John von Neumann Institute for Computing

Optimizing Lattice QCD Simulations on BlueGene/L

Stefan Krieg

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 543-550, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Optimizing Lattice QCD Simulations on BlueGene/L

Stefan Krieg

Department of Physics
Faculty of Mathematics and Natural Sciences
University of Wuppertal
Gaußstraße 20, D-42119 Wuppertal, Germany
Jülich Supercomputing Centre (JSC)
Research Centre Jülich
52425 Jülich, Germany
E-mail: krieg@fz-juelich.de

I describe how Lattice QCD simulations can be optimised on IBM's BlueGene/L eServer Solution computer system, focusing more on the software side of such simulations rather than on simulation algorithms. I sketch the requirements of Lattice QCD simulations in general and describe an efficient implementation of the simulation code on this architecture.

1 Introduction

The strong nuclear force binds protons and nucleons to form the atomic nucleus, countering the electric repulsion between the protons. However protons and nucleons are not fundamental particles, but are comprised of "up" and "down" quarks, two representatives of a family of 6 quarks. These are the fundamental degrees of freedom of the theory of the strong force, called Quantum Chromodynamics (QCD). The gluons are the gauge bosons in QCD, mediating the strong nuclear force, similar to photons in (Quantum) Electrodynamics. Because the coupling describing the strength of QCD interactions scales with the energy, and is large at low energy, Taylor expansions in this coupling are not valid, meaning that there is no known method of analytically calculating the low energy observables of the theory, such as particle masses. Thus it is difficult to compare the theory with experiment.

However a specific formulation of the theory, called Lattice QCD (LQCD), allows a numerical calculation of many of these observables, using a Monte-Carlo approach. A similarity between LQCD and Statistical Mechanics has allowed a fruitful interaction between the two fields, so that similar simulation algorithms are employed in both areas. The standard Monte Carlo methods used in LQCD are the Hybrid Monte Carlo algorithm[1] and its variants, comprised of a Molecular Dynamics evolution of the gluon fields, followed by a Metropolis accept/reject step. The main difference between LQCD and Statistical Mechanics is the dimensionality: QCD, being a relativistic theory, has four (space-time) dimensions. In LQCD, the quarks and gluons "live" on a four dimensional lattice and all "interactions" are essentially local, so that only nearest neighbouring (and in a few formulations next to neighbouring) lattice sites interact.

The Wilson Dirac Kernel

The run-time of a LQCD simulation is dominated by a small kernel, which multiplies the sparse Dirac Matrix with a vector. There are several different variants of the Dirac Matrix.

A commonly used option is the Wilson Dirac Matrix

$$M_{xy} = 1 - \kappa \sum_{\mu} (r - \gamma_\mu) \otimes U_{\mu\, x} \delta_{y\, x+\hat{\mu}} + (r + \gamma_\mu) \otimes U^{\dagger}_{\mu\, x-\hat{\mu}} \delta_{y\, x-\hat{\mu}}. \qquad (1.1)$$

It connects only lattice points which are nearest neighbours. γ^μ represent 4×4 complex matrices with only four nonzero entries each, the $U_\mu(x)$ matrices are 3×3 (dense complex) SU(3) matrices (these matrices are tensor multiplied in qE. 1.1) and κ encodes the quark mass. The γ matrices are fixed for every direction μ, whereas the U matrices are dynamical variables and thus different for every μ and lattice site x. The latter will therefore have to be stored in memory but are *the only* memory required for this matrix. Also apparent from Eq. 1.1 is the sparseness of the Wilson Dirac matrix: The number of flops required for a matrix-vector multiplication scales linearly with V, the number of lattice sites x in the simulation volume. The matrix dimension N is simply given by the dimension of the tensor product of a γ and a U matrix and the lattice volume V and is thus just $N = 12 \times V$. This matrix-vector multiplication (the kernel) is thus sufficiently small and simple to be optimised by hand.

2 The BlueGene/L System

The IBM Blue Gene/L eServer Solution (BGL) stems from a family of customized LQCD machines. The family history begins in 1993 with the development of QCDSP, built by a collaboration of different LQCD groups. The QCDSP systems were fully installed in 1998 and won the Gordon Bell prize for "Most Cost Effective Supercomputer" that year. Most members of the QCDSP collaboration, now joined by IBM, went on to design the QCDOC ("QCD on a Chip") computer. The development phase began in 1999 and the first machines were installed in 2004.

