This short note summarizes the requirements for a linear algebra API that combines data-parallel operations with communications. It also presents sufficient implementation ideas to demonstrate that this API can be implemented with high performance.

The API is intended to be sufficiently flexible for all Lattice QCD applications, and execute efficiently on all existing and anticipated platforms, so that there is no need to directly call non-portable message passing routines.

It is intended for this API to be built on the message passing (communications) API, but no direct calls into that API are apparent here.

**Capability Requirements**

1. Data-parallel operations (logically SIMD) on all sites across a lattice.
2. Synchronous (blocking) shifts on lattice level objects among virtual grid directions
3. Broadcast operations (filling a lattice quantity from a scalar value(s))
4. Global reduction operations
5. Ability to map problem grid onto machine grid and handle subnode looping for site operations and face communications between nodes for shifts.
6. Use underlying geometry configured by Message Passing API.
7. Perform site level operations on various data-types primitives, like matrices, vectors, products of matrices (propagators).
8. Lattice objects have a unique (internally known) data-layout.
9. Support SU(3) and SU(2) (? ? ? ?)

**API Design: Performance Requirements**

1. Design must allow for overlapping of computation and communications.
2. Operations must allow some internal layout that is possibly checkerboarded, but otherwise not directly known. E.g., checkerboarding affects how communications are performed. A query can be made to return layout info
3. Operations do not (by default) make conversions of objects from one layout into another layout so as to perform the operation. An error is generated if objects are in incompatible layouts.
4. Allow different underlying layouts to optimize implementations on a wide range of machines. Notably, (super)-scalar microprocessors with primary and secondary caches, and also classic vector machines.
API Open Issues

1. In order to allow the implementation to optimize data layout for the architecture, we may want to implement methods to create lattice objects (factory methods or object constructors).

Hardware Issues

A design (probably not the only one possible) that addresses these performance constraints is something along the lines of data-parallel operations akin to Fortran 90, but with support for more complex primitive types:

1. Predefined channels are opened between neighboring nodes of a d-dimensional grid.
2. On clusters of scalar nodes, the face of a subnode is sent non-blocking to a neighboring node say in the forward direction. The neighboring node in the corresponding backward direction sends its face into a preallocated buffer. While this is going on, the specified operation is performed on the interior sites. A wait is issued and the operation is performed on the face.

API Motivation

Ideally, overloaded routines would be called making names easier to remember. Since this is not possible directly in C, unique names must be used. To see how the idealized case might look, below are a few representative calls, written as a C api. Translation into C++ is possible.

Example

Consider the equation  
\[ \text{dest}(x) = \text{src1}(x) \times \text{src2}(x+\text{dir}) \quad (\text{for all } x) \]

This can be decomposed as  
\[ \text{tmp1}(x) = \text{src2}(x+\text{dir}) \]
\[ \text{dest}(x) = \text{src1}(x) \times \text{tmp1}(x) \]

Implementation 1: This can be implemented as two functions as follows:

Shift(tmp1, src2, FORWARD, dir);
Multiply(dest, src1, tmp1, REPLACE);

where FORWARD and dir give the sign of the shift and the direction, and REPLACE is an option indicating that dest is overwritten (allowing other options, like += later).

Implementation 2: If the Shift is changed to also return its result, this can be rewritten as:

Multiply(dest, src1, Shift(src2, FORWARD, dir), REPLACE);

In implementation 1 a lattice temporary is explicitly used, whereas in implementation 2, it appears that a temporary is implicitly used (returned by the Shift function). However, if the implementation of Shift were to merely create an object (structure) which records the need to shift in a particular direction and keeps a pointer to the unshifted data, then in implementation 2 Multiply could detect this object (structure) and the operations of multiply...
and shifting could be combined together into one operation. This eliminates the lattice temporary.

In general, Shift is an example of some generic pseudo-function that we don’t want to evaluate immediately but defer it to be performed with inside of the Multiply. The objects tmp1, src1, src2, and dest are some yet unspecified lattice objects. They could be implemented as pointers to a structure containing info on the (subgrid-)geometry, primitive type and data-layout.

Note that for parallel machines, Shift would encapsulate the need to send a sub-lattice face to a neighboring node and index local data offset by one in a particular dimension. Shift therefore hides the explicit communications calls.

