FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Highly Optimized Code for Lattice Quantum Chromodynamics on the CRAY T3E

Norbert Attig, Stephan Güsken[b], Pierre Lacock[a],
Thomas Lippert[b], Klaus Schilling[a,b],
Peer Ueberholz[b], Jochen Viehoff[b]

FZJ-ZAM-IB-9822

Januar 1998

(letzte Änderung: 12.01.98)

Preprint:
Proceedings of the International Conference on Parallel Computing (ParCo'97), Parallel Computing: Fundamental Applications, and New Directions
In: Advances in Parallel Computing, Vol. 12, Editors: E.H. D'Orlander, G.R. Joubert, F.J. Peters, U. Trottenberg, R. Völpel, North Holland, 1998, pp. 557-564

([a]) German Supercomputing Center (HLRZ), c/o Research Centre Jülich,
D-52425 Jülich, Germany
([b]) Department of Physics, University of Wuppertal,
D-42097 Wuppertal, Germany

Highly Optimized Code for Lattice Quantum Chromodynamics on the CRAY T3E

N. Attig[a], S. Güsken[c], P. Lacock[b], Th. Lippert[c], K. Schilling[b,c], P. Ueberholz[c] *, and J. Viehoff[c]

[a]Central Institute for Applied Mathematics (ZAM), Research Centre Jülich, D-52425 Jülich, Germany

[b]German Supercomputing Center (HLRZ), c/o Research Centre Jülich, D-52425 Jülich, Germany

[c]Department of Physics, University of Wuppertal, D-42097 Wuppertal, Germany

In lattice quantum chromodynamics, large systems of linear equations have to be solved to compute physical quantities. The availability of efficient parallel Krylov subspace solvers plays a vital role in the solution of these systems. We present a detailed analysis of the performance of the stabilised biconjugate gradient (BiCGStab) algorithm with symmetric successive over-relaxed (SSOR) preconditioning on a massively parallel CRAY T3E system.

1. Lattice Gauge Theory Computations

The numerical investigation of quantum chromodynamics (QCD) on a four-dimensional space-time grid is one of the grand challenges in high-performance scientific computing [1]. QCD is generally considered to be the fundamental theory which describes the strong forces binding quarks with gluons to form the known hadrons like the proton or neutron [2]. Even after 20 years of research, QCD still has not been solved in a non-perturbative analytical approach, and it is by now widely believed that the controlled numerical treatment of the theory on the lattice on very fast parallel supercomputers is the only viable scheme to extract quantitative physical results [3]. The results from lattice gauge theory (LGT) simulations are urgently needed as theoretical input for current and future accelerator experiments that attempt to observe new physics beyond the Standard Model of elementary particle physics [4].

LGT computes functional integrals using Monte Carlo methods known from statistical physics [5]. A representative ensemble of field configurations is generated by a Markov process (simulation phase). These configurations are subsequently analysed by constructing (and averaging over) hadronic correlators from quark Green's functions (analysis phase).

*Talk presented by P. Ueberholz.

The correlators then serve to extract physical observables of interest like hadron masses and decay constants.

2. The numerical problem and computational effort

The collaborations SESAM[2] [6] and TχL[3] [7] are two large-scale projects [8] which want to study full QCD with two flavours of dynamical Wilson fermions following the method described above. This task needs high-performance computers which are also able to handle Tbytes of data.

First, a representative ensemble of gauge field configurations on a space-time lattice is generated using a Hybrid Monte Carlo (HMC) algorithm on the parallel supercomputer APE100/Quadrics (simulation phase). The SESAM simulation ($16^3 \times 32$ lattice) has taken place on a 256-node machine with a theoretical peak performance of 12.8 Gflops in DESY/Zeuthen, while the TχL production ($24^3 \times 40$ lattice) is still continuing on two 512-node systems (peak performance of 25.6 Gflops) in Rome and DESY/Zeuthen. The total computer time used so far is about 250 Tflop hours.