In December 1999, IBM announced a five-year effort to build a massively parallel computer, to be applied to the study of biomolecular phenomena such as protein folding. In 2004 the BGL, the first of a series of computers built and planned within this effort, took the position as the world's fastest computer system away from the "Earth Simulator" who held this position from 2002 to 2004.

The BGL shares several properties with the two dedicated QCD machines. It is a massively parallel machine, with the largest installation at Los Alamos National Laboratory being comprised of 64 racks containing 65,536 nodes, with 2 CPUs per node. These nodes are connected by a 3d torus network (QCDOC: 6d) with nearest neighbour connections. The two CPUs of a node can be operated in two ways: The so called "Virtual Node" (VN) mode divides the 512 MB and 4 MB 3^{rd} level cache between the two nodes which then work independently, whereas the so called "Coprocessor mode" has the first CPU doing all the computations and the second assisting with the communications. The VN mode proved to be more interesting for QCD, especially to match the 4d space-time lattice to the communication hardware topology. However, in VN mode, the communication cannot be offloaded to any communication hardware but has to be managed by the CPU itself. But the gain in peak performance by having both CPU's performing the calculations more than compensates for this.

The CPU integrates a ppc440 embedded processor and a "double" FPU optimised for complex arithmetic. The FPU operates on 128 bit registers with a 5 cycle pipeline latency,

but is capable of double precision calculations only. Single precision calculations can be performed, but they imply an on the fly conversion to double precision during the load and a conversion to single precision during the store operation. The FPU can perform one multiply-add instruction on two doubles stored in a 128 bit register, which results in 2.8 GFlops peak performance per CPU with the 700 MHz clock rate used in BGL. The double precision load/store operations always load two floating point numbers, with the first being aligned to a 16 Byte boundary, similar to the SSE2 instructions.

The compilers are capable of rearranging the code to satisfy this requirement ("auto-simdization"), but the simulation code usually is too complicated for this strategy to work without problems. When auto-simdization proves problematic, it is also possible to compile the code using the first FPU only making auto-simdization unnecessary.

3 Performance Optimization

The two cores of a BGL compute node can be run in two different modes: In Coprocessor (CO) mode the primary core does all calculations and a small part of the communications and the other core performs the remaining part of the communications. In Virtual Node (VN) mode both cores act as independent nodes. As far as MPI is concerned this doubles the number of available MPI tasks.

For LQCD with its rather simple communication patterns, the time spent in the communication part of the kernel is relatively small compared to the time spent in the serial part of the kernel. For that reason the VN mode is advantageous since it doubles the machine's peak performance. In other words: in order to achieve the same performance as in VN mode, the CO mode kernel has to have more than twice the performance per core of the parallelised VN mode kernel, since not all communications can be handed off to the secondary core. With a sustained performance of over 25.5% of machine peak (see below) that would require the CO mode kernel to reach over 51% of machine peak in order to be competitive. This is not likely since the kernel is memory bound and a single core cannot sustain the whole memory bandwidth. As a consequence most (if not all) LQCD applications including the Wilson Dirac kernel described here use VN mode.

In the remainder of this section I shall describe how the kernel's serial performance and communication part was optimised.

Optimising Single Core Performance

Since the kernel is sufficiently small and simple, it is possible and customary to optimise the serial part by hand. Several approaches are usually considered here:

- Writing the kernel in assembly
- Writing the kernel using GCC inline assembly
- Writing the kernel using compiler macros
- Writing the kernel using some assembly generator (eg. BAGEL)
- Writing some core parts of the Kernel using one of the above methods

I will focus here on a version of the kernel written using the "intrinsics" of the IBM XLC compiler. These compiler macros are easy to use and the code generated this way performs almost equally well as assembly code.

All arithmetic operations in the kernel use complex numbers. The FPU is optimized for such calculations and is for example capable of performing a multiplication of two complex numbers (`a,b`) in only 2 clock cycles (`c` contains the result):

```
fxpmul a, b, tmp
fxcxnpma c, b, tmp, a
```

The same code using intrinsics (including load/store):

```
double _Complex alpha,beta,gamma, tmp;
alpha = __lfpd(&a);
beta = __lfpd(&b);
__fxpmul(alpha, beta, tmp);
__fxcxnpma(gamma, beta, tmp, alpha);
__stfpd(&c, gamma);
```

As can be seen from the second example, the intrinsics match the assembly instructions, but do not use explicit registers. The compiler will select the registers as well as schedule the instructions.

