Grid Documentation

Peter Boyle, Guido Cossu, Antonin Portelli, Azusa Yamaguchi

Sep 25, 2018

CONTENTS:

CHAPTER

ONE

PRELIMINARIES

Attention: manual version 1 (CD)

Grid is primarily an *application development interface* (API) for structured Cartesian grid codes and written in C++11. In particular it is aimed at Lattice Field Theory simulations in general gauge theories, but with a particular emphasis on supporting SU(3) and U(1) gauge theories relevant to hadronic physics.

1.1 Who will use this library

As an application development interface *Grid* is primarily a programmers tool providing the building blocks and primitives for constructing lattice gauge theory programmes.

Grid functionality includes:

- Data parallel primitives, similar to QDP++
- gauge and fermion actions
- solvers
- gauge and fermion force terms
- integrators and (R)HMC.
- parallel field I/O
- object serialisation (text, XML, JSON. . .)

Grid is intended to enable the rapid and easy development of code with reasonably competitive performance.

It is first and foremost a *library* to which people can programme, and develop new algorithms and measurements. As such, it is very much hoped that peoples principle point of contact with Grid will be in the wonderfully rich C++ language. Since import and export procedures are provided for the opaque lattice types it should be possible to call Grid from other code bases.

Grid is most tightly coupled to the Hadrons package developed principally by Antonin Portelli. This package is entirely composed against the Grid data parallel interface.

Interfacing to other packages is also possible.

Several regression tests that combine Grid with Chroma are included in the Grid distribution. Further, Grid has been successfully interfaced to

- The Columbia Physics System
- The MILC code

1.2 Data parallel interface

Most users will wish to interact with Grid above the data parallel *Lattice* interface. At this level a programme is simply written as a series of statements, addressing entire lattice objects.

Implementation details may be provided to explain how the code works, but are not strictly part of the API.

Example

> For example, as an implementation detail, in a single programme multiple data (SPMD) message passing supercomputer the main programme is trivially replicated on each computing node. The data parallel operations are called *collectively* by all nodes. Any scalar values returned by the various reduction routines are the same on each node, resulting in (for example) the same decision being made by all nodes to terminate an iterative solver on the same iteration.

1.3 Internal development

Internal developers may contribute to Grid at a level below the data parallel interface.

Specifically, development of new lattice Dirac operators, for example, or any codes directly interacting with the

- Communicators
- Simd
- Tensor
- Stencil

will make use of facilities provided by to assist the creation of high performance code. The internal data layout complexities will be exposed to some degree and the interfaces are subject to change without notice as HPC architectures change.

Since some of the internal implementation details are needed to explain the design strategy of grid these will be documented, but labelled as *implementation dependent*

Reasonable endeavours will be made to preserve functionality where practical but no guarantees are made.

CHAPTER

TWO

REPORTING BUGS

To help us tracking and solving more efficiently issues with Grid, please report problems using the issue system of GitHub rather than sending emails to Grid developers.

When you file an issue, please go though the following checklist:

- Check that the code is pointing to the HEAD of develop or any commit in master which is tagged with a version number.
- Give a description of the target platform (CPU, network, compiler). Please give the full CPU part description, using for example cat /proc/cpuinfo | grep 'model name' | uniq (Linux) or sysctl machdep.cpu.brand_string (macOS) and the full output the –version option of your compiler.
- Give the exact configure command used.
- Attach config.log.
- Attach grid.config.summary.
- Attach the output of make V=1.
- Describe the issue and any previous attempt to solve it. If relevant, show how to reproduce the issue using a minimal working example.

CHAPTER

THREE

DOWNLOAD, INSTALLATION AND BUILD

3.1 Required libraries

- GMP,
- MPFR
- Eigen: bootstrapping grid downloads and uses for internal dense matrix (non-QCD operations) the Eigen library.

Grid optionally uses:

- HDF5
- LIME for ILDG and SciDAC file format support.
- FFTW either generic version or via the Intel MKL library.
- LAPACK either generic version or Intel MKL library.

3.2 Compilers

- Intel ICPC v17 and later
- Clang v3.5 and later (need 3.8 and later for OpenMP)
- GCC v4.9.x
- GCC v6.3 and later (recommended)

Important:

Some versions of GCC appear to have a bug under high optimisation (-O2, -O3).

The safety of these compiler versions cannot be guaranteed at this time. Follow Issue 100 for details and updates.

GCC v5.x

GCC v6.1, v6.2

3.3 Quick start

First, start by cloning the repository:

```
git clone https://github.com/paboyle/Grid.git
```

Then enter the cloned directory and set up the build system:

```
cd Grid
./bootstrap.sh
```

Now you can execute the *configure* script to generate makefiles (here from a build directory):

```
mkdir build; cd build
../configure --enable-precision=double --enable-simd=AVX --enable-comms=mpi-auto \
    --prefix=<path>
```

where:

```
--enable-precision=single|double
```

sets the **default precision**. Since this is largely a benchmarking convenience, it is anticipated that the default precision may be removed in future implementations, and that explicit type selection be made at all points. Naturally, most code will be type templated in any case.:

```
--enable-simd=GEN|SSE4|AVX|AVXFMA|AVXFMA4|AVX2|AVX512|NEONv8|QPX
```

sets the **SIMD architecture**,

```
--enable-comms=mpi|none
```

selects whether to use MPI communication (mpi) or no communication (none).

```
--prefix=<path>
```

should be passed the prefix path where you want to install Grid.

Other options are detailed in the next section, you can also use

```
configure --help
```

to display them.

Like with any other program using GNU autotool, the

```
CXX, CXXFLAGS, LDFLAGS, ...
```

environment variables can be modified to customise the build.

Finally, you can build, check, and install Grid:

```
make;
make check;
make install
```

If you want to build all the tests just use *make tests*.

3.3.1 Detailed build configuration options

Option	usage
`--prefix=path`	installation prefix for Grid.
`--with-gmp=path`	look for GMP in the UNIX prefix <*path*>
`--with-mpfr=path`	look for MPFR in the UNIX prefix <*path*>
`--with-fftw=path`	look for FFTW in the UNIX prefix <*path*>
`--with-lime=path`	look for c-lime in the UNIX prefix <*path*>
`--enable-lapack[=path]`	enable LAPACK support in Lanczos eigensolver. A UNIX prefix containing the library can be specified (optional).
–enable-mkl[=path]	use Intel MKL for FFT (and LAPACK if enabled) routines. A UNIX prefix containing the library can be specified (optional).
–enable-simd=code	setup Grid for the SIMD target <*code*>*'(default: 'GEN*). A list of possible SIMD targets is detailed in a section below.
–enable-gen-simd-width=size	select the size (in bytes) of the generic SIMD vector type (default: 32 bytes). E.g. SSE 128 bit corresponds to 16 bytes.
–enable-precision=single|double	set the default precision (default: *double*).
–enable-comms=mpi|none	use <*comm*> for message passing (default: *none*).
–enable-rng=sitmo|ranlux48|mt19937	choose the RNG (default: *sitmo*).
–disable-timers	disable system dependent high-resolution timers.
–enable-chroma	enable Chroma regression tests.
–enable-doxygen-doc	enable the Doxygen documentation generation (build with *make doxygen-doc*)

3.3.2 Possible communication interfaces

The following options can be use with the *–enable-comms=* option to target different communication interfaces:

<comm>	Description
none	no communications
mpi	MPI communications with compiler CXX
mpi-auto	MPI communications with compiler CXX but clone flags from MPICXX

For the MPI interfaces the optional *-auto* suffix instructs the *configure* scripts to determine all the necessary compilation and linking flags. This is done by extracting the informations from the MPI wrapper specified in the environment variable *MPICXX* (if not specified *configure* will scan though a list of default names). The *-auto* suffix is not supported by the Cray environment wrapper scripts. Use the standard wrappers (*CXX=CC*) set up by Cray *PrgEnv* modules instead.

3.3.3 Possible SIMD types

The following options can be use with the *–enable-simd=* option to target different SIMD instruction sets:

<simd>	Description
GEN	generic portable vector code
SSE4	SSE 4.2 (128 bit)
AVX	AVX (256 bit)
AVXFMA	AVX (256 bit) + FMA
AVXFMA4	AVX (256 bit) + FMA4
AVX2	AVX 2 (256 bit)
AVX512	AVX 512 bit
NEONv8	[ARM NEON](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch07s03.html) (128 bit)
QPX	IBM QPX (256 bit)

Alternatively, some CPU codenames can be directly used:

<simd>	Description
KNL	[Intel Xeon Phi codename Knights Landing](http://ark.intel.com/products/codename/48999/Knights-Landing)
SKL	[Intel Skylake with AVX512 extensions](https://ark.intel.com/products/codename/37572/Skylake#@server)
BGQ	Blue Gene/Q

3.3.4 Notes

- We currently support AVX512 for the Intel compiler and GCC (KNL and SKL target). Support for clang will appear in future versions of Grid when the AVX512 support in the compiler is more advanced.
- For BG/Q only [bgclang](http://trac.alcf.anl.gov/projects/llvm-bgq) is supported. We do not presently plan to support more compilers for this platform.
- BG/Q performances are currently rather poor. This is being investigated for future versions.
- The vector size for the *GEN* target can be specified with the *configure* script option *–enable-gen-simd-width*.

3.4 Build setup for Intel Knights Landing platform

The following configuration is recommended for the Intel Knights Landing platform:

```
../configure --enable-precision=double\
             --enable-simd=KNL        \
             --enable-comms=mpi-auto  \
             --enable-mkl             \
             CXX=icpc MPICXX=mpiicpc
```

The MKL flag enables use of BLAS and FFTW from the Intel Math Kernels Library.

If you are working on a Cray machine that does not use the *mpiicpc* wrapper, please use:

```
../configure --enable-precision=double\
             --enable-simd=KNL        \
             --enable-comms=mpi       \
             --enable-mkl             \
             CXX=CC CC=cc
```

If gmp and mpfr are NOT in standard places (/usr/) these flags may be needed:

```
--with-gmp=<path>        \
--with-mpfr=<path>
```

where <*path*> is the UNIX prefix where GMP and MPFR are installed.

Knight's Landing with Intel Omnipath adapters with two adapters per node presently performs better with use of more than one rank per node, using shared memory for interior communication. We recommend four ranks per node for best performance, but optimum is local volume dependent.

```
../configure --enable-precision=double\
          --enable-simd=KNL        \
          --enable-comms=mpi-auto \
          --enable-mkl             \
          CC=icpc MPICXX=mpiicpc
```

3.5 Build setup for Intel Haswell Xeon platform

The following configuration is recommended for the Intel Haswell platform:

```
../configure --enable-precision=double\
           --enable-simd=AVX2       \
           --enable-comms=mpi-auto \
           --enable-mkl             \
           CXX=icpc MPICXX=mpiicpc
```

The MKL flag enables use of BLAS and FFTW from the Intel Math Kernels Library.

If gmp and mpfr are NOT in standard places (/usr/) these flags may be needed:

```
--with-gmp=<path>        \
--with-mpfr=<path>
```

where <*path*> is the UNIX prefix where GMP and MPFR are installed.

If you are working on a Cray machine that does not use the *mpiicpc* wrapper, please use:

```
../configure --enable-precision=double\
           --enable-simd=AVX2       \
           --enable-comms=mpi      \
           --enable-mkl             \
           CXX=CC CC=cc
```

Since Dual socket nodes are commonplace, we recommend MPI-3 as the default with the use of one rank per socket. If using the Intel MPI library, threads should be pinned to NUMA domains using:

```
export I_MPI_PIN=1
```

This is the default.

3.6 Build setup for Intel Skylake Xeon platform

The following configuration is recommended for the Intel Skylake platform:

```
../configure --enable-precision=double\
             --enable-simd=AVX512     \
             --enable-comms=mpi       \
             --enable-mkl             \
             CXX=mpiicpc
```

The MKL flag enables use of BLAS and FFTW from the Intel Math Kernels Library.

If gmp and mpfr are NOT in standard places (/usr/) these flags may be needed:

```
--with-gmp=<path>        \
--with-mpfr=<path>       \
```

where <*path*> is the UNIX prefix where GMP and MPFR are installed.

If you are working on a Cray machine that does not use the *mpiicpc* wrapper, please use:

```
../configure --enable-precision=double\
             --enable-simd=AVX512     \
             --enable-comms=mpi       \
             --enable-mkl             \
             CXX=CC CC=cc
```

Since Dual socket nodes are commonplace, we recommend MPI-3 as the default with the use of one rank per socket. If using the Intel MPI library, threads should be pinned to NUMA domains using:

```
export I_MPI_PIN=1
```

This is the default.

3.7 Build setup for AMD EPYC / RYZEN

The AMD EPYC is a multichip module comprising 32 cores spread over four distinct chips each with 8 cores. So, even with a single socket node there is a quad-chip module. Dual socket nodes with 64 cores total are common. Each chip within the module exposes a separate NUMA domain. There are four NUMA domains per socket and we recommend one MPI rank per NUMA domain. MPI-3 is recommended with the use of four ranks per socket, and 8 threads per rank.

The following configuration is recommended for the AMD EPYC platform:

```
../configure --enable-precision=double\
             --enable-simd=AVX2       \
             --enable-comms=mpi \
             CXX=mpicxx
```

If gmp and mpfr are NOT in standard places (/usr/) these flags may be needed:

```
--with-gmp=<path>        \
--with-mpfr=<path>
```

where <*path*> is the UNIX prefix where GMP and MPFR are installed.

Using MPICH and g++ v4.9.2, best performance can be obtained using explicit GOMP_CPU_AFFINITY flags for each MPI rank. This can be done by invoking MPI on a wrapper script omp_bind.sh to handle this.

It is recommended to run 8 MPI ranks on a single dual socket AMD EPYC, with 8 threads per rank using MPI and shared memory to communicate within this

command line
: mpirun -np 8 ./omp_bind.sh ./Benchmark_dwf –mpi 2.2.2.1 –dslash-unroll –threads 8 –grid 16.16.16.16 –cacheblocking 4.4.4.4

Where omp_bind.sh does the following:

```
#!/bin/bash

numanode=` expr $PMI_RANK % 8 `
basecore=`expr $numanode \* 16`
core0=`expr $basecore + 0 `
core1=`expr $basecore + 2 `
core2=`expr $basecore + 4 `
core3=`expr $basecore + 6 `
core4=`expr $basecore + 8 `
core5=`expr $basecore + 10 `
core6=`expr $basecore + 12 `
core7=`expr $basecore + 14 `

export GOMP_CPU_AFFINITY="$core0 $core1 $core2 $core3 $core4 $core5 $core6 $core7"
echo GOMP_CUP_AFFINITY $GOMP_CPU_AFFINITY

$@
```

3.7.1 Build setup for laptops, other compilers, non-cluster builds

Many versions of g++ and clang++ work with Grid, and involve merely replacing CXX (and MPICXX), and omit the enable-mkl flag.

Single node builds are enabled with:

```
--enable-comms=none
```

FFTW support that is not in the default search path may then enabled with:

```
--with-fftw=<installpath>
```

BLAS will not be compiled in by default, and Lanczos will default to Eigen diagonalisation.

CHAPTER

FOUR

EXECUTION MODEL

Grid is intended to support performance portability across a many of platforms ranging from single processors to message passing CPU clusters and accelerated computing nodes.

The library provides data parallel C++ container classes with internal memory layout that is transformed to map efficiently to SIMD architectures. CSHIFT facilities are provided, similar to HPF and cmfortran, and user control is given over the mapping of array indices to both MPI tasks and SIMD processing elements.