The analyses of these gauge field configurations are performed on a CRAY T3E (512 nodes, 300 Gflops peak performance) installed at the Research Centre Jülich. In the analysis phase the most time-consuming part is the frequent computation of the QCD-Green's functions as solutions of huge sparse systems of linear equations [9] on each gauge field configuration

$$M\psi = \phi \,, \tag{1}$$

where ϕ is some input vector, M is the Dirac fermion matrix and ψ the required solution. From the solution (quark propagator) we can then investigate the properties of hadrons by constructing the appropriate hadronic correlators. For the choice of lattice fermions considered here, namely the so-called Wilson fermions, (1) has the explicit form

$$\psi(x)^c_\alpha - \kappa \sum_{\mu=1}^{4} (U_{-\mu}(x)^{c,c\prime} m_{-\mu,\alpha,\alpha\prime} \psi(x-\mu)^{c\prime}_{\alpha\prime} + U_\mu(x)^{c,c\prime} m_{\mu,\alpha,\alpha\prime} \psi(x+\mu)^{c\prime}_{\alpha\prime}) = \phi(x)^c_\alpha \,, \tag{2}$$

where $U_\mu(x)^{c,c\prime}$ is an SU(3) matrix for the gauge field (gluon) with a three-component colour index c, computed in the simulation phase, $\psi(x)^c_\alpha$ is a 4×3 complex matrix for the particle (quark) with four Dirac components α, and m is a 4×4 matrix containing the spin components. The index x runs over all space-time lattice points (4 dimensions). In particular, M is a non-hermitian complex sparse matrix with the following structure:

- 1 in the diagonal.
- 8 non-diagonal 12×12 matrices (nearest neighbours in space-time).
- Each 12×12 matrix is a tensor product of an $SU(3)$ matrix U times the Dirac 4×4 matrix m.

In order to solve this system of linear equations we apply Krylov subspace solvers[4], complemented by preconditioning techniques.

[2]Sea Quark Effects in Spectrum and Matrix Elements

[3]Towards the chiral Limit

[4]The coefficients of the regular sparse matrix are related to a stochastic background gauge field, hence multigrid methods are not efficient here!

3. Inversion algorithms

The class of Krylov subspace iterative methods for solving (1) is characterised by the following generic form:

1. choose an initial guess ψ^0 and set $r^0 = \phi - M\psi^0$,
2. compute iteratively ψ^m of the form $\psi^m = \psi^0 + q_{(m-1)}(M)r^0$.

Here $q_{(m-1)}(M)$ is a polynomial of degree $< m$. Examples are the conjugate gradient or the minimal residual algorithms. At present, the state-of-the-art method for inverting the fermion matrix is the stabilised biconjugate gradient (BiCGStab) algorithm [10].

Further improvement of the inversion algorithm can be obtained by using preconditioning techniques, which should reduce the number of iterations and computing time necessary to achieve a given accuracy. To precondition (1) we take two non-singular matrices V_1 and V_2 which act as left and right preconditioners respectively, i.e. we replace (1) by

$$M\psi = \phi \;\; \longrightarrow \;\; V_1^{-1} M V_2^{-1} \widetilde{\psi} \;=\; \widetilde{\phi} \tag{3}$$

The matrix $V = V_1 V_2$ is called the preconditioner. The efficiency of the preconditioning method depends on how good an approximation V is of M, as well as the computational overhead entailed in the method: solving systems with V_1 and V_2 should be cheap.

Currently the most efficient way is to use SSOR preconditioning (for details see [11,12]):

$$V_1 = I - L; \quad V_2 = I - U \tag{4}$$

with $M = I - L - U$, I the diagonal part, L the strictly lower triangular part and U the strictly upper triangular part. The convergence rate depends on the ordering of the sites on the lattice.