The most important part of the kernel is a multiplication of the (complex dense) 3×3 SU(3) matrix, representing the gluon fields, with a vector. The intrinsics code for the first component of the result vector is:

```
tmp  = __fxpmul(su3_00, __creal(vec0));
tmp  = __fxcxnpma(tmp, su3_00, __cimag(vec0));
tmp  = __fxcpmadd(tmp, su3_01, __creal(vec1));
tmp  = __fxcxnpma(tmp, su3_01, __cimag(vec1));
tmp  = __fxcpmadd(tmp, su3_02, __creal(vec2));
tmp  = __fxcxnpma(tmp, su3_02, __cimag(vec2));
```

This code could run at about 92% of peak, (not considering pipeline issues). In order to avoid stalling the pipeline, two complete matrix vector multiplications should be performed simultaneously.

Typically LQCD calculations in general and the kernel operations in particular are not only limited by the memory bandwidth but also feel the memory latency because of non-sequential memory access patterns. Thus prefetching is another very important ingredient for the kernel optimisation. The intrisic prefetch instruction (`__dcbt(*void)`) will prefetch one 32 byte cacheline including the specified pointer. All scheduling will be done by the compiler, so one only has to identify the data that has to be prefetched. A strategy that proved successful is to prefetch the next SU(3) matrix and next vector, while performing the previous calculation. This will optimize for the case that the data is already in the 3^{rd} level cache, which can be assumed since memory bandwidth is limited.

Optimising Communications

Since LQCD uses a 4d space-time lattice and only nearest neighbour interactions, the obvious strategy for communication is to match the lattice dimensions with the communication

hardware layout, that is to parallelise in all 4 dimensions (the 4th direction is along the two CPU's on a node). A typical MPI communicator will have the dimension $8 \times 8 \times 8 \times 2$, here matching the communication layout to the smallest partition of BGL (being a torus), the so called "mid-plane" (512 nodes/1024 CPUs). The latest versions of MPI will usually make sure that a node's coordinates in a communicator agree with its hardware coordinates in the job's partition. This is rather important, since a mismatch can severely impact performance[a]. A useful tool to check whether this really is the case are the run-time system calls "`rts_get_personality(..)`", which returns a structure containing the node's coordinates, and "`rts_get_processor_id()`", which returns the calling CPU's ID inside the node (0 or 1).

Thin interplay between communication and calculation can be highly optimised if the special LCQD communication APIs are used. In that case many different communication strategies can be applied. Since for single precision the performance of the kernel is the same with the QCD API or MPI, I will focus on the MPI implementation here. For MPI in general there are two different communication strategies that can be used (on a machine that cannot offload communications):

- Either first communicate the boundaries of the local sub-lattices and then do all the calculations
- or first carry out some part of the calculation, communicate intermediate results cutting the amount of data to be communicated by half, and then do the remaining calculations.

On the BGL the second strategy proved to be best, and all results quoted are for a kernel using this method. The communication buffers needed within this approach can be located in an area of memory with a special caching strategy: Since they are only written during the preparation of the communication and only read once, it is advantageous to put them in a chunk of memory with "store-without-allocate" caching strategy. The run-time call for this is "`rts_get_dram_window(.., RTS_STORE_WITHOUT_ALLOCATE, ..)`". Data stored to memory allocated in this way will not pollute the L1 cache but be directly stored to the lower memory hierarchies, in this case the L3.

Since in the case of the kernel all pointers to memory that are involved in the communications do never change, the use of persistent sends and receives ("`MPI_PSend, MPI_PRecv, MPI_Startall, MPI_Waitall`") has proved to be the best choice[b]. This also allows the MPI to make sure that all send/receive fifos are kept busy simultaneously, thus optimizing the bandwidth.

Making Use of Single Precision Arithmetics

Since the FPU of the BGL compute node supports only double precision arithmetics both single and double precision calculations have the same peak performance. The compute kernel however is memory bandwidth limited. Thus the single precision kernel will reach

[a] The mapping of the MPI Cartesian communicator to the hardware torus can be easily set with the "`BGLMPI_MAPPING=TZYX`" environment variable by changing the order of the default "`XYZT`".

