Grid Documentation

Peter Boyle, Guido Cossu, Antonin Portelli, Azusa Yamaguchi

Sep 06, 2018

CONTENTS:

CHAPTER

ONE

PRELIMINARIES

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).
–enable-precision={single|double}	set the default precision (default: *double*).
–enable-precision=<comm>	use *<comm>* for message passing (default: *none*). A list of possible SIMD targets is detailed in a section below.
–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. This is the mpi3 communications implementation. We recommend four ranks per node for best performance, but optimum is local volume dependent.

```
../configure --enable-precision=double\
          --enable-simd=KNL        \
          --enable-comms=mpi3-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=mpi3-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=mpi3      \
           --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=mpi3      \
             --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=mpi3      \
             --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=mpi3 \
             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 MPI3 and shared memory to communicate within this node:

```
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] ; }
```

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.

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 we declare:

```
template<typename vtype>
using iLorentzColourMatrix = iVector<iScalar<iMatrix<vtype, Nc> >, Nd > ;

typedef iLorentzColourMatrix<ComplexD > LorentzColourMatrixD;
```

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)

// 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)
```

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
////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
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";

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)
```

(continues on next page)

(continued from previous page)

```
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 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,
              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.

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());
```

(continues on next page)

(continued from previous page)

```
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);
```

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	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	LatticeHalfSpinColourVec-torD	LatticeHalf-FermionD
Vector	Scalar	Ma-trix	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.

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);
```

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.

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);
```

(continues on next page)

(continued from previous page)

```
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 operatons 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 N-th dimension. 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);
```

(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 an accelerated_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();
accelerator_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];
});
```

5.7.4 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]; ;
      }
    }
  }
}
```

5.7.5 CovariantCshift

Covariant Cshift operations are provided for common cases of 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);
```

5.7.6 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))));
}
```

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,
```

(continues on next page)

(continued from previous page)

```
                  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.

5.9 Input/Output facilities

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);
}

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

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

template <class sobj> void random(GridSerialRNG &rng,sobj &l)   { rng.fill(l,rng._
↪uniform  ); }
template <class sobj> 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;
};
```

Member	Description
M	
Mdag	
MdagM	
Mdiag	
Mdir	
Meooe	
Mooee	
MooeInv	
MeooeDag	
MooeeDag	
MooeeInvDag	

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	
OpDir	
Op	
AdjOp	
HermOpAndNorm	
HermOp	

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	
SchurDiagOneLH	
SchurDiagOneRH	
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

10.1 Approximation

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) ;
};
```

CHAPTER

ELEVEN

LATTICE GAUGE THEORY UTILITIES

CHAPTER

TWELVE

LATTICE ACTIONS

12.1 Gauge

12.2 Fermion

12.3 Pseudofermion

CHAPTER

THIRTEEN

HMC

CHAPTER

FOURTEEN

DEVELOPMENT OF THE INTERNALS

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