the data of one block is sent to SPE, the 6 neighbor planes are grouped together and are also sent to SPE. The performance curves are depicted in plot 3.13. Here, the performance improvement gained from vectorization is not so obvious as in matrix multiply because the routines are not carefully pipelined and the time for DMA transfer exceeds the time for SPE computing. The largest speedup for vector Jacobi occurred while 5 SPEs are used concurrently.

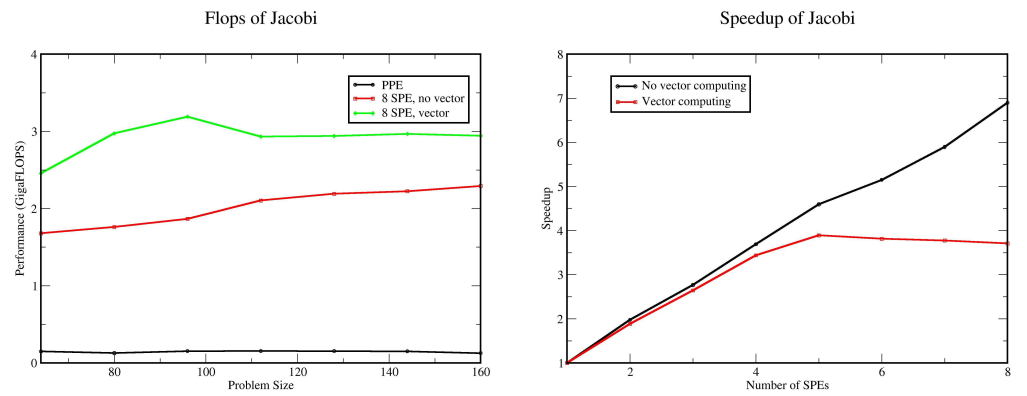


Figure 3.13: Performance and speedup for Jacobi method

Triple matrix multiply

Triple matrix multiply represents the compute intensive kernel of a Wavelet based evaluation of Coulomb potentials in molecular systems.

$$\tilde{A} \ = W \ A \ W^T \tag{3.15}$$

In this equation, W is a sparse wavelet matrix with a band structure. For the code compiled by CellSs, the dense matrix A is computed first and kept in main memory. Then two steps of sparse matrix-matrix multiply are executed. Results gotten with that scheme are graphed in figure 3.14.

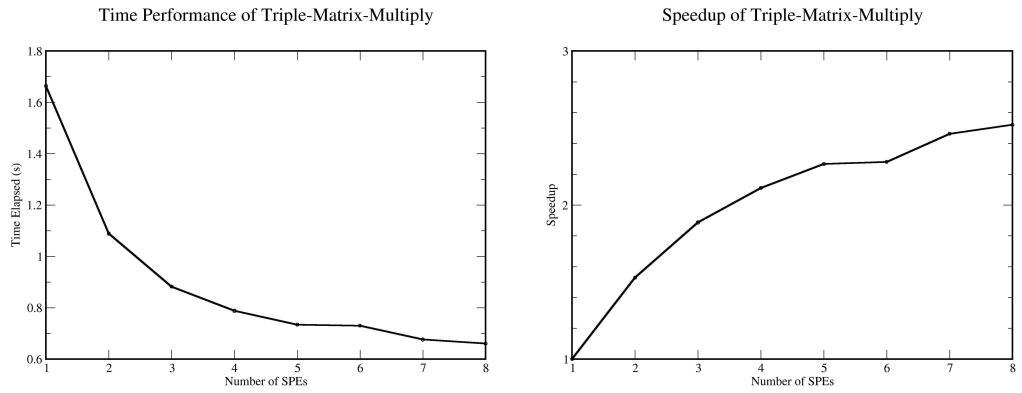


Figure 3.14: Performance and speedup for triple matrix multiply

3.5 MPI on Cell *(Norbert Eicker)*

In order to build clusters out of Cell nodes a highly capable interconnect is necessary. To enable tightly coupled parallel applications to make efficient use of a cluster system this network has to provide both, low latency and high bandwidth.

On the QS20 Cell blade for this reason InfiniBand daughter-cards based on Mellanox silicon are available. Since InfiniBand is a widespread technology as a high speed / low latency interconnect in the context of cluster systems, we did not expect severe problems concerning the availability of corresponding software – even if the combination of hard- and software in use within JUICE is not officially supported by the OpenFabrics software stack.

Nevertheless, we had to start our work on MPI within the project with a complete different solution: Due to the fact that the InfiniBand daughter-cards where late we started our experiments by implementing a MPI system based on the on-board Gigabit-Ethernet NICs of the QS20. While this was quite helpful in order to start with software development early, the achievable performance numbers for this setup were – as expected – sub-optimal. Even worse, the NICs integrated in the QS20's southbridge turned out to be quite limited in performance. While ParaStation MPI on other processor and Ethernet platforms has a latency of $10 - 20\mu$sec, the best results achievable on JUICE were almost 40μsec. Also the single-direction bandwidth of slightly more than 70 MB/s is by far off the expected results of > 100 MB/s. So we were lucky to finally be able to start with real hardware in May 2007.

In order to connect the InfiniBand host channel adapter (HCA) to the Cell system we made use of the PCIe $4x$ socket available on the QS20 blade. By saying that it is clear that for the InfiniBand bandwidth the expectation should not be set too high: On more mainstream platforms like Intel XEON or AMD Opteron the full performance of this kind of interconnect can only be achieved, if at least a $8x$ socket is available.

Different from other BladeCenter solutions provided by IBM the HCA are not connected directly to the BladeCenters backplane but the external connector is attached to the front bezel of the blade. From here ordinary CX4 cables are used to setup the connection to the Voltaire ISR 9024 switch.

The software-stack we chose is the 4th release candidate of the Open Fabrics Enterprise Distribution (aka OFED 1.2rc4). Since there were no precompiled packages for the Cell platform running Fedore Core 6 – the Linux distribution running on our blades at that time – we had to compile the software by ourself. It turned out that this worked with only slight modifications of OFED's build system.