In case of even-odd ordering, where one labels all even sites before the odd ones on each processor like a checkerboard (Fig. 1), the preconditioned matrix $V_1^{-1} M V_2^{-1}$ can be computed explicitely. For larger sub-systems we use parallel SSOR preconditioning. In particular, the sub-blocks have the same size as the partitions of the lattice assigned to a processor. Thus, in parallel SSOR, the parallelism is adapted to the parallel system. The use of the Eisenstat trick is essential:

$$V_1^{-1} M V_2^{-1} = (I - U)^{-1} + (I - L)^{-1}(I - (I - U)^{-1}) \tag{5}$$

In this way the SSOR preconditioning is about as expensive as the multiplication $M\psi$, because both $(I - L)x = y$ (forward solve) and $(I - U)x = y$ (backward solve) are easy to solve.

The largest vector computed in the SESAM and TχL projects is of a size of 12 Mwords. This number is given by the volume factor (e.g. $24^3 \times 40$), the four Dirac components, three colour components, and a factor 2 due to complex elements. As a consequence, we are faced in the analysis phase, where about ten thousand configurations have to be used for calculating the propagators, with a requirement of about 10 Teraflop hours computing time on the order of 1 Terabyte of data. It is therefore evident that optimization of the QCD analysis codes on parallel supercomputers is mandatory in order to carry out ambitious computations of this kind efficiently.

4. CRAY T3E architecture

The CRAY T3E, which is the second generation of Cray Research MPP systems, is the ideal machine for this kind of application. It is a fully scalable MIMD system with distributed memory and global address space. The application processing elements (application PE) are connected by a network which has the topology of a three-dimensional torus. The system is self-hosting and scalable from 8 to 2048 PEs. For every 16 application PE an operating system node (OS PE) is required.

Each processing node is equipped with a DEC Alpha EV5 microprocessor with a clock rate of 300 MHz. The instruction rate is up to 4 per clock cycle (2 floating-point, 2 integer/logic) which results in a peak performance of 600 Mflops. The microprocessor uses IEEE 64-bit arithmetic. Furthermore, each microprocessor has 3 on-chip caches: one 8 KByte data cache (Dcache), segmented into 256 cache lines with 4 64-bit words each, one 8 KByte instruction cache (Icache) with the same segmentation and one 96 KByte secondary cache (Scache), separated in 3 associative sets of 512 cache lines each, serving either Dcache and Icache. The load latencies are 2 clock periods (CP) from Dcache and 8 to 10 CP from Scache, the load bandwidths are 2 loads/CP from Dcache or Scache, or 1 store/CP.

Each PE is equipped with a local memory (DRAM) of 128, 256 or 512 Mbyte. To maximize local memory bandwidth, 6 data stream buffers are available. They do a prefetch from DRAM to cache for vector-like data references leading to a better performance. Additionally, a set of 512 off-chip memory mapped external (E) registers can be used, which directly load/store into/from the cache registers from/to the global memory.

For QCD simulations it is sometimes sufficient to perform the calculations in half-precision. On the T3E operations can only be performed in 64-bit precision due to the 64-bit arithmetic of the processor. Nevertheless, it is possible to use half-precision variables (32 bit). Before being processed, they are extented to 64 bit, then processed and finally reduced again to 32 bit. So it is not possible on the T3E to increase the number of operations per clock period but one can profit from a better cache usage.

The CRAY T3E-600 at the Research Centre Jülich is equipped with 512 application PE, which together offer a peak performance of 300 Gflops. The memory is 128 Mbyte on each PE. Unfortunately, the application PE are from a series of EV5 chips which show a hardware design problem in the memory control unit. Using the stream buffers on these PE may lead to stability problems of the system. Therefore, stream buffers are deactivated and may in general only be used for tests when the system is in maintanance mode.

5. Implementing the inversion algorithm on the CRAY T3E

Our strategy for computing the quark propagators is as follows. First, a four-dimensional processor grid with n_{proc} processors is defined, where $p_x \cdot p_y \cdot p_z \cdot p_t = n_{proc}$. Then, the system is partitioned by dividing the four-dimensional lattice of size $N_x \cdot N_y \cdot N_z \cdot N_t = V$ into local sublattices of size $n_x \cdot n_y \cdot n_z \cdot n_t = V_{loc}$ with $n_i = N_i/p_i$.

