Thomas Streuer[1], Hinnerk Stüben

# Simulations of QCD in the Era of Sustained Tflop/s Computing

[1]Department of Physics and Astronomy, University of Kentucky, Lexington, KY, USA

Contribution to Parallel Computing (ParCo) 2007, 4–7 September 2007, Aachen and Jülich, Germany

ZIB-Report 07–48 (December 2007)

# Simulations of QCD in the Era of Sustained Tflop/s Computing

**Thomas Streuer**[1] **and Hinnerk Stüben**[2]

[1] Department of Physics and Astronomy,
University of Kentucky, Lexington, KY, USA
*E-mail: thomas.streuer@desy.de*

[2] Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB),
Takustr. 7, 14195 Berlin, Germany
*E-mail: stueben@zib.de*

The latest machine generation installed at supercomputer centres in Germany offers a peak performance in the tens of Tflop/s range. We study performance and scaling of our quantum chromodynamics simulation programme BQCD that we obtained on two of these machines, an IBM Blue Gene/L and an SGI Altix 4700. We compare the performance of Fortran/MPI code with assembler code. The latter allows to exploit concurrency at more levels, in particular in overlapping communication and computation as well as prefetching data from main memory.

## 1 Introduction

The first computer delivering a performance of more than 1 Tflop/s peak as well as in the Linpack benchmark appeared on the Top500 list in June 1997[1]. For German QCD[a] researchers it has taken until the installation of the current generation of supercomputers at national centres until a sustained Tflop/s was available in everyday runs of their simulation programmes.

In this paper we report on how the sustained performance was obtained on these machines. There are two machines, the IBM BlueGene/L at NIC/ZAM Jülich and the SGI Altix 4700 at LRZ Garching/Munich. Both started user operation in 2006. The BlueGene/L has 16.384 CPUs (cores) and offers a peak performance of 45 Tflop/s. The Altix 4700 originally had 4096 cores delivering 26 Tflop/s peak. It was upgraded in 2007 to 9726 cores delivering 62 Tflop/s peak. The performance figures we present were measured on the upgraded system.

It is well known that the performance of QCD programmes can be significantly improved by using low level programming techniques like programming in assembler. In general compilers are not able to generate most efficient code for the multiplication of small complex matrices which is the typical operation in computational QCD (see Sec. 2), even if all data needed for the computations is in the data cache (see Table 2). In assembler one can in addition exploit concurrency at more levels. At one level there are low level communication calls on the BlueGene, by which one can achieve that communication and computation overlap. Another level is prefetching data from main memory, which will be important on the Altix[b].

[a] Quantum chromodynamics (QCD) is the theory of strongly interacting elementary particles.
[b] On the Altix there are two potential methods for overlapping communication and computation. (a) Since mem-

As will be explained in Sec. 2 simulations of QCD are communication intensive. Therefore overlapping communication and computation is an important issue. On systems with larger SMP-nodes one can overlap communication and computation by combining OpenMP and MPI[2]. The nodes of the machines we consider are too small nodes for this high level programming technique to work efficiently.

## 2 Computational QCD

The starting point of QCD is an infinite-dimensional integral. To deal with the theory on the computer space-time continuum is replaced by a four-dimensional regular finite lattice with (anti-) periodic boundary conditions. After this discretisation the integral is finite-dimensional but rather high-dimensional.

The standard algorithm employed today in simulations of QCD is Hybrid Monte Carlo[3] (HMC). HMC programmes have a pronounced kernel, which is an iterative solver of a large system of linear equations. In BQCD we use the standard conjugate gradient (cg) solver. Depending on the physical parameters 80 % or up to more than 95 % of the execution time is spent in the solver. The dominant operation in the solver is the matrix times vector multiplication. In the context of QCD the matrix involved is called *fermion matrix*. This paper is about optimising one part of fermions matrix multiplication which is the multiplication of a vector $\psi$ with the *hopping matrix* $D$: $\phi(i) = \sum_{j=1}^{n} D(i,j)\psi(j)$, where $n$ is the lattice volume. The hopping matrix is large and sparse. The entries in row $i$ are the nearest neighbours of entry $i$ of the vector $\psi$ (for an illustration see Fig. 1). The entries of the hopping matrix are $3 \times 3$ complex matrices and for Wilson fermion, which are used in BQCD, the entries of the vectors are $4 \times 3$ complex matrices or with internal indices $s, s' = 1,2,3,4$ and $c, c' = 1,2,3$ spelled out