3.5.1 Results for InfiniBand

Fig. 3.15 shows the results we got from two tests of the Intel MPI Benchmark (IMB). This synthetic low-level communication benchmark is used in order to experimentally determine the two fundamental parameters of our interconnect: latency and bandwidth.

All data presented in this section results from four independent runs of the same benchmark. The actual data-points in the diagrams are the resulting average. Correspondingly the error-bars depict the variation of these runs.

The left diagram of fig. 3.15 gives latency results for the `pingpong` test. The four lines depicted are from four different benchmark setups. Each setup uses two processes running on two different nodes of the cluster sending messages to each other. The difference between

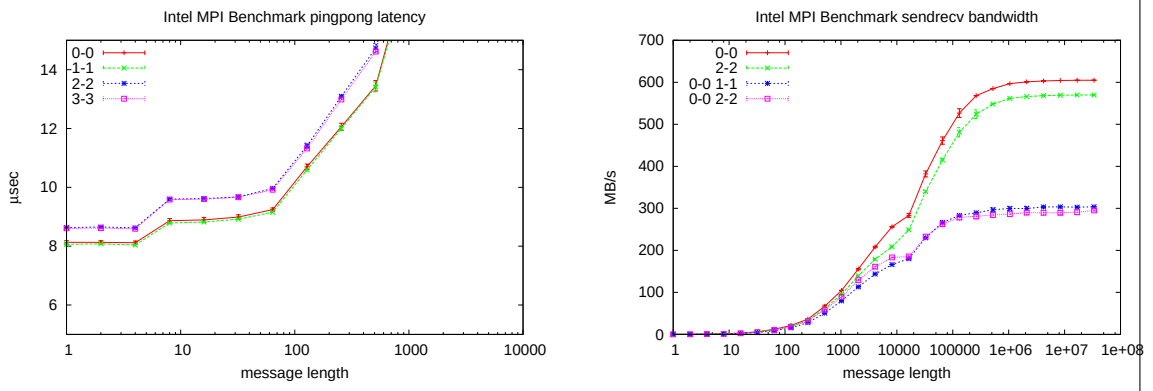


Figure 3.15: MPI performance for InfiniBand on QS20 blades.

the setup is that the processes are pinned[1] to one of the four different (virtual) cores of the blades[2]. Since the cores with numbers 0 and 1 in fact are the same physical core, we expect their results to be identical. The diagram clearly shows that this expectation holds. The same is true for the cores with number 2 and 3. On the other hand the latency for communication between cores 0 is different from the one we get between cores 2. We interpret this behavior as a result of the fact that the southbridge is directly connected to the CBE hosting core 0. The extra latency experienced by the other CBE results from the extra time the messages need from one CBE to the other.

Furthermore the absolute results for InfiniBand latency are not exhilarating, too. A latency of $> 8\mu$sec is about a factor 2 larger than experienced on other processor platforms. This might either be due to the sub-optimal southbridge of the QS20 or result from the strict in-order design of the CBE's PPE.

Similar findings are shown in the right diagram of fig. 3.15 for the bandwidth. The numbers presented here are created by IMB's `sendrecv` test. Let us first concentrate on the two lines marked as `0-0` and `2-2`. Again, the processes are pinned to the cores 0 and cores 2, respectively. For large messages the bandwidth achieved on core 2 is almost 6% smaller than the one seen on core 0. Again we hold the BIF connecting the two CBEs responsible for this result.

The absolute bandwidth for the (bi-directional) `sendrecv` test is disappointing, too. In fact it is only 4% more than the 582 MB/s achieved by the one-directional `pingpong` test. Again we think the southbridge is responsible for that. Since only a PCIe $4x$ slot is available bandwidth is already missing on the hardware level.

3.5.2 Local communication

Since the QS20 blades host two CBEs local communication is an interesting topic, too. We expect many parallel applications to make use only of one CBE per process, i.e. there will run two processes on each node of a Cell cluster. Therefore these processes have to exchange messages locally.

[1] In fact not only the processes are pinned but also memory-binding to the corresponding NUMA node is enabled.

[2] While the QS20 blade has two CBEs with one PPE, each PPE provides symmetric multi-threading (SMT) capability. Therefore, from the operating system's point of view each of the two PPEs looks like two. Thus the Linux kernel reports four processors in `/proc/cpuinfo`.

While setting up our tests we experienced some strange behavior of our parallel applications: During the call of `MPI_Init` the processes stopped running for some minutes. After that the program started up normally and we got results quite similar to the ones we got from the Ethernet-based ParaStation MPI – as we expected for local communication. It took some time to find out that this is due to strange process pinning done by the MPI implementation of MVAPICH included within the OFED. In fact this MPI implementation tries to pin processes to exclusive cores. The strategy is to use core 0 for the first process sent to a node, core 1 for the second one, etc. Unfortunately on the QS20 the virtual cores 0 and 1 belong to the same physical one. Setting up some shared memory segment for local communication now seems to trigger some race-conditions between the two processes pinned to the same core leading to a significant delay of a few minutes.

By setting some magic environment variables[3] we were able to switch of MVAPICH's process pinning. This prevents the behavior described above and reduces startup time to the normal means.

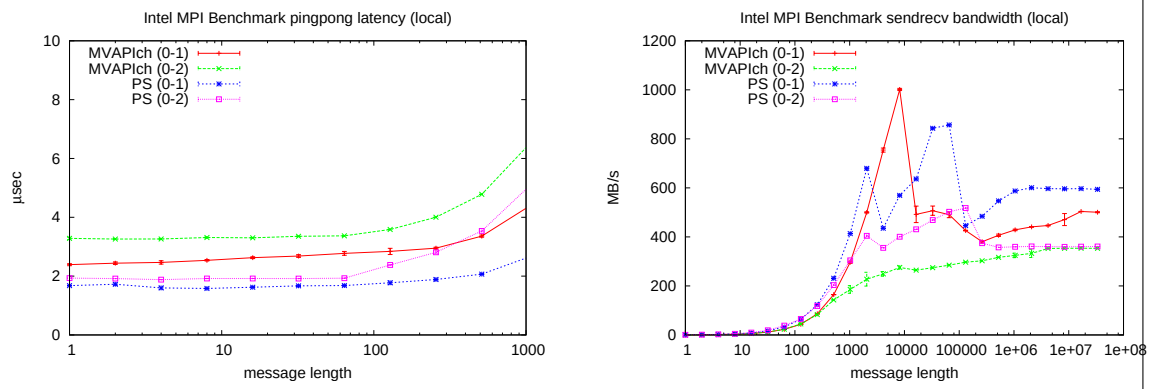


Figure 3.16: MPI performance for local communication on QS20 blades.

