Outstanding Issues on C++ QDP Interface

Robert Mawhinney and Kostya Petrov

Version 1.0, November 14, 2002

This document is a short note outlining differences in the two approaches to designing the C++ QDP API aka QDP++. This document represents solely a Columbia version, though it is derived from and evolved through conversations with Robert Edwards (RGE), whose comments we have tried to preserve. It is not a full specification, but rather an overview of an approach we find appealing.

Columbia is motivated to create a new API, not necessarily imitating/extending the old ones. An important consideration in this process are the people, particularly students, who will come to Lattice QCD during the next few years. In order for them to build on existing codes and knowledge, an easy to use API is vital. It should also be built using a language that they are likely to know. Thus neither similarity to the C interface, nor similarity to any existing languages like high-performance Fortran are vital constraints.

A design path to achieve this is to have syntax of the API reflect closely the mathematics and common usage of analytical calculations. An underlying data-parallel character for members of a high level object suits the basic structure of our problems and our machines. However, users need to easily be able to access subsets of the data, perform transfers using many directions simultaneously while still having the API achieve reasonable efficiency. Through C++, we propose users manipulate objects of the general form

A[x][mu]

where x and mu are generalized indices, which are class objects, appropriate to a vector field This makes possible] statements like

*A[even][mu]=B[even][mu]+C[even][mu]*D[even][mu]*

where *even* represents a subset of all even sites and mu is a vector index.

Two implementations of such an expression would likely be done. The first, low-performance implementation, could evaluate the expression on a site-by-site basis. The second would use C++ containers to have the expression resolve to a single function call for evaluation. Since this call would have information about the entire, aggregate operation, optimizations for a specific platform could be done inside the call. This will require some run-time calculation. Such run-time calculation seems inevitable since, for simulataneous multi-dimensional transfers, optimal blocks and strides may need to be determined for each transfer direction. These cannot in general be determined at compile time, since the local lattice volume is only determined when the calculation is started on a hardware platform of a particular geometry. This will require effort and only a limited set of such operations may be optimized at any time. However, all the information is available for future optimization.

In the broadest terms, the proposed API would use C++ for gathering information, via class objects, from the user to present to a function. This seems an attractive method for freeing users from complicated argument lists, while still making all needed information available to the implementer.

1. Basic objects

   Columbia advocates using as many advantages of object-oriented design (OO) as possible. Thus, following the directive "everything that looks like an object should be an object" the basic objects of the API are the actual multi-dimensional fields living inside a container class (STL-like). It will, when and if we need, behave like an array but with more flexibility. **RGE** advocates standard array constructs.

   Declarations:

   // LatticeGauge class contains a matrix valued vector field, as in
   // QDP/C
   LatticeGauge U;

   // integer to use as an index (like an array in C)
   int i;

   // Get a single matrix, using lexical ordering give by i
   ColorMatrix u = U[i];

   // Object containing a single lattice site
   SingleSite x(1,2,3,4)

   // Object containing a single vector direction
   SingleDir mu(3)

   // Get matrix from site (1,2,3,4) with vector index mu
   ColorMatrix u = U[x][mu];

2. Referring to Basic objects - Shifts and Subsets.

   Columbia uses postfix operators.

   LatticeGauge A,B,C,D;
   LatticeFermion X,Y;

   MultiSite even; // Object describing all even sites.
   MultiDir mu=3dSpace; // Object containing vector directions for 3 dimensions

   // Both of these could be allowed. Implicit sum over mu
   *A[even][mu]=B+C*D;*
   *A[even][mu]=B[even][mu]+C[even][mu]*D[even][mu];*

   // Common subsets (*even*, ...) predefined
   *Y[even]=A[even][mu] * X[even+mu]; Y[even]=A*A[even+mu];* // identical to the previous

   See below why we may want subsets (practically) on the RHS. We think that general logic assumes that once you have a field living on some manifold, we should be able to refer to its part living on some part of the manifold.

   RGE allows subset only on the left-hand side.
   *A(even) = B * C; // result of mult on only even sites*
   Double f = sum(B+C); // sum over all sites
   Double g = sum(B+C,even);// sum over even sites. Here B+C not lvalue

   Columbia advocates multidimensional shifts to be a part of API, **RGE** defers to later API. There is assumed factor of two penalty on performance for not using such on QCDOC. We will come again to this later.