The speed of the algorithm and the convergence depends on the layout and the ordering of the fields on each processor. The best usage of the cache is given by running the space-time index last. Two efficient orderings of the lattice sites are the even-odd or

checkerboard ordering and the local lexicographic ordering. The first one is easy to vectorize or parallelize, because the even (odd) points are independent from each other (Fig. 1), while the second one is the most efficient way with regard to the number of iterations [11]. In the forward (backward) solves the neighbours before (after) the current site have to be considered. Thus one needs to communicate the results of the points on the boundary as soon as they are computed. In order to reduce the overhead for the communication we introduce a further blocking within local lexicographic (BLL) ordering on each processor (Fig. 1), which accelerates the Mflop rate by a factor of 3.

In Fig. 1 the ordering of the sites corresponds to the alphabetic ordering $a - q$. The arrows illustrate the points, which have to be taken into account for the update of one of the sites in a two-dimensional analogue.

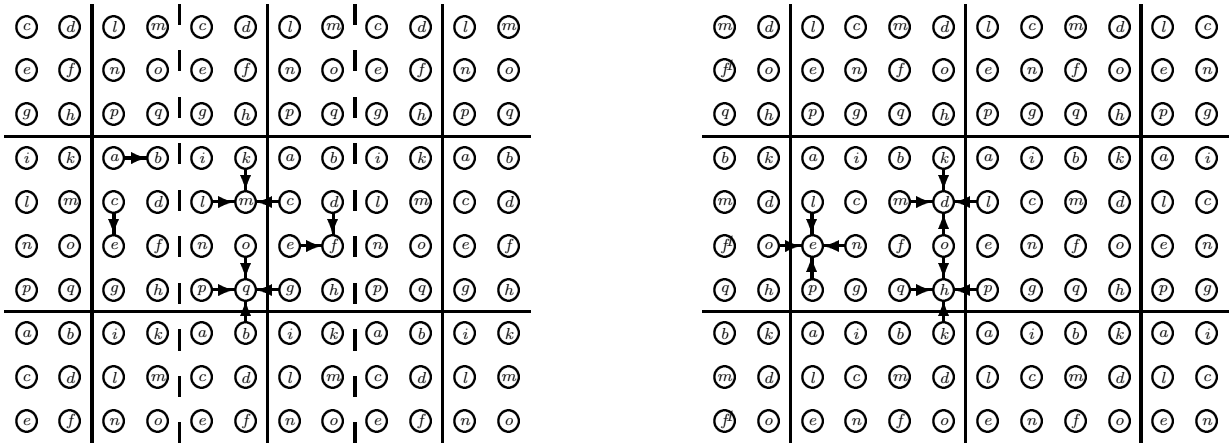

Figure 1. Blocked local lexicographic (left) and even-odd ordering (right).

Before the inversion algorithm can start, the input of the gauge fields (U) and sometimes the starting vectors (ϕ, ψ_o) has to be done. These data are given in 32-bit precision because they were generated on the APE100 in 32-bit precision. The I/O is implemented in parallel by letting the p_t processors read n_t files and send the data to those processors belonging to the same time slices. Doing this, an I/O-bandwidth of up to 75 Mbytes/s can be reached. When the data are read in and distributed, they finally have to be converted to 64 bit.

One iteration step for solving the linear equation $M\psi = \phi$ involves four basic operations:

1. communicate the boundary values,
2. build the vector inner products,
3. perform AXPY vector operations,
4. compute matrix-vector multiplications.

For the communication we use the highly optimized Cray-specific shared memory routines (*shmemput* and *shmemget*) which transfer the data directly between the local and remote memories. For standard message passing routines like MPI about 10% of the total

computer time is spent on communication. With shared memory routines, communication is a factor of 2 to 3 faster.

In case of the inner products and AXPY operations, Fortran 90, BLAS, and assembler routines can be compared. For our application nearly no performance difference between the different routines can be seen. However, on the Fortran level it is worthwhile taking care that all cache entries are used as often as possible and to pad the arrays to avoid cache conflicts. E.g. the operation $z = z + \beta * y$ followed by $z = x + \alpha * z$ runs with 75 Mflops while the performance increases to 140 Mflops if we do both operations in one step $z = x + \alpha * (z + \beta * y)$ as it is needed in the BiCGStab algorithm.

