The BAGEL assembler generation library

Peter A Boyle [1]

Abstract

This paper presents two coupled software packages which receive widespread use in the field of numerical simulations of Quantum Chromo-Dynamics. These consist of the BAGEL library and the BAGEL Fermion sparse-matrix library, BFM.

The Bagel library can generate assembly code for a number of architectures and is configurable – supporting several precision and memory pattern options to allow architecture specifig optimisation. It provides high performance on the QCDOC, BlueGene/L and BlueGene/P parallel computer architectures that are popular in the the field of lattice QCD. The code includes a complete conjugate gradient implementation for the Wilson and Domain Wall fermion actions, making it easy to use for third party codes including the Jefferson Laboratory's CHROMA, UKQCD's UKhadron, and the Riken-Brookhaven-Columbia collaboration's CPS packages.

PACS: 11.15.Ha, 12.38.Gc

PROGRAM SUMMARY

Manuscript Title: The BAGEL QCD assembler generation library
Authors: Peter Boyle
Program Title: Bagel
Licensing provisions: GNU Public License V2.
Programming language: C++, assembler.
Computer: Massively parallel message passing. BlueGene/QCDOC/others
Operating system: POSIX, Linux and compatible
Number of processors used: 16,384
Keywords: Assembler, optimisation, domain specific compiler, PowerPC, BlueGene
Classification: 11.5 Quantum Chromodynamics, Lattice Gauge Theory
External routines/libraries: QMP, QDP++
Nature of problem: Quantum chromodynamics sparse matrix inversion for Wilson and Domain Wall Fermion formulations
Solution method: Optimised Krylov linear solver
Typical running time 1h per matrix inversion; multi-year simulations

[1] SUPA, School of Physics, University of Edinburgh, Edinburgh, EH9 3JZ, UK.

Unusual features: Domain specific compiler generates optimised assembly code

1 Introduction

Quantum chromo-dynamics (QCD) is an SU(3) gauge theory believed to correctly describe the strong nuclear force. The low energy regime is highly non-linear. The problem of solving the theory is amenable only to computer simulation.

Precise calculation of the matrix elements of low energy QCD hadronic bound states is required for a number of important experimental determinations of the CKM matrix elements that parametrise the standard model. Theoretical uncertainty is dominant in a number of these quantities and the field of *lattice QCD* is critical to measuring these constants of nature.

Lattice QCD simulations directly evaluate the Feynman path integral in Euclidean space, using importance sampled Markov-Chain-Monte-Carlo techniques to integrate over four (and in some cases five) space-time dimensional quantum fields. The fermion fields have 4×3 spin-color structure, and the gauge fields have 3×3 color structure. When expressed as a multi-dimensional integral the number of independent degrees of freedom can exceed 10^7 in contemporary calculations.

Current simulations are performed on expensive massively parallel computers. Both our natural desire to advance our research, impatience at waiting several years for the results of simulations, the requirements of scientific competiveness, and the phenomenal cost of the fastest supercomputers are strong motivating influences to ensure that we invest in software infrastructure to maximise the performance our calculations.

Compiled code typically performs quite poorly. Runtimes are dominated by sparse matrix inversion of the Dirac operator, typically using some variant of red-black preconditioned Conjugate Gradient (CG). A good speed-up can be obtained by careful optimisation of a bounded body of code covering the sparse matrix (Dirac operator) application and the linear algebra used in the innermost CG loop.

This paper describes a system, collectively called BAGEL, for generating efficient QCD code for multiple platforms and for a several of the most popular of representations of the Dirac sparse matrix operator.

It supports two different memory layouts – favouring linear memory streaming or memory locality – either of which can be selected in order to tune to a given

memory system.

The structure of the paper is as follows: Section 2 contains background discussion of why compilers typically fail to produce optimal code. Section 3 describes the design goals of the BAGEL system. Section 4 describes the software interface to the BAGEL system, and Section 5 describes compilation and usage. Considerations for prudent scientific exploitation, such as software and hardware test infrastructure, are described in section 6. Section 7 discusses performance and precision considerations.