$$\phi_{sc}(i) = \sum_{\mu=1}^{4} \left[ (1+\gamma_\mu)_{ss'} U^\dagger_{\mu,cc'}(i-\hat{\mu})\psi_{s'c'}(i-\hat{\mu}) + (1-\gamma_\mu)_{ss'} U_{\mu,cc'}(i)\psi_{s'c'}(i+\hat{\mu}) \right] . \quad (1)$$

$U^\dagger$ denotes hermitian conjugation. The $\gamma$-matrices

$$\gamma_1 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}, \gamma_2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \gamma_3 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}, \gamma_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

lead to different access patterns to the entries of $\psi$. No floating point operations are needed in their multiplications.

A static performance analysis of the hopping matrix multiplication yields that the ratio of floating point multiplications to floating point additions is about 85 % which gives the maximal theoretical performance on processors with fused multiply-adds. Per memory access 13 floating point operations have to be performed on the average.

QCD programmes are parallelised by decomposing the lattice into regular domains. The domains become relatively small. For example, the size of a CPU local domain is

ory is shared between all nodes, it is possible to exchange data simply by using loads or stores (via so-called *shmem pointers*), combined with prefetches as needed in order to hide latency. We have tried this promising method in assembler and in Fortran/C/MPI (without explicit prefetching). In both cases performance decreased. (b) One could try to employ *hyper-threading* where one would use on thread per core for computation and a second thread for communication. In principle there should be no hardware bottlenecks. However, hyper-threading is switched off on the machine we were using.

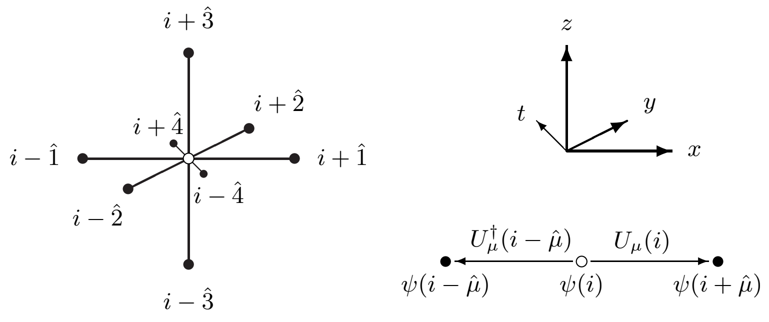

Figure 1. Nearest neighbour stencil underlying the hopping matrix $D$. The central point is $i$. On the righthand side the corresponding Cartesian coordinate system and the variables appearing in Eq. (1) are indicated for one dimension. $U$ is called the *gauge field* which is defined on the links of the lattice. The field $\psi$ is defined on the lattice sites.

$8^3 \times 4 = 2048$ lattice sites when putting a $32^3 \times 64$ lattice (a typical size used in today's simulations) on 1024 CPUs. The size of the surface of this local volume is 2560 sites, i.e. the surface to volume ratio is 1.25. Here is the challenge of QCD programmes. In every iteration of the solver data of the size of the input vector $\psi$ has to be communicated to neighbouring processes.

The basic optimisation is to calculate the projections $(1 \pm \gamma_\mu)\psi(j)$ before the communication. Due to the symmetries of the projections the amount of data to be transfered can be halved. Even with this optimisation, the problem is communication intensive. Very good communication hardware is needed and overlapping communication and computation is desired to scale QCD programmes to large numbers of processes.