Figure 3.16 present results for local communication corresponding to fig. 3.15 for real communication via InfiniBand. Additionally it compares two implementations of the MPI standard concerning local communication: MVAPICH as contained in the OFED stack on the one hand, ParaStation MPI (denoted as PS) on the other hand. The latter is a highly optimized MPI stack provided by ParTec and used as a reference implementation for shared memory communication.

Again, we expect two different classes of results: Doing communication between cores of the same socket and between different sockets[4]. The left diagram of fig. 3.16 presents the latencies as determined by IMB's `pingpong` test. It shows clearly that ParTec's implementation is outperforming MVAPICH by more than 50% in the relevant case.

The bandwidth results displayed in the right diagram of fig. 3.16 approve this tendency. Furthermore, the absolute bandwidth we were able to achieve using our MPI tests is by far off the 25.6 GB/s the CBE's memory interface is able to provide to the SPEs and not even near the bandwidth of the BIFs connecting the CBEs. Most probably this is due to the very special capabilities of the PPE. Since implementation of the PPC core within the CBE is a strict in-order one it is quite different from any other current node of a HPC-cluster. As a comparison, on the JS21 blade hosting normal PPC970 CPU we were able to achieve more

[3] `export VIADEV_ENABLE_AFFINITY=0`

[4] While the first class is quite artificial since both virtual cores map in fact to the identical physical one, the latter is more relevant in practice

than double the bandwidth with the same software even on a much less capable memory subsystem.

3.5.3 Conclusions

We have investigated the MPI capabilities of the QS20 blades equipped with IBM's InfiniBand option. Our results show that the current hardware is not well suited to serve as a HPC platform. This is due to many reasons:

- The southbridge of the QS20 only provides a PCIe $4x$ socket. This is definitely too narrow to serve a normal $4x$ InfiniBand HCA.
- The implementation of the southbridge seems to be sub-optimal. The total latency for MPI communication is double the one observed on any other hardware platform.
- Also the strict in-order implementation of the PPC core seems to introduce additional problems – at least for local communication within a node.

IBM promises to address this problems in the next generations of their Cell blade by implementing a new southbridge.

In addition to that, a radical new design of a communication interface might be the right answer to the questions introduced by the new design of the CBE: Since almost all of the computing-capabilities of the Cell are concentrated in the SPEs, at least on a logical level most communication operations are messages to be sent between SPEs. As long as the interconnect is not directly accessible by the SPEs, each communication operation has to be composed out of three steps:

1. Copy data to send from the SPE's local store to the memory. This will occupy precious memory bandwidth.
2. Initiate sending the actual message into the memory of the remote node. Again, memory bandwidth is occupied on both sides.
3. Retrieve the message from main memory into the local store of the destination SPE. This occupies the memory interface, too.

Thus it would be much nicer if the communication interface would be connected to the CBE's BIF in a way that it is directly accessible by the SPE. This will release some of the pressure on the memory interface and at least in principle enable much smaller latencies.

3.6 MPI performance of matrix-matrix-multiplication *(Inge Gutheil)*

3.6.1 Matrix-matrix-multiplication

Matrix-matrix-multiplication is the application with the best computation to memory access ratio. Thus it is the candidate to get almost peak performance on modern computers where memory access is one order slower than CPU performance.

The starting-point

The SDK 2.1 is shipped with some example programs one of which is a matrix-matrix-multiplication program which achieves almost peak performance on a single cell blade. It computes $C = A \cdot B$ for square matrices with sizes of 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048, or 4096x4096 and up to 16 SPEs per blade with a single PPE. To achieve this very good performance the matrices are partitioned into 64×64 blocks and these blocks are stored row wise in the main memory. The blocks of C to be computed form a work pool and each SPE takes one of these blocks from the work pool marking the index of this block as being worked on. It then reads the necessary blocks of A and B and accumulates the block of C in the local store until it is computed. Then the block of C is written to the main memory and C's index is marked as computed. Each SPE takes new blocks to be computed until no more blocks are available. This leads to very good load balance even if the SPEs do not start at the same time.
When more than 8 SPEs are used the matrices to be multiplied are copied to both parts of the main memory so that each SPE only reads from the part of the main memory which can be accessed without using the BIC. The result matrix is held only once. With this constellation the performance for the largest matrix size and 16 SPEs was 379 GFLOPS.
We changed the program slightly to allow all multiples of 64 as matrix size and to allow rectangular matrices, too. In the original program there is an option to test the result for correctness which makes it necessary to store another matrix. In order to enlarge the problem size we wrote a second version without result test needing only the matrices A and B (still stored twice to use 16 SPEs) and the result matrix C. In this version 5 matrices of size n^2 have to be stored in 1 GB of main memory. This means that we expected good performance only for matrices of size $n < 7000$.

Farming

In the next step we did some kind of farming for the multiplication of larger matrices with more cell blades. Each of the np blades computes a part of size $n/np \cdot n$ of the whole matrix by multiplying the $n/np \cdot n$ part of A with the whole matrix B. This means that each blade has to store its part of A and C and the whole matrix B. In this approach we only used 8 SPE per PPE but sometimes both PPEs per blade so that we could use up to 24 MPI processes. There was no need to store matrices twice during these tests but n/np had to be a multiple of 64 and each MPI node had to store 2 matrices of size $n/np \cdot n$ and one matrix of size $n \cdot n$ in the main memory, now being limited to 512 MB. This means that the memory limits for the matrix sizes were the following:
$np = 2 :\ n < 8192,\ \ np = 4 :\ n < 9459,\ \ np = 8 :\ n < 10362,$

$np = 16 : \quad n < 10923, \quad np = 24 : \quad n < 11130$. From figure 3.17 we can see that the performance decreases shortly before that limit is reached.