2 Problem characterisation

We are interested in applying Krylov space solvers, for example the Conjugate Gradient algorithm to solve the manifestly Hermitian (complex valued) sparse matrix equation for a matrix M,

$$\psi = (M^\dagger M)^{-1}\eta$$

where indices are suppressed. Red-black checkerboard preconditioning is used. In four dimensional systems, the checkerboard is defined by even and odd parities $p = (x + y + z + t)|2$, and M is block decomposed into

$$M = \begin{pmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ M_{oe}M_{ee}^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{pmatrix} \begin{pmatrix} 1 & M_{ee}^{-1}M_{eo} \\ 0 & 1 \end{pmatrix}$$

The preconditioned system solves the Hermitian system

$$\psi_o = (M_{\rm prec}^\dagger M_{\rm prec})^{-1}\eta'_o$$

on the odd checkerboard with the Schur complement

$$M_{\rm prec} = M_{oo} - M_{oe}M_{ee}^{-1}M_{eo}.$$

For the non-Hermitian system relevant to valence propagator calculations we use CG-NE, where $\eta'_o = \eta_o - M_{oe}M_{ee}^{-1}\eta_e$, and $\psi_e = M_{ee}^{-1}\eta_e - M_{ee}^{-1}M_{eo}\psi_o$. For Wilson fermions both M_{ee} and its inverse are trivial and can be scaled to the identity, making their application cost free.

2.1 Wilson fermions

For Wilson fermions the vectors ψ and η represent $4 \times 3 \times L^3 \times T$ complex numbers: here $L \simeq 32$ and $T \simeq 64$ represent a discrete spatial and temporal grid. The matrix M is complex valued with naive dimension $(4 \times 3 \times L^3 \times T)^2$. However it is sparse with a nearest neighbour coupling term representing the gauge covariant Dirac operator, and a mass term proportional to the identity. We use the DeGrand-Rossi basis for the Dirac matrices γ_μ in common with CHROMA [1] and CPS [2].

$$D_W(m) \equiv M_{x,x'} = m - \sum_\mu \frac{1}{2}\Big[U_\mu(x)(1-\gamma_\mu)\delta_{x+\hat{\mu},x'} + U^\dagger_\mu(x')(1+\gamma_\mu)\delta_{x-\hat{\mu},x'} - 2\delta_{x,x'}\Big]$$

The Wilson-Dirac hopping term involves 12 additions for two spinor projection for each of 8 forward and backward directions, 12×8 multiplies and 60×8 multiply-adds associated with SU(3) multiplication and $24 \times (8-1)$ additions to accumulate the result vector. The total is 1320 floating point operations per site, composed of 96 multiplies, 480 multiply-accumulates, and 264 additions. The unpaired multiplies and adds in the Wilson kernel means that for microprocessors whose peak floating point throughput is attained only when issuing multiply-add instructions the "speed-of-light" performance that QCD is guaranteed not to exceed is 78% of peak.

2.2 Domain wall fermions

Domain wall fermions are represented with a five dimensional system, with fifth dimension length L_s and coordinate s [3–5]. It uses the Wilson operator as a key building block. Here,

$$M = D^{\rm dwf}_{x,s;x',s'}(M_5, m_f) = \delta_{s,s'} D^{\parallel}_{x,x'}(M_5) + \delta_{x,x'} D^{\perp}_{s,s'}(m_f)$$

$$D^{\parallel}_{x,x'}(M_5) = D_W(-M_5)$$

$$\begin{aligned} D^{\perp}_{s,s'}(m_f) = & \frac{1}{2}\Big[(1-\gamma_5)\delta_{s+1,s'} + (1+\gamma_5)\delta_{s-1,s'} - 2\delta_{s,s'}\Big] \\ & - \frac{m_f}{2}\Big[(1-\gamma_5)\delta_{s,L_s-1}\delta_{0,s'} + (1+\gamma_5)\delta_{s,0}\delta_{L_s-1,s'}\Big]. \end{aligned} \quad (1)$$

The additional dimension introduces an ambiguity in the choice of checkerboarding and two different preconditioning schemes are used by the CPS and CHROMA. The CPS approach defines a five dimensional checkerboarding, *rb5d*, $p = (x + y + z + t + s)|2$. This has the attraction that M_{ee} remains free (as in the Wilson case) and parallelisation in the fifth dimension is very effective.

Alternatively, one can continue to define a four dimensional checkerboarding of this five dimensional system [6], $p = (x + y + z + t)|2$. This facilitates uniform treatment of both four dimensional and five dimensional Fermion formulations as an important practical consideration in the CHROMA code base. In this *rb4d* scheme M_{ee} contains a nearest neighbour coupling in the fifth dimension, and its inverse is non-local in the fifth dimension. The cost is of $O(L_s)$ and is not prohibitive, however it introduces a serial dependency between different coordinates in the fifth dimension that prevents effective geometric parallelisation in this direction.