Identically shaped arrays then be processed with perfect data parallelisation. Such identically shaped arrays are called conformable arrays. The transformation is based on the observation that Cartesian array processing involves identical processing to be performed on different regions of the Cartesian array.

The library will both geometrically decompose into MPI tasks and across SIMD lanes. Local vector loops are parallelised with OpenMP pragmas.

Data parallel array operations can then be specified with a SINGLE data parallel paradigm, but optimally use MPI, OpenMP and SIMD parallelism under the hood. This is a significant simplification for most programmers.

The two broad optimisation targets are:

- MPI, OpenMP, and SIMD parallelism

Presently SSE4, ARM NEON (128 bits) AVX, AVX2, QPX (256 bits), and AVX512 (512 bits) targets are supported with aggressive use of architecture vectorisation intrinsic functions.

- MPI between nodes with and data parallel offload to GPU's.

For the latter generic C++ code is used both on the host and on the GPU, with a common vectorisation granularity.

4.1 Accelerator memory model

For accelerator targets it is assumed that heap allocations can be shared between the CPU and the accelerator. This corresponds to lattice fields having their memory allocated with *cudaMallocManaged* with Nvidia GPU's.

Grid does not assume that stack or data segments share a common address space with an accelerator.

- This constraint presently rules out porting Grid to AMD GPU's which do not support managed memory.
- At some point in the future a cacheing strategy may be implemented to enable running on AMD GPU's

CHAPTER

FIVE

DATA PARALLEL API

Data parallel array indices are divided into two types.

- Internal indices, such as complex, colour, spin degrees of freedom
- spatial (space-time) indices.

The ranges of all internal degrees are determined by template parameters, and known at compile time. The ranges of spatial indices are dynamic, run time values and the Cartesian structure information is contained and accessed via *Grid* objects.

Grid objects are the controlling entity for the decomposition of a distributed *Lattice* array across MPI tasks, nodes, SIMD lanes, accelerators. Threaded loops are used as appropriate on host code.

(binary) Data parallel operations can only be performed between Lattice objects constructed from the same Grid pointer. These are called *conformable* operations.

We will focus initially on the internal indices as these are the building blocks assembled in Lattice container classes. Every Lattice container class constructor requires a Grid object pointer.

5.1 Tensor classes

The Tensor data structures are built up from fundamental scalar matrix and vector classes:

```
template<class vobj      > class iScalar { private: vobj _internal ; }
template<class vobj,int N> class iVector { private: vobj _internal[N] ; }
template<class vobj,int N> class iMatrix { private: vobj _internal[N][N] ; }
```

These are template classes and can be passed a fundamental scalar or vector type, or nested to form arbitrarily complicated tensor products of indices. All mathematical expressions are defined to operate recursively, index by index.

Presently the constants

- Nc
- Nd

are globally predefined. However, this is planned for changed in future and policy classes for different theories (e.g. QCD, QED, SU2 etc...) will contain these constants and enable multiple theories to coexist more naturally.

Arbitrary tensor products of fundamental scalar, vector and matrix objects may be formed in principle by the basic Grid code.

For Lattice field theory, we define types according to the following tensor product structure ordering. The suffix "D" indicates either double types, and replacing with "F" gives the corresponding single precision type.

The test cases have R, which takes the compiled default precision (either F or D). This is for convenience only and may be deprecated in future forcing code external to Grid to choose the specific word size.

Type definitions are provided in qcd/QCD.h to give the internal index structures of QCD codes. For example:

```
template<typename vtype>
using iSinglet                    = iScalar<iScalar<iScalar<vtype> > >;
using iSpinMatrix                 = iScalar<iMatrix<iScalar<vtype>, Ns> >;
using iColourMatrix               = iScalar<iScalar<iMatrix<vtype, Nc> > > ;
using iSpinColourMatrix           = iScalar<iMatrix<iMatrix<vtype, Nc>, Ns> >;
using iLorentzColourMatrix        = iVector<iScalar<iMatrix<vtype, Nc> >, Nd > ;
using iDoubleStoredColourMatrix   = iVector<iScalar<iMatrix<vtype, Nc> >, Nds > ;
using iSpinVector                 = iScalar<iVector<iScalar<vtype>, Ns> >;
using iColourVector               = iScalar<iScalar<iVector<vtype, Nc> > >;
using iSpinColourVector           = iScalar<iVector<iVector<vtype, Nc>, Ns> >;
using iHalfSpinVector             = iScalar<iVector<iScalar<vtype>, Nhs> >;
using iHalfSpinColourVector       = iScalar<iVector<iVector<vtype, Nc>, Nhs> >;
```

Giving the type table:

Lattice	Lorentz	Spin	Colour	scalar_type	Field
Scalar	Scalar	Scalar	Scalar	RealD	RealD
Scalar	Scalar	Scalar	Scalar	ComplexD	ComplexD
Scalar	Scalar	Scalar	Matrix	ComplexD	ColourMatrixD
Scalar	Vector	Scalar	Matrix	ComplexD	LorentzColourMatrixD
Scalar	Scalar	Vector	Vector	ComplexD	SpinColourVectorD
Scalar	Scalar	Vector	Vector	ComplexD	HalfSpinColourVectorD
Scalar	Scalar	Matrix	Matrix	ComplexD	SpinColourMatrixD

The types are implemented via a recursive tensor nesting system.

Example

Here, the prefix "i" indicates for internal use, preserving the template nature of the class. Final types are declared with vtype selected to be both scalar and vector, appropriate to a single datum, or stored in a partial SoA transformed lattice object:

```
// LorentzColour
typedef iLorentzColourMatrix<Complex  > LorentzColourMatrix;
typedef iLorentzColourMatrix<ComplexF > LorentzColourMatrixF;
typedef iLorentzColourMatrix<ComplexD > LorentzColourMatrixD;

typedef iLorentzColourMatrix<vComplex > vLorentzColourMatrix;
typedef iLorentzColourMatrix<vComplexF> vLorentzColourMatrixF;
typedef iLorentzColourMatrix<vComplexD> vLorentzColourMatrixD;
```

Arbitrarily deep tensor nests may be formed. Grid uses a positional and numerical rule to associate indices for contraction in the Einstein summation sense.

Symbolic name	Number	Position
LorentzIndex	0	left
SpinIndex	1	middle
ColourIndex	2	right

The conventions are that the index ordering left to right are: Lorentz, Spin, Colour. A scalar type (either real or complex, single or double precision) is be provided to the innermost structure.

5.1.1 Tensor arithmetic rules (lib/tensors/Tensor_arith.h)

Arithmetic rules are defined on these types

The multiplication operator follows the natural multiplication table for each index, index level by index level.

Operator *

x	S	V	M
S	S	V	M
V	S	S	V
M	M	V	M

The addition and subtraction rules disallow a scalar to be added to a vector, and vector to be added to matrix. A scalar adds to a matrix on the diagonal.

Operator + and *Operator* -

+/-	S	V	M
S	S		M
V		V	
M	M		M

The rules for a nested objects are recursively inferred level by level from basic rules of multiplication addition and subtraction for scalar/vector/matrix. Legal expressions can only be formed between objects with the same number of nested internal indices. All the Grid QCD datatypes have precisely three internal indices, some of which may be trivial scalar to enable expressions to be formed.

Arithmetic operations are possible where the left or right operand is a scalar type.

Example:

```
LatticeColourMatrixD U(grid);
LatticeColourMatrixD Udag(grid);

Udag = adj(U);

RealD unitary_err = norm2(U*adj(U) - 1.0);
```

Will provide a measure of how discrepant from unitarity the matrix U is.

5.1.2 Internal index manipulation (lib/tensors/Tensor_index.h)

General code can access any specific index by number with a peek/poke semantic:

```
// peek index number "Level" of a vector index
template<int Level,class vtype>  auto peekIndex (const vtype &arg,int i);

// peek index number "Level" of a vector index
template<int Level,class vtype>  auto peekIndex (const vtype &arg,int i,int j);

// poke index number "Level" of a vector index
template<int Level,class vtype>
void pokeIndex (vtype &pokeme,arg,int i)
```

(continues on next page)

(continued from previous page)

```
// poke index number "Level" of a matrix index
template<int Level,class vtype>
void pokeIndex (vtype &pokeme,arg,int i,int j)
```

Example:

```
for (int mu = 0; mu < Nd; mu++) {
  U[mu] = PeekIndex<LorentzIndex>(Umu, mu);
}
```

Similar to the QDP++ package convenience routines are provided to access specific elements of vector and matrix internal index types by physics name or meaning aliases for the above routines with the appropriate index constant.

- peekColour
- peekSpin
- peekLorentz

and

- pokeColour
- pokeSpin
- pokeLorentz

For example, we often store Gauge Fields with a Lorentz index, but can split them into polarisations in relevant pieces of code.

Example:

```
for (int mu = 0; mu < Nd; mu++) {
  U[mu] = peekLorentz(Umu, mu);
}
```

For convenience, direct access as both an l-value and an r-value is provided by the parenthesis operator () on each of the Scalar, Vector and Matrix classes. For example one may write

Example:

```
ColourMatrix A, B;

A()()(i,j) = B()()(j,i);
```

bearing in mind that empty parentheses are need to address a scalar entry in the tensor index nest.

The first (left) empty parentheses move past the (scalar) Lorentz level in the tensor nest, and the second (middle) empty parantheses move past the (scalar) spin level. The (i,j) index the colour matrix.

Other examples are easy to form for the many cases, and should be obvious to the reader. This form of addressing is convenient and saves peek, modifying, poke multiple temporary objects when both spin and colour indices are being accessed. There are many cases where multiple lines of code are required with a peek/poke semantic which are easier with direct l-value and r-value addressing.

5.1.3 Matrix operations

Transposition and tracing specific internal indices are possible using:

```
template<int Level,class vtype>
auto traceIndex (const vtype &arg)

template<int Level,class vtype>
auto transposeIndex (const vtype &arg)
```

These may be used as

Example:

```
LatticeColourMatrixD Link(grid);
ComplexD link_trace = traceIndex<ColourIndex> (Link);
```

Again, convenience aliases for QCD naming schemes are provided via

- traceColour
- traceSpin
- transposeColour
- transposeSpin

Example:

```
ComplexD link_trace = traceColour (Link);
```

The operations only makes sense for matrix and scalar internal indices.

The trace and transpose over all indices is also defined for matrix and scalar types:

```
template<class vtype,int N>
auto trace(const iMatrix<vtype,N> &arg) -> iScalar

template<class vtype,int N>
auto transpose(const iMatrix<vtype,N> &arg  ) -> iMatrix
```

Similar functions are:

- conjugate
- adjoint

The traceless anti-Hermitian part is taken with:

```
template<class vtype,int N> iMatrix<vtype,N>
Ta(const iMatrix<vtype,N> &arg)
```

SU(N) Reunitarisation (or reorthogonalisation) is enabled by:

```
template<class vtype,int N> iMatrix<vtype,N>
ProjectOnGroup(const iMatrix<vtype,N> &arg)
```

Example:

```
LatticeColourMatrixD Mom(grid);
LatticeColourMatrixD TaMom(grid);

TaMom  = Ta(Mom);
```

5.1.4 Querying internal index structure

Templated code may find it useful to use query functions on the Grid datatypes they are provided. For example general Serialisation and I/O code can inspect the nature of a type a routine has been asked to read from disk, or even generate descriptive type strings:

```
////////////////////////////////////////////////
// Support type queries on template params:
////////////////////////////////////////////////
// int _ColourScalar  =  isScalar<ColourIndex,vobj>();
// int _ColourVector  =  isVector<ColourIndex,vobj>();
// int _ColourMatrix  =  isMatrix<ColourIndex,vobj>();
template<int Level,class vtype>  int isScalar(void)
template<int Level,class vtype>  int isVector(void)
template<int Level,class vtype>  int isMatrix(void)
```

Example (lib/parallelIO/IldgIO.h):

```
template<class vobj> std::string ScidacRecordTypeString(int &colors, int &spins, int &
↪ typesize,int &datacount) {

////////////////////////////////////////
// Encode a generic tensor as a string
////////////////////////////////////////

typedef typename getPrecision<vobj>::real_scalar_type stype;

int _ColourN       = indexRank<ColourIndex,vobj>();
int _ColourScalar  =  isScalar<ColourIndex,vobj>();
int _ColourVector  =  isVector<ColourIndex,vobj>();
int _ColourMatrix  =  isMatrix<ColourIndex,vobj>();

int _SpinN       = indexRank<SpinIndex,vobj>();
int _SpinScalar  =  isScalar<SpinIndex,vobj>();
int _SpinVector  =  isVector<SpinIndex,vobj>();
int _SpinMatrix  =  isMatrix<SpinIndex,vobj>();

int _LorentzN       = indexRank<LorentzIndex,vobj>();
int _LorentzScalar  =  isScalar<LorentzIndex,vobj>();
int _LorentzVector  =  isVector<LorentzIndex,vobj>();
int _LorentzMatrix  =  isMatrix<LorentzIndex,vobj>();

std::stringstream stream;

stream << "GRID_";
stream << ScidacWordMnemonic<stype>();

if ( _LorentzVector )   stream << "_LorentzVector"<<_LorentzN;
if ( _LorentzMatrix )   stream << "_LorentzMatrix"<<_LorentzN;

if ( _SpinVector )   stream << "_SpinVector"<<_SpinN;
if ( _SpinMatrix )   stream << "_SpinMatrix"<<_SpinN;

if ( _ColourVector )   stream << "_ColourVector"<<_ColourN;
if ( _ColourMatrix )   stream << "_ColourMatrix"<<_ColourN;

if ( _ColourScalar && _LorentzScalar && _SpinScalar )   stream << "_Complex";
```

(continues on next page)

(continued from previous page)

```
typesize = sizeof(typename vobj::scalar_type);

if ( _ColourMatrix ) typesize*= _ColourN*_ColourN;
else                 typesize*= _ColourN;

if ( _SpinMatrix )   typesize*= _SpinN*_SpinN;
else                 typesize*= _SpinN;

};
```

5.1.5 Inner and outer products

We recursively define (tensors/Tensor_inner.h), ultimately returning scalar in all indices:

```
////////////////////////////////////////////////////////////////////////
// innerProduct Scalar x Scalar -> Scalar
// innerProduct Vector x Vector -> Scalar
// innerProduct Matrix x Matrix -> Scalar
////////////////////////////////////////////////////////////////////////
template<class l,class r>
auto innerProductD (const iScalar<l>& lhs,const iScalar<r>& rhs)

template<class l,class r,int N>
auto innerProductD (const iVector<l,N>& lhs,const iVector<r,N>& rhs)

template<class l,class r,int N>
auto innerProductD (const iMatrix<l,N>& lhs,const iMatrix<r,N>& rhs)

template<class l,class r>
auto innerProduct (const iScalar<l>& lhs,const iScalar<r>& rhs)

template<class l,class r,int N>
auto innerProduct (const iVector<l,N>& lhs,const iVector<r,N>& rhs)

template<class l,class r,int N>
auto innerProduct (const iMatrix<l,N>& lhs,const iMatrix<r,N>& rhs)
```

The sum is always performed in double precision for the innerProductD variant.

We recursively define (tensors/Tensor_outer.h):

```
////////////////////////////////////////////////////////////////////////
// outerProduct Scalar x Scalar -> Scalar
//              Vector x Vector -> Matrix
////////////////////////////////////////////////////////////////////////
template<class l,class r>
auto outerProduct (const iScalar<l>& lhs,const iScalar<r>& rhs)

template<class l,class r,int N>
auto outerProduct (const iVector<l,N>& lhs,const iVector<r,N>& rhs)
```

5.1.6 Functions of Tensor

The following unary functions are defined, which operate element by element on a tensor data structure:

```
sqrt();
rsqrt();
sin();
cos();
asin();
acos();
log();
exp();
abs();
Not();
toReal();
toComplex();
```

Element wise functions are defined for:

```
div(tensor,Integer);
mod(tensor,Integer);
pow(tensor,RealD);
```

Matrix exponentiation (as opposed to element wise exponentiation is implemented via power series in:

```
Exponentiate(const Tensor &r  ,RealD alpha, Integer Nexp = DEFAULT_MAT_EXP)
```

the exponentiation is distributive across vector indices (i.e. proceeds component by component for a LorentzColourMatrix).