Around 80% of the computer time is spent for the forward/backward solve or, in case of even-odd ordering, for the matrix-vector multiplication, which from the programming point of view is essentially the same. Therefore, we spent most of our effort in optimizing these operations. Looking at the fermion matrix in more detail, the number of $SU(3)$-multiplications can be reduced by a factor of 2 by noting that the 4×4 matrix $m_{\alpha,\alpha\prime}$ can be split into the matrices p of size 4×2 and q of size 2×4. This is known as the Wilson trick. Matrix M of (2) takes the form

$$M^{c,c\prime}_{\alpha,\alpha\prime}(x,y) = \delta^{c,c\prime}_{\alpha,\alpha\prime}\delta_{x,y} - \kappa \sum_{\mu=1}^{4} \sum_{\alpha\prime\prime=1}^{2} \Big(\quad q_{-\mu,\alpha,\alpha\prime\prime} U^{+c,c\prime}_{\mu}(x-\mu) p_{-\mu,\alpha\prime\prime,\alpha\prime} \delta_{(x-\mu),y} \\ + \quad q_{\mu,\alpha,\alpha\prime\prime} U^{c,c\prime}_{\mu}(x) p_{\mu,\alpha\prime\prime,\alpha\prime} \delta_{(x+\mu),y} \Big) . \tag{6}$$

For a further discussion of the implementation, let us consider the last term:

$$\sum_{\alpha\prime=1}^{4} \sum_{\alpha\prime\prime=1}^{2} \sum_{c\prime=1}^{3} q_{\mu,\alpha,\alpha\prime\prime} U_{\mu}(x)^{c,c\prime} p_{\mu,\alpha\prime\prime\alpha\prime} \chi(x+\mu)^{c\prime}_{\alpha\prime} = y(x)^{c}_{\alpha} . \tag{7}$$

A standard way to compute (7) in four steps is:

1.
   - Communicate the boundary of χ (done globally for the whole vector χ).
   - Compute $p_{\mu,\alpha\prime\prime\alpha\prime}\chi(x+\mu)^{c\prime}_{\alpha\prime} = \chi(x)\prime^{c\prime}_{\alpha\prime\prime}$ for a specific site x.
   - Compute $U_{\mu}(x)^{c,c\prime}\chi(x)\prime^{c\prime}_{\alpha\prime\prime} = \chi(x)\prime\prime^{c}_{\alpha\prime\prime}$ for a specific site x.
   - Compute $q_{\mu,\alpha,\alpha\prime\prime}\chi(x)\prime\prime^{c}_{\alpha\prime\prime} = y(x)$ for a specific site x.

Using even-odd ordering there is another possible way to proceed in five steps:

2.
   - Compute $p_{\mu,\alpha\prime\prime\alpha\prime}\chi(x)^{c\prime}_{\alpha\prime} = \omega(x)^{c\prime}_{\alpha\prime\prime}$ for all sites x.
   - Communicate $\omega(x)$ to its neighbour $x+\mu$.
   - Gather result in $\chi(x)\prime$.
   - Compute $U_{\mu}(x)^{c,c\prime}\chi(x)\prime^{c\prime}_{\alpha\prime\prime} = \chi(x)\prime\prime^{c}_{\alpha\prime\prime}$ for all sites x.
   - Compute $q_{\mu,\alpha,\alpha\prime\prime}\chi(x)\prime\prime^{c}_{\alpha\prime\prime} = y(x)$ for all sites x.