Both these approaches are supported by Bagel. We note that while the rb5d scheme is simpler and requires fewer floating point operations, the rb4d scheme empirically produces more precise results for a given CG stopping condition. In particular in mixed precision schemes the required number of double precision iterations tends to be reduced, as a result of the different the conditioning effects of the different schemes.

3 Compiler deficiencies

If compilers produced efficient code the BAGEL system would not exist. Poor performance is typically delivered by the compiled C++ typically in existence. C++ code in popular QCD packages can yield as little as 1% of peak performance on modern microprocessors, while even "hero programmed" C++ still yields as little as 10% of peak. As shall be seen, carefully tuned assembler achieves in the range 20-50% of peak on modern systems. Analysis of intermediate assembler generated by both the GCC compiler and vendor compilers such as IBM's xlc identifies some weaknesses [7]. Indeed, one is often amazed that the ingenuity of microprocessor designers allows such poor code to perform as well as it does.

Register allocation : Compiled QCD code typically makes sub-optimal use of the large available register set in modern RISC chips. QCD codes are dominated by 3×3 complex matrix operations, and benefit greatly from using a large register pool to eliminate repeated loads and stores. The GCC compiler is particularly poor using the same two or three registers for otherwise indepdent operations which introduces artificial serial dependencies. This is likely

a result of the historical importance of x86 in the development of GCC's optimisations. The x86 architecture both has an exceedingly small register set, and (consequently!) the hardware register renaming is particularly good at avoiding this artificial serialisation. This is less often the case in RISC microprocessors.

Software prefetching : Compilers are typically rather poor at detecting memory access patterns in code, beyond simple AXPY type operations. Software prefetching via explicit L1 cache touch instructions are rarely generated for code containing non-trivial loop structures. On systems with hardware stream prefetching there one often still gains from software prefetching if the hardware fetch engine is external to the L1 cache.

4 BAGEL design

The first package described in this paper is the BAGEL assembler generation library. This is, in essence, a domain specific compiler for QCD. The programming interface is a quirky C++ API, rather than a parsed computer programming language. It is used to create and manipulate a list of objects representing an abstracted RISC-like assembler code sequence. This sequence is then translated to and optimised for a number of architectures, table 1.

Table 1
Bagel output targets

ppc440, powerIII	PowerPC and Power standard floating point
bgl	PowerPC with double Hummer extensions
noarch	C-code output with complex scalar datatype
noarch-simd	C-code output with complex vector datatype
usparcII	SPARC (historically Sun) Fujitsu currently
alpha	Alpha, mainly historical

The interface is designed to force the programmer to clearly specify important information that compilers fail to extract from C++ code. This includes deciding the register usage and loop unrolling strategies and declaring the memory access/prefetch pattern. The interface is also designed to automate the tasks that are done well and without fuss by computer algorithms - loop unrolling, software pipelining and pipeline scheduling.

The approach provides facilities for load/store/operate on complex data types. Some bounded number of complex and scalar point variables can be allocated, in the form of scalars or one, two, and three dimensional arrays. The total number is limited by the register set of the target architecture, and is typically

Table 2
Some examples of the programming interface provided by BAGEL

Scalar complex variable	`int x = alreg(Cregs);`
Scalar float variable	`int x = alreg(Fregs);`
Scalar integer variable	`int i = alreg(Iregs);`
Array complex variable	`reg_array_1d(psi,Cregs,3);`
Complex $z = x + y$	`complex_add(z,x,y)`
Complex $z = x - y$	`complex_sub(z,x,y)`
Complex $z = x * y$	`complex_mul(z,x,y)`
Complex load	`complex_load(z,offset,ptr)`
Complex store	`complex_store(z,offset,ptr)`
Stream declaration	`s = create_stream(block,ptr,STRIDED,stride);`
Stream prefetch	`prefetch_stream(s);`
Stream iterate	`iterate_stream(s);`
Unrolled loop	`for(int i=0;i<3;i++) {}`
Executed loop	`r = start_loop(count);`
	`stop_loop(r);`

either 32 or 64. Complex arithmetic operations use dedicated hardware where available, e.g. BlueGene/L and BlueGene/P. These primitives are cracked into multiple scalar floating point operations on other architectures.

Pointer/integer data types and integer arithmetic operations are provided. Pointers are dereferenced in explicit load/store calls that allow modest offsets to be specified (mapping to address offsets directly encoded in a processor instruction). While integer arithmetic can be used for pointer update, using the stream facility enables both automatic pointer update and simultaneously advises the system to automatically generate software prefetch for the requested pattern. Stream patterns include linear, strided, and the lookup table driven gather/scatter idiom common in sparse matrix code.