Determinant is similar:

```
iScalar Determinant(const Tensor &r )
```

5.2 Vectorisation

Internally, Grid defines a portable abstraction SIMD vectorisation, via the following types:

- vRealF
- vRealD
- vComplexF
- vComplexD

These have the usual range of arithmetic operators and functions acting upon them. They do not form part of the API, but are mentioned to (partially) explain the need for controlling the layout transformation in lattice objects. They contain a number consecutive elements of the appropriate Real/Complex type, where number is architecture depemendent. The number may be queried at runtime using:

```
vComplexF::Nsimd();
```

The layout transformations in indexing functions in the Grid objects as completely parameterised by this Nsimd(). They are documented further in the Internals chapter.

5.3 Coordinates

The Grid is define on a N-dimensional set of integer coordinates.

The maximum dimension is eight, and indexes in this space make use of the Coordinate class. The coordinate class shares a similar interface to *std::vector<int>*, but contains all data within the object, and has a fixed maximum length (template parameter).

Example:

```
const int Nd=4;
Coordinate point(Nd);

for(int i=0;i<Nd;i++)
  point[i] = 1;

std::cout<< point <<std::endl;

point.resize(3);

std::cout<< point <<std::endl;
```

This enables the coordinates to be manipulated without heap allocation or thread contention, and avoids introducing STL functions into GPU code, but does so at the expense of introducing a maximum dimensionality. This limit is easy to change (lib/util/Coordinate.h).

5.4 Grids

A Grid object defines the geometry of a global cartesian array, and through inheritance provides access to message passing decomposition, the local lattice, and the message passing primitives.

The constructor requires parameters to indicate how the spatial (and temporal) indices are decomposed across MPI tasks and SIMD lanes of the vector length. We use a partial vectorisation transformation, must select which space-time dimensions participate in SIMD vectorisation. The Lattice containers are defined to have opaque internal layout, hiding this layout transformation.

We define GridCartesian and GridRedBlackCartesian which both inherit from GridBase:

```
class GridCartesian        : public GridBase
class GridRedBlackCartesian: public GridBase
```

The simplest Cartesian Grid constructor distributes across *MPI_COMM_WORLD*:

```
////////////////////////////////////////////////////////////////////////
// Construct from comm world
////////////////////////////////////////////////////////////////////////
GridCartesian(const Coordinate &dimensions,
              const Coordinate &simd_layout,
              const Coordinate &processor_grid);
```

A second constructor will create a child communicator from a previously declared Grid. This allows to subdivide the processor grid, and also to define lattices of differing dimensionalities and sizes, useful for both Chiral fermions, lower dimensional operations, and multigrid:

```
////////////////////////////////////////////////////////////////////////
// Constructor takes a parent grid and possibly subdivides communicator.
////////////////////////////////////////////////////////////////////////
GridCartesian(const Coordinate &dimensions,
              const Coordinate &simd_layout,
```

(continues on next page)

(continued from previous page)

```
              const Coordinate &processor_grid,
              const GridCartesian &parent,int &split_rank);
```

The Grid object provides much *internal* functionality to map a lattice site to a node and lexicographic index. These are not needed by code interfacing to the data parallel layer.

When the requested processor grid is smaller than the parent's processor grid, multiple copies of the same geometry communicator are created, indexed by spli_rank. This can be convenient to split a job into multiple independent sub jobs (a long present feature of MPI). It can be particularly effective in valence analysis, where for example, many inversions are performed on a single configuration. These can be made on smaller communicators in parallel and communications overheads minimised. Routines:

```
Grid_split
Grid_unsplit
```

are provided to communicate fields between different communicators (e.g. between inversion and contraction phases).

Example (tests/solver/Test_split_grid.cc):

```
const int Ls=8;

//////////////////////////////////////////
// Grids distributed across full machine
// pick up default command line args
//////////////////////////////////////////
Grid_init(&argc,&argv);

Coordinate latt_size   = GridDefaultLatt();
Coordinate simd_layout = GridDefaultSimd(Nd,vComplex::Nsimd());
Coordinate mpi_layout  = GridDefaultMpi();

GridCartesian         * UGrid   = SpaceTimeGrid::makeFourDimGrid(GridDefaultLatt(),
                                                                 GridDefaultSimd(Nd,
↪vComplex::Nsimd()),
                                                                 GridDefaultMpi());
GridCartesian         * FGrid   = SpaceTimeGrid::makeFiveDimGrid(Ls,UGrid);
GridRedBlackCartesian * rbGrid  = SpaceTimeGrid::makeFourDimRedBlackGrid(UGrid);
GridRedBlackCartesian * FrbGrid = SpaceTimeGrid::makeFiveDimRedBlackGrid(Ls,UGrid);

//////////////////////////////////////////
// Split into N copies of 1^4 mpi communicators
//////////////////////////////////////////
Coordinate mpi_split (mpi_layout.size(),1);
GridCartesian         * SGrid = new GridCartesian(GridDefaultLatt(),
                                                  GridDefaultSimd(Nd,
↪vComplex::Nsimd()),
                                                  mpi_split,
                                                  *UGrid);

GridCartesian         * SFGrid   = SpaceTimeGrid::makeFiveDimGrid(Ls,SGrid);
GridRedBlackCartesian * SrbGrid  = SpaceTimeGrid::makeFourDimRedBlackGrid(SGrid);
GridRedBlackCartesian * SFrbGrid = SpaceTimeGrid::makeFiveDimRedBlackGrid(Ls,SGrid);

//////////////////////////////////////////////////////////////
// split the source out using MPI instead of I/O
//////////////////////////////////////////////////////////////
Grid_split  (Umu,s_Umu);
```

(continues on next page)

(continued from previous page)

```
Grid_split  (src,s_src);
```

Internals

The processor Grid is defined by data values in the Communicator object:

```
int              _Nprocessors;     // How many in all
std::vector<int> _processors;      // Which dimensions get relayed out over␣
↪processors lanes.
int              _processor;       // linear processor rank
std::vector<int> _processor_coor;  // linear processor coordinate
unsigned long    _ndimension;
Grid_MPI_Comm    communicator;
```

The final of these is potentially an MPI Cartesian communicator, mapping some total number of processors to an N-dimensional coordinate system. This is used by Grid to geometrically decompose the subvolumes of a lattice field across processing elements. Grid is aware of multiple ranks per node and attempts to ensure that the geometrical decomposition keeps as many neigbours as possible on the same node. This is done by reordering the ranks in the constructor of a Communicator object once the topology requested has been indicated, via an internal call to the method OptimalCommunicator(). The reordering is chosen by Grid to trick MPI, which makes a simple lexicographic assignment of ranks to coordinate, to ensure that the simple lexicographic assignment of the reordered ranks is the optimal choice. MPI does not do this by default and substantial improvements arise from this design choice.

5.5 Lattice containers

Lattice objects may be constructed to contain the local portion of a distribued array of any tensor type. For performance reasons the tensor type uses a vector Real or Complex as the fundamental datum.

Every lattice requires a GridBase object pointer to be provided in its constructor. Memory is allocated at construction time. If a Lattice is passed a RedBlack grid, it allocates half the storage of the full grid, and may either store the red or black checkerboard. The Lattice object will automatically track through assignments which checkerboard it refers to. For example, shifting a Even checkerboard by an odd distance produces an Odd result field.

Struct of array objects are defined, and used in the template parameters to the lattice class.

Example (lib/qcd/QCD.h):

```
template<typename vtype> using iSpinMatrix = iScalar<iMatrix<iScalar<vtype>, Ns> >;
typedef iSpinMatrix<ComplexF>           SpinMatrixF; //scalar
typedef iSpinMatrix<vComplexF>          vSpinMatrixF;//vectorised
typedef Lattice<vSpinMatrixF>           LatticeSpinMatrixF;
```

The full range of QCD relevant lattice objects is given below.

Lattice	Lorentz	Spin	Colour	scalar_type	Field	Synonym
Vector	Scalar	Scalar	Scalar	Integer	LatticeInteger	N/A
Vector	Scalar	Scalar	Scalar	RealD	LatticeRealD	N/A
Vector	Scalar	Scalar	Scalar	ComplexD	LatticeComplexD	N/A
Vector	Scalar	Scalar	Matrix	ComplexD	LatticeColourMatrixD	LatticeGaugeLink
Vector	Vector	Scalar	Matrix	ComplexD	LatticeLorentzColourMatrixD	LatticeGaugeFieldD
Vector	Scalar	Vector	Vector	ComplexD	LatticeSpinColourVectorD	LatticeFermionD
Vector	Scalar	Vector	Vector	ComplexD	LatticeHalfSpinColourVectorD	LatticeHalfFermionD
Vector	Scalar	Matrix	Matrix	ComplexD	LatticeSpinColourMatrixD	LatticePropagatorD

Additional single precison variants are defined with the suffix "F". Other lattice objects can be defined using the sort of typedef's shown above if needed. LatticeInteger is typically only used in the form of predicate fields for where statements.

5.5.1 Opaque containers

The layout within the container is complicated to enable maximum opportunity for vectorisation, and is opaque from the point of view of the API definition. The key implementation observation is that so long as data parallel operations are performed and adjacent SIMD lanes correspond to well separated lattice sites, then identical operations are performed on all SIMD lanes and enable good vectorisation.

Because the layout is opaque, import and export routines from naturally ordered x,y,z,t arrays are provided (lib/lattice/Lattice_transfer.h):

```
unvectorizeToLexOrdArray(std::vector<sobj> &out, const Lattice<vobj> &in);
vectorizeFromLexOrdArray(std::vector<sobj> &in , Lattice<vobj> &out);
```

sobj and vobj should be a matching pair of scalar and vector objects of the same internal structure. The compiler will let you know with a long and verbose complaint if they are not.

The Lexicographic order of data in the external vector fields is defined by (lib/util/Lexicographic.h):

```
Lexicographic::IndexFromCoor(const Coordinate &lcoor, int &lex,Coordinate *local_
↪dims);
```

This ordering is $x + L_x * y + L_x * L_y * z + L_x * L_y * L_z * t$

Peek and poke routines are provided to perform single site operations. These operations are extremely low performance and are not intended for algorithm development or performance-critical code.

The following are *collective* operations and involve communication between nodes. All nodes receive the same result by broadcast from the owning node:

```
void peekSite(sobj &s,const Lattice<vobj> &l,const Coordinate &site);
void pokeSite(const sobj &s,Lattice<vobj> &l,const Coordinate &site);
```

The following are executed independently by each node:

```
void peekLocalSite(sobj &s,const Lattice<vobj> &l,Coordinate &site);
void pokeLocalSite(const sobj &s,Lattice<vobj> &l,Coordinate &site);
```

Lattices of one tensor type may be transformed into lattices of another tensor type by peeking and poking specific indices in a data parallel manner:

```
template<int Index,class vobj> // Vector data parallel index peek
auto PeekIndex(const Lattice<vobj> &lhs,int i);

template<int Index,class vobj> // Matrix data parallel index peek
auto PeekIndex(const Lattice<vobj> &lhs,int i,int j);

template<int Index,class vobj>   // Vector poke
void PokeIndex(Lattice<vobj> &lhs,const Lattice<> & rhs,int i)

template<int Index,class vobj>   // Matrix poke
void PokeIndex(Lattice<vobj> &lhs,const Lattice<> & rhs,int i,int j)
```

The inconsistent capitalisation on the letter P is due to an obscure bug in g++ that has not to our knowledge been fixed in any version. The bug was reported in 2016.

Todo: CD: Do you want to mention/expose PropToFerm and FermToProp? Are there other such convenience routines to make part of the API?

5.5.2 Global Reduction operations

Reduction operations for any lattice field are provided. The result is identical on each computing node that is part of the relevant Grid communicator:

```
template<class vobj>
RealD norm2(const Lattice<vobj> &arg);

template<class vobj>
ComplexD innerProduct(const Lattice<vobj> &left,const Lattice<vobj> &right);

template<class vobj>
vobj sum(const Lattice<vobj> &arg)
```

5.5.3 Site local reduction operations

Internal indices may be reduced, site by site, using the following routines:

```
template<class vobj>
auto localNorm2 (const Lattice<vobj> &rhs)

template<class vobj>
auto localInnerProduct (const Lattice<vobj> &lhs,const Lattice<vobj> &rhs)
```

5.5.4 Outer product

A site local outer product is defined:

```
template<class ll,class rr>
auto outerProduct (const Lattice<ll> &lhs,const Lattice<rr> &rhs)
```

5.5.5 Slice operations

Slice operations are defined to operate on one lower dimension than the full lattice. The omitted dimension is the parameter orthogdim:

```
template<class vobj>
void sliceSum(const Lattice<vobj> &Data,
              std::vector<typename vobj::scalar_object> &result,
              int orthogdim);

template<class vobj>
void sliceInnerProductVector( std::vector<ComplexD> & result,
                              const Lattice<vobj> &lhs,
                              const Lattice<vobj> &rhs,
                              int orthogdim);

template<class vobj>
void sliceNorm (std::vector<RealD> &sn,
                const Lattice<vobj> &rhs,
                int orthogdim);
```

5.6 Data parallel expression template engine

The standard arithmetic operators and some data parallel library functions are implemented site by site on lattice types.

Operations may only ever combine lattice objects that have been constructed from the **same** grid pointer.

Example:

```
LatticeFermionD A(&grid);
LatticeFermionD B(&grid);
LatticeFermionD C(&grid);

A = B - C;
```

Such operations are said to be **conformable** and are the lattice are guaranteed to have the same dimensions and both MPI and SIMD decomposition because they are based on the same grid object. The conformability check is lightweight and simply requires the same grid pointers be passed to the lattice objects. The data members of the grid objects are not compared.

Conformable lattice fields may be combined with appropriate scalar types in expressions. The implemented rules follow those already documented for the tensor types.

5.6.1 Unary operators and functions

The following sitewise unary operations are defined:

Operation	Description
operator-	negate
adj	Hermitian conjugate
conjugate	complex conjugate
trace	sitewise trace
transpose	sitewise transpose
Ta	take traceles anti Hermitian part
ProjectOnGroup	reunitarise or orthogonalise
real	take the real part
imag	take the imaginary part
toReal	demote complex to real
toComplex	promote real to complex
timesI	elementwise +i mult (0 is not multiplied)
timesMinusI	elementwise -i mult (0 is not multiplied)
abs	elementwise absolute value
sqrt	elementwise square root
rsqrt	elementwise reciprocal square root
sin	elementwise sine
cos	elementwise cosine
asin	elementwise inverse sine
acos	elementwise inverse cosine
log	elementwise logarithm
exp	elementwise exponentiation
operator!	Logical negation of integer field
Not	Logical negation of integer field

The following sitewise applied functions with additional parameters are:

```
template<class obj> Lattice<obj> pow(const Lattice<obj> &rhs_i,RealD y);

template<class obj> Lattice<obj> mod(const Lattice<obj> &rhs_i,Integer y);

template<class obj> Lattice<obj> div(const Lattice<obj> &rhs_i,Integer y);

template<class obj> Lattice<obj>
expMat(const Lattice<obj> &rhs_i, RealD alpha, Integer Nexp = DEFAULT_MAT_EXP);
```

5.6.2 Binary operators

The following binary operators are defined:

```
operator+
operator-
operator*
operator/
```

Logical are defined on LatticeInteger types:

```
operator&
operator|
operator&&
operator||
```

5.6.3 Ternary operator, logical operations and where

Within the data parallel level of the API the only way to perform operations that are differentiated between sites is use predicated execution.