Matrix-matrix-multiplication with MPI communication

We used an easier parallelization than the standard SUMMA[12] algorithm. In the SUMMA algorithm the communication part consists of broadcasts along rows and columns of a rectangular processor grid, which should be as square as possible. If the number of processors np is square in the SUMMA algorithm $\sqrt{np}$ steps are performed (each one perhaps not in one step but in several smaller ones so that not too much memory for MPI communication is needed) and in each step one of the processors does two broadcasts, one to the $\sqrt{np}-1$ processors in its row and one to the $\sqrt{np}-1$ processors in its column. In each broadcast n^2/np elements are sent. The other processors do at most one broadcast along their processor row or column. As the execution time of the whole parallel program can not be less than the time for the slowest processor, in each step the time for the two broadcasts have to be taken into account which gives an amount of $2\sqrt{np}$ broadcasts of n^2/np Elements sent to $\sqrt{np}-1$ processors (see [6]).
In our simpler algorithm only the matrix B is sent around by MPI_Sendreceive_replace in a ring topology. The matrices are distributed in a row block manner. Each processor keeps its part of A and C. In the first step each processor copies its part of B to a send-receive buffer and does the MPI_Sendreceive_replace with its predecessor. After having done the computation of the correct part of the local part of A times local part of B the buffer received is copied to B and MPI_Sendreceive_replace is done again with the buffer. The multiplications are done with B again. In the last step only MPI_Sendreceive_replace is done the npth time and the matrix received is copied to B so that each processor has the same part of B in the end as in the beginning.
In [6] the communication time per step for the SUMMA algorithm is given without latency as

$$t_{comm} = \frac{msgsize \cdot nmsg}{bw}$$

where $msgsize$ is the length of the message sent in that step and $nmsg$ is the number of messages per step and bw is the bandwidth for broadcast. In total n/nb steps are performed where nb is some well suited block size. If latency plays an important role the number of steps must not be too large, i.e. the blocks mustn't be too small.
For the case of a square processor grid with np processors in each step there is one processor which has to do two broadcasts of messages of length $\frac{n}{\sqrt{np}} \cdot nb \cdot fpsize$, one along its processor row and one along its column. Each of this broadcasts results in $\sqrt{np}-1$ messages for small np or in $\log_2 \sqrt{np}$ messages for a good implementation of broadcast and $np \geq 16$. This results in $msgsize = \frac{n}{\sqrt{np}} \cdot nb \cdot fpsize$ and $nmsg = 2(\sqrt{np}-1)$. In total with n/nb steps this gives

$$t_{comm} = \frac{4n^2}{bw} \frac{2(\sqrt{np}-1)}{\sqrt{np}}$$

and

$$\frac{4n^2}{bw} \leq t_{comm} \leq \frac{8n^2}{bw}$$

For a non square processor grid with $np = np_1 \cdot np_2$ the formula for the communication time for SUMMA is more complicated and the total communication time can be

higher (see Appendix B). Our algorithm depends on a processor ring topology and the possibility to send messages independently so that all processors in the ring can do the MPI_Sendreceive_replace in parallel. If that is true and we denote by t_{srr} the factor by which MPI_Sendreceive_replace per message is slower than a simple send we get np steps with the amount of communication mentioned above which makes

$$t_{comm} = \frac{4n^2}{bw} t_{srr}$$

in total for the algorithm with MPI_Sendreceive_replace. Thus it depends on t_{srr} and the choice of the processor grid for SUMMA whether the communication time for SUMMA or for the simple matrix multiplication algorithm is higher.
Nearly all the communication can be overlapped by computation if there is enough computation to do. This is the case, if the computation time per step is higher than the communication time per step.
For our algorithm this is the case if

$$\begin{aligned} t_{comp} = \frac{2n^3/np^2}{peak} &\geq t_{comm} = \frac{4n^2/np}{bw} t_{srr} \\ \iff \frac{n}{np \cdot peak} &\geq 2\frac{t_{srr}}{bw} \\ \iff n &\geq 2peak \cdot \frac{t_{srr}}{bw} \cdot np \end{aligned} \tag{3.16}$$

3.6.2 Results of performance measurements

Performance on a single blade using one PPE and all SPEs

First we measured the performance of the only slightly modified workload program for square matrices with n up to 7168 and could see that for $2432 \leq n \leq 6400$ more than 370 GFLOPS were achieved for any n. For larger n we found more than 370 GFLOPS for $n = 6528,\ 6656$, and $n = 6784$ whereas for $n = 6464,\ 6592$, and $n = 6720$ only 300 or 310 GFLOPS could be achieved. For larger n the performance broke in dramatically from almost 200 GFLOPS for $n = 6848$ to only 73 GFLOPS for $n > 7000$. This result agrees well with the memory limit for the matrix size mentioned before.

Performance with farming

For the tests with farming we only measured the performance for square matrices with sizes up to 9216. The results can be seen in figure 3.17. The performance achieved is always more than 90 % of the peak performance.
The memory limits are a bit smaller than expected. With two PPEs and 8 SPEs per PPE the maximum performance achieved is 380 GFLOPS which is a little less than the performance with one PPE and 16 SPEs. The performance begins to decrease with $n = 7296$ and breaks in for $n > 7900$ where it is less than 200 GFLOPS. With four PPEs we get almost perfect scaling for $4600 < n < 8700$ where we get 760 GFLOPS. For $n > 9000$ the performance is less than 200 GFLOPS again. The region with a performance of about 1500 GFLOPS with 8 PPEs is $3584 < n < 8704$, with 12 PPEs more than 2260 GFLOPS were achieved for $n > 3840$, with 16 PPEs more than 3000 GFLOPS were achieved for $n > 4096$, and with 24 PPEs more than 4500 GFLOPS for $n > 4608$. As we did not measure problems larger

than 9984 we did not see the performance break-in for 12 and more PPEs.