Some examples are tabulated in table 2

The BAGEL package performs a greedy issue optimisation pass, where a plan of usage for each of the processor pipelines is constructed. The instruction stream is reordered such that each instruction is raised, consistent with allowed dependency conflicts, to the earliest available pipeline issue slot not already occupied by an earlier instruction.

Register RAW (read-after-write), WAW and WAR dependencies are tracked

and respected in the reordering. Dependencies through the memory system are not tracked, and if present the programmer must required to insert explicit load/store barriers. This is not typically required in high performance kernels, however, and the restriction is somewhat analogous to assuming that the `__restrict` C++ keyword implicitly used.

The instructions are classed in related issue groups. Each target processor provides a model detailing the number of pipelines, their produce-to-use latency, and the issue groups served by each pipeline. The processor file also specifies opcode translations from the abstracted assembler into the target assembly language. Support is provided for several generations of each of the Alpha, Sparc and PowerPC microprocessor lines. Support is included for special variants of PowerPC such as the double Hummer (complex) instruction set extensions in BlueGene/L and BlueGene/P systems. This automates the tedious task of translating between broadly similar RISC instruction sets and scheduling instructions for the details of a particular microprocessor implementation. Thus, after a single (and certainly non-trivial) coding effort one supports many targets, each with the performance of hand tuned assembler.

5 Bagel Fermion Matrix package

We describe a second software package, BFM, which makes use of the above BAGEL package and presents an increasingly widely used software interface for QCD. This is made use of by both the UKQCD hadronic measurement code, and the USQCD CHROMA [1] and CPS [2] software packages to obtain high performance on US and UK national lattice gauge theory facilities including QCDOC [8,9,7,10], BlueGene/L, and BlueGene/P systems. The package has also been used by a number of groups throughout Europe, and has been used throught RBC and UKQCD programme of Domain Wall Fermion simulations [11–20]. In modern microprocessors, memory system considerations are at least as important as floating point pipeline issues. BFM implements two distinct approaches to the memory system, and the optimal one for each architecture should be empirically determined. We consider the specific example of the implementation of the 4-d Wilson-Dirac operator D_W.

The *stream* memory strategy produces long patterns of sequential reads and benefits from prefetch hardware. In the loop ordering detailed below, no more than two concurrent read streams are required, and contiguous sequences run the full length of the vector. The implementation uses an index ordering where the left most index moves fastest in memory, and $\vec{x}$ represents a four dimensional coordinate in x, y, z, t order. This order is optimal for QCDOC which has no L2 cache, but rather a similar sized addressable on-chip embedded DRAM memory. This memory system had a write bandwidth that exceeded

read bandwidth and would only prefetch two concurrent read streams.

$$\forall \vec{x}, \forall \mu$$
$$\chi(0, \mu, \vec{x} + \hat{\mu}) = U(\mu, \vec{x})^{\dagger}(1 + \gamma_{\mu})\psi(\vec{x})$$
$$\chi(1, \mu, \vec{x} - \hat{\mu}) = (1 - \gamma_{\mu})\psi(\vec{x})$$

$$\forall \vec{x}$$
$$\phi(\vec{x}) = \sum_{\mu} \chi(0, \mu, \vec{x}) + U(\mu, \vec{x})\chi(1, \mu, \vec{x})$$

The *cache* memory strategy produces high reuse at the L2 cache level but with only short sequences of contiguous reads. Gauge fields are "double-stored" with $U(+, \mu, \vec{x}) \equiv U^{\dagger}(-, \mu, \vec{x} + \hat{\mu})$. Here, depending on the L2 capacity, up $2 * N_d = 8$ fold reuse of a loaded site of ψ is available, however the typical access length is only $N_c \times N_s = 24$ words, or 192 bytes. Therefore, this does not make particularly good use of any prefetch engine closer than the L2 cache.

$$\forall \vec{x}$$
$$\phi(x) = \textstyle\sum_{\mu} U(+, \mu, \vec{x})^{\dagger}(1 - \gamma_{\mu})\psi(\vec{x} + \hat{\mu}) + U(-, \mu, \vec{x})^{\dagger}(1 + \gamma_{\mu})\psi(\vec{x} - \hat{\mu})$$