The predicate takes the form of a LatticeInteger which is confromable with both the iftrue and iffalse argument:

```
template<class vobj,class iobj> void where(const Lattice<iobj> &pred,
                                           Lattice<vobj> &iftrue,
                                           Lattice<vobj> &iffalse);
```

This plays the data parallel analogue of the C++ ternary operator:

```
a == b ? c : d;
```

In order to create the predicate in a coordinate dependent fashion it is often useful to use the lattice coordinates.

The LatticeCoordinate function:

```
template<class iobj> LatticeCoordinate(Lattice<iobj> &coor,int dir);
```

fills an Integer field with the coordinate in the direction specified by "dir". A usage example is given

Example:

```
int dir = 3;
int block = 4;
LatticeInteger coor(FineGrid);

LatticeCoordinate(coor,dir);

result = where(mod(coor,block)==(block-1),x,z);
```

This example takes result to be either "x" or "z" in a coordinate dependent way. When third (z) lattice coordinate lies at the boundaries of a block size (periodic arithmetic). This example is lifted and paraphrased from code that (data parallel) evaluates matrix elements for a coarse representation of the Dirac operator in multigrid.

Other usage cases of LatticeCoordinate include the generation of plane wave momentum phases.

5.7 Site local fused operations

The biggest limitation of expression template engines is that the optimisation visibility is a single assignment statement in the original source code.

There is no scope for loop fusion between multiple statements. Multi-loop fusion gives scope for greater cache locality.

Two primitives for hardware aware parallel loops are provided. These will operate directly on the site objects which are expanded by a factor of the vector length (in our struct of array datatypes).

Since the mapping of sites to data lanes is opaque, these vectorised loops are *only* appropriate for optimisation of site local operations.

5.7.1 View objects

Due to an obscure aspect of the way that Nvidia handle device C++11 lambda functions, it is necessary to disable the indexing of a Lattice object.

Rather, a reference to a lattice object must be first obtained.

The reference is copyable to a GPU, and is able to be indexed on either accelerator code, or by host code.

In order to prevent people developing code that dereferences Lattice objects in a way that works on CPU compilation, but fails on GPU compilation, we have decided to remove the ability to index a lattice object on CPU code.

As a result of Nvidia's constraints, all accesses to lattice objects are required to be made through a View object.

In the following, the type is LatticeView<vobj>, however it is wise to use the C++11 auto keyword to avoid naming the type. See code examples below.

5.7.2 thread_loops

The first parallel primitive is the thread_loop

Example:

```
LatticeField r(grid);
LatticeField x(grid);
LatticeField p(grid);
LatticeField mmp(grid);
auto r_v = r.View();
auto x_v = x.View();
auto p_v = p.View();
auto mmp_v = mmp.View();
thread_loop(s , r_v, {
  r_v[s] = r_v[s]   - a * mmp_v[s];
  x_v[s] = x_v[s]   + a*p_v[s];
  p_v[s] = p_v[s]*b + r_v[s];
});
```

5.7.3 accelerator_loops

The second parallel primitive is the "accelerator_loop".

The thread loop runs on host processor cores only. If the enabled architecture is VGPU, if Grid is configured with

–enable-simd=VGPU,

the acccelerator_loop may run on a GPU if present. On non-accelerated architectures, the accelerator_loop will simply run as a an OpenMP thread_loop.

It is planned to support multiple forms of acccelerator in future, including OpenMP 5.0 offload, and possibly SyCL based offload.

Example:

```
LatticeField r(grid);
LatticeField x(grid);
LatticeField p(grid);
LatticeField mmp(grid);
auto r_v = r.View();
auto x_v = x.View();
auto p_v = p.View();
auto mmp_v = mmp.View();
accelerator_loop(s , r_v, {
  r_v[s] = r_v[s]   - a * mmp_v[s];
```

(continues on next page)

(continued from previous page)

```
  x_v[s] = x_v[s]    + a*p_v[s];
  p_v[s] = p_v[s]*b + r_v[s];
});
```

5.7.4 Cshift

Site shifting operations are provided using the Cshift function:

```
template<class vobj>
Lattice<vobj> Cshift(const Lattice<vobj> &rhs,int dimension,int shift)
```

This shifts the whole vector by any distance shift in the appropriate dimension.

For the avoidance of doubt on direction conventions,a positive shift moves the lattice site $x_\mu = 1$ in the rhs to $x_\mu = 0$ in the result.

Example (benchmarks/Benchmark_wilson.cc):

```
{ // Naive wilson implementation
  ref = Zero();
  for(int mu=0;mu<Nd;mu++){

    tmp = U[mu]*Cshift(src,mu,1);
    {
      auto ref_v = ref.View();
      auto tmp_v = tmp.View();
      for(int i=0;i<ref_v.size();i++){
        ref_v[i]+= tmp_v[i] - Gamma(Gmu[mu])*tmp_v[i]; ;
      }
    }

    tmp =adj(U[mu])*src;
    tmp =Cshift(tmp,mu,-1);
    {
      auto ref_v = ref.View();
      auto tmp_v = tmp.View();
      for(int i=0;i<ref_v.size();i++){
        ref_v[i]+= tmp_v[i] + Gamma(Gmu[mu])*tmp_v[i]; ;
      }
    }
  }
}
```

5.8 Inter-grid transfer operations

Transferring between different checkerboards of the same global lattice:

```
template<class vobj>
void pickCheckerboard(int cb,Lattice<vobj> &half,const Lattice<vobj> &full);

template<class vobj>
void setCheckerboard(Lattice<vobj> &full,const Lattice<vobj> &half);
```

These are used to set up Schur red-black decomposed solvers, for example.

Multi-grid projection between a fine and coarse grid:

```
template<class vobj,class CComplex,int nbasis>
void blockProject(Lattice<iVector<CComplex,nbasis > > &coarseData,
                  const             Lattice<vobj>   &fineData,
                  const std::vector<Lattice<vobj> > &Basis);
```

Multi-grid promotion to a finer grid:

```
template<class vobj,class CComplex,int nbasis>
void blockPromote(const Lattice<iVector<CComplex,nbasis > > &coarseData,
                  Lattice<vobj>   &fineData,
                  const std::vector<Lattice<vobj> > &Basis)
```

Support for subblock Linear algebra:

```
template<class vobj,class CComplex>
void blockZAXPY(Lattice<vobj> &fineZ,
                const Lattice<CComplex> &coarseA,
                const Lattice<vobj> &fineX,
                const Lattice<vobj> &fineY)

template<class vobj,class CComplex>
void blockInnerProduct(Lattice<CComplex> &CoarseInner,
                       const Lattice<vobj> &fineX,
                       const Lattice<vobj> &fineY)

template<class vobj,class CComplex>
void blockNormalise(Lattice<CComplex> &ip,Lattice<vobj> &fineX)

template<class vobj>
void blockSum(Lattice<vobj> &coarseData,const Lattice<vobj> &fineData)

template<class vobj,class CComplex>
void blockOrthogonalise(Lattice<CComplex> &ip,std::vector<Lattice<vobj> > &Basis)
```

Conversion between different SIMD layouts:

```
template<class vobj,class vvobj>
void localConvert(const Lattice<vobj> &in,Lattice<vvobj> &out)
```

Slices between grid of dimension N and grid of dimentions N+1:

```
template<class vobj>
void InsertSlice(const Lattice<vobj> &lowDim,Lattice<vobj> & higherDim,int slice, int␣
↪orthog)

template<class vobj>
void ExtractSlice(Lattice<vobj> &lowDim,const Lattice<vobj> & higherDim,int slice,␣
↪int orthog)
```

Growing a lattice by a multiple factor, with periodic replication:

```
template<class vobj>
void Replicate(Lattice<vobj> &coarse,Lattice<vobj> & fine)
```

That latter is useful to, for example, pre-thermalise a smaller volume and then grow the volume in HMC. It was written while debugging G-parity boundary conditions.

CHAPTER

SIX

RANDOM NUMBER GENERATORS

Grid provides three configure time options for random the number generator engine.

- sitmo
- ranlux48
- mt19937

The selection is controlled by the –enable-rng=<option> flag.

Sitmo is the default Grid RNG and is recommended. It is a hash based RNG that is cryptographically secure and has

1. passed the BigCrush tests
2. can Skip forward an arbitrary distance (up to 2^256) in O(1) time

We use Skip functionality to place each site in an independent well separated stream. The Skip was trivially parallelised, important in a many core node, and gives very low overhead parallel RNG initialisation.

Our implementation of parallel RNG

- Has passed the BigCrush tests **drawing once from each site RNG** in a round robin fashion.

This test is applied in tests/testu01/Test_smallcrush.cc

The interface is as follows:

```
class GridSerialRNG {
  GridSerialRNG();
  void SeedFixedIntegers(const std::vector<int> &seeds);
  void SeedUniqueString(const std::string &s);
}

class GridParallelRNG {
  GridParallelRNG(GridBase *grid);
  void SeedFixedIntegers(const std::vector<int> &seeds);
  void SeedUniqueString(const std::string &s);
}
```

- Seeding

The SeedUniqueString uses a 256bit SHA from the OpenSSL library to construct integer seeds. The reason for this is to enable reproducible seeding in the measurement sector of physics codes. For example, labelling a random drawn by a string representation the physics information, and the appending trajectory number will give a unique set of seeds for each measurement on each trajectory. This string based functionality is probably not expected to be used in a lattice evolution, except for possibly the initial state. Subsequent evolution should checkpoint and restore lattice RNG state using the interfaces below.

These may be drawn as follows:

```
void random(GridParallelRNG &rng,Lattice<vobj> &l)   { rng.fill(l,rng._uniform);  }
void gaussian(GridParallelRNG &rng,Lattice<vobj> &l) { rng.fill(l,rng._gaussian); }

void random(GridSerialRNG &rng,sobj &l)   { rng.fill(l,rng._uniform  ); }
void gaussian(GridSerialRNG &rng,sobj &l) { rng.fill(l,rng._gaussian ); }
```

- Serial RNG's are used to assign scalar fields.
- Parallel RNG's are used to assign lattice fields and must subdivide the field grid (need not be conformable).

It is the API users responsibility to initialise, manage, save and restore these RNG state for their algorithm. In particular there is no single globally managed RNG state.

Input/Output routines are provided for saving and restoring RNG states.

lib/parallelIO/BinaryIO.h:

```
////////////////////////////////////////////////////////////////////////////
// Read a RNG;  use IOobject and lexico map to an array of state
////////////////////////////////////////////////////////////////////////////
static void readRNG(GridSerialRNG &serial,
                    GridParallelRNG &parallel,
                    std::string file,
                    Integer offset,
                    uint32_t &nersc_csum,
                    uint32_t &scidac_csuma,
                    uint32_t &scidac_csumb)
////////////////////////////////////////////////////////////////////////////
// Write a RNG; lexico map to an array of state and use IOobject
////////////////////////////////////////////////////////////////////////////
static void writeRNG(GridSerialRNG &serial,
                     GridParallelRNG &parallel,
                     std::string file,
                     Integer offset,
                     uint32_t &nersc_csum,
                     uint32_t &scidac_csuma,
                     uint32_t &scidac_csumb)
```

lib/parallelIO/NerscIO.h:

```
void writeRNGState(GridSerialRNG &serial,GridParallelRNG &parallel,std::string file);

void readRNG(GridSerialRNG &serial,
             GridParallelRNG &parallel,
             std::string file,
             Integer offset,
             uint32_t &nersc_csum,
             uint32_t &scidac_csuma,
             uint32_t &scidac_csumb);
```

Example:

```
NerscIO::writeRNGState(sRNG,pRNG,rfile);
```

CHAPTER

SEVEN

INPUT OUTPUT FACILITIES

Grid introduces both high performance parallel I/O routines, making use of MPI-2 parallel I/O internally, and also features for automatic serialisation of rich object types to various common formats. The SciDAC file formats are supported.

7.1 Serialisation

Serialisable classes can be defined by

- Deriving from Serializable
- Declaring all class data members in the GRID_SERIALIZABLE_CLASS_MEMBERS macro

An example is

Example:

```
class myclass : public Serializable {
  GRID_SERIALIZABLE_CLASS_MEMBERS(
     myclass,
     myenum, e,
     std::vector<myenum>, ve,
     std::string, name,
     int, x,
     double, y,
     bool , b,
     std::vector<double>, array,
     std::vector<std::vector<double> >, twodimarray,
     std::vector<std::vector<std::vector<Complex> > >, cmplx3darray);
 };
```

We make use of variadic macros to automatically generate both the class declaration and appropriate I/O routines. A virtual Reader and Writer interface is used. Specific cases included in Grid are

Writer	Reader	Format
XmlWriter	XmlReader	XML
BinaryWriter	BinaryReader	Binary
TextWriter	TextReader	ASCII
JSONWriter	JSONReader	json
Hdf5Writer	Hdf5Reader	HDF5

Write interfaces, similar to the XML facilities in QDP++ are presented. However, the serialisation routines are automatically generated by the macro, and a virtual reader adn writer interface enables writing to any of a number of

formats.

Example:

```
class myclass : public Serializable {
  GRID_SERIALIZABLE_CLASS_MEMBERS(
    myclass,
    myenum, e,
    std::vector<myenum>, ve,
    std::string, name,
    int, x,
    double, y,
    bool , b,
    std::vector<double>, array,
    std::vector<std::vector<double> >, twodimarray,
    std::vector<std::vector<std::vector<Complex> > >, cmplx3darray);
 };

 myclass Instance;
 {
    XmlWriter WR("bother.xml");
    push(WR,"NestedDocumentExample");
    write(WR,Instance);
    pop(WR);
 }
```

7.2 Data parallel field IO

Support is provided to perform I/O operations for distributed Lattice fields.

Binary, NERSC, ILDG, and SciDAC formats are supported.

The ILDG and SciDAC formats require that Grid be compiled with the LIME library.

Parallel I/O makes use of MPI-2 collective parallel I/O interfaces, and relies on this to deliver good performance.

7.2.1 Binary Interface

The Binary I/O interface defines the following API functions:

```
template<class vobj,class fobj,class munger>
static void BinaryIO::readLatticeObject(Lattice<vobj> &Umu,
                                        std::string file,
                                        munger munge,
                                        Integer offset,
                                        const std::string &format,
                                        uint32_t &nersc_csum,
                                        uint32_t &scidac_csuma,
                                        uint32_t &scidac_csumb)

template<class vobj,class fobj,class munger>
static void BinaryIO::writeLatticeObject(Lattice<vobj> &Umu,
                                         std::string file,
                                         munger munge,
                                         Integer offset,
                                         const std::string &format,
```

(continues on next page)

(continued from previous page)

```
                                    uint32_t &nersc_csum,
                                    uint32_t &scidac_csuma,
                                    uint32_t &scidac_csumb)


////////////////////////////////////////////////////////////////////////
// Read a RNG;  use IOobject and lexico map to an array of state
////////////////////////////////////////////////////////////////////////
static void BinaryIO::readRNG(GridSerialRNG &serial,
                              GridParallelRNG &parallel,
                              std::string file,
                              Integer offset,
                              uint32_t &nersc_csum,
                              uint32_t &scidac_csuma,
                              uint32_t &scidac_csumb)

////////////////////////////////////////////////////////////////////////
// Write a RNG; lexico map to an array of state and use IOobject
////////////////////////////////////////////////////////////////////////
static void BinaryIO::writeRNG(GridSerialRNG &serial,
                               GridParallelRNG &parallel,
                               std::string file,
                               Integer offset,
                               uint32_t &nersc_csum,
                               uint32_t &scidac_csuma,
                               uint32_t &scidac_csumb)
```

The *offset* parameters allow the other file formats, with merged *headers* of various formats to build upone the binary I/O facilities.