## 3 QCD on the IBM BlueGene/L

### 3.1 Hardware

The IBM BlueGene/L is a massively parallel computer. The topology of the network connecting the compute nodes is a three-dimensional torus. Each compute nodes has links to its six nearest neighbours. The hardware bandwidth per link is 175 MByte/s. The network latency is[4]:

$$\text{One way Latency} = (2.81 + .0993 \times \text{Manhattan Distance})\ \mu s$$

In addition to the torus network which is used for point-to-point communication there is a tree network for collective communications.

A BlueGene/L compute chip[5] contains two standard PowerPC 440 cores running at a clock speed of 700 MHz. Each core has a *double hummer* floating point unit[6] (FPU) which operates as a vector processor on a set of $2 \times 32$ registers. Besides pure vector arithmetic operations, there is a set of instructions which operates differently on both registers as it is needed for performing complex arithmetic. Since each FPU can perform one vector-multiply-add operation per clock cycle, the peak performance of the chip is 5.6 Gflop/s.

Each compute node contains 512 MByte of main memory. Each core has a 32 kByte L1 data cache. There is a 4 MByte L3 cache on the chip which is shared between the two cores. Coherency between the L1 caches of the two cores is not enforced by hardware, so software has to take care of it. To facilitate data exchange between the two cores, each chip contains 1 kByte of static ram (SRAM) which is not cached.

The torus network is accessed from the chips through a set of memory-mapped FIFOs. There are 6 injection FIFOs and 12 reception FIFOs on each chip[7]. Data can be written to or read from these FIFOs using some of the double-hummer load/store instructions. We used this feature in our code. We made no special use of the independent tree network.

The BlueGene/L operating systems supports two modes of operation: (a) *communication coprocessor mode*, where one of the cores is dedicated to communication, while the other does the computational work, and (b) *virtual node mode*, where both cores perform both communication and computation operations. We always run our programs in virtual node mode.

### 3.2 Assembler kernel

The $y$-, $z$-, and $t$-directions of the lattice are decomposed in such a way that the decomposition matches the physical torus network exactly. The $x$-direction is split between the two cores of a node. Since the L1 caches are not coherent, communication in the $x$-dimension cannot be done via shared memory in the most straightforward way. Instead, we use the 1 kByte SRAM for communication between the two CPUs.

Ideally communication and computation should overlap. In a QCD programme on the BlueGene/L this can only be achieved by programming in assembler. For the floating pointing operations and communication double-hummer instructions are used. In the course of the computation, each node needs to receive part of the data from the boundary of its neighbouring nodes, and likewise it has to send part of the data from its boundary to neighbouring nodes. In order to hide communication latency, the assembler kernel always looks ahead a few iterations and sends data that will be needed by a remote node. Data is sent in packets of 96 bytes (plus 32 bytes for header and padding), which is the size of a projected spinor $(1 \pm \gamma_\mu)\psi(j)$ in double precision. When a CPU needs data from another node, it polls the respective reception FIFO until a data packet arrives. Since each node sends data packets in the same order in which they are needed on the receiving side, it is not necessary to do any reordering of the packets or to store them temporarily. For comparison with a similar implementation see Reference[8].

### 3.3 Performance results

In scaling tests the performance of the *cg*-kernel was measured. For performance measurements the code was instrumented with timer calls and for the kernel all floating point operations were counted manually.

In order to get good performance it is important that the lattice fits the physical torus of the machine. In the assignment of MPI process ranks the four torus directions have to be permuted. On the BlueGene/L this can be accomplished by setting the environment variable `BGLMPI_MAPPING` appropriately. The settings of that variable were `TXYZ` on 1, 2, and 4 racks and `TYZX` on 8 racks.