For domain wall fermions, we consider the four-dimensional component $D_{\parallel}$. There is an added attraction for this loop ordering. On the BlueGene architecture the register capacity is 32 complex words, which is ample to retain both an entire 3×3 SU(3) matrix and maintain temporary vectors on which it operates. Since the same gauge link is applied on multiple 4-d space-time planes with different coordinates in the fifth dimension, the following loop ordering allows better use of prefetching with a typical read run-length around 3KB. It also retains the benefit of substantial L2 cache reuse, and the implementation applies the same SU(3) matrix L_s times without reloading the data to registers. This roughly halves the load store overhead.

$$\forall \vec{x}\{$$
$$\forall \mu, \forall s\{$$
$$\chi(+, \mu, s) = U(+, \mu, \vec{x})^{\dagger}(1 - \gamma_{\mu})\psi(s, \vec{x} + \hat{\mu})$$
$$\chi(-, \mu, s) = U(-, \mu, \vec{x})^{\dagger}(1 + \gamma_{\mu})\psi(s, \vec{x} - \hat{\mu})$$
$$\}$$
$$\phi(s, x) = \textstyle\sum_{s} \sum_{\mu} \chi(+, \mu, s) + \chi(-, \mu, s)$$
$$\}$$

5.1 Software interface

The software interface to BFM is flexible and designed to enable users to access it at a number of levels. Firstly, there is a code neutral interface that does not have dependencies on third party QCD code bases. BFM supports a number of different memory layout options according to architecture it is most natural to present an interface that seperates internal from external representation of data vectors, with import/export facilities provided. Vectors in the internal format can be allocated, imported/exported and manipulated using opaque references of type *Fermion_t*.

```
template <class Float> class bfmbase  {

  Fermion_t allocFermion   (int mem_type=mem_slow);
  void      freeFermion    (Fermion_t handle);

  void      impexFermion  (double *psi,
                           Fermion_t handle,
                           int import,
                           int cb,
                           int s) ;

  void      importFermion  (double *psi,
                            Fermion_t handle,
                            int cb);

  void      exportFermion  (double *psi,
                            Fermion_t handle,
                            int cb);

  void      importGauge(double *gauge, int dir) ;

}
```

BFM contains an implementation of the conjugate gradient algorithm. However, the interface presents appropriate sparse matrix and linear algebra building blocks that can easily be used to implement other Krylov solvers.

The internal precision used can be selected with a template argument, and the interface consists of the following methods.

```
  /*Linalg*/
  double    norm(Fermion_t psi);
```

```
void      scale(Fermion_t psi,double a);

void      axpy(Fermion_t z, Fermion_t x, Fermion_t y,double a);

void      axpby(Fermion_t z,
                Fermion_t x,
                Fermion_t y,
                double a,
                double b);

double    axpy_norm(Fermion_t z,
                    Fermion_t x,
                    Fermion_t y,
                    double a);

double    axpby_norm(Fermion_t z,
                     Fermion_t x,
                     Fermion_t y,
                     double a,
                     double b);

void      chiralproj_axpby(Fermion_t y,
                           Fermion_t x,
                           double a,
                           int sign,
                           int Ns);

/*Fermion matrix application*/
double Mprec(Fermion_t chi,    // Preconditioned on checkerboard 0.
             Fermion_t psi,    // Returns norm of result if donorm!=0
             Fermion_t tmp,
             int dag,int donrm=0) ;

void Munprec(Fermion_t chi[2],  // Unpreconditioned
             Fermion_t psi[2],
             Fermion_t tmp,
             int dag) ;

/*Conjugate gradient implementation*/
int CGNE(Fermion_t sol_guess[2], Fermion_t source[2]);
int CGNE_prec(Fermion_t sol_guess, Fermion_t source);
```

An interface is provided to enable interaction with the codes making use of the SciDAC QDP++ library, and its' *LatticeFermion* and *LatticeColorMatrix* types. This includes both UKQCD's measurement system and Jefferson

Laboratory's CHROMA code base. The additional methods are

```
  void importGauge(multi1d<LatticeColorMatrix> &u);

  void importFermion  (LatticeFermion &psi,
                       Fermion_t handle,
                       int cb,
                       int s=0);

  void exportFermion  (LatticeFermion &psi,
                       Fermion_t handle,
                       int cb,
                       int s=0);

  // for DWF
  void importFermion  (multi1d<LatticeFermion> &psi,
                       Fermion_t handle,
                       int cb);

  void exportFermion  (multi1d<LatticeFermion> &psi,
                       Fermion_t handle,
                       int cb);
```