[b] The environment variable "`BGLMPI_PACING=N`" can be set to switch off the packet pacing, since the dominant communication is nearest neighbours only. This results in slightly smaller communication time.

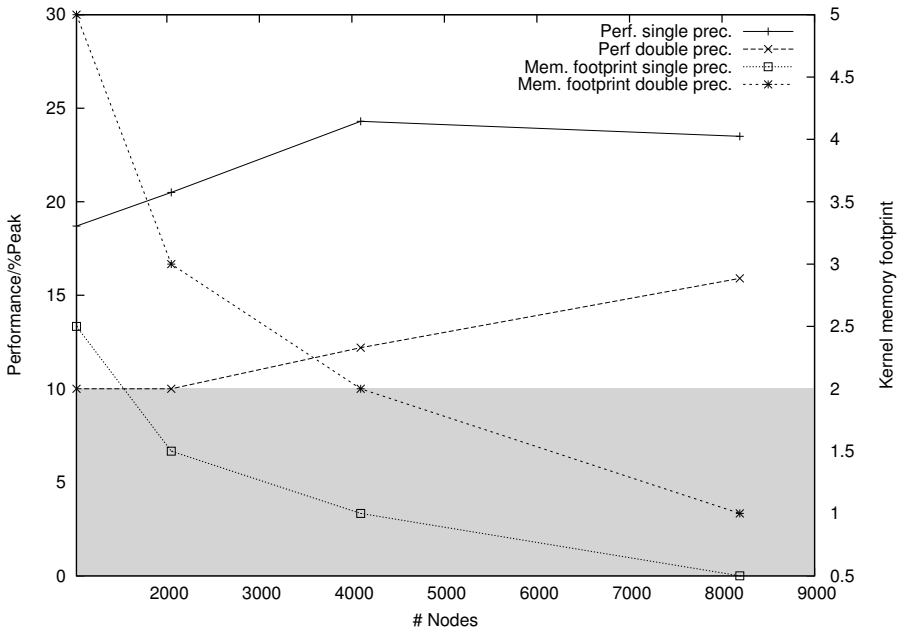

Figure 1. Performance and memory footprint of the single and the double precision kernels as a function of the number of nodes in a strong scaling analysis (global $48^3 \times 96$ lattice). The grey shaded area is the L3 cache size.

a higher performance since it has half the memory and communication bandwidth requirements. Since it also has half the memory footprint of the double precision kernel, scaling will also be improved (see Fig.1). This improvement comes at virtually no coding costs: the only difference between the single precision and the double precision kernel is the load/store instructions and prefetches.

Making use of the single precision kernel is not trivial from a mathematical point of view, since most LQCD calculation require double precision accuracy. For our simulations with dynamical overlap fermions however we can use our inverter scheme, the relaxed GMRESR with SUMR preconditioner[2] and replace the double precision with a single precision preconditioner. Depending on the lattice size almost all computer time will be spent in this preconditioner and thus the performance of the single precision kernel will, to a good accuracy, dominate the performance of the whole code. There certainly is a price to be paid by switching to the single precision SUMR: the total number of calls to the kernel increases slightly compared to the full double precision calculation. Up to lattice sizes of $48^3 \times 64$ this increase however always stayed well below 10% and is thus much smaller than the gain of using double precision (comp. figures 1 and 2).

4 Performance Results and Scaling

Since all communications are constrained to a node's nearest neighbours, and the BGL's network is a torus, weak scaling is perfectly linear. In Fig.2 both the performance for