At first glance one would expect case 1 to be better suited in exploiting cache reuse, while case 2 should be able to exploit the stream buffers more effectively. This is confirmed by the single processor results listed in Tab. 1, where one clearly sees that the multiplication

Table 1
Single-processor performance in Mflops. Single-precision results are given in parentheses.

	case 1	case 2
$p_{\mu,\alpha\prime\prime\alpha\prime}\chi^{c\prime}_{\alpha\prime}$	12 (20)	13 (27)
$U^{c,c\prime}_{\mu}\chi\prime^{c\prime}_{\alpha\prime\prime}$	70 (73)	94 (124)
$q_{\mu,\alpha,\alpha\prime\prime}\chi\prime\prime^{c}_{\alpha\prime\prime}$	26 (30)	20 (45)
overall	48 (63)	39 (61)

$U\chi\prime$ has a higher Mflop rate in case 2, while for the final multiplication case 1 has the higher rate since it has better cache usage.

Also, as expected, nearly all floating point operations are generated by the complex 3×3 matrix-vector multiplication with $U_\mu(x)$. On the other hand, the multiplications with p and q are mainly load and store, because m has a simple form, e.g. for μ=2

$$p_2 = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

As a result, these multiplications are hard to optimize since the performance is determined by the bandwidth between memory and cache.

In Fortran, case 1 is clearly the better way to proceed. To conclude this section we quote the overall inversion performances for the BiCGStab algorithm with SSOR preconditioning: BLL ordering: 59 (75) Mflops; even-odd ordering: 62 (80) Mflops. Single-precision results are given in parentheses.

6. Single-processor optimization

Coding kernel routines in assembler can improve the performance considerably. Therefore we rewrote in a first step the multiplication with U in assembler. The performance results are listed in Tab. 2. A significant improvement can be observed in both cases, indicating that the hand-written assembler code has a much better register use.

Table 2
Single-processor performance of the assembler codes in Mflops.

	case 1	case 2
$U^{c,c\prime}_{\mu}\chi\prime^{c\prime}_{\alpha\prime\prime}$	150 (158)	168 (240)
overall	67 (81)	45 (72)

Further improvements are expected by extending the assembler code to more steps needed to calculate (7). In case 2 it is possible to make efficient use of the stream buffers. One possibility is to combine the last three steps in one assembler program. In this case, we find a decrease in the Mflops rate for the assembler code from 168 to 125

Mflops. However, the assembler code now contains the gather instruction of the nearest neighbour sites as well as the multiplication with the matrix q, which have both very low Mflop rates. Combining these steps with the matrix-vector multiplication one can make more effective use of data already in the cache and eliminate some of the intermediate steps (e.g. explicitly calculating χ''). The result is an increase in case 2 from 45 (72) to 70 (118) Mflops in the *overall* performance.

Another possiblity is to include all the steps in a single assembler routine except the communication, which is done globally as in case 1. In this case the single-processor performance of the assembler routine decreases (*overall* performance increases) to a value of 85 (140) Mflops. The reason for this are the extra load and store instructions that have been added to the assembler routine. To understand why this happens one has to consider the architecture of the EV5 RISC chip: even though 4 instructions are issued per clock period, some instructions cannot perform simultaneously (e.g. loads and stores). In addition, only one floating point multiply or add may be issued in the same clock period. These factors make it very difficult to balance the requirements of the processor with the practical nature of the problem, where the assembler code now becomes saturated with load and store instructions, while the number of floating point operations only increases marginally. On the other hand, it is an improvement in the *overall* performance of around 20%. Our current best performance values for the BiCGStab algorithms are

even-odd ordering:	87(115) Mflops
BLL ordering:	72 (86) Mflops

However, the number of iterations with BLL ordering is about the half compared to even-odd ordering. Therefore, in real time, BLL ordering is still about 70% faster than even-odd ordering. All performance numbers given above are with activated stream buffers. Without streams, the performance for BLL ordering drops to 51 (72) Mflops.

7. Summary

We have presented a discussion of optimized codes for lattice QCD on the CRAY T3E. To improve the state-of-the-art BiCGStab inversion algorithm, we implemented SSOR preconditioning with blocked lexicographic ordering which can be used efficiently on parallel machines. Optimization efforts are concentrated on the matrix-vector multiplications since most of the computational effort is contained therein. Programming Fortran kernel routines or parts thereof in assembler improves the performance substantially.