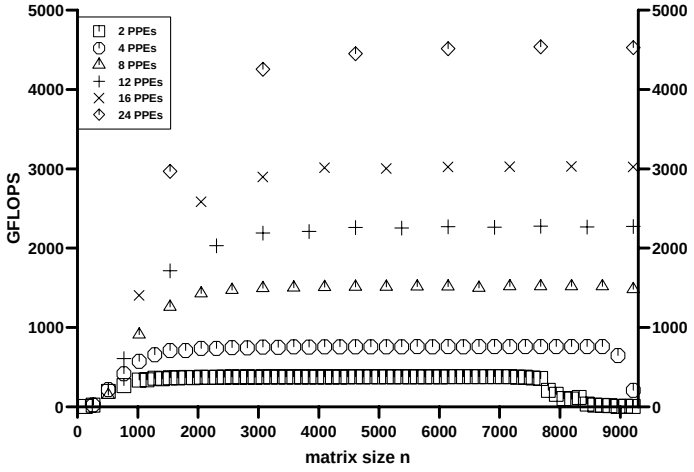


Figure 3.17: Matrix-matrix-multiplication on JUICE, farming, using various numbers of PPEs each with 8 SPEs

Performance with MPI communication

We measured the performance of the simple algorithm without overlap and Gigabit Ethernet and it was terribly slow. With InfiniBand it became much better, and when we overlapped communication with computation, the performance became even better. To overlap communication by computation a copy operation of a part of the matrix has to be done which is also rather slow, thus the performance gain of the overlapping could only be seen for large n.
The measured bandwidth for broadcast as well as simple MPI_Send is $bw = 600$ MB/sec. We measured the communication time for the SUMMA algorithm without computations and found for $n = 1024$ and $np = 4$ as best time measured 13.178 ms compared to

$$\frac{2^{10} \cdot 2^{10} \cdot 4}{600 \cdot 2^{20}}\text{sec} = \frac{1}{150}\text{sec} = 6.67\text{ms}$$

expected time.
Isolated measurements of the performance of MPI_Sendreceive_replace showed that

$$\frac{bw}{t_{srr}} \approx 1.55 \cdot 10^8 \text{Byte/sec} = 155 \text{ MB/s} \tag{3.17}$$

which means that t_{srr} is almost 4 compared to the 600 MB/sec for broadcast but only about 2 compared to the communication time measured for the SUMMA communication. Thus it has further to be investigated whether the SUMMA algorithm should be preferred. Measurements of the total communication time for the simple algorithm showed that for matrix sizes of $n = 2048$ and larger the communication time measured within the complete matrix-matrix multiplication was slightly less than 4 times the communication time of one

MPI_Sendreceive_replace cycle with 4 processors on 4 nodes. This is a little surprising but perhaps repeated MPI communication of the same type are faster because some overhead is reduced. For the complete programs we still have to add the copy times, which are about half the times for the communication in the simple algorithm. We shall thus take the $\frac{bw}{t_{srr}}$ measured without computations (equation 3.17) to guess the problem sizes necessary to overlap communication by computation.
The peak floating point performance with 8 SPEs is approximately 200 GFLOPS. Thus looking at the problem sizes necessary to overlap communication by computation (equations 3.16 and 3.17) we see the following: (The additional overhead for the first copy operation is not taken into account.)

$$\begin{aligned} 2peak\frac{t_{srr}}{bw} &= 2 \cdot 2 \cdot 10^{11} \cdot \frac{1}{1.55 \cdot 10^8} \approx \frac{4}{1.55} 10^3 \approx 2600 \\ &\Longrightarrow n \geq 2600 \cdot np \end{aligned}$$

The values of n resulting from this condition are given in the second column of table 3.4. On the other side the amount of memory available per blade is 1 GB=2^{30} Byte and for the simple matrix multiplication each processor has to store four matrices of size $n \cdot n/np \cdot sizeof(float) = 4n^2/np$ Byte. MPI_Sendreceive_replace needs at least one more buffer of the same size, from our measurements we assume that it even needs twice that amount of memory, which means that each MPI-process has to store $6 \cdot 4n^2/np = 24n^2/np$ Byte. In order to fit into the main memory of 2^{30} Byte there is a limit of

$$n^2 < \frac{np \cdot 2^{30}}{24} \Longleftrightarrow n < \sqrt{np/24} \cdot 2^{15}$$

(first column of table 3.4) if we use one MPI process per blade.

np	maximum n	minimum n for overlap	maximum n measured without swapping	maximum performance measured [GFLOPS]
2	9460	5200	9728	335 ($n = 9600$)
4	13400	10400	13568	581 ($n = 13312$)
6	16400	15600	16512	809 ($n = 16512$)
8	18900	20800	18944	1012 ($n = 18944$)
10	21200	26000	21120	1157 ($n = 21120$)
12	23200	31200		

Table 3.4: Limitations for n

If we tried to use two MPI processes per blade, each with 8 SPEs, the memory limit per process would be half the memory, thus the maximum size would not allow the overlap of communication by computation. The same holds if we take 16 SPEs per MPI process. This doubles computation speed and with that the minimum n to overlap.

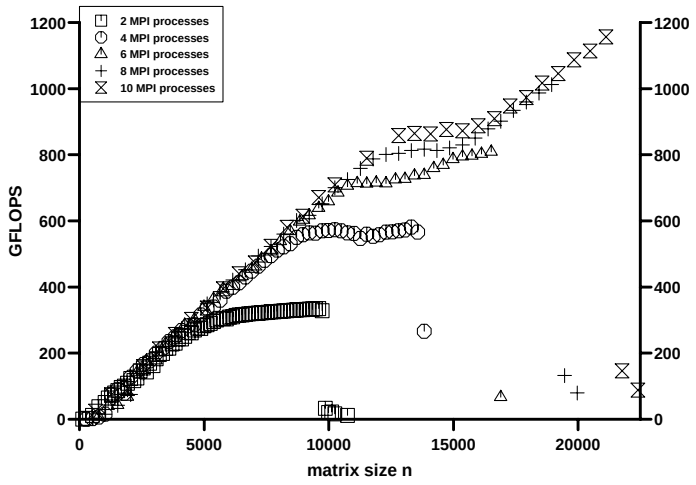


Figure 3.18: M atrix-matrix-multiplication on JUICE, MPI communication, overlapped by computation

The performance results for the matrix-matrix multiplication using MPI_Sendreceive_replace with overlapping communication by computation are depicted in figure 3.18.
Table 3.4 shows that even for one MPI process per blade and 8 SPEs per MPI process the memory is only slightly larger than necessary for overlap of communication by computation for up to 6 MPI processes, for more MPI processes the complete overlap of communication by computation is not possible.