**Example 2**

A pseudo-code example of the calculation of the average plaquette using data parallel operations is given. The example presumes name overloading; for C it would be necessary to use unique method names.

```c
/* Nd = number of dimensions
 * Nc = number of colors
 * Volume = Nd-volume
 * Forward is an enum. There is also a Backward.
 * The type specifiers below are suggestive, but not fleshed out yet */
Lattice_Gauge u[Nd], tmp_0, tmp_1;
Lattice_Real wplaq_tmp;
double tmp, w_plaq;
int mu, nu;

w_plaq = 0;
for(mu=1; mu < Nd; ++mu)
    for(nu=0; nu < mu; ++nu)
    {
        /* tmp_0 = u(x+mu,nu)*u_dag(x+nu,mu) */
        Multiply(tmp_0,
            Shift(u[nu],FORWARD,mu),
            Shift(Conjugate(u[mu]),FORWARD,nu),
            REPLACE);

        /* tmp_1 = tmp_0*u_dag(x,nu)=u(x+mu,nu)*u_dag(x+nu,mu)*u_dag(x,nu) */
        Multiply(tmp_1,tmp_0,Conjugate(u[nu]),REPLACE);

        /* tmp_0 = u(x,mu)*tmp_1
                = u(x,mu)*u(x+mu,nu)*u_dag(x+nu,mu)*u_dag(x,nu) */
        Multiply(tmp_0,u[mu],tmp_1,REPLACE);
    }
```

/* wplaq_tmp = tr(tmp_0)
   = tr(u(x,mu)*u(x+mu,nu)*u_dag(x+nu,mu)*u_dag(x,nu)) */
Trace(wplaq_tmp,tmp_0,REAL_PART);

/* w_plaq += sum(wplaq_tmp); */
Sum(w_plaq,wplaq_tmp,ADD);
}

w_plaq = 2.0*w_plaq/(float)(volume*Nd*(Nd-1)*Nc);

This example uses a few lattice temporaries. By combining some operations, these intermediates can be eliminated.

**Classification of operations**
The set of operations can be divided into 3 major categories:
1. N-aray operations (unary, binary, ...) on lattice objects producing a lattice object
2. Broadcasts (fills) of lattice objects from a lattice scalar (like a complex number)
3. Reductions (global sums) of lattice objects to lattice scalars

**Unary:**
A unary operation performs a transformation on a single source and stores the results in a destination. The operation may have additional arguments, such as the direction of a shift.

void Shift(Lattice_Field dest, Lattice_field source, enum sign, int direction);
void Copy(Lattice_Field dest, Lattice_Field source, enum option);
void Trace(double dest, Lattice_Field source, enum option);

**Binary:**
Binary operations take two sources and one destination:
void Multiply(Lattice_Field dest, Lattice_Field src1, Lattice_Field src2, enum option);
void Compare(Lattice_Bool dest, Lattice_Field src1, Lattice_Field src2, enum compare_func);

**Broadcasts:**
void Fill(Lattice_Field dest, float val);

**Reductions:**
void Sum(double dest, Lattice_Field source);
**N-aray operations**

While all operations can be expressed as combinations of unary and binary operations, implementing the library in this fashion will perform poorly because of the number of temporaries generated.

**Implementation of N-aray operations as macros**

The functions (whether C-based or C++-based) could be implemented in terms of generic n-aray macros. These in term take as an argument the site level operation to be performed on each primitive. The pseudo-functions *Shift* and *Conjugate* above would just set flags to be interpreted. A pseudo-code implementation of unary and binary is given for a single processor workstation.

Binary(Lattice_Field dest, Lattice_Field src1, Lattice_Field src2, BINARYMACRO func, enum OPTION);

where *source* can possibly be shifted or conjugated. An example for no shifting or conjugating.