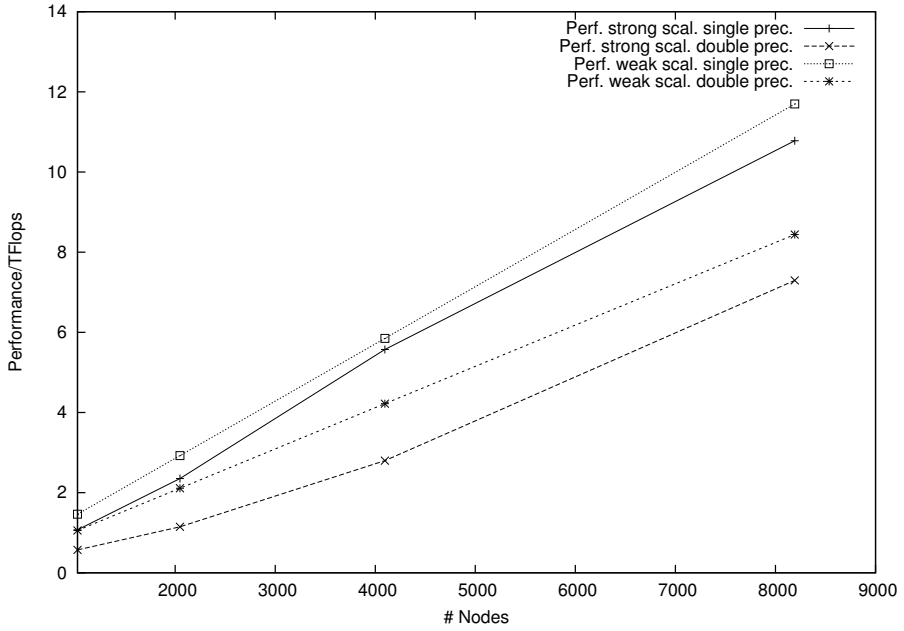

Figure 2. Performance of the single and the double precision kernels as a function of the number of nodes in a weak (local $4^3 \times 8$ lattice) and strong scaling analysis (strong $48^3 \times 96$).

a weak and strong scaling analysis of the kernel are shown. For weak scaling, the kernel reaches 25.5% of machine peak for single and 18.4% of machine peak for double precision accuracy. The performance numbers for the strong scaling analysis are plotted in Fig.1 and reach 24.5% of machine peak single and 16.0% of machine peak in double precision accuracy.

As long as the kernel memory footprint fits into the L3 cache, the performance of the kernel matrix always exceeds 20% of machine peak. (The relative dip of performance at 2k nodes is due to a suboptimal local lattice layout.) The scaling region is therefore quite large and reaches up to 8k nodes for the global lattice chosen here.

5 Conclusions and Outlook

I have shown how to optimise the Wilson Dirac kernel for the IBM Blue Gene/L architecture with the IBM XLC compiler macros and MPI. The kernel scales up to the whole machine and reaches a performance of over 11 TFlop/s in single precision and over 8.4 TFlop/s in double precision on the whole 8 Rack Blue Gene/L "JUBL" at the Jülich Supercomputing Centre (JSC) of the Research Centre Jülich.

The JSC will install a 16 Rack Blue Gene/P (BGP) with 220 TFlop/s peak performance end of 2007. A first implementation of the (even/odd preconditioned) Wilson Dirac kernel shows that this too is a particularly well suited architecture for LQCD: the sustained per node performance of the kernel went up from 1.3 GFlop/s (1.0 GFlop/s) on BGL to

4.3 Gflop/s (3.3 GFlop/s) or 31.5% (24.3%) of machine peak on BGP in single precison (double precision) accuracy.

Acknowledgements

I would like to thank Pavlos Vranas, then IBM Watson Research, for many valuable discussions, Bob Walkup of IBM Watson Research for his help during the first stages of the project, Charles Archer of IBM Rochester for his help with the BGL MPI and Nigel Cundy for his ongoing collaboration. I am indebted to Jutta Docter and Michael Stephan, both JSC, for their constant help with the machine.

References

1. S. Duane, A. Kennedy, B. Pendelton and D. Roweth, Phys. Lett. B, **195**, 216 (1987).
2. N. Cundy, J. van den Eshof, A. Frommer, S. Krieg, Th. Lippert and K. Schäfer, *Numerical methods for the QCD overlap operator. III: Nested iterations*, Comp. Phys. Comm., **165**, 841, (2005). hep-lat/0405003.
3. P. Vranas et al., The Blue Gene/L Supercomputer and Quantum Chromo Dynamics
4. N. R. Adiga et al, *BlueGene/L torus interconnect network*, IBM Journal of Research and Development Vol. **49**, Number 2/3, (2005).
5. IBM Redbook "Blue Gene/L: Application Development"
6. IBM J. Res. & Dev. Vol. **49**, March/May, (2005)