Conclusions

Starting in January 2007, the JUICE project experienced a rapid development. The Cell platform was in great demand and shortly after the cluster setup there were many scientists who applied for access to the cell nodes. The tutorials and example programs shipped with the SDK were a good starting point. Software projects evolved and in May, first results were presented at the Cell Cluster Meeting 2007 in Jülich [16].

The Cell chip is a very promising architecture for compute-intensive tasks like the simulation of strongly correlated materials. The current cell blades, however, come with a complicated NUMA architecture which makes it difficult to reliably achieve high performance. We feel that it would be completely sufficient for our purposes to have a single chip per board/node. Also the operating system does not quite fit the philosophy of the Cell chip. It is a conventional full-featured Linux system, where instead a lightweight kernel on the compute nodes (CNK) similar to the one found on BlueGene super computers would be more appropriate. An important consideration for the development of Cell architectures is the balance of floating point performance, memory access, and network. Currently the small size of the local store is a severe limitation to efficiency.

The Cell broadband engine architecture with its limited local storage and the availability of DMA transfers for load/store operations to/from the main memory is well-suited for the solution of linear systems arising from the discretization of partial differential equations on structured grids. In the multigrid method the smoothing iteration plays an important role and is computationally most expensive. Offloading the Jacobi smoother to the SPUs reveals that the performance benefits a lot from the high available memory bandwidth, although we only get about 5 percent of the possible floating point performance.

The calculation of the Coulomb potential based on a Wavelet transformation requires sparse matrix operations which are difficult to implement on the Cell. Use of storage formats like CSR entails heavy use of indirect addressing. So these algorithms cannot fully exploit the SIMD operations of the SPUs. Memory alignment also turns out to be problematic because it is hard to follow a regular pattern to memory on aligned locations. Additionally we have to consider, that the matrices can become very large.

After the setup of an InfiniBand network parallel computing via MPI became more important. SDK's matrix-matrix-multiplication example was modified and using a version without communication (farming) a performance of 4.5 TFLOPS (90% peak) could be attained. A parallel version of this code with MPI communication achieved 56% of the peak using 80 SPEs on 10 blades (1.157 TFLOPS).

Investigating the MPI capabilities of the QS20 blades equipped with IBM's InfiniBand option revealed some deficiencies. The southbridge of the QS20 only provides a PCIe $4x$ socket. This is definitely too narrow to serve a normal $4x$ InfiniBand HCA. The total latency for MPI communication is double the one observed on any other hardware platform. Also the strict in-order implementation of the PPC core seems to introduce additional problems.

While IBM has launched successor systems of the JUICE components - BladeCenterH and

QS21 blades with double memory size and including a new southbridge - the big step in Cell architecture is expected in 2008 with the enhanced double precision Cell processor as part of the the Cell-accelerated petascale system Roadrunner[19].

Acknowledgement

We would like to express our gratitude to IBM, in particular to Michael Hennecke and Uwe Holzinger, for their efforts in the deployment phase and the setup of the JUICE cluster. As well, we would like to thank all who joined us in our regular meetings or helped us in getting more and more familiar with Cell: Ralph Altenfeld, Ulrich Detert, Daniel Hackenberg, Stefan Krieg, Jan Meinke, Markus Stürmer, Brian Wylie.

Appendix A

QS20 Memory Management

Two variants of dense matrix-matrix-multiplication are considered to reveal some information about the QS20 memory subsystem: The mmikj variant with a regular unit stride and mmijk featuring a kernel with a large stride in the innermost loop.

```
mmikj core loops:
for (i=0; i<size; i++) {
   for (j=0; j<size; j++)
      cmat[i*size+j] = 0.0f;
   for (k=0; k<size; k++) {
      for (j=0; j<size; j++) {
       cmat[i*size+j]+=amat[i*size+k]
         *bmat[k*size+j];
      }
   }
}
```

```
mmijk core loops:
for (i=0; i<size; i++) {
   for (j=0; j<size; j++) {
      cmat[i*size+j] = 0.0f;
      for (k=0; k<size; k++) {
       cmat[i*size+j]+=amat[i*size+k]
         *bmat[k*size+j];
      }
   }
}
```

The package *numactl* allows starting processes with a specific NUMA scheduling or memory placement policy and the package *cellperfctr-tools* provides access to hardware counters, e.g. L1 and L2 cache misses. Performance tests have been run with the following code sequence (SDK-2.1):

```
//cpc -e: events to count: D-cache load miss and L2 loads
//cpc -E: enable performance counter
system("cpc -e DL1_Miss_t0,cache_miss; cpc -E");
system("cpc -z");         //cpc -z: zero the counters
gtime = times(&tbuf);
mmikj(a, b, c, n );
etime1 = (double)(times(&tbuf) - gtime) / (double)sysconf(_SC_CLK_TCK);
system("cpc -r");         //cpc -r: read counters and display on stdout
system("cpc -z");
gtime = times(&tbuf);
mmijk(a, b, c, ln );
etime2 = (double)(times(&tbuf) - gtime) / (double)sysconf(_SC_CLK_TCK);
system("cpc -r");
```

Compilation:

```
ppu32-gcc -m32 -mabi=altivec -maltivec -include altivec.h -O3 -c matrix_mul.c
ppu32-gcc -o matrix_mul matrix_mul.o -m32 -Wl,-m,elf32ppc -lm -lnuma
```

Accepting the default NUMA policy the measured runtimes for the matrix multply variants mmikj (red) and mmijk (magenta) are depicted in figure A.1. In addition, for the mmijk