Thus the bulk of the data transferred is through a common code across all formats, and only the header generation differs. The Binary format has no headers, and it is the responsibility of the programmer to retain and check checksums.

WE DO NOT RECOMMEND EVERY USING A BULK DATA FILE THAT HAS NOT HAD A CHECKSUM VERIFIED.

7.2.2 Field meta data

in order to maximise code reuse, Grid uses an internal meta data field to represent gauge configuration in the API. This combines elements of the ILDG defined metadata and the NERSC header metadata:

```
class FieldMetaData : Serializable {
public:

GRID_SERIALIZABLE_CLASS_MEMBERS(FieldMetaData,
                                int, nd,
                                std::vector<int>, dimension,
                                std::vector<std::string>, boundary,
                                int, data_start,
                                std::string, hdr_version,
                                std::string, storage_format,
                                double, link_trace,
                                double, plaquette,
                                uint32_t, checksum,
                                uint32_t, scidac_checksuma,
                                uint32_t, scidac_checksumb,
```

(continues on next page)

(continued from previous page)

```
                                    unsigned int, sequence_number,
                                    std::string, data_type,
                                    std::string, ensemble_id,
                                    std::string, ensemble_label,
                                    std::string, ildg_lfn,
                                    std::string, creator,
                                    std::string, creator_hardware,
                                    std::string, creation_date,
                                    std::string, archive_date,
                                    std::string, floating_point);
};
```

A routine:

```
void PrepareMetaData(Lattice<vobj> & field, FieldMetaData &header)
```

are provided to automatically fill these data structures with the dimensions, the users name, machine name, dates and data types. The checksums are computed and returned by the I/O routines themselves.

For gauge configurations:

```
void GaugeStatistics(Lattice<vLorentzColourMatrixD> & data,FieldMetaData &header);
```

Will compute the physical (plaquette, link trace) attributes stored.

7.2.3 NERSC format and generalisations

The class NerscIO has static methods providing public methods for gauge and random number generator I/O as follows:

```
template<class vsimd>
static void NerscIO::readConfiguration(Lattice<iLorentzColourMatrix<vsimd> > &Umu,
                               FieldMetaData& header,
                               std::string file);

template<class vsimd>
static void NerscIO::writeConfiguration(Lattice<iLorentzColourMatrix<vsimd> > &Umu,
                                        std::string file,
                                        int two_row,
                                        int bits32);

static void NerscIO::writeRNGState(GridSerialRNG &serial,GridParallelRNG &parallel,
↪std::string file);

static void NerscIO::readRNGState(GridSerialRNG &serial,GridParallelRNG & parallel,
                                  FieldMetaData& header,std::string file);
```

Implementation detail

The lattice field routines internally use the above Binary routines to write the bulk data at an offset using MPI-2 I/O.

7.2.4 SciDAC and ILDG formats

The routines in this section rely on the c-lime library being enabled in the Grid compilation (http://usqcd-software.github.io/c-lime/). The configure script searches for *lime*, but the location can be indicated with the

–with-lime=prefix

flag.

General writers into Lime record files are presented:

```
class GridLimeReader : public BinaryIO {
  void open(const std::string &_filename);
  void close(void);
  template<class vobj>
  void readLimeLatticeBinaryObject(Lattice<vobj> &field,std::string record_name);
  template<class serialisable_object>
  void readLimeObject(serialisable_object &object,std::string object_name,std::string␣
↪record_name)
};

class GridLimeWriter : public BinaryIO {
  void open(const std::string &_filename);
  void close(void);
  int createLimeRecordHeader(std::string message, int MB, int ME, size_t PayloadSize);
  template<class vobj>
  void writeLimeLatticeBinaryObject(Lattice<vobj> &field,std::string record_name);
  template<class serialisable_object>
  void writeLimeObject(int MB,int ME,serialisable_object &object,std::string object_
↪name,std::string record_name);
};
```

These are specialised to SciDAC writers, introducing facilities for generating type strings and checksum information:

```
template<class vobj> std::string
ScidacRecordTypeString(int &colors, int &spins, int & typesize,int &datacount);

template<class vobj> void ScidacMetaData(Lattice<vobj> & field,
                                         FieldMetaData &header,
                                         scidacRecord & _scidacRecord,
                                         scidacFile   & _scidacFile);


class ScidacWriter : public GridLimeWriter {
  template<class SerialisableUserFile>
  void writeScidacFileRecord(GridBase *grid,SerialisableUserFile &_userFile);
  template <class vobj, class userRecord>
  void writeScidacFieldRecord(Lattice<vobj> &field,userRecord _userRecord);
};

class ScidacReader : public GridLimeReader {
  template<class SerialisableUserFile>
  void readScidacFileRecord(GridBase *grid,SerialisableUserFile &_userFile);
  template <class vobj, class userRecord>
  void readScidacFieldRecord(Lattice<vobj> &field,userRecord &_userRecord);
};
```

They are also specialised to ILDG format writers, available and defined only for Gauge configurations:

```
class IldgWriter : public ScidacWriter {

  template <class vsimd>
  void writeConfiguration(Lattice<iLorentzColourMatrix<vsimd> > &Umu,int sequence,
↪std::string LFN,std::string description);

};

class IldgReader : public GridLimeReader {
  template <class vsimd>
  void readConfiguration(Lattice<iLorentzColourMatrix<vsimd> > &Umu, FieldMetaData &
↪FieldMetaData_) ;

};
```

Implementation detail

The lattice field routines internally use the above Binary routines to write the bulk data at an offset using MPI-2 I/O. The cooperation with c-lime is functional but inelegant, using *ftell* on the File pointer that we provide to c-lime to find the current offset when we write the payload.

Example (tests/IO/Test_ildg_io.cc):

```
GridCartesian     Fine(latt_size,simd_layout,mpi_layout);
GridParallelRNG   pRNG(&Fine); pRNG.SeedFixedIntegers(std::vector<int>({45,12,81,9}));
LatticeGaugeField Umu(&Fine);
SU<Nc>::HotConfiguration(pRNG,Umu);
FieldMetaData header;

std::cout <<GridLogMessage<<"*************************************"<<std::endl;
std::cout <<GridLogMessage<<"** Writing out  ILDG conf    *********"<<std::endl;
std::cout <<GridLogMessage<<"*************************************"<<std::endl;
std::string file("./ckpoint_ildg.4000");
IldgWriter _IldgWriter;
_IldgWriter.open(file);
_IldgWriter.writeConfiguration(Umu,4000,std::string("dummy_ildg_LFN"),std::string(
↪"dummy_config"));
_IldgWriter.close();
```

Example (tests/lanczos/Test_dwf_compressed_lanczos_reorg.cc):

```
void checkpointFine(std::string evecs_file,std::string evals_file)
{
  assert(this->_Aggregate.subspace.size()==nbasis);
  emptyUserRecord record;
  Grid::ScidacWriter WR;
  WR.open(evecs_file);
  for(int k=0;k<nbasis;k++) {
    WR.writeScidacFieldRecord(this->_Aggregate.subspace[k],record);
  }
  WR.close();

  XmlWriter WRx(evals_file);
  write(WRx,"evals",this->evals_fine);
}
```

CHAPTER

EIGHT

LINEAR OPERATORS

As a basic interface to fermion Dirac operators, we define an abstract sparse matrix base class:

```
////////////////////////////////////////////////////////////////////////////////////
// Interface defining what I expect of a general sparse matrix, such as a Fermion action
////////////////////////////////////////////////////////////////////////////////////
template<class Field> class SparseMatrixBase {
public:
  virtual GridBase *Grid(void) =0;
  // Full checkerboar operations
  virtual RealD M    (const Field &in, Field &out)=0;
  virtual RealD Mdag (const Field &in, Field &out)=0;
  virtual void  MdagM(const Field &in, Field &out,RealD &ni,RealD &no) {
    Field tmp (in.Grid());
    ni=M(in,tmp);
    no=Mdag(tmp,out);
  }
  virtual  void Mdiag    (const Field &in, Field &out)=0;
  virtual  void Mdir     (const Field &in, Field &out,int dir, int disp)=0;
};
```

And a derived class adding methods suitable to red black preconditioning:

```
////////////////////////////////////////////////////////////////////////////////////
// Interface augmented by a red black sparse matrix, such as a Fermion action
////////////////////////////////////////////////////////////////////////////////////
template<class Field> class CheckerBoardedSparseMatrixBase : public SparseMatrixBase
<Field> {
public:
  virtual GridBase *RedBlackGrid(void)=0;
  // half checkerboard operaions
  virtual  void Meooe    (const Field &in, Field &out)=0;
  virtual  void Mooee    (const Field &in, Field &out)=0;
  virtual  void MooeeInv (const Field &in, Field &out)=0;

  virtual  void MeooeDag    (const Field &in, Field &out)=0;
  virtual  void MooeeDag    (const Field &in, Field &out)=0;
  virtual  void MooeeInvDag (const Field &in, Field &out)=0;
};
```

A checkerboard is defined as a parity $(x + y + z + t)|2$, and half checker board operations supported for red black preconditioning.

Member	Description
M	Apply matrix
Mdag	Apply matrix adjoint
MdagM	Apply Matrix then adjoin matrix
Mdiag	Apply site diagonal part of matrix
Mdir	Apply terms involving hopping one direction
Meooe	Apply even/odd matrix. ResultCB determined by InputCB
Mooee	Apply site diagonal terms to a CB field
MooeInv	Apply inverse of site diagonal terms to a CB field
MeooeDag	Apply adjoint of Meooe
MooeeDag	Apply adjoint of Mooee
MooeeInvDag	Apply adjoint inverse of Mooee

All Fermion operators will derive from this base class.

8.1 Linear Operators

We introduce a second, decoupled class of LinearOperators. Linear operators may compose the elements of sparse in different ways.

By sharing the class for Sparse Matrix across multiple operator wrappers, we can share code between RB and non-RB variants. Sparse matrix is like the fermion action def, and then the wrappers implement the specialisation of "Op" and "AdjOp" to the cases minimising replication of code.

Abstract base:

```
template<class Field> class LinearOperatorBase {
public:

  // Support for coarsening to a multigrid
  virtual void OpDiag (const Field &in, Field &out) = 0; // Abstract base
  virtual void OpDir  (const Field &in, Field &out,int dir,int disp) = 0; // Abstract␣
↪base

  virtual void Op     (const Field &in, Field &out) = 0; // Abstract base
  virtual void AdjOp  (const Field &in, Field &out) = 0; // Abstract base
  virtual void HermOpAndNorm(const Field &in, Field &out,RealD &n1,RealD &n2)=0;
  virtual void HermOp(const Field &in, Field &out)=0;
};
```

Member	Description
OpDiag	Diagonal term
OpDir	Terms involving hopping in one direction
Op	Full operator
AdjOp	Full operator adjoint
HermOp	A hermitian version of operator (possibly squared)
HermOpAndNorm	A hermitian version of operator (possibly squared) returns norm result

8.1.1 MdagMLinearOperator

This Linear operator takes a SparseMatrix (Fermion operator) and implements the unpreconditioned MdagM operator with the above interface:

```
template<class Matrix,class Field>
class MdagMLinearOperator : public LinearOperatorBase<Field> {
public:
  MdagMLinearOperator(Matrix &Mat): _Mat(Mat){};
....
}
```

8.1.2 ShiftedMdagMLinearOperator

This Linear operator takes a SparseMatrix (Fermion operator) and implements the unpreconditioned *MdagM* + *shift* operator with the above interface:

```
template<class Matrix,class Field>
class ShiftedMdagMLinearOperator : public LinearOperatorBase<Field> {
  ShiftedMdagMLinearOperator(Matrix &Mat,RealD shift): _Mat(Mat), _shift(shift){};
};
```

8.1.3 HermitianLinearOperator

This Linear operator takes an already Hermitian SparseMatrix (Fermion operator) and implements the *M* operator with the above interface:

```
template<class Matrix,class Field>
class HermitianLinearOperator : public LinearOperatorBase<Field> {
  HermitianLinearOperator(Matrix &Mat): _Mat(Mat){};
  void Op     (const Field &in, Field &out)   { _Mat.M(in,out);  }
  void AdjOp     (const Field &in, Field &out){ _Mat.M(in,out);  }
}
```

This operator is suitable for staggered Fermion inversions for example, or for $\gamma_5 D_W$ inversions.

8.2 Red Black

We introduce a base operator for Schur decomposed red black solves:

```
template<class Field>
class SchurOperatorBase :  public LinearOperatorBase<Field> {
public:
  virtual  RealD Mpc      (const Field &in, Field &out) =0;
  virtual  RealD MpcDag   (const Field &in, Field &out) =0;
  virtual void MpcDagMpc(const Field &in, Field &out,RealD &ni,RealD &no);
  virtual void HermOpAndNorm(const Field &in, Field &out,RealD &n1,RealD &n2);
  virtual void HermOp(const Field &in, Field &out);
  void Op     (const Field &in, Field &out);
  void AdjOp     (const Field &in, Field &out);
  void OpDiag (const Field &in, Field &out) ;
  void OpDir  (const Field &in, Field &out,int dir,int disp) ;
};
```

Since there are a number of ways to perform Schur decomposition and solution there are a number of derived classes.:

```
template<class Matrix,class Field>
class SchurDiagMooeeOperator :  public SchurOperatorBase<Field> {
  SchurDiagMooeeOperator (Matrix &Mat): _Mat(Mat){};
};

template<class Matrix,class Field>
class SchurDiagOneLH :  public SchurOperatorBase<Field> {
  SchurDiagOneLH (Matrix &Mat): _Mat(Mat){};
};

template<class Matrix,class Field>
class SchurDiagOneRH :  public SchurOperatorBase<Field> {
SchurDiagOneRH (Matrix &Mat): _Mat(Mat){};
};
```

For staggered fermions the Schur decomposition operator is:

```
template<class Matrix,class Field>
class SchurStaggeredOperator :  public SchurOperatorBase<Field> {
}
```

The meaning of these different operators is

Operator	Description
SchurDiagMooeeeOperator	$M_{oo} + M_{oe} M_e e^{-1} M_{eo}$
SchurDiagOneLH	$1 + M_{oo}^{-1} M_{oe} M_e e^{-1} M_{eo}$
SchurDiagOneRH	$1 + M_{oe} M_e e^{-1} M_{eo} M_{oo}^{-1}$
SchurStaggeredOperator	

CHAPTER

NINE

OPERATOR FUNCTIONS

Can this be simplified ???

I think a single class could do both OperatorFunction and LinearFunction roles Just need to pass the Operator in as a constructor???