We find that one of the main difficulties in optimizing codes on the CRAY T3E is the efficient implementation of instruction scheduling with respect to the complex memory system (set of on-chip caches, stream buffers, E-registers), as well as the quad-instruction issue nature of the processor.

Acknowledgements

The authors gratefully acknowledge the computer time granted by the HLRZ on the CRAY T3E of the Research Centre Jülich. They would like to thank E. Anderson of SGI/Cray Research for his advice and efforts with respect to the assembler programming and R. Vogelsang from SGI GmbH/Cray Research for his continuous support.

REFERENCES

1. Aoki, S.: The QCD Teraflops Project, Int. J. Mod. Phys. **C2** (1991) 829.
2. Zerwas, P.M., Kastrup, H.A. (eds.), *QCD 20 years later* (World Scientific, Singapore, 1992).
3. DeGrand, T.: Lattice Gauge Theory for QCD, COLO-HEP-378 (hep-ph/9610391), October 1996.
4. Particle data group, Phys. Rev. **D54** (1996) 1.
5. Creutz, M. in *Quarks, Gluons and Lattices* (Cambridge University Press, Cambridge, 1983).
6. Glässner, U., Güsken, S., Hoeber, H., Lippert, Th., Luo, X., Ritzenhöfer, G., Schilling, K., Siegert, G.: QCD with dynamical Wilson fermions - first results from SESAM, Nucl. Phys. **B** (Proc. Suppl.) **47** (1996) 386;
   Glässner, U., Güsken, S., Hoeber, H., Lippert, Th., Ritzenhöfer, G., Schilling, K., Siegart, G., Wachter, A.: First Evidence of N_f-Dependence in the QCD Interquark Potential, Phys. Lett. **B383** (1996) 98;
   Eicker, N., Glässner, U., Güsken, S., Hoeber, H., Lippert, Th., Ritzenhöfer, G., Schilling, K., Siegart, G., Spitz, A., Ueberholz, P., Viehoff, J.: Evaluating sea quark contributions to flavour singlet operators in lattice QCD, Phys. Lett. **B389** (1996) 720.
7. Conti, L., Eicker, N., Giusti, L., Glässner, U., Güsken, S., Hoeber, H., Lippert, Th., Martinelli, G., Rapuano, F., Ritzenhöfer, G., Schilling, K., Siegert, G., Spitz, A., Viehoff, J.: Full QCD with dynamical Wilson fermions on a $24^3 \times 40$ lattice - A feasibility study, Nucl. Phys. **B** (Proc. Suppl.) **53** (1997) 222;
   Lippert, Th., Bali, G., Eicker, N., Giusti, L., Glässner, U., Güsken, S., Hoeber, H., Lacock, P., Martinelli, G., Rapuano, F., Ritzenhöfer, G., Schilling, K., Siegert, G., Spitz, A., Ueberholz, P., Viehoff, J.: SESAM and TχL Results for Wilson Action: A Status Report, Nucl. Phys. **B** (Proc. Suppl.) **60A** (1998) 311.
8. Attig, N.: QCD on Parallel Computers at the HLRZ Supercomputing Center, Proceedings of Physics Computing '96 (PC'96), Krakow, Poland, (1996) 536.
9. Lippert, Th., Schilling, K. and Petkov, N.: Quark Propagator on the Connection Machine, Parallel Computing **18** (1992) 1291.
10. Frommer, A., Hannemann, V., Lippert, Th., Nöckel, B., Schilling, K.: Accelerating Wilson Fermion Matrix Inversions by Means of the Stabilized Biconjugate Gradient Algorithm, Int. J. Mod. Phys. **C5** (1994) 1073.
11. Fischer, S., Frommer, A., Glässner, U., Lippert, Th., Ritzenhöfer, G., Schilling, K.: A Parallel SSOR Preconditioner for Lattice QCD, Comp. Phys. Comm. **98** (1996) 20.
12. Eicker, N., Frommer, A., Hoeber, H., Lippert, Th., Medecke, B., Schilling, K., Weufen, G.: Parallel SSOR preconditioners for Improved Actions in Lattice Field Theory, this conference.