Multiply(dest,src1,src2,option) expands to

Binary(dest,src1,src2,Multiply_site,option,NULL)

which in turn expands to:

```c
{int site;
 for(site=0; site < geometry.volume; ++site)
  { Multiply_site(dest[site], src1[site], src2[site], option) }
}
```

Multiply(dest,src1,Shift(src2,sign,dir),option) expands to

Binary(dest,src1,src2,Multiply_site,option,"COMM2,sign,dir")

(where COMM2 is a tag which remembers the need for the shift) which in turn expands to:

```c
{int site;
 for(site=0; site < geometry.volume; ++site)
  { Multiply_site(dest[site], src1[site], src2[geometry.offsets[sign][dir][site]], option) }
}
```
Implementation of N-array operations using lazy type constructors
Describe the tricks using lazy type constructors in C++

Data Types
Describe the type structure of primitives. E.g., in a fiber-bundle language, bundle is the base manifold (the lattice) and at site there is a fiber with some type. Mathematically, the primitive types we want at a site are either simple scalar values, or live within the product space of a vector space over color components with a vector space over spin components. So, we want generically

- Gauge fields: \( \text{Product}(\text{Matrix}(N_c), \text{Scalar}) \)
- Fermions: \( \text{Product}(\text{Vector}(N_c), \text{Vector}(N_s)) \)
- Scalars: \( \text{Scalar} \)
- Propagators: \( \text{Product}(\text{Matrix}(N_c), \text{Matrix}(N_s)) \)

where \( N_c \) indicates the dimension of the color vector space and \( N_s \) indicates the dimension of the spin vector space. For example, Scalars can multiply gauge fields or fermions. A gamma matrix can multiply a fermion, but not a gauge field. A gauge field can multiply a fermion, but a fermion cannot multiply another fermion unless there is a conjugation involved.

For example, a fermion can be laid out in memory as a \( N_c \times N_s \) matrix, but mathematically a fermion is not equivalent to a matrix.

Given these motivations, the following scalardata-types are supported:

- **ComplexF**: single precision complex real (total of 8 bytes.)
- **SU2F**: single precision SU(2) matrix (16 bytes)
- **GaugeF**: SU(N) single precision matrix (72 bytes for \( N=3 \).)
- **RealF**: scalar real, single precision (4 bytes)
- **IntF**: scalar integer (4 bytes)
- **BooleanF**: scalar Boolean (unfortunately, still 4 bytes???)
- **HalfFermionF**: two spin component SU(N) fundamental fermion, single precision (48 bytes \( N=3 \).)
- **DiracFermionF**: four spin component SU(N) Dirac fermion (96 bytes \( N=3 \))
- **StaggeredFermionF**: one spin component SU(N) staggered fermion (24 bytes \( N=3 \))
- **PropagatorF**: SU(N) 12 source color-spin and 12 sink color-spin components, single precision (576 bytes \( N=3 \)).

Single precision versions have a trailing \( F \) and double precision versions have a trailing \( D \).
There are also lattice variants of these names by prepending \texttt{Lattice}, so for example there is \texttt{LatticeGaugeF}.

\section*{Naming conventions}

All library routines begin with \texttt{QCDF} for single precision and \texttt{QCDD} for double precision. If relevant for the particular operation, the next letter is the \textit{N} in SU(\textit{N}). The type names above are shortened for compactness. Use uppercase for lattice objects and lowercase for scalar objects:

\begin{itemize}
  \item \texttt{R} \quad \texttt{scalar LatticeRealF}
  \item \texttt{I} \quad \texttt{scalar LatticeIntF}
  \item \texttt{G} \quad \texttt{LatticeGaugeF}
  \item \texttt{H} \quad \texttt{LatticeHalfFermionF}
  \item \texttt{D} \quad \texttt{LatticeDiracFermionF}
  \item \texttt{S} \quad \texttt{LatticeStaggeredFermionF}
  \item \texttt{P} \quad \texttt{LatticePropagatorF}
  \item \texttt{r} \quad \texttt{scalar RealF}
  \item \texttt{i} \quad \texttt{scalar IntF}
  \item \texttt{g} \quad \texttt{GaugeF}
  \item \texttt{h} \quad \texttt{HalfFermionF}
  \item \texttt{d} \quad \texttt{DiracFermionF}
  \item \texttt{s} \quad \texttt{StaggeredFermionF}
  \item \texttt{p} \quad \texttt{PropagatorF}
\end{itemize}

(should \texttt{DiracFermionF} be a \texttt{WilsonFermion} and then can have \texttt{DomainWallFermionF} ???)

The destination is the first argument and the sources come next.