Audit this:

```
template<class Field> class OperatorFunction {
  virtual void operator() (LinearOperatorBase<Field> &Linop, const Field &in, Field &
↪out) = 0;
};

template<class Field> class LinearFunction {
  virtual void operator() (const Field &in, Field &out) = 0;
};

////////////////////////////////////////////////////////////
// Base classes for Multishift solvers for operators
////////////////////////////////////////////////////////////
template<class Field> class OperatorMultiFunction {
  virtual void operator() (LinearOperatorBase<Field> &Linop, const Field &in,␣
↪std::vector<Field> &out) = 0;
};

////////////////////////////////////////////////////////////////////////////////////////
↪//////
// Hermitian operator Linear function and operator function
////////////////////////////////////////////////////////////////////////////////////////
↪//////
template<class Field>
class HermOpOperatorFunction : public OperatorFunction<Field> {
  void operator() (LinearOperatorBase<Field> &Linop, const Field &in, Field &out);
};

template<typename Field>
class PlainHermOp : public LinearFunction<Field> {
  PlainHermOp(LinearOperatorBase<Field>& linop);
};

template<typename Field>
class FunctionHermOp : public LinearFunction<Field> {
  FunctionHermOp(OperatorFunction<Field> & poly,LinearOperatorBase<Field>& linop) ;
};
```

CHAPTER

TEN

ALGORITHMS

In this section we describe a number of algorithmic areas present in the core Grid library.

- Approximation: Chebyshev and Rational
- Krylov solvers: Conjugate Gradient
- Eigensolver: Chebyshev preconditioned Lanczos
- FFT: multidimensional FFT of arbitrary lattice fields

10.1 Approximation

Both Chebyshev and Rational approximation codes are included.

10.1.1 Polynomial

A polynomial of an operator with a given set of coefficients can be applied:

```
template<class Field>
class Polynomial : public OperatorFunction<Field> {
    Polynomial(std::vector<RealD> &_Coeffs) ;
};
```

10.1.2 Chebyshev

Class:

```
template<class Field> class Chebyshev : public OperatorFunction<Field>
```

Defines constructors:

```
Chebyshev(ChebyParams p);
Chebyshev(RealD _lo,RealD _hi,int _order, RealD (* func)(RealD) );
Chebyshev(RealD _lo,RealD _hi,int _order) ;
```

and methods:

```
RealD approx(RealD x);

void operator() (LinearOperatorBase<Field> &Linop, const Field &in, Field &out) ;
```

This will apply the appropriate polynomial of the LinearOperator Linop to the type Field. The coefficient for the standard Chebyshev approximation to an arbitrary function, over tha range [hi,lo] can be set up with the appropriate constructor call.

10.1.3 Remez

We adopt the Remez class written by Kate Clark with minimal modifications for compatibility

Class:

```
class AlgRemez
```

10.2 Iterative solvers and algorithms

We document a number of iterative algorithms of topical relevance to Lattice Gauge theory. These are written for application to arbitrary fields and arbitrary operators using type templating, by implementating them as arbitrary OperatorFunctions.

Most of these algorithms these algorithms operate on a generic matrix class, which derives from LinearOperatorBase.

10.2.1 Linear operators

By sharing the class for Sparse Matrix across multiple operator wrappers, we can share code between RB and non-RB variants. Sparse matrix is an abstract fermion action def, and then the LinearOperator wrappers implement the specialisation of "Op" and "AdjOp" to the cases minimising replication of code.

algorithms/LinearOperator.h

Class:

```
template<class Field> class LinearOperatorBase {
public:

  // Support for coarsening to a multigrid
  virtual void OpDiag (const Field &in, Field &out) = 0; // Abstract base
  virtual void OpDir  (const Field &in, Field &out,int dir,int disp) = 0; // Abstract␣
↪base

  virtual void Op     (const Field &in, Field &out) = 0; // Abstract base
  virtual void AdjOp  (const Field &in, Field &out) = 0; // Abstract base
  virtual void HermOpAndNorm(const Field &in, Field &out,RealD &n1,RealD &n2) = 0;
  virtual void HermOp(const Field &in, Field &out)=0;
};
```

The specific operators are:

> template<class Matrix,class Field> class MdagMLinearOperator : public LinearOperatorBase<Field> template<class Matrix,class Field> class ShiftedMdagMLinearOperator : public LinearOperatorBase<Field> template<class Matrix,class Field> class HermitianLinearOperator : public LinearOperatorBase<Field> template<class Field> class SchurOperatorBase : public LinearOperatorBase<Field> template<class Matrix,class Field> class SchurDiagOneRH : public SchurOperatorBase<Field> template<class Matrix,class Field> class SchurDiagOneLH : public SchurOperatorBase<Field> template<class Matrix,class Field> class SchurStaggeredOperator : public SchurOperatorBase<Field>

10.2.2 Conjugate Gradient

algorithms/iterative/ConjugateGradient.h

Class:

```
template <class Field>  class ConjugateGradient : public OperatorFunction<Field>
```

with methods:

```
ConjugateGradient(RealD tol, Integer maxit, bool err_on_no_conv = true);

void operator()(LinearOperatorBase<Field> &Linop, const Field &src, Field &psi) ;
```

10.2.3 Multishift Conjugate Gradient

algorithms/iterative/ConjugateGradientMultiShift.h

Class:: template<class Field> class ConjugateGradientMultiShift : public OperatorMultiFunction<Field>, public OperatorFunction<Field>

with methods:

```
ConjugateGradient(RealD tol, Integer maxit, bool err_on_no_conv = true);

void operator()(LinearOperatorBase<Field> &Linop, const Field &src, Field &psi) ;
```

10.2.4 Block Conjugate Gradient

algorithms/iterative/BlockConjugateGradient.h

Class:

```
template <class Field> class BlockConjugateGradient : public OperatorFunction<Field>
```

Several options are possible. The behaviour is controlled by an enumeration.

Enum:

```
enum BlockCGtype { BlockCG, BlockCGrQ, CGmultiRHS };
```

Constructor:

```
BlockConjugateGradient(BlockCGtype cgtype,int _Orthog,RealD tol, Integer maxit, bool␣
↪err_on_no_conv = true)
```

With operator:

```
void operator()(LinearOperatorBase<Field> &Linop, const Field &Src, Field &Psi)
```

- CGmultiRHS

This applies conjugate gradient to multiple right hand sides concurrently making use of a separate Krylov space for each. There is no cross coupling and the routine is equivalent to running each of these independently one after the other in term of iteration count.

- BlockCGrQ

This applies block conjugate gradient to multiple right hand sides concurrently making use of a shared Krylov space for each. The cross coupling may in some cases lead to acceleration of convergence and reduced matrix multiplies for multiple solves.

10.2.5 Mixed precision Conjugate Gradient

Class:

```
template<class FieldD,class FieldF>  class MixedPrecisionConjugateGradient : public␣
↪LinearFunction<FieldD>
```

Applies an inner outer mixed precision Conjagate Gradient. It has constructor:

```
MixedPrecisionConjugateGradient(RealD tol, Integer maxinnerit, Integer maxouterit,␣
↪GridBase* _sp_grid, LinearOperatorBase<FieldF> &_Linop_f, LinearOperatorBase<FieldD>
↪ &_Linop_d) :
```

Where the linear operators are for the single precision and double precision operators respectively. The operator to apply the inversion is:

```
void operator() (const FieldD &src_d_in, FieldD &sol_d){
```

10.2.6 Preconditioned Conjugate Residual

Class:

```
template<class Field> class PrecConjugateResidual : public OperatorFunction<Field>
```

Constructor:

```
PrecConjugateResidual(RealD tol,Integer maxit,LinearFunction<Field> &Prec)
```

Solve method:

```
void operator() (LinearOperatorBase<Field> &Linop,const Field &src, Field &psi)
```

10.2.7 Implicitly restarted Lanczos

Class:

```
template<class Field> class ImplicitlyRestartedLanczos
```

Solve method:

```
void calc(std::vector<RealD>& eval, std::vector<Field>& evec,  const Field& src, int&␣
↪Nconv, bool reverse=false)
```

10.2.8 Schur decomposition

Operator	Description
SchurDiagMooeeeOperator	$M_{oo} + M_{oe} M_e e^{-1} M_{eo}$
SchurDiagOneLH	$1 + M_{oo}^{-1} M_{oe} M_e e^{-1} M_{eo}$
SchurDiagOneRH	$1 + M_{oe} M_e e^{-1} M_{eo} M_{oo}^{-1}$
SchurStaggeredOperator	$m^2 - M_{oe} M_{eo}$

Associated with these operators are convenience wrappers for Schur decomposed solution of the full system are provided (red-black preconditioning, algorithms/iterative/SchurRedBlack.h):

Class:

```
template<class Field> class SchurRedBlackStaggeredSolve
template<class Field> class SchurRedBlackDiagMooeeSolve
template<class Field> class SchurRedBlackDiagOneLHSolve
template<class Field> class SchurRedBlackDiagOneRHSolve
```

Constructors:

```
SchurRedBlackStaggeredSolve(OperatorFunction<Field> &HermitianRBSolver);
SchurRedBlackDiagMooeeSolve(OperatorFunction<Field> &HermitianRBSolver,int cb=0);
SchurRedBlackDiagOneLHSolve(OperatorFunction<Field> &HermitianRBSolver,int cb=0);
SchurRedBlackDiagOneRHSolve(OperatorFunction<Field> &HermitianRBSolver,int cb=0);
```

The cb parameter specifies which checkerboard the SchurDecomposition factorises around, and the HermitianRBSolver parameter is an algorithm class, such as conjugate gradients, for solving the system of equations on a single checkerboard.

All have the operator method, returning both checkerboard solutions:

```
template<class Matrix> void operator() (Matrix & _Matrix,const Field &in, Field &out)
```

In order to allow for deflation of the preconditioned system, and external guess constructor is possible:

```
template<class Matrix, class Guesser> void operator() (Matrix & _Matrix,const Field &
↪in, Field &out,Guesser &guess){
```

CHAPTER

ELEVEN

LATTICE GAUGE THEORY UTILITIES

11.1 Spin

See, for example, tests/core/Test_gamma.cc for a complete self test of the Grid Dirac algebra, spin projectors, and Gamma multiplication table.

The spin basis is:

$$\gamma_x = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$$

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

$$\gamma_z = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$$

$$\gamma_t = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$\gamma_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

These can be accessed via a strongly typed enumeration to avoid multiplication by zeros. The values are considered opaque, and symbolic names must be used. These are signed (prefixes like MinusIdentity also work):

```
Gamma::Algebra::Identity
Gamma::Algebra::Gamma5
Gamma::Algebra::GammaT
Gamma::Algebra::GammaTGamma5
Gamma::Algebra::GammaX
Gamma::Algebra::GammaXGamma5
Gamma::Algebra::GammaY
Gamma::Algebra::GammaYGamma5
```

(continues on next page)

(continued from previous page)

```
Gamma::Algebra::GammaZ
Gamma::Algebra::GammaZGamma5
Gamma::Algebra::SigmaXT
Gamma::Algebra::SigmaXZ
Gamma::Algebra::SigmaXY
Gamma::Algebra::SigmaYT
Gamma::Algebra::SigmaYZ
Gamma::Algebra::SigmaZT
```

Example

They can be used, for example (benchmarks/Benchmark_wilson.cc):

```
Gamma::Algebra Gmu [] = {
  Gamma::Algebra::GammaX,
  Gamma::Algebra::GammaY,
  Gamma::Algebra::GammaZ,
  Gamma::Algebra::GammaT
};

{ // Naive wilson implementation
  ref = zero;
  for(int mu=0;mu<Nd;mu++){

    tmp = U[mu]*Cshift(src,mu,1);
    for(int i=0;i<ref._odata.size();i++){
      ref._odata[i]+= tmp._odata[i] - Gamma(Gmu[mu])*tmp._odata[i]; ;
    }

    tmp =adj(U[mu])*src;
    tmp =Cshift(tmp,mu,-1);
    for(int i=0;i<ref._odata.size();i++){
      ref._odata[i]+= tmp._odata[i] + Gamma(Gmu[mu])*tmp._odata[i]; ;
    }
  }
}
ref = -0.5*ref;
RealD mass=0.1;
```

Two spin projection is also possible on non-lattice fields, and used to build high performance routines such as the Wilson kernel:

```
template<class vtype>  void spProjXp (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProjYp (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProjZp (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProjTp (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProj5p (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProjXm (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProjYm (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProjZm (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
```

(continues on next page)

(continued from previous page)

```
template<class vtype>  void spProjTm (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
template<class vtype>  void spProj5m (iVector<vtype,Nhs> &hspin,const iVector<vtype,
↪Ns> &fspin)
```

and there are associated reconstruction routines for assembling four spinors from these two spinors:

```
template<class vtype>  void spReconXp (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconYp (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconZp (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconTp (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spRecon5p (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconXm (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconYm (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconZm (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spReconTm (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)
template<class vtype>  void spRecon5m (iVector<vtype,Ns> &fspin,const iVector<vtype,
↪Nhs> &hspin)

template<class vtype>  void accumReconXp (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconYp (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconZp (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconTp (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumRecon5p (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconXm (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconYm (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconZm (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumReconTm (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
template<class vtype>  void accumRecon5m (iVector<vtype,Ns> &fspin,const iVector
↪<vtype,Nhs> &hspin)
```

These ca

11.2 SU(N)

A generic Nc qcd/utils/SUn.h is provided. This defines a template class:

```
template <int ncolour> class SU ;
```

The most important external methods are:

```
static void printGenerators(void) ;
template <class cplx>  static void generator(int lieIndex, iSUnMatrix<cplx> &ta) ;

static void SubGroupHeatBath(GridSerialRNG &sRNG, GridParallelRNG &pRNG, RealD beta, ␣
↪// coeff multiplying staple in action (with no 1/Nc)
                             LatticeMatrix &link,
                             const LatticeMatrix &barestaple,  // multiplied by␣
↪action coeffs so th
                             int su2_subgroup, int nheatbath, LatticeInteger &
↪wheremask);

static void GaussianFundamentalLieAlgebraMatrix(GridParallelRNG &pRNG,
                                                LatticeMatrix &out,
                                                Real scale = 1.0) ;
static void GaugeTransform( GaugeField &Umu, GaugeMat &g)
static void RandomGaugeTransform(GridParallelRNG &pRNG, GaugeField &Umu, GaugeMat &g);

static void HotConfiguration(GridParallelRNG &pRNG, GaugeField &out) ;
static void TepidConfiguration(GridParallelRNG &pRNG,GaugeField &out);
static void ColdConfiguration(GaugeField &out);

static void taProj( const LatticeMatrixType &in,  LatticeMatrixType &out);
static void taExp(const LatticeMatrixType &x, LatticeMatrixType &ex) ;

static int su2subgroups(void) ; // returns how many subgroups
```

Specific instantiations are defined:

```
typedef SU<2> SU2;
typedef SU<3> SU3;
typedef SU<4> SU4;
typedef SU<5> SU5;
```

For example, Quenched QCD updating may be run as (tests/core/Test_quenched_update.cc):

```
for(int sweep=0;sweep<1000;sweep++){

  RealD plaq = ColourWilsonLoops::avgPlaquette(Umu);

  std::cout<<GridLogMessage<<"sweep "<<sweep<<" PLAQUETTE "<<plaq<<std::endl;

  for( int cb=0;cb<2;cb++ ) {

    one.checkerboard=subsets[cb];
    mask= zero;
    setCheckerboard(mask,one);

    //      std::cout<<GridLogMessage<<mask<<std::endl;
    for(int mu=0;mu<Nd;mu++){

      // Get Link and Staple term in action; must contain Beta and
      // any other coeffs
      ColourWilsonLoops::Staple(staple,Umu,mu);
```

(continues on next page)

(continued from previous page)

```
      link = PeekIndex<LorentzIndex>(Umu,mu);

      for( int subgroup=0;subgroup<SU3::su2subgroups();subgroup++ ) {

        // update Even checkerboard
        SU3::SubGroupHeatBath(sRNG,pRNG,beta,link,staple,subgroup,20,mask);

      }

      PokeIndex<LorentzIndex>(Umu,link,mu);

      //reunitarise link;
      ProjectOnGroup(Umu);
    }
  }
}
```

11.3 Space time grids

CHAPTER

TWELVE

LATTICE ACTIONS

We discuss in some detail the implementation of the lattice actions. The action is a sum of terms, each of which must inherit from and provide the following interface.

lib/qcd/action/ActionBase.h:

```
class Action
{

 public:
  bool is_smeared = false;
  // Heatbath?
  virtual void refresh(const GaugeField& U, GridParallelRNG& pRNG) = 0; // refresh␣
↪pseudofermions
  virtual RealD S(const GaugeField& U) = 0;                             // evaluate␣
↪the action
  virtual void deriv(const GaugeField& U, GaugeField& dSdU) = 0;        // evaluate␣
↪the action derivative
  virtual std::string action_name()    = 0;                             // return the␣
↪action name
  virtual std::string LogParameters()  = 0;                             // prints␣
↪action parameters
  virtual ~Action(){}
};
```

Fermion Lattice actions are defined in the qcd/action/fermion subdirectory and in the qcd/action/gauge subdirectories. For Hybrid Monte Carlo and derivative sampling algorithm Pseudofermoin actions are defined in the qcd/action/pseudofermion subdirectory.