For convenience, a QDP++ wrapper is provided to call the BFM domain wall wall solver, and handle the four dimensional to five dimensional fermion field projection. The routine also computes the standard five dimensional observables required in DWF analyses with a low memory footprint implementation.

```
template <class Float_f,class Float_d>
int dwf_CG(LatticePropagator &sol,
           LatticePropagator &src,
           multi1d<LatticeColorMatrix> &U,
           std::basic_string<char> output_stem,
           int Ls,
           Real mass,
           Real M5,int Ncg,Real residual[],int max_iter[])
```

The implementation uses a *Ncg* complete restarts of the CG algorithm, with the first *Ncg-1* passes using precision of template type *Float_f* and the final pass using *Float_d*. These can be either single or double precision. In the mixed precision case, one gets most of the gain in bandwidth/performance of single precision, while retaining the double precision accuracy of the final results.

6 Compilation

The package is available online at

`http://www.ph.ed.ac.uk/~paboyle/bagel/bagel.html`

It is often used in conjunction with QMP, QDP, and CHROMA or CPS

```
http://usqcd.jlab.org/usqcd-docs/qmp/
http://usqcd.jlab.org/usqcd-docs/qdp++/
http://usqcd.jlab.org/usqcd-docs/chroma/
http://qcdoc.phys.columbia.edu/chulwoo_index.html
```

The packages use the GNU autoconf tools as their build system. There are a number of options, and we give the appropriate commands for a BlueGene/P system as examples for the bagel package:

```
./configure --enable-isize=4 \
            --enable-itype="unsigned long" \
            --enable-ifmt=%lx \
            --prefix=<install>
make all install
```

and the bfm package:

```
./configure --host=none \
            --build=none \
            --enable-strategy=stream \
            --enable-allocator=memalign \
            --enable-target-cpu=bgl \
            --with-bagel=<install>\
            --prefix=<install>\
            CXX=mpicxx
make all install
```

7 Test infrastructure

The test infrastructure is in the bfm/tests subdirectory. These largely rely on regression to the QDP++ and CHROMA libraries, and these are prerequisites for most of the tests, however a single node regression test against reference data files is provided for stand-alone compiles of BFM.

The most important of these are listed in table 3.

Table 3
BFM test infrastructure

bfm/tests subdirectory	Functionality
linalg	Linear combination, dot product etc...
wilson	Wilson fermion checkerboarded matrix application
wilson_singlenode	Standalone (neutral) wilson fermion matrix
dwf_mooee	DWF rb4d checkerboarded building block
dwf_mooeeinv	DWF rb4d checkerboarded building block
dwf_prec	DWF rb4d checkerboarded matrix
dwf_cg	DWF rb4d checkerboarded CG for 5d system
dwf_rb5d_dperp	DWF rb5d checkerboarded 5d hopping term
dwf_rb5d	DWF rb5d checkerboarded matrix
dwf_rb5d_cg	DWF rb5d checkerboarded CG for 5d system
dwf_rb5d_prop	12 component 4d propagator solution
	with $4d \leftrightarrow 5d$ projection.
majority_vote	Hardware test and diagnostic

In addition to software correctness tests, this code is used on massively parallel computers and consumes a large amount of computer time - many thousands of nodes for several years in the case of RBC-UKQCD simulations. Silent node failures are a reality in any sufficiently large machine, and scientific prudence requires hardware self test be carried out. Further, when reproducibility problems are discovered it is imperative that accurate identification of the failing components be made.

The subdirectory *bfm/tests/majority_vote* contains a hardware majority vote test. This runs the same conjugate gradient algorithm on the same randomly generated data vectors on every node concurrently, having all nodes use the same RNG sequence. The conjugate gradient is run many times over in an infinite loop with the random numbers refreshed after each inversion. The virtual global sum method is overridden in this test to substitute a cross-comparison of each node to the corresponding data on node zero. Any "broken" nodes presenting a local norm contribution that differs from that on node zero will conveniently identify itself for replacement.

It is possible to run this test with or without boundary face communication between nodes. In the non-communicating mode, the nodes decouple their calculations entirely and any local computational failures (e.g. memory system, FPU etc...) will attributed to a unique node.

The test can also be run with boundary communication enabled. Silent hardware errors that arise in communications themselves are highly non-trivial to debug as attribution to a unique entity is not possible. Whether the problem is send side or receive side cannot be determined in this scenario and intelligent diagnostic tests are invaluable. This test does as well as possible: and a set of nodes including faulty hardware is identified. Multiple runs can be used to narrow down the intersection of all nodes to which the error propagates to enable replacablement.