| implementation: Fortran/MPI | | lattice: $48^3 \times 96$ | | |
|---|---|---|---|---|
| #racks | Mflop/s per core | overall Tflop/s | speed-up | efficiency |
| 1 | 280 | 0.57 | 1.00 | 1.00 |
| 2 | 292 | 1.20 | 2.09 | 1.04 |
| 4 | 309 | 2.53 | 4.41 | 1.10 |
| 8 | 325 | 5.32 | 9.29 | 1.16 |
| implementation: Fortran/MPI | | lattice: $32^4 \times 64$ | | |
| #racks | Mflop/s per core | overall Tflop/s | speed-up | efficiency |
| 1 | 337 | 0.69 | 1.00 | 1.00 |
| 2 | 321 | 1.32 | 1.91 | 0.95 |
| 4 | 280 | 2.30 | 3.33 | 0.83 |
| 8 | 222 | 3.65 | 5.28 | 0.66 |
| implementation: assembler | | lattice: $32^3 \times 64$ | | |
| #racks | Mflop/s per core | overall Tflop/s | speed-up | efficiency |
| 1 | 535 | 1.10 | 1.00 | 1.00 |
| 2 | 537 | 2.20 | 2.01 | 1.00 |
| 8 | 491 | 8.05 | 7.34 | 0.92 |

Table 1. Performance of the conjugate gradient kernel on the BlueGene/L for two implementations and two lattices.

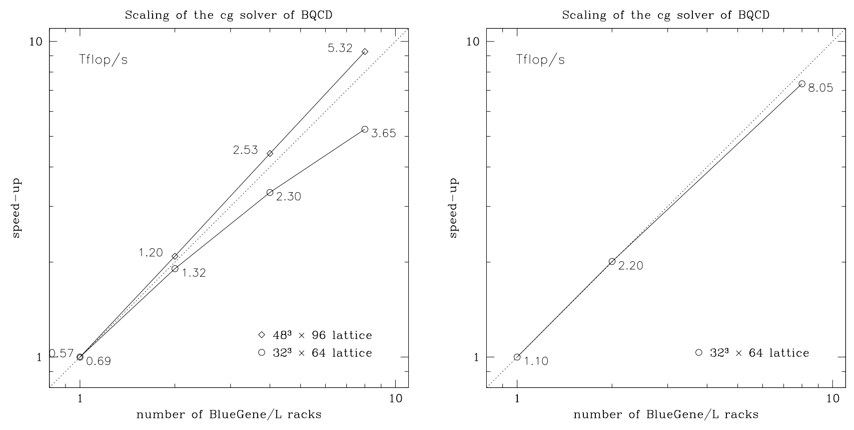

Figure 2. Scaling of the conjugate gradient kernel of BQCD on the BlueGene/L for the Fortran 90/MPI version (left) and for the assembler version (right). The dotted lines indicate linear scaling.

Performance results are given in Table 1. In Fig. 2 results are shown on double logarithmic plots. One can see from the table and the plots that the Fortran/MPI version exposes super-linear scaling on the $48^3 \times 96$ lattice. Even the $32^3 \times 64$ lattice scales quite well given the fact that the lattice volumes per core become tiny (down to $16 \times 2^3$). The scaling of the assembler version is excellent. For the same tiny local lattices the scaling is considerably better than for the Fortran/MPI version. This means that in the assembler version computation and communication really overlap.

## 4 QCD on the SGI Altix 4700

### 4.1 Hardware

The SGI Altix 4700 is a scalable ccNUMA parallel computer, i.e. its memory is physically distributed but logically shared and the memory is kept coherent automatically by the hardware. For the programmer (or a programme) all of the machine's memory is visible to all nodes, i.e. there is a global address space.