There is no trailing character (needed) for the lattice variants without communications. The communication variants end are below:

\begin{itemize}
  \item \texttt{< >} \quad \text{(empty) lattice operation with no communications}
  \item \texttt{S1} \quad \text{Shift the first source argument}
  \item \texttt{S2} \quad \text{Shift second source argument}
\end{itemize}

The arguments for communications follow the source. The directions are integer based \([0\ldots N_d-1]\). The sign of the direction is an enum type \texttt{Sign} with values \texttt{FORWARD}, \texttt{BACKWARD}. 
Linear Algebra API

The following assumes that C is the implementation language, so does not use function overloading, and hides all intermediate objects (structures) as opaque types.

List some basic operations we will want.

**Norms and Inner Products**

**Norms**

void QCDF_norm2_T(RealF *r, const Type *a)

Computes \( r = |a|^2 \) where \( T \) is a shortened type name for the type lattice Type

LatticeRealF, LatticeComplexF, LatticeGaugeF, LatticeDiracF

void QCDF_norm2add_T(RealF *r, const Type *a)

LatticeRealF, LatticeComplexF, LatticeGaugeF, LatticeDiracF

Norms of integers are also useful.

void QCDF_norm2_I(IntF *r, const IntF *a)

Computes \( r = |a|^2 \)

**Inner Products**

void QCDF_dot_T(ComplexF *r, const Type *a)

Computes \( r = \sum_{n,x} \text{conj}(a_n(x))*b_n(x) \) where \( T \) is a shortened type name for the type Type

LatticeComplexF, LatticeGaugeF, LatticeDiracF

**Division**

Division is only supported on very basic fields. This is not something to encourage using since the inverse of a field can be used for multiplication which is typically more efficient

void QCDF_div_R (LatticeRealF *r, const LatticeRealF *a, const LatticeRealF *b)

\( r = a / b \).

**Addition and Subtraction**

Basic addition and subtraction of two fields into a third field
void QCDF_mult_T_op2(Type *r, const Type *a, const Type *b)

Binary operations where T is a shortened type name for the type Type
LatticeRealF, LatticeIntF, LatticeComplexF, LatticeGaugeF,
LatticeDiracFermionF, LatticeHalfFermion, LatticePropagatorF

and op2 is chosen from below where the types must be conforming, e.g. a gauge field
adds to a gauge field to produce a gauge field

add r = a + b
sub r = a − b

Copying

Copying onto a target with optional replacement or addition is a special case of Addition
above, but is more optimal. Note, with the C1 variant a communication is possible

void QCDF_copy_T_op2(Type *r, const Type *a)
void QCDF_copy_T_op2_S1(Type *r, const Type *a, int dir1, Sign sign1)

Binary operations where T is a shortened type name for the type Type
RealF, IntF, ComplexF, GaugeF, DiracFermionF, HalfFermion,
StaggeredFermionF, PropagatorF

and op2 is chosen from

rep r = a
neg r = -a
add r = r + a
sub r = r − a

Conjugation

Stand-alone conjugation

void QCDF_conj_T_op2(Type *r, const Type *a)

Binary operations where T is a shortened type name for the type Type
LatticeComplexF, LatticeGaugeF, LatticePropagatorF

and op2 is chosen from

rep r = conj(a)
neg r = -conj(a)
add r = r + conj(a)
sub r = r − conj(a)

Multiplication

Multiplication is a fundamental operation and quite important for speed. It should support
the many variants
Multiply operations where \(T1\), \(T2\), \(T3\) are shortened type names for the type \(Type1\), \(Type2\) and \(Type3\)

- \(LatticeGaugeF\), \(LatticeDiracFermionF\), \(LatticeHalfFermionF\), \(LatticePropagatorF\)

and \(op3\) is chosen from below where the types must be conforming, e.g. a gauge field multiplies a dirac field, or a \(\text{conj}(LatticeGaugeF)*LatticeDiracFermionF\) is required. Note, only \(\text{conj}(LatticeGaugeF)\) is supported