An earlier version of this test included software checksumming of send and receive buffers in this latter scenario and can be made available on request. This earlier form of this diagostic has been used extensively on by RBC and UKQCD on QCDOC, and the present code has also been transferred to IBM Research for use in BlueGene/P systems. In particular it has been used during shakeout of the petaflop/s system at Argonne National Laboratory.

8 Performance and mixed precision

Performance figures quoted in this section are relative to the true peak performance of the chip, and the reader should bear in mind that a figure of 78% of peak would represent perfect pipeline usage for the (dominant) Wilson-Dirac operator, and that the entire CG algorithm has a "speed-of-light" a little below this due to the additional linear algebra.

Table 4 gives the performance of the full CG algorithm, including all communication costs on a 4^4 volume on QCDOC, BG/L and BG/P systems. For comparison we include the performance of the CHROMA C++ package. This is done under the caveat that the CHROMA authors broadly expect and support calling optimised sparse matrix code on major architectures, and a number of optimised codes exist [21–24].

Single precision acceleration in Lattice QCD must be used with care. Is certainly safe to use during the molecular dynamics phase of an HMC evolution if a precise double precision solution is used for the Metropolis accept/reject step and the initial guess for the solution does not in itself violate Metropolis reversibility. The worst that can then happen is a loss of acceptance which will be evident in the simulation performance.

For valence propagator measurements double precision accuracy remains imperative, and the restarted CG approach described above recommended. Any gain is determined by the combination of the ratio of single precision to double precision performance, and the combination of the number of iterations required for a single double precision solve and the number of iterations in

Table 4
Per-core performance of *entire* DWF conjugate gradient algorithm on a $4^4 \times 8$ local volume where the code is spread out in all four dimensions. Lattice QCD scales well in toroidal machines when weak scaling is considered until global sums become dominant, and this is representative of the performance per core of code on very large systems with the same local volume. On BlueGene systems all cores are used concurrently, to fairly represent memory system contention. The global volume is chosen according to the number of processors to keep this local volume and thus the surface/volume ratio fixed.

Platform	precision	nodes	performance/node	% peak
QCDOC	double	64	270 Mflop/s	33%
QCDOC	single	64	320 Mflop/s	40%
BG/L	double	512	420 Mflop/s	15%
BG/L	single	512	700 Mflop/s	25%
BG/P	double	512	500 Mflop/s	15%
BG/P	single	512	830 Mflop/s	24%

each of passes of a restarted mixed precision approach.

For illustration, the counts for *rb5d* on the lightest quark mass in RBC-UKQCD's 24^3, 2.13 GeV simulation is given in table 5 with a stopping condition of $1.0e^{-8}$.

Table 5
Comparison of conjugate gradient iteration counts for double precision and mixed precision. A similar quality result can be obtained with around 90% of iterations performed on single precision with only a 10% increase in the total number of iterations. Given single precision code can perform up to twice as fast, this is a substantial gain on certain architectures. Indeed were exotic hardware such as graphics related considered, the gains could be even higher.

double	1000
single/single/double	800,200,100

9 Future developments

9.1 Additional Fermion actions

All the actions supported by Bagel are broadly speaking Wilson-like. The generalisation to cover Clover and Twisted-mass actions is relatively simple, and the relevant assembler building blocks have been in existence for quite some

time. A revision of the package to include Clover and Twisted mass actions is envisaged later this year once the author has had time to develop the C++ wrapping code, and more importantly the regression and test infrastructure.

The plethora of approaches to the Overlap operator suggest to the author that the current implemention of a fast Wilson operator coupled to a more flexible programming environments, such as CHROMA is the best approach.

9.2 Wider SIMD support

This new version of BFM and Bagel also contains an extension of the abstraction model to cover short-vector SIMD instructions. These are increasingly important targets, with the x86/SSE, Altivec (VMX) and CELL platforms. The latter two fit well with the RISC specific nature of Bagel, while x86 support is planned using the equivalent "C" output (target `noarch-simd`) for Bagel coupled to a future use of vector datatypes.

The complex datatype abstraction is generalised to a short vector of complex, of architecture dependent length. Current architectures use length 1. The strategy taken for greater lengths is worth some discussion as it quite naturally generalises to longer lengths, such as the planned 16 for the forthcoming Intel Larrabee graphics/HPC product once the author has access.