The simplest lattice action is the Wilson plaquette action, and we start by considering the Wilson loops facilities as this is illustrative of the implementation policy design approach.

12.1 Wilson loops

Wilson loops are common objects in Lattice Gauge Theory. A utility class is provided to assist assembling these as they occur both in common observable construction but also in actions such as the Wilson plaquette and the rectangle actions.

Since derivatives with respect to gauge links are required for evolution codes, non-closed staples of various types are also provided. The gauge actions are all assembled consistently from the Wilson loops class.

Implementation policies

The Wilson loops class is templated to take a implementation policy class parameter. The covarian Cshift is inherity from the policy and implements boundary conditions, such as period, anti-period or charge conjugate. In this way the

Wilson loop code can automatically transform with the boundary condition and give the right plaquette, force terms etc... for the boundary conditions passed in external to the class.

This implementation policy class is called an "impl", and a class that bundles together all the required rules to assemble a gauge action is called a Gimpl.

There are several facilities provided by a Gimpl.

These include Boundary conditions and consequently a CovariantCshift.

12.1.1 CovariantCshift

Covariant Cshift operations are provided for common cases of the boundary condition. These may be further optimised in future:

```
template<class covariant,class gauge>
Lattice<covariant> CovShiftForward(const Lattice<gauge> &Link, int mu,
                                   const Lattice<covariant> &field);

template<class covariant,class gauge>
Lattice<covariant> CovShiftBackward(const Lattice<gauge> &Link, int mu,
                                    const Lattice<covariant> &field);
```

12.1.2 Boundary conditions

The covariant shift routines occur in namespaces PeriodicBC and ConjugateBC. The correct covariant shift for the boundary condition is passed into the gauge actions and wilson loops via an "Impl" template policy class.

The relevant staples, plaquettes, and loops are formed by using the provided method:

```
Impl::CovShiftForward
Impl::CovShiftBackward
```

etc... This makes physics code transform appropriately with externally supplied rules about treating the boundary.

Example (lib/qcd/util/WilsonLoops.h):

```
static void dirPlaquette(GaugeMat &plaq, const std::vector<GaugeMat> &U,
                         const int mu, const int nu) {
  // ___
  //|   |
  //|<__|
  plaq = Gimpl::CovShiftForward(U[mu],mu,
                  Gimpl::CovShiftForward(U[nu],nu,
                      Gimpl::CovShiftBackward(U[mu],mu,
                         Gimpl::CovShiftIdentityBackward(U[nu], nu))));
}
```

Currently provided predefined implementations are (qcd/action/gauge/GaugeImplementations.h):

```
typedef PeriodicGaugeImpl<GimplTypesR> PeriodicGimplR; // Real.. whichever prec
typedef PeriodicGaugeImpl<GimplTypesF> PeriodicGimplF; // Float
typedef PeriodicGaugeImpl<GimplTypesD> PeriodicGimplD; // Double

typedef PeriodicGaugeImpl<GimplAdjointTypesR> PeriodicGimplAdjR; // Real.. whichever␣
↪prec
```

(continues on next page)

(continued from previous page)

```
typedef PeriodicGaugeImpl<GimplAdjointTypesF> PeriodicGimplAdjF; // Float
typedef PeriodicGaugeImpl<GimplAdjointTypesD> PeriodicGimplAdjD; // Double

typedef ConjugateGaugeImpl<GimplTypesR> ConjugateGimplR; // Real.. whichever prec
typedef ConjugateGaugeImpl<GimplTypesF> ConjugateGimplF; // Float
typedef ConjugateGaugeImpl<GimplTypesD> ConjugateGimplD; // Double
```

12.2 Gauge Actions

lib/qcd/action/gauge/Photon.h defines the U(1) field:

```
class Photon
```

using Fourier techniques.

lib/qcd/action/gauge/WilsonGaugeAction.h defines the standard plaquette action:

```
template <class Gimpl>
class WilsonGaugeAction : public Action<typename Gimpl::GaugeField> ;
```

This action is suitable to use in a Hybrid Monte Carlo evolution as an action term and has constructor:

```
WilsonGaugeAction(RealD beta_);
```

lib/qcd/action/gauge/PlaqPlusRectangleAction.h defines the standard plaquette plus rectangle class of action:

```
template<class Gimpl>
class PlaqPlusRectangleAction : public Action<typename Gimpl::GaugeField> ;
```

The constructor is:

```
PlaqPlusRectangleAction(RealD b,RealD c);
```

Due to varying conventions, convenience wrappers are provided:

```
template<class Gimpl>    class RBCGaugeAction : public PlaqPlusRectangleAction<Gimpl>;
template<class Gimpl>    class IwasakiGaugeAction : public RBCGaugeAction<Gimpl> ;
template<class Gimpl>    class SymanzikGaugeAction : public RBCGaugeAction<Gimpl> ;
template<class Gimpl>    class DBW2GaugeAction : public RBCGaugeAction<Gimpl> ;
```

With convenience constructors to set the rectangle coefficient automatically to popular values:

```
SymanzikGaugeAction(RealD beta) : RBCGaugeAction<Gimpl>(beta,-1.0/12.0) {};
IwasakiGaugeAction(RealD beta) : RBCGaugeAction<Gimpl>(beta,-0.331) {};
DBW2GaugeAction(RealD beta) : RBCGaugeAction<Gimpl>(beta,-1.4067) {};
```

12.3 Fermion

These classes all make use of a Fermion Implementation (Fimpl) policy class to provide things like boundary conditions, covariant transportation rules etc.

All Fermion operators actions inherit from a common base class,

that conforms to the CheckerBoardedSparseMatrix interface and constrains these objects to conform to the interface expected by general algorithms in Grid:

```
////////////////////////////////////////////////////////////////////////////////////
// Interface defining what I expect of a general sparse matrix, such as a Fermion action
////////////////////////////////////////////////////////////////////////////////////
template<class Field> class SparseMatrixBase {
public:
  virtual GridBase *Grid(void) =0;
  // Full checkerboar operations
  virtual RealD M    (const Field &in, Field &out)=0;
  virtual RealD Mdag (const Field &in, Field &out)=0;
  virtual void  MdagM(const Field &in, Field &out,RealD &ni,RealD &no) {
    Field tmp (in._grid);
    ni=M(in,tmp);
    no=Mdag(tmp,out);
  }
  virtual  void Mdiag    (const Field &in, Field &out)=0;
  virtual  void Mdir     (const Field &in, Field &out,int dir, int disp)=0;
};

////////////////////////////////////////////////////////////////////////////////////
// Interface augmented by a red black sparse matrix, such as a Fermion action
////////////////////////////////////////////////////////////////////////////////////
template<class Field> class CheckerBoardedSparseMatrixBase : public SparseMatrixBase<Field> {
public:
  virtual GridBase *RedBlackGrid(void)=0;
  // half checkerboard operaions
  virtual  void Meooe    (const Field &in, Field &out)=0;
  virtual  void Mooee    (const Field &in, Field &out)=0;
  virtual  void MooeeInv (const Field &in, Field &out)=0;

  virtual  void MeooeDag    (const Field &in, Field &out)=0;
  virtual  void MooeeDag    (const Field &in, Field &out)=0;
  virtual  void MooeeInvDag (const Field &in, Field &out)=0;

};
```

The base class for Fermion Operators inherits frmo these:

```
template<class Impl>
class FermionOperator : public CheckerBoardedSparseMatrixBase<typename Impl::FermionField>, public Impl
```

These all make use of an implementation template class, and the possible implementations include:

```
typedef WilsonImpl<vComplexF, FundamentalRepresentation, CoeffReal > WilsonImplF;  // Float
typedef WilsonImpl<vComplexD, FundamentalRepresentation, CoeffReal > WilsonImplD;  // Double
```

Staggered fermions make us of a spin index free field via the StaggeredImpl:

```
typedef StaggeredImpl<vComplexF, FundamentalRepresentation > StaggeredImplF;   // Float
typedef StaggeredImpl<vComplexD, FundamentalRepresentation > StaggeredImplD;   //␣
↪Double
```

A number of alternate, non-fundamental Fermion representations are supported. Note that the Fermion action code is common to each of these, demonstrating the utility of the template Fimpl classes for separating the code that varies from the invariant sections:

```
typedef WilsonImpl<vComplexF, AdjointRepresentation, CoeffReal > WilsonAdjImplF;   //␣
↪Float
typedef WilsonImpl<vComplexD, AdjointRepresentation, CoeffReal > WilsonAdjImplD;   //␣
↪Double

typedef WilsonImpl<vComplexF, TwoIndexSymmetricRepresentation, CoeffReal >␣
↪WilsonTwoIndexSymmetricImplF;   // Float
typedef WilsonImpl<vComplexD, TwoIndexSymmetricRepresentation, CoeffReal >␣
↪WilsonTwoIndexSymmetricImplD;   // Double

typedef WilsonImpl<vComplexF, TwoIndexAntiSymmetricRepresentation, CoeffReal >␣
↪WilsonTwoIndexAntiSymmetricImplF;   // Float
typedef WilsonImpl<vComplexD, TwoIndexAntiSymmetricRepresentation, CoeffReal >␣
↪WilsonTwoIndexAntiSymmetricImplD;   // Double
```

G-parity boundary conditions are supported, and an additional flavour index inserted on the Fermion field via the Gparity implementation:

```
typedef GparityWilsonImpl<vComplexF, FundamentalRepresentation,CoeffReal>␣
↪GparityWilsonImplF;   // Float
typedef GparityWilsonImpl<vComplexD, FundamentalRepresentation,CoeffReal>␣
↪GparityWilsonImplD;   // Double
```

ZMobius Fermions use complex rather than real action coefficients and are supported via an alternate implementation:

```
typedef WilsonImpl<vComplexF, FundamentalRepresentation, CoeffComplex > ZWilsonImplF;␣
↪// Float
typedef WilsonImpl<vComplexD, FundamentalRepresentation, CoeffComplex > ZWilsonImplD;␣
↪// Double
```

Some example constructor calls are given below for Wilson and Clover fermions:

```
template <class Impl> class WilsonFermion;
```

With constructor:

```
WilsonFermion(GaugeField &_Umu, GridCartesian &Fgrid,
              GridRedBlackCartesian &Hgrid, RealD _mass,
              const ImplParams &p = ImplParams(),
              const WilsonAnisotropyCoefficients &anis =␣
↪WilsonAnisotropyCoefficients() );
```

and:

```
template <class Impl> class WilsonCloverFermion : public WilsonFermion<Impl>;
```

with constructor:

```
WilsonCloverFermion(GaugeField &_Umu, GridCartesian &Fgrid,
                    GridRedBlackCartesian &Hgrid,
                    const RealD _mass,
                    const RealD _csw_r = 0.0,
                    const RealD _csw_t = 0.0,
                    const WilsonAnisotropyCoefficients &clover_anisotropy =␣
↪WilsonAnisotropyCoefficients(),
                    const ImplParams &impl_p = ImplParams());
```

Additional paramters allow for anisotropic versions to be created, which take default values for the isotropic case.

The constuctor signatures can be found in the header files in qcd/action/fermion/ A complete list of the 4D ultralocal Fermion types is:

```
WilsonFermion<WilsonImplF>                              WilsonFermionF;
WilsonFermion<WilsonAdjImplF>                           WilsonAdjFermionF;
WilsonFermion<WilsonTwoIndexSymmetricImplF>             WilsonTwoIndexSymmetricFermionF;
WilsonFermion<WilsonTwoIndexAntiSymmetricImplF>         ␣
↪WilsonTwoIndexAntiSymmetricFermionF;
WilsonTMFermion<WilsonImplF>                            WilsonTMFermionF;
WilsonCloverFermion<WilsonImplF>                        WilsonCloverFermionF;
WilsonCloverFermion<WilsonAdjImplF>                     WilsonCloverAdjFermionF;
WilsonCloverFermion<WilsonTwoIndexSymmetricImplF>       ␣
↪WilsonCloverTwoIndexSymmetricFermionF;
WilsonCloverFermion<WilsonTwoIndexAntiSymmetricImplF>␣
↪WilsonCloverTwoIndexAntiSymmetricFermionF;
ImprovedStaggeredFermion<StaggeredImplF>                ImprovedStaggeredFermionF;
```

Cayley form chiral fermions (incl. domain wall):

```
DomainWallFermion<WilsonImplF>                    DomainWallFermionF;
DomainWallEOFAFermion<WilsonImplF>                DomainWallEOFAFermionF;
MobiusFermion<WilsonImplF>                        MobiusFermionF;
MobiusEOFAFermion<WilsonImplF>                    MobiusEOFAFermionF;
ZMobiusFermion<ZWilsonImplF>                      ZMobiusFermionF;
ScaledShamirFermion<WilsonImplF>                  ScaledShamirFermionF;
MobiusZolotarevFermion<WilsonImplF>               MobiusZolotarevFermionF;
ShamirZolotarevFermion<WilsonImplF>               ShamirZolotarevFermionF;
OverlapWilsonCayleyTanhFermion<WilsonImplF>       OverlapWilsonCayleyTanhFermionF;
OverlapWilsonCayleyZolotarevFermion<WilsonImplF> OverlapWilsonCayleyZolotarevFermionF;
```

Continued fraction overlap:

```
   OverlapWilsonContFracTanhFermion<WilsonImplF>      ␣
↪OverlapWilsonContFracTanhFermionF;
   OverlapWilsonContFracZolotarevFermion<WilsonImplF>␣
↪OverlapWilsonContFracZolotarevFermionF;

Partial fraction overlap::

   OverlapWilsonPartialFractionTanhFermion<WilsonImplF>␣
↪OverlapWilsonPartialFractionTanhFermionF;
   OverlapWilsonPartialFractionZolotarevFermion<WilsonImplF>␣
↪OverlapWilsonPartialFractionZolotarevFermionF;

 Gparity cases; a partial list is defined until tested::

   WilsonFermion<GparityWilsonImplF>         GparityWilsonFermionF;
```

(continues on next page)

(continued from previous page)

```
   DomainWallFermion<GparityWilsonImplF>      GparityDomainWallFermionF;
   DomainWallEOFAFermion<GparityWilsonImplF> GparityDomainWallEOFAFermionF;

   WilsonTMFermion<GparityWilsonImplF>        GparityWilsonTMFermionF;
   MobiusFermion<GparityWilsonImplF>          GparityMobiusFermionF;
   MobiusEOFAFermion<GparityWilsonImplF>      GparityMobiusEOFAFermionF;
```

For each action, the suffix "F" can be replaced with "D" to obtain a double precision version. More generally, it is possible to perform communications with a different precision from computation. The number of combinations is rather large to list, but in the above listing the substitution is the obvious one.