The compute nodes of the Altix 4700 consist of 256 dual core processors. One processor is reserved for the operating system, 255 processors can be used for computation. Inside a node processors are connected via the fat tree *NUMAlink 4* network with a theoretical bandwidth of 6.4 GB/s per link. The nodes are connected via a two-dimensional torus type of network. However, the network is not homogeneous, which *a priori* makes it difficult to scale our problem to very large numbers of cores. The machine at LRZ has the following bisection bandwidths per processor[9]:

| | |
|---|---|
| intra-node | $2 \times 0.8$ GByte/s. |
| any two 'vertical' nodes | $2 \times 0.4$ GByte/s. |
| four nodes (shortest path) | $2 \times 0.2$ GByte/s. |
| total system | $2 \times 0.1$ GByte/s. |

The processors of the Altix 4700 are Intel Itanium2 *Montecito* Dual Core CPUs, clocked at 1.6 GHz. Each core contains two floating point units, each of which is capable of performing one multiply-add operation per cycle, leading to a peak performance of 6.4 Gflop/s per core (12.8 Gflop/s per processor).

There are three levels of cache, but only two of them (L2 and L3) are used for floating point data. The L3 cache has a size of 9 MByte and a maximum bandwidth of 32 bytes/cycle, which is enough to feed the floating point units even for memory-intensive operations. The bandwidth to main memory is substantially lower.

### 4.2 Assembler kernel

Because the memory bandwidth is so much lower than the L3 cache bandwidth, it is important that we partition our problem in such a way that we can keep the fields which we need during the conjugate gradient iterations in the L3 cache, so that in principle no access to local memory is required. From Table 2 one can see that lattices up to about $8^4$ sites fit

| lattice | #cores | Fortran [Mflop/s] | assembler [Mflop/s] |
|---|---|---|---|
| $4^4$ | 1 | 3529 | 4784 |
| $6^4$ | 1 | 3653 | 4813 |
| $8^4$ | 1 | 3245 | 4465 |
| $10^4$ | 1 | 1434 | 3256 |
| $12^4$ | 1 | 1329 | 2878 |
| $14^4$ | 1 | 1103 | 2766 |
| $16^4$ | 1 | 1063 | 2879 |

Table 2. Performance of the hopping matrix multiplication on a single core on the Altix 4700.

| weak scaling for local $8^4$ lattices | | | |
|---|---|---|---|
| lattice | #cores | Fortran [Mflop/s] | assembler [Mflop/s] |
| $8^4$ | 1 | 2553 | 3655 |
| $16^4$ | 16 | 1477 | 2235 |
| $24^4$ | 81 | 1273 | 1978 |
| $32^4$ | 256 | 1251 | 1750 |
| $32^3 \times 64$ | 512 | 1195 | 1619 |
| $40^3 \times 64$ | 1000 | 1156 | 1485 |

| strong scaling for the $32^3 \times 64$ lattice | | | |
|---|---|---|---|
| lattice | #cores | Fortran [Mflop/s] | assembler [Mflop/s] |
| $32^3 \times 64$ | 512 | 1195 | 1619 |
| $32^3 \times 64$ | 1024 | 1395 | 1409 |
| $32^3 \times 64$ | 2048 | 996 | 841 |

Table 3. Scaling on the Altix 4700 for the conjugate gradient solver. Performance figures are in Mflop/s per core.

into the L3 cache. When staying inside the L3 cache assembler code is roughly a factor of 1.3 faster. Outside the L3 cache the assembler is faster up to a factor of 2.7. The reason for this speed-up is prefetching. Prefetching is important in the parallel version even if the local lattice would fit into the cache, because data that stems form remote processes will not be in the cache but rather in main memory.

## 4.3 Performance results

Performance results are given in Table 3 and plotted in Fig. 3. Weak scaling results are shown on the left hand side of Fig. 3. From the weak scaling we see that parallel performance is dominated by data communication overhead. When going from one core to the