We note that QCD codes are constructed on a simple Cartesian lattice and are most naturally thought of in a data parallel fashion. This is evidenced by the success of the Connection Machines, APE projects, and the QDP++ data parallel package. QCD codes are sadly shoehorned into the more commercially successful SPMD programming model, and the global lattice is subdived geometrically into domains of responsibility. Scalar variables and operations, (as opposed to the truly parallel, site-indexed, vector computation) are replicated in a redundant fashion on each of the processors.

With (short) vector SIMD, of length N_{simd}, it is quite natural to think of the global lattice being subdived geometrically into N_{simd} *more* domains of responsibility than there are processing cores. Each processing cores handles N_{simd} domains of responsibility in parallel, and the usual problem of suitability of the operations and loops to longer vector lengths is solved.

The only cross-talk between different elements of SIMD vectors will arise quite naturally in the infrequent communications phases, and are suppressed by the surface to volume ratio. Reasonably powerful permute/extract/merge operations suffice to ensure that this is efficient.

BFM supports this mixed SIMD and MIMD geometric decomposition, and a

"C" output noarch-simd with $N_{\mathrm{simd}} = 2$ has been implemented, corresponding to vectors of two complex numbers $(\mathrm{re}, \mathrm{im}, \mathrm{re}, \mathrm{im})$. This preparatory work does not yet have manifest significance, it is expected to become highly significant in the next few years as new and wider SIMD architectures become available.

10 Conclusions

This paper presents a quite sophisticated approach to the problem of generating high performance code for QCD, and in principle other fields. Domain specific compiler technology has been developed that enables machine details to be obscured and multiple architectures covered by a single code base. Pipeline scheduling is automated and development time is reduced compared to hand coded assembler. More complex kernels than one might care to hand code are enabled.

Significant effort has been made to ensure the optimised code can be easily used by some of the most ubiquitous packages in the field.

The software is released under the terms of the GNU Public License v2. The author politely requests that any scientific papers containing results that depend on data produced with any version of BAGEL whether through gauge configurations, propagators, or correlation functions should acknowledge that use with reference to this paper.

References

[1] Edwards, R. G. and Joo, B., Nucl. Phys. Proc. Suppl. **140** (2005) 832.

[2] Boyle, P. A. et al., Nucl. Phys. Proc. Suppl. **140** (2005) 829.

[3] Blum, T. and Soni, A., Phys. Rev. **D56** (1997) 174.

[4] Shamir, Y., Nucl. Phys. **B406** (1993) 90.

[5] Kaplan, D. B., Phys. Lett. **B288** (1992) 342.

[6] Brower, R. C., Neff, H., and Orginos, K., Nucl. Phys. Proc. Suppl. **153** (2006) 191.

[7] Boyle, P. A., Jung, C., and Wettig, T., ECONF **C0303241** (2003) THIT003.

[8] Boyle, P. A. et al., Qcdoc: A 10 teraflops computer for tightly-coupled calculations, in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 40, Washington, DC, USA, 2004, IEEE Computer Society.

[9] Boyle, P. et al., IBM JRD **492 2/3** (2004) 351.

[10] Boyle, P. et al., Nucl. Phys. Proc. Suppl. **140** (2005) 169.

[11] Antonio, D. J. et al., Phys. Rev. **D75** (2007) 114501.

[12] Allton, C. et al., Phys. Rev. **D76** (2007) 014504.

[13] Antonio, D. J. et al., Phys. Rev. **D77** (2008) 014509.

[14] Boyle, P., Proceedings of Science **PoS(Lat2007)** (2007) 005.

[15] Antonio, D. J. et al., Phys. Rev. Lett. **100** (2008) 032001.

[16] Boyle, P. A. et al., JHEP **07** (2008) 112.

[17] Allton, C. et al., Phys. Rev. **D78** (2008) 114509.

[18] Aoki, Y. et al., Phys. Rev. **D78** (2008) 054510.

[19] Boyle, P. A. et al., Phys. Rev. Lett. **100** (2008) 141601.

[20] Boyle, P. A., Flynn, J. M., Juttner, A., Sachrajda, C. T., and Zanotti, J. M., JHEP **05** (2007) 016.

[21] McClendon, C., (2001).

[22] Pochinsky, A., Nucl. Phys. Proc. Suppl. **140** (2005) 859.

[23] Pochinsky, A., J. Phys. Conf. Ser. **46** (2006) 157.

[24] Doi, J., PoS **LAT2007** (2007) 032.