- \(\text{nnr}\ r = a*b\)
- \(\text{nnn}\ r = -a*b\)
- \(\text{ncr}\ r = a*\text{conj}(b)\)
- \(\text{ncn}\ r = -a*\text{conj}(b)\)
- \(\text{cnr}\ r = \text{conj}(a)*b\)
- \(\text{cnn}\ r = -\text{conj}(a)*b\)
- \(\text{ccr}\ r = \text{conj}(a)*\text{conj}(b)\)
- \(\text{ccn}\ r = -\text{conj}(a)*\text{conj}(b)\)
- \(\text{nna}\ r = r + a*b\)
- \(\text{nns}\ r = r - a*b\)
- \(\text{nca}\ r = r + a*\text{conj}(b)\)
- \(\text{ncs}\ r = r - a*\text{conj}(b)\)
- \(\text{cna}\ r = r + \text{conj}(a)*b\)
- \(\text{cna}\ r = r - \text{conj}(a)*b\)
- \(\text{cca}\ r = r + \text{conj}(a)*\text{conj}(b)\)
- \(\text{ccs}\ r = r - \text{conj}(a)*\text{conj}(b)\)

For the types \(LatticeRealF\) and \(LatticeIntF\), there is not a huge need for communication variants and certainly conjugation variants are not needed, so

Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)

There is also need for cross type multiplication of \(RealF\) and \(ComplexF\) against other types

Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
Пóйдóм мóн \(\text{nnr}\ r = a*b\)
Пóйдóм мóн \(\text{nnn}\ r = -a*b\)
and \( op3 \) is chosen from below where now the second source and the target types must be the same. Note, the case of \( \text{conj(RealF)} \) is not supported.

\[
\begin{align*}
nnr \quad r &= a \cdot b & nnn \quad r &= -a \cdot b \\
cnr \quad r &= \text{conj}(a) \cdot b & cnn \quad r &= -\text{conj}(a) \cdot b \\
nna \quad r &= r + a \cdot b & nns \quad r &= r - a \cdot b \\
cna \quad r &= r + \text{conj}(a) \cdot b & cna \quad r &= r - \text{conj}(a) \cdot b
\end{align*}
\]

**Filling**

- **void QCDF_zero_T (Type *r)**
  - Fill \( r \) with the 0 field (e.g., a matrix) where \( T \) is a shortened type name for the type \( \text{Type} \):
    - \( \text{LatticeRealF, LatticeIntF, LatticeComplexF, LatticeGaugeF, LatticeDiracF, LatticePropagatorF} \)

- **void QCDF_one_T (Type *r)**
  - Fill \( r \) with the identity field (e.g., a matrix) where \( T \) is a shortened type name for the type \( \text{Type} \):
    - \( \text{LatticeRealF, LatticeIntF, LatticeComplexF, LatticeGaugeF} \)

- **void QCDF_I_T (Type *r)**
  - Fill \( r \) with the imaginary \( I \) times the identity field (maybe a matrix) [useful in HMD] where \( T \) is a shortened type name for the type \( \text{Type} \):
    - \( \text{LatticeComplexF, LatticeGaugeF} \)

- **void QCDF_random_T (Type *r)**
  - Fill all elements with uniform \([0,1]\) random numbers where \( T \) is a shortened type name for the type \( \text{Type} \):
    - \( \text{LatticeRealF, LatticeComplexF, LatticeGaugeF, LatticeDiracFermionF} \)

- **void QCDF_gaussian_T (Type *r)**
  - Fill all elements with normal \( N(0,1) \) gaussian random numbers where \( T \) is a shortened type name for the type \( \text{Type} \):
    - \( \text{LatticeRealF, LatticeComplexF, LatticeGaugeF, LatticeDiracFermionF} \)

Note, need ways to set and read random number seeds, specify random numbers, etc.
Trace
void QCDF_realtrace_G(RealF *r, const GaugeF *a)
    Compute r = ReTr(a)

void QCDF_imagtrace_G(RealF *r, const GaugeF *a)
    Compute r = ImTr(a)

void QCDF_trace_G(ComplexF *r, const GaugeF *a)
    Compute complex part of trace r = Tr(a)

Spin Product
Multiply Dirac fermion by a product of gamma-matrices and store the result. \{idx\} is interpreted according to the following scheme:

The indexing scheme for the gamma matrices is as follows. There are four matrices, gamma_1, gamma_2, gamma_3 and gamma_4, and the basis of 16 matrices
\{d\} is indexed by an integer \(d\)
\(d = 8b_3 + 4b_2 + 2b_1 + b_0\).