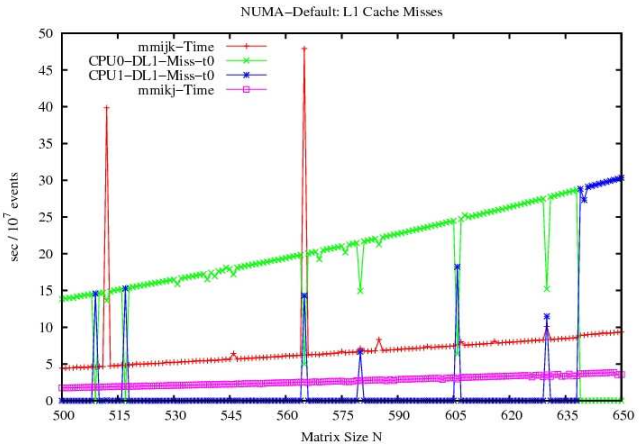


Figure A.1: Default NUMA policy: Runtimes and L1 cache misses

variant (memory access with stride N) the corresponding numbers of L1 cache miss events on cpu0 (green) and cpu1 (blue) are displayed. Cache misses for variant mmikj are negligible and not displayed. The curve for mmikj shows the expected behaviour according to the numerical complexity $O(N^3)$ whereas the curve for mmijk depicts some peaks (N=512, 565,...). The type of cache miss event (green for cpu0 and blue for cpu1) unveils which cpu is currently executing. The measured runtimes for mmikj and mmijk (disregarding N=512, 565) indicate that the L1 cache misses with mmijk are responsible for the higher execution times. The peaks at N=512 and 565 cannot be attributed to L1 cache misses.

Pinning the program execution to CPU 0 and memory allocation to the local memory can be achieved by: numactl –membind=0 –physcpubind=0 ./mmrun

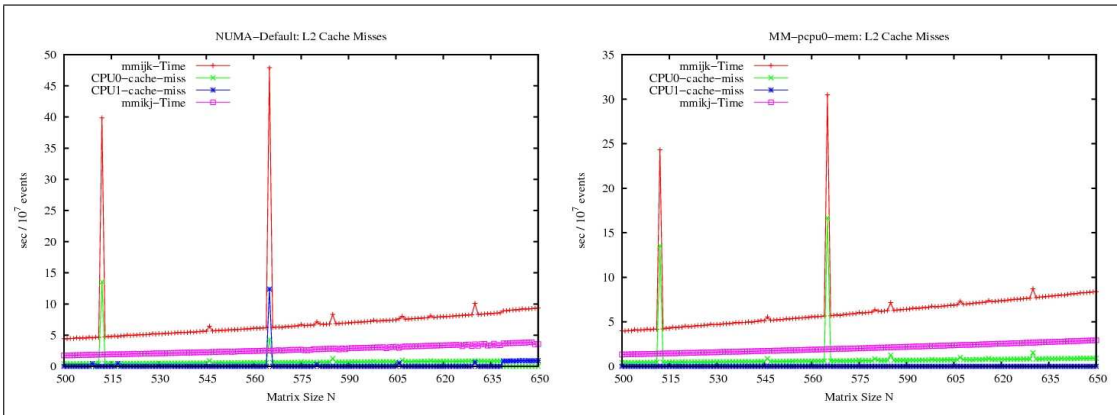
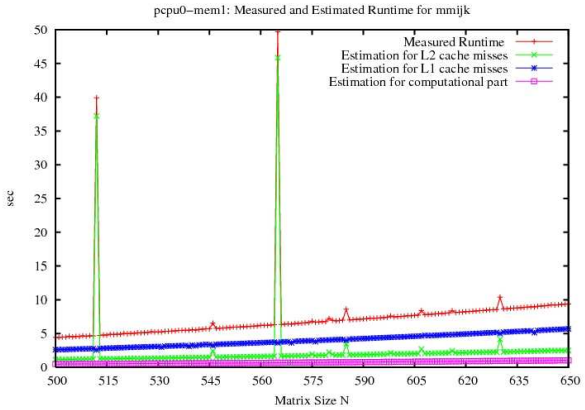


Figure A.2: Runtimes and L2 cache misses: NUMA default (left) and pinning to CPU 0

Figure A.2 displays runtimes and L2 cache misses for both policies. One can conclude that the peaks at N=512 and 565 are caused by the increased number of L2 cache misses for these problem sizes. When process execution is pinned to CPU 0 the peaks are considerably reduced.



A good approximation for the measured mmijk runtime is given by assessing

14 cyc	(4,4ns)	for computation in the innermost loop
58 cyc	(18,1ns)	per L1 cache miss
497 cyc	(155,3ns)	per L2 cache miss
368 cyc	(115,0ns)	for the cache line transfer from remote to local L2 cache

Appendix B

SUMMA Algorithm

The amount of communication per step for the SUMMA algorithm in the case where $np = np_1 \cdot np_2$ can be predicted in the following way.
In each step there is one processor sending $\frac{n}{np_1} \cdot nb$ 4-Byte data to $np_2 - 1$ processors and $\frac{n}{np_2} \cdot nb$ 4-Byte data to $np_1 - 1$ processors, so that the communication time per step is

$$t_{comm} = \frac{4n \cdot nb}{bw} \cdot \left(\frac{np_2 - 1}{np_1} + \frac{np_1 - 1}{np_2} \right)$$

The second term can be larger than 2 if the np_1 and np_2 differ much. In the examples we considered this is the case for a 2×5 grid, where $\frac{2-1}{5} + \frac{5-1}{2} = 2 + \frac{1}{5}$.
The total communication time in $\frac{n}{nb}$ steps then is

$$t_{comm} = \frac{4n^2}{bw} \left(\frac{np_2 - 1}{np_1} + \frac{np_1 - 1}{np_2} \right)$$