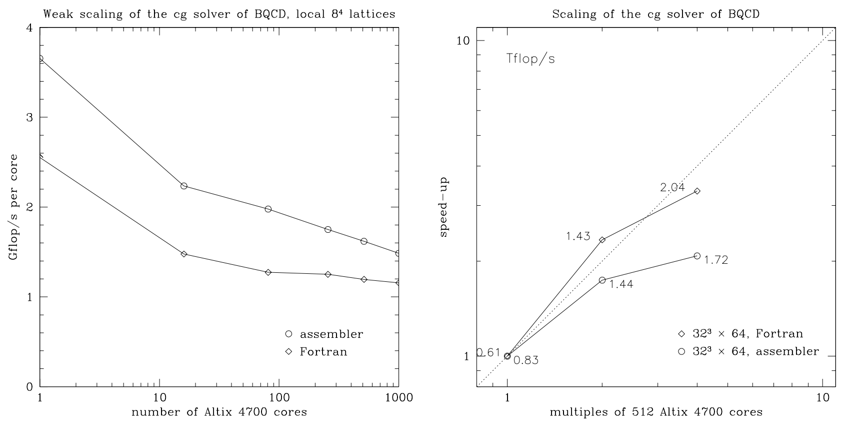

Figure 3. Weak (left) and strong scaling (right) of the conjugate gradient solver. The dotted line in the strong scaling plot indicates linear scaling.

general case of $3^4 = 81$ cores (on a single node) performance drops by a factor of about two and continues to decrease slowly when increasing the number of cores further.

Strong scaling results are shown on the right hand side of Fig. 3. The Fortran code scales super-linearly when going from 512 to 1024 cores which is clearly an effect of the large L3 cache. Remarkably, Fortran outperforms the assembler on 2048 cores. This indicates that the MPI calls, that are handled in the assembler part, lead in this case to an inefficient communication pattern.

Note that 1024 and 2048 cores do not fit into two or four nodes respectively. For a production run a 'sweet spot' had been searched and filling two nodes with local $8^4$ lattices was chosen. The overall lattice size was $40^3 \times 64$ which was put onto $5^3 \times 8$ = 1000 cores. The average overall performance sustained was 1.485 Tflop/s which is 23 % of the peak performance.

## 5 Summary

Using lower level programming techniques improves the performance of QCD programmes significantly. The speed-up that can be achieved in comparison to programming in Fortran/MPI is a factor of 1.3–2.0.

We found that our code scales up to the whole Blue Gene/L in Jülich. The highest performance measured was 8.05 Tflop/s on the whole machine. In production typically one rack (2048 cores) is used on which a performance 1.1 Tflop/s or 19 % of peak is sustained. The high performance and scaling could be obtained by using *double hummer* instructions and techniques to overlap communication and computation.

On the Altix 4700 the large L3 data cache helps a lot to boost performance. Due to the hierarchical nature of the communication network performance measurement depend to some degree on the placement of programmes. In production an average sustained performance of 1.485 Tflop/s or 23 % of peak is achieved when using 1000 cores.

*Acknowledgements.* The performance measurements were done on the BlueGene/L at NIC/ZAM Jülich at the BlueGene Scaling Workshop, Jülich, 3–5 December 2006 and on the Altix 4700 at LRZ Garching/Munich. T.S. is supported by a Feodor-Lynen Fellowship.

## References

1. `www.top500.org`
2. G. Schierholz and H. Stüben, *Optimizing the Hybrid Monte Carlo Algorithm on the Hitachi SR8000,* in S. Wagner, W. Hanke, A. Bode and F. Durst (eds.), High Performance Computing in Science and Engineering, Munich 2004.
3. S. Duane, A. Kennedy, B. Pendleton, D. Roweth, Phys. Lett. B **195**, 216–222 (1987).
4. *Unfolding the IBM eServer Blue Gene Solution*, IBM Redbook, `ibm.com/redbooks`.
5. A. A. Bright *et al.*, IBM J. Res. & Dev. **49**, 277–287 (2005).
6. C. D. Wait *et al.*, IBM J. Res. & Dev. **49**, 249–254 (2005).
7. N. R. Adiga *et al.*, IBM J. Res. & Dev. **49**, 265–276 (2005).
8. P. Vranas *et al.*, Proceedings of the ACM/IEEE SC2006 Conference on High Performance Computing and Networking, Tampa, Florida, USA, November 2006.
9. `www.lrz-muenchen.de/services/compute/hlrb/batch/batch.html`