Note, this construction is gamma matrix basis independent.

void QCDF_spprod_T_op(Type *r, const Type *a, int d)
    \(r = Gamma^d * a\) where Type is
        LatticeDiracFermionF, LatticePropagatorF
    where op is
        rep \(r = Gamma^d * a\) \quad neg \(r = -Gamma^d * a\)
        add \(r = r + Gamma^d * a\) \quad sub \(r = r - Gamma^d * a\)

Spin Projection and Reconstruction
The spin projection trick is useful for Wilson-like fermions, namely 4 spin component fermions

\textit{NOTE: WANT MANY MU VERSION AS WELL??}

\textbf{THIS SECTION NEEDS WORK}
void QCDF_spproj_HD(LatticeHalfFermionF *r, const LatticeDiracFermionF *a, int mu)
   r = spin_project(a, mu) -> (1+gamma_mu)*a

void QCDF_sprecon_HD(LatticeDiracFermionF *r, const LatticeHalfFermionF *a, int mu)
   r = spin_reconstruct(a, mu)

void QCDF_spproj_HGD_op(LatticeHalfFermionF *r, const LatticeGaugeF *u, const LatticeDiracFermionF *a, int mu)
void QCDF_spproj_HGD_op_S2(LatticeHalfFermionF *r, const LatticeGaugeF *u, const LatticeDiracFermionF *a, int mu)
void QCDF_spproj_HGD_op_SX(LatticeHalfFermionF *r, const LatticeGaugeF *u, const LatticeDiracFermionF *a, int mu)
   r = U*spin_project(a, mu)
   where op is
      mnr r = U*spin_project(a, mu)       mcnr r = conj(U)*spin_project(a, mu)

void QCDF_sprecon_DGH_op(LatticeDiracFermionF *r, const LatticeHalfFermionF *a, int mu)
void QCDF_sprecon_DGH_op_S1(LatticeDiracFermionF *r, const LatticeGaugeF *u, const LatticeHalfFermionF *a, int mu)
void QCDF_sprecon_DGH_op_S2(LatticeDiracFermionF *r, const LatticeGaugeF *u, const LatticeHalfFermionF *a, int mu)
void QCDF_sprecon_DGH_op_SX(LatticeDiracFermionF *r, const LatticeGaugeF *u, const LatticeHalfFermionF *a, int mu)
   r = U*spin_reconstruct(a, mu)
   where op is
      mnr r = spin_reconst(U*a, mu)       mcnr r = spin_reconst(conj(U)*a, mu)

Comparisons
Use comparisons to generate Boolean mask fields later used in copy under a mask

void QCDF_comp_T1T2_op(BooleanF *r, const Type *a, const Type *b)
   r = a op b   where T is a shortened name for Type
              RealF, IntF, LatticeRealF, LatticeIntF
   where op is
      lt       r = a < b       le       r = a <= b
      gt       r = a > b       ge       r = a >= b
      eq       r = a = b       ne       r = a != b
**Boolean Operations**

Boolean operations on Boolean values

```c
void QCDF_bool_B_op(BooleanF *r, const BooleanF *a, const BooleanF *b) {
    r = a op b
    where op is
        or    r = a | b    and    r = a & b
        xor   r = xor(a,b)
}
```

```c
void QCDF_bool_B_not(BooleanF *r, const BooleanF *a) {
    r = not(a)
}
```

**Copymask**

Rather than replicate all possible operations with versions for a bit mask field, just make a stand-alone copy under a mask. This is not optimal, but dramatically lower the explosion of possible variants

```c
void QCDF_copymask_T_op(TypeF *r, const BooleanF *a, const Type *b) {
    where T is a shortened name for Type and op is
    rep   r = (a)?b
    neg   r =-(a)?b
    add   r +=(a)?b
    sub   r -= (a)?b
}
```

**Data-Layout**

There seems to be a fixation on knowing exactly what data-layouts are in use.

**SZIN**

In fortran style ordering for indices (note, row major ordering):

- `site(checkerboarded x,y,z,t)`
- `checkerboard x = int(x/2)`
- `checkerboard index = (x+y+z+t)%2`

Lattice_GaugeField(real/imag, color row, color column, site, checkerboard, direction)
Lattice_DiracFermion(real/imag, color row, spin column, site, checkerboard)
Lattice_HalfFermion(real/imag, color row, spin column[2 components], site)

This is the generic ordering. For optimized Dirac operators, a common internal usage is Lattice_GaugeField(real/imag, color row, color column, direction, site, checkerboard)
Spin conventions

Same as MILC and CPS. A chiral basis with a negative sign convention on $\gamma_2$. 