This means that the SUMMA algorithm should be preferred if

$$\left(\frac{np_2 - 1}{np_1} + \frac{np_1 - 1}{np_2} \right) < t_{srr}$$

With a measured bandwidth for broadcast to one processor and point-to-point communication of 600 MB/sec for messages longer than 2^{20} Bytes this means that for $np = 4$ processors the total communication time for the SUMMA algorithm should be

$$t_{comm} = \frac{4n^2}{600 \cdot 2^{20}} (\frac{1}{2} + \frac{1}{2}) = \frac{n^2}{150 \cdot 2^{20}} [\text{sec}]$$

for $n >= 1024$. (For $n = 1024$ the local pieces of the matrix are $512 \cdot 512$ floats which makes $2^9 \cdot 2^9 \cdot 4 = 2^{20}$ Bytes.)
The minimum matrix size n for the total overlap of communication by computation in the SUMMA algorithm is

$$\begin{aligned} t_{comp} = \frac{ops}{peak} = \frac{2 \cdot n^2 / np * nb}{peak} \geq t_{comm} = \frac{4n \cdot nb}{bw} \cdot \left(\frac{np_2 - 1}{np_1} + \frac{np_1 - 1}{np_2} \right) \\ \iff \frac{n}{np \cdot peak} \geq \frac{2}{bw} \cdot \left(\frac{np_2 - 1}{np_1} + \frac{np_1 - 1}{np_2} \right) \\ \iff n \geq \frac{2peak}{bw} \cdot \left(\frac{np_2 - 1}{np_1} + \frac{np_1 - 1}{np_2} \right) \cdot np \end{aligned}$$

Bibliography

[1] George Almasi, Ralph Bellofatto, Jose Brunheroto, Calin Cascaval, Jose G. Castanos, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, et al.
An Overview of the Blue Gene/L System Software Organization,
Technical report, IBM Thomas J. Watson Research Center, 2003

[2] Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta,
CellSs: A Programming Model for the Cell BE Architecture,
in proceedings of the ACM/IEEE SC 2006 Conference, Nov. 2006

[3] L.S. Blackford, J. Choi, A.Cleary et al.,
ScaLAPACK Users' Guide, SIAM Philadelphia, 1997

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

[5] Barcelona Supercomputing Center (BSC), Cell Superscalar (CellSs)
User's Manual, Barcelona Supercomputing Center, Jan. 2007
http://www.bsc.es/media/976.pdf

[6] A. Buttari, J. Dongarra, and J. Kurzak,
Limitations of the PlayStation 3 for High Performance Cluster Computing,
Manchester Institute for Mathematical Sciences, School of Mathematics,
July 2007, http://www.manchester.ac.uk/mims/eprints

[7] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, G. Bosilka,
SCOP3, A Rough Guide to Scientific Computing on the PlayStation 3
Technical Report UT-CS-07-595 Version 1.0, Innovative Computing Laboratory,
University of Tennessee, Knoxville, May 2007
http://www.emsl.pnl.gov:2080/docs/global/ga.html

[8] Sean Dague, System Installation Suite, Massive Installation for Linux,
http://dague.net/sean/sispaper.pdf, http://systemimager.org

[9] Ulrich Detert, Andreas Thomasch, Norbert Eicker, Jeff Broughton,
JULI Project - Final Report, Technical Report FZJ-ZAM-IB-2007-05

[10] Andreas Dolfen, Eva Pavarini, and Erik Koch,
New Horizons for the Realistic Description of Materials with Strong Correlations,
Inside, 1, 2006,
http://inside.hlrs.de/htm/Edition_01_06/article_05.htm.

[11] Andreas Dolfen,
Massively parallel exact diagonalization of strongly correlated systems,
Master's thesis, Forschungszentrum Juelich, October 2006

[12] Robert van de Geijn and Jerrell Watts,
SUMMA: Scalable Universal Matrix Multiplication Algorithm
Concurrency: Practice and Experience , Vol. 9 (4), pp. 255-274 (April 1997)

[13] IBM, Cell Boradband Engine, Software Development Kit 2.1,
Installation Guide, March 2007

[14] IBM, CBEA JSRE Series
SPE Runtime Management Library Version 2.1, IBM 2007

[15] IBM, Cell Broadband Engine, Software Development Kit 2.1,
SPE Runtime Management Library Version 1.2 to 2.1 Migration Guide, March 2007

[16] Jülich Supercomputing Centre (JSC), Cell Cluster Meeting in Jülich
http://www.fz-juelich.de/jsc/juice/cell_cluster_meeting

[17] Jakub Kurzak, Jack Dongarra,
Implementation of a Mixed-Precision in Solving Systems of Linear Equations on the Cell Processor, Nov. 2006
http://www.netlib.org/utk/people/JackDongarra/PAPERS/cell-linpack-2006.pdf

[18] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Lack Dongarra,
Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems),
Lapack Working Note, 2006

[19] Ken Koch,
Roadrunner System Overview,
http://www.lanl.gov/orgs/hpc/roadrunner.shtml

[20] Mark Mendell,
IBM XL Compiler Support for the Cell Broadband Engine,
Carleton University, November 2006

[21] Josep M. Perez, Pieter Bellens, Rosa M. Badia and Jesus Labarta,
CellSs: Programming the Cell/B.E. made easier,
IBM Journal of R&D. vol. 51, Aug. 2007

[22] Annika Schiller,
A Fast Wavelet Based Implematation to Calculate Coulomb Potentials on the Cell,
Technical Report FZJ-ZAM-IB-2007-06

[23] G. Sutmann and S. Wädow,
A Fast Wavelet Based Evaluation of Coulomb Potentials in Molecular Systems
published in: NIC Workshop 2006, From Comp. Biophysics to Systems Biology,
Jan Meinke, Olav Zimmermann, Sandipan Mohanty, Ulrich H.E. Hansmann (Editors)
John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 34,
ISBN-10: 3-9810843-0-6, ISBN-13: 978-3-9810843-0-6, pp. 185-188 , 2006.

[24] U. Trottenberg and C. Oosterlee and A. Schüller,
Multigrid, Academic Press, San Diego, 2001

[25] S.Williams, J.Shalf, L.Oliker et. al.,
The Potential of the Cell Processor for Scientific Computing,
Proceedings of the 3rd Conference on Computing Frontiers, 2006