3. **Columbia** advocates using of composite types for observables of double type which are global (overhead is small). Then we can write

   Double p; //note capital D, can of course use less ambiguous name, e.g. scalar

   <double>

   ColorMatrix u; LatticeGauge B,C,D;

u=B+C; // will be a sum over all lattice

u=B[even]; //sum over all lattice sites

u=B[even]+C[even]; // same but first we add fields;

p=A[even]+B[even]; *//same as above, but as long as p is <scalar>double, not a matrix, we take trace in the end.*

Subset plane;

p=A[plane]; //summation over a plane or whatever object you might like;

Possible ambiguities arise (courtesy of RGE):

*Double p,q,r,s; p=Tr(A);q=Tr(B);r=p*q; s=Tr(A)*Tr(B);*

here s is not equal to r. User has to know that reduction happens at the equation sign, and know what he is doing. More thought is needed here to for simplicity for users.

Columbia advocates also implied summation over objects of type MultiDir if they appear only on right hand side (as we do in mathematics). SingleDir types will not be summed over. A third type is likely needed to avoid terms where two implicit sums over vector indices should not include the common term (like in the plaquette). The implied summation allows the implementation to do communication in many directions concurrently. This is a vital feature of our problem and our machines.

RGE considers this awkward and ambiguous.

4. Composite shifts

   Columbia finds them natural. Columbia overall strategy relies on data/view representation of the lattice objects. Thus if we change the view once, we can do it twice and so on. Sequence is not ambiguous. And we do write $U_{x+\mu+\nu,\nu}$ in mathematics. We do not see a lot of overhead on it. Run-time overhead is incurred here in any approach that uses more than a single pre-defined shift, since the various sites from which the shifted objects originate must be determined.

   *A[even] = B*C[dir1]*D[dir1+dir2]*E[dir1+dir2+dir3];*

   RGE recommends no composite shifts, considers overhead high and does not see real gain.

*A[even] = B*shift(C*shift(D*shift(E,dir1),dir2),dir3);*

Columbia The shift above of RGE involves run-time overhead, since the sites from which E is to be shifted can only be found by determining *even+dir1+dir2+dir3*.

RGE argues that only compile-time optimizations are sensible. This would rule out any either of the composite shifts above.

Columbia formally allows nested map/subset statements but would not advocate/document use of such. These are objects of the form

5. Examples Wilson-Dirac and plaquette.

   RGE has explicit directions (see 1.)

   *chi[even] = reconMinus(u[0] * lshift(spprojMinus(psi, 0), 0), 0)*

   *+ reconPlus(rshift(conj(u[0]) * spprojPlus(psi, 0), 0), 0)*

   *+ reconMinus(u[1] * lshift(spprojMinus(psi, 1), 1), 1)*

   *+ reconPlus(rshift(conj(u[1]) * spprojPlus(psi, 1), 1), 1)*

   *+reconMinus(u[2] * lshift(spprojMinus(psi, 2), 2), 2)*

   *+ reconPlus(rshift(conj(u[2]) * spprojPlus(psi, 2), 2), 2)*

   *+ reconMinus(u[3] * lshift(spprojMinus(psi, 3), 3), 3)*

   *+ reconPlus(rshift(conj(u[3]) * spprojPlus(psi, 3), 3), 3);* //modified by Columbia to be 4D

   RGE proposes introducing another function *reduce* and write

   *chi[even] = reduce(reconMinus(u * lshift(spprojMinus(psi)))*

   *+ reconPlus(rshift(conj(u) * spprojPlus(psi)))); //how is reduce supposed to know how many fields are there in U? and over which of them are we going to sum? (Columbia)*

   Columbia

   Dimension mu=4dSpace;

   *chi[even] = reconMinus(u[even][mu] * spprojMinus(psi[even+mu]))*

   *+ reconPlus(conj(u[even-mu][mu]) * spprojPlus(psi[even-mu]));*

   Here index μ appears only on one side so will be summed over. For plaquette subset X=ALL;

*p=tr_d(U[X][mu]*U[X+mu][nu]*dag(U[X+nu][mu])*dag(U[X][nu])); or*

*p=U[X][mu]*U[X+mu][nu]*dag(U[X+nu][mu])*dag(U[X][nu]);*

Columbia does not push on any level of implicitness. We leave it for committee to decide whether we want to be cute or bullet-proof. We also leave user the ability to check if the API does what he wants to do, using following syntax:

```
SingleSite x;
SingleDir mu,nu;

for(x.start;x++) {
for(mu.start;mu++) {
for(nu.start;nu++) {

if (mu != mu)

p+=U[x][mu]*U[x+mu][nu]*dag(U[x+nu][mu])*dag(U[x][nu];
} } }
```

We view this as a major strength of this approach. A user, or implementor, can check any aggregate API expression involving subsets against a site-by-site evaluation of the same expression. The syntax is almost identical, providing a powerful debugging tool. Users can also implement complicated, but not time critical, expressions in this manner.