Suffix	Computation	Communication
F	fp32	fp32
D	fp64	fp64
R	default	default
FH	fp32	fp16
DF	fp64	fp32
RL	default	lower

12.4 Pseudofermion

Pseudofermion actions are defined in qcd/action/pseudofermion/ . These action terms are built from template classes:

```
// Base even odd HMC on the normal Mee based schur decomposition.
//
//     M = (Mee Meo) =  (1             0 )    (Mee   0               )   (1 Mee^{-1}␣
↪Meo)
//         (Moe Moo)    (Moe Mee^-1    1 )    (0   Moo-Moe Mee^-1 Meo)   (0   1       ␣
↪)
//
// Determinant is det of middle factor. This assumes Mee is indept of U.
template<class Impl> class SchurDifferentiableOperator ;

// S = phi^dag (Mdag M)^-1 phi
template <class Impl>  class TwoFlavourPseudoFermionAction ;

// S = phi^dag V (Mdag M)^-1 Vdag phi
template<class Impl>   class TwoFlavourRatioPseudoFermionAction ;

// S = phi^dag (Mdag M)^-1 phi  (odd)
//   + phi^dag (Mdag M)^-1 phi  (even)
template <class Impl>  class TwoFlavourEvenOddPseudoFermionAction;

// S = phi^dag V (Mdag M)^-1 Vdag phi
template <class Impl>  class TwoFlavourEvenOddRatioPseudoFermionAction ;
```

Rational HMC pseudofermion terms:

```
// S_f = chi^dag *  N(M^dag*M)/D(M^dag*M) * chi
//
// Here, M is some operator
// N and D makeup the rat. poly
template<class Impl> class OneFlavourRationalPseudoFermionAction;
```

(continues on next page)

(continued from previous page)

```
// S_f = chi^dag* P(V^dag*V)/Q(V^dag*V)* N(M^dag*M)/D(M^dag*M)* P(V^dag*V)/Q(V^
↪dag*V)* chi
//
// Here P/Q \sim R_{1/4}  ~ (V^dagV)^{1/4}
// Here N/D \sim R_{-1/2} ~ (M^dagM)^{-1/2}
template<class Impl> class OneFlavourRatioRationalPseudoFermionAction;


// S = phi^dag (Mdag M)^-1/2 phi
template <class Impl> class OneFlavourEvenOddRationalPseudoFermionAction;

// S_f = chi^dag* P(V^dag*V)/Q(V^dag*V)* N(M^dag*M)/D(M^dag*M)* P(V^dag*V)/Q(V^
↪dag*V)* chi
//
// Here P/Q \sim R_{1/4}  ~ (V^dagV)^{1/4}
// Here N/D \sim R_{-1/2} ~ (M^dagM)^{-1/2}
template<class Impl> class OneFlavourEvenOddRatioRationalPseudoFermionAction;
```

The relevant Fermion operators are constructed externally, and references are passed in to these object constructors. Thus, they work for any Fermion operator and the code can be shared. For example, one of the constructors is given as:

```
TwoFlavourEvenOddRatioPseudoFermionAction(FermionOperator<Impl>  &_NumOp,
                                          FermionOperator<Impl>  &_DenOp,
                                          OperatorFunction<FermionField> & DS,
                                          OperatorFunction<FermionField> & AS) :
```

The exact one flavour algorithm for Domain Wall Fermions is present but is not documented here:

```
#include <Grid/qcd/action/pseudofermion/ExactOneFlavourRatio.h>
```

CHAPTER

THIRTEEN

HMC

There are a large number of examples under tests/hmc/

The most important data structure associated with (R)HMC describes the action and integration scheme.

The action is a sum of terms, possibly with nested timesteps.

This is assembled in an ActionSet object (qcd/action/ActionSet.h). The timesteps are managed by Levels. The ActionSet object maintains a list of ActionLevel objects:

```
// Define the ActionSet
template <class GaugeField, class R>
using ActionSet = std::vector<ActionLevel<GaugeField, R> >;
```

Each ActionLevel is associated with each level of the nested integration scheme, schematically:

```
template <class Field>
struct ActionLevel {
public:
  unsigned int multiplier;
  // Fundamental repr actions separated because of the smearing
  typedef Action<Field>* ActPtr;
  std::vector<ActPtr>& actions;
}
```

The outer loop, running MD trajectories, Metropolis steps, saving and restoring configurations is generic and managed by a "Runner" class..

There are a number of possible integrators: LeapFrog, MinimumNorm2, ForceGradient

These are a template parameter to the HMCRunner class. We will take as an example Test_hmc_EODWFRatio.cc:

```
typedef GenericHMCRunner<MinimumNorm2> HMCWrapper;  // Uses the default minimum norm
```

The test defines the Fermion action and Gauge action

> typedef WilsonImplR FermionImplPolicy; typedef DomainWallFermionR FermionAction; typedef typename FermionAction::FermionField FermionField; HMCWrapper TheHMC;

The HMC runner is given the Grid information (lifted from standard Grid –grid Lx.Ly.Lz.Lt command line):

```
TheHMC.Resources.AddFourDimGrid("gauge");
```

The checkpointing strategy is defined:

```
// Checkpointer definition
CheckpointerParameters CPparams;
CPparams.config_prefix = "ckpoint_EODWF_lat";
CPparams.rng_prefix = "ckpoint_EODWF_rng";
CPparams.saveInterval = 5;
CPparams.format = "IEEE64BIG";

TheHMC.Resources.LoadNerscCheckpointer(CPparams);
```

Todo: HOW TO resume from saved RNGs. Guido changed this.

Random number generators are seeded:

```
RNGModuleParameters RNGpar;
RNGpar.serial_seeds = "1 2 3 4 5";
RNGpar.parallel_seeds = "6 7 8 9 10";
TheHMC.Resources.SetRNGSeeds(RNGpar);
```

Observables measured at the end of the trajectory can be registered:

```
// Construct observables
// here there is too much indirection
typedef PlaquetteMod<HMCWrapper::ImplPolicy> PlaqObs;
TheHMC.Resources.AddObservable<PlaqObs>();
```

The action must be defined:

```
RealD beta = 5.6 ;
WilsonGaugeActionR Waction(beta);

const int Ls = 8;
auto GridPtr   = TheHMC.Resources.GetCartesian();
auto GridRBPtr = TheHMC.Resources.GetRBCartesian();
auto FGrid     = SpaceTimeGrid::makeFiveDimGrid(Ls,GridPtr);
auto FrbGrid   = SpaceTimeGrid::makeFiveDimRedBlackGrid(Ls,GridPtr);

LatticeGaugeField U(GridPtr);

Real mass = 0.04;
Real pv   = 1.0;
RealD M5  = 1.5;

FermionAction DenOp(U,*FGrid,*FrbGrid,*GridPtr,*GridRBPtr,mass,M5);
FermionAction NumOp(U,*FGrid,*FrbGrid,*GridPtr,*GridRBPtr,pv,  M5);

double StoppingCondition = 1.0e-8;
double MaxCGIterations = 10000;
ConjugateGradient<FermionField>  CG(StoppingCondition,MaxCGIterations);
TwoFlavourEvenOddRatioPseudoFermionAction<FermionImplPolicy> Nf2(NumOp, DenOp,CG,CG);

  // Set smearing (true/false), default: false
Nf2.is_smeared = false;

// Collect actions
ActionLevel<HMCWrapper::Field> Level1(1);
```

(continues on next page)

(continued from previous page)

```
Level1.push_back(&Nf2);

ActionLevel<HMCWrapper::Field> Level2(4);
Level2.push_back(&Waction);

TheHMC.TheAction.push_back(Level1);
TheHMC.TheAction.push_back(Level2);
```

And the HMC can be setup and run:

```
TheHMC.Parameters.MD.MDsteps = 20;
TheHMC.Parameters.MD.trajL   = 1.0;
TheHMC.ReadCommandLine(argc, argv); // these can be parameters from file

TheHMC.Run();  // no smearing
```

This puts together the pieces of the previous sections (actions, Fermion operators, solver algorithms etc. . .) into a full application.

CHAPTER

FOURTEEN

DEVELOPMENT OF THE INTERNALS

Todo: CD: The whole section needs to be completed, of course

The interfaces used in this chapter of the manual are subject to change without notice as new architectures are addressed.

The intent is to document the approach taken by Grid to optimised code, and provide guidelines to those developers working on the internals.

14.1 Simd classes

14.2 Communications facilities

14.3 Cartesian Grid facilities and field layout

14.4 Stencil construction

14.5 Optimised fermion operators

14.6 Optimised communications

CHAPTER

FIFTEEN

INTERFACING WITH EXTERNAL SOFTWARE

Grid provides a number of important modules, such as solvers and eigensolvers, that are highly optimized for complex vector/SIMD architectures, such as the Intel Xeon Phi KNL and Skylake processors. This growing library, with appropriate interfacing, can be accessed from existing code. Here we describe interfacing issues and provide examples.

15.1 MPI initialization

Grid supports threaded MPI sends and receives and, if running with more than one thread, requires the MPI_THREAD_MULTIPLE mode of message passing. If the user initializes MPI before starting Grid, the appropriate initialization call is:

```
MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);
assert(MPI_THREAD_MULTIPLE == provided);
```

15.2 Grid Initialization

Grid itself is initialized with a call:

```
Grid_init(&argc, &argv);
```

Todo: CD: Where are the command-line arguments explained above?

where *argc* and *argv* are constructed to simulate the command-line options described above. At a minimum one must provide the *–grid* and *–mpi* parameters. The latter specifies the grid of processors (MPI ranks).

The following Grid procedures are useful for verifying that Grid is properly initialized.

Grid procedure	returns
std::vector<int> GridDefaultLatt();	lattice size
std::vector<int> GridDefaultSimd(int Nd,vComplex::Nsimd());	SIMD layout
std::vector<int> GridDefaultMpi();	MPI layout
int Grid::GridThread::GetThreads();	number of threads

15.3 MPI coordination

Grid wants to use its own numbering of MPI ranks and its own assignment of the lattice coordinates with each rank. Obviously, the calling program and Grid must agree on these conventions. It is convenient to use Grid's Cartesian communicator class to discover the processor assignments. For a four-dimensional processor grid one can define:

```
static Grid::CartesianCommunicator *grid_cart = NULL;
grid_cart = new Grid::CartesianCommunicator(processors);
```

where *processors* is of type *std::vector<int>*, with values matching the MPI processor-layout dimensions specified with the *–mpi* argument in the *Grid_Init* call. Then each MPI rank can obtain its processor coordinate using the Cartesian communicator instantiated above. For example, in four dimensions:

```
std::vector<int> pePos(4);
for(int i=0; i<4; i++)
   pePos[i] = grid_cart->_processor_coor[i];
```

and each MPI process can get its world rank from its processor coordinates using:

```
int peRank = grid_cart->RankFromProcessorCoor(pePos)
```

Conversely, each MPI process can get its processor coordinates from its world rank using:

```
grid_cart->ProcessorCoorFromRank(peRank, pePos);
```

If the calling program initialized MPI before initializing Grid, it is then important for each MPI process in the calling program to reset its rank number so it agrees with Grid:

```
MPI_Comm comm;
MPI_Comm_split(MPI_COMM_THISJOB,jobid,peRank,&comm);
MPI_COMM_THISJOB = comm;
```

where *MPI_COMM_THISJOB* is initially a copy of *MPI_COMM_WORLD* (with *jobid = 0*), or it is a split communicator with *jobid* equal to the index number of the subcommunicator. Once this is done,:

```
MPI_Comm_rank(MPI_COMM_THISJOB, &myrank);
```

returns a rank that agrees with Grid's *peRank*.

15.4 Mapping fields between Grid and user layouts

In order to map data between layouts, it is important to know how the lattice sites are distributed across the processor grid. A lattice site with coordinates *r[mu]* is assigned to the processor with processor coordinates *pePos[mu]* according to the rule:

```
pePos[mu] = r[mu]/dim[mu]
```

where *dim[mu]* is the lattice dimension in the *mu* direction. For performance reasons, it is important that the external data layout follow the same rule. Then data mapping can be done without requiring costly communication between ranks. We assume this is the case here.

When mapping data to and from Grid, one must choose a lattice object defined on the appropriate grid, whether it be a full lattice (4D *GridCartesian*), one of the checkerboards (4D *GridRedBlackCartesian*), a five-dimensional full grid (5D *GridCartesian*), or a five-dimensional checkerboard (5D *GridRedBlackCartesian*). For example, an improved staggered fermion color-vector field *cv* on a single checkerboard would be constructed using

Example:

```
std::vector<int> latt_size   = GridDefaultLatt();
std::vector<int> simd_layout = GridDefaultSimd(Nd,vComplex::Nsimd());
std::vector<int> mpi_layout  = GridDefaultMpi();

GridCartesian               Grid(latt_size,simd_layout,mpi_layout);
GridRedBlackCartesian       RBGrid(&Grid);

typename ImprovedStaggeredFermion::FermionField  cv(RBGrid);
```

To map data within an MPI rank, the external code must iterate over the sites belonging to that rank (full or checkerboard as appropriate). To import data into Grid, the external data on a single site with coordinates *r* is first copied into the appropriate Grid scalar object *s*. Then it is copied into the Grid lattice field *l* with *pokeLocalSite*:

```
pokeLocalSite(const sobj &s, Lattice<vobj> &l, Coordinate &r);
```

To export data from Grid, the reverse operation starts with:

```
peekLocalSite(const sobj &s, Lattice<vobj> &l, Coordinate &r);
```

and then copies the single-site data from *s* into the corresponding external type.

Here is an example that maps a single site's worth of data in a MILC color-vector field to a Grid scalar ColourVector object *cVec* and from there to the lattice colour-vector field *cv*, as defined above.

Example:

```
indexToCoords(idx,r);
ColourVector cVec;
for(int col=0; col<Nc; col++)
    cVec._internal._internal._internal[col] =
        Complex(src[idx].c[col].real, src[idx].c[col].imag);

pokeLocalSite(cVec, cv, r);
```

Here the *indexToCoords()* function is a MILC mapping of the MILC site index *idx* to the 4D lattice coordinate *r*.

Grid provides block- and multiple-rhs conjugate-gradient solvers. For this purpose it uses a 5D lattice. To map data to and from Grid data types, the index for the right-hand-side vector becomes the zeroth coordinate of a five-dimensional vector *r5*. The remaining components of *r5* contain the 4D space-time coordinates. The *pokeLocalSite/peekLocalSite* operations then accept the coordinate *r5*, provided the destination/source lattice object is also 5D. In the example below data from a single site specified by *idx*, belonging to a set of *Ls* MILC color-vector fields, are copied into a Grid 5D fermion field *cv5*.

Example:

```
GridCartesian * UGrid = SpaceTimeGrid::makeFourDimGrid(GridDefaultLatt();
GridRedBlackCartesian * FrbGrid = SpaceTimeGrid::makeFiveDimRedBlackGrid(Ls,UGrid)
↪typename ImprovedStaggeredFermion5D::FermionField  cv5(FrbGrid);

std::vector<int> r(4);
indexToCoords(idx,r);
std::vector<int> r5(1,0);
for( int d = 0; d < 4; d++ ) r5.push_back(r[d]);

for( int j = 0; j < Ls; j++ ){
  r5[0] = j;
  ColourVector cVec;
```

(continues on next page)

(continued from previous page)

```
  for(int col=0; col<Nc; col++){
      cVec._internal._internal._internal[col] =
          Complex(src[j][idx].c[col].real, src[j][idx].c[col].imag);
  }
  pokeLocalSite(cVec, *(out->cv), r5);
}
```