February 21, 2003

QDP++ Primer

Draft 1.0

This is a first attempt at a QDP++ primer; we welcome feedback on its utility, and the type of material you would like us to include as it is extended.

A QDP++ Primer

1 Introduction

QDP++ is a implementation of the Level-2 data-parallel QCD-API in C++. Broadly, the aim is to provide a suite of data-parallel routines and data types appropriate to lattice gauge theory that enable efficient writing of C++ lattice QCD codes. QDP++ codes will be portable, with details of the low-level communication routine transparent to the user, and benefit from the SciDAC optimisation developments on the national facilities available to the lattice community.

For the user, the application code is written in C++, with QDP++ included as a library, whose routines are called in the same way as for any other library routines. Thus the aim of the primer is not to describe the inner workings of QDP++, but how to use it for writing application codes. We will assume that the machine you are using already has the QDP++ library installed, or that you install it according to the instructions in the Appendix.

Why C++? C++ encourages an object-orientated programming model, and the construction of *classes*; a corollary is the concept of *operator overloading*. Classes are user-defined data types, but in addition their definition generally includes a set of allowed operations on them. Operator overloading means that the precise definition of a function or operator depends on the *class* of the arguments. Thus, if `A` and `B` are matrices, the multiplication operation might mean something different from the case where `A` and `B` are real numbers.

The primer is written for users who are familiar with C, but are novices to C++, a category which the authors satisfy admirably. The layout of the document is as follows. We begin with how to write source code and a Makefile that allows programs to employ the QDP++ library. We then describe how to initialise QDP++, and how the geometry and size of the problem is specified.

We discuss the lattice-specific data types of QDP++, and proceed to describe its data-parallel features: data-parallel operations, communications and global reductions. As the first example, we consider the ubiquitous "Hello, world" program, illustrating the initialisation of QDP++. We then proceed to the evaluation of the plaquette, introducing such lattice matrix

algebra, nearest-neighbour communication, and global sums. To allow you to extend these examples to "real" applications, we provide a few small lattices in NERSC format.

This document is a *Primer*, rather that an *reference manual*; the latter can be found at `http://www.jlab.org/~edwards/qcdapi/qdp`+/doc/html/+. The aim initially is to discover the language features of QDP++, and the details of linking in the QMP library, and compiling for a parallel machine, will appear in a later version of the primer.

2 The nitty-gritty

2.1 Including QDP++

From the user's perspective, QDP++ is a library, and therefore the application has to know what the library contains by means of including a header file, `qdp.h`,. and where to look for the library. To satisfy the first requirement, every file using QDP++ needs two lines at the top:

```
#include "qdp.h"

using NAMESPACE QDP;
```

The second line, which may be unfamiliar to C programmers, tells the application to use specific functions and routines defined within QDP. Of course, we now have to tell the compiler where to find both the include files and the library. We assume that QDP++ is installed in HOME/qcd/src/qdp++, in which case these are simply accomplished by adding a couple of lines to `Makefile`:

```
LDFLAGS=-L${HOME}/qcd/src/qdp++/lib -lqdp

CXXFLAGS=-I${HOME}/qcd/src/qdp++/include/
```

The first line tells the compiler where to look for the library, whilst the second line tells the compiler where to look for the include files. Note that CXX is the built-in for the C++ compiler.

Finally, for our stand-alone `hello_world` program, we specify a target as follows:

```
hello_world: hello_world.o
	$(CXX) $(CXXFLAGS) -o $@ $< ${LDFLAGS}
```

2.2 Hello, world

Our simple hello_world code is written out below:

```
int main(int argc, char *argv[])
{

  QDP_initialize(&argc, &argv);

  const int foo[] = {4,4,4,8};
  multi1d<int> nrow(Nd);
  nrow = foo;  // Use only Nd elements
  Layout::setLattSize(nrow);
  Layout::create();

  cout << "Hello, world" << endl;

  {
    int lattice_volume;
    lattice_volume = Layout::vol();

    cout << "Volume is " << lattice_volume << endl;
  }


  QDP_finalize();

  return 0;
}
```

We will now go through this program line-by-line. QDP_initialize is the first line in any QDP++ code; it puts the machine into a known state, and in particular in the case of a parallel machine would initialise the communication hardware.

The next few lines are associated with setting up the geometry of our problem, and the routines to specifying the querying the geometry are contained withing the namespace Layout. Certain elements of the geometry are fixed in a given installation of QDP++, specifically Nc (3), the number of colors, Nd (4), the number of space-time dimensions, and Ns (4), the num-

ber of spinor components; the default values for the QDP++ installation are specified in the brackets. Code targeted at differnt values of these parameters requires a different installation of QDP++. Certain parameters then have to be set either at compile time, or at run time, notably the dimensions of our lattice. For this example, we will specify the dimensions within the code, but a practical code would probably specify them at run-time; we will employ a $4^3 \times 8$ lattice. The space-time dimensions are contained in the one-dimensional array `nrow`; we will return to the construction `multi1d` in the next section. We pass the dimensions to the layout through the command `Layout::setLattSize(nrow)`. Once the required values have been specified, we initialise the geometry through `Layout::create()`. This evaluates the volume of the lattice, sets up the neighbour tables for commuications, creates the subsets associated with checkerboarding.

One of the enhancements of C++ with respect to C is the greatly improved I/O support; we take advantage of these in our simple "hello, world" print statement to "standard out". Now that we have gone to the trouble of setting up our lattice geometry, we can access certain information about the lattice that was computed during `Layout::create()`, such as the lattice volume. This is done with the *Accessor* function `Layout::vol()`; note the ability of C to specify variables anywhere within a program unit, with the memory for the variables only defined within the *scope* specified by `{` and `}`.

The final element in any QDP++ code is `QDP`finalize()+; this ensures an orderly exit for QDP, and on a parallel machine would be called by all nodes.

3 QDP++ Basics

3.1 Data Types

C++ has a brief set of predefined data types, or *classes* in C++ parlance, such as `float`, `int`, `char`, `bool` etc.. QDP++ includes additional "user-defined", or rather predefined, classes that aid the construction of lattice QCD codes. All QDP-defined classes begin with an upper-case letter.

The basic classes are of two forms:

Scalar classes:

- `Real` A floating-point number, which can be defined to be either a *float* or a *double*

- `Complex` A complex number, constructed from two `Real`
- `ColorMatrix` A general SU(`Nc`) matrix
- `Fermion` A complex object with `Nc` colour indices and `Ns` spinor indices

We can create *arrays*, such an array `nsize` that defines the lattice dimensions

```
multi1d<int> nsize(Nd)
```

where here `Nd` is the number of lattice dimensions.

Lattice classes:

These are objects that exist on every site of the lattice, and typically are site-wide version of the scalar classes. They all begin with `Lattice`, and some examples corresponding to the scalar classes are

- `LatticeReal`
- `LatticeComplex`
- `LatticeColorMatrix`
- `LatticeFermion`

As for scalar quantities, we can create an array of lattice quantities

```
multi1d<LatticeColorMatrix> u(Nd).
```

which would correspond, say, to all four directions of a gauge field.

3.2 QDP++ Operations

The key to QDP++, inherited from C++, is the idea of *operator overloading*, the idea that the definition of an operator can depend on the *types* of the argument, i.e. be object-oriented. As a simple example, consider multiplication. For simple real numbers a, b and c, then multiplication has the obvious meaning

$$c = a \times b.$$

However, in the case of matrices A, B and C, we would like multiplication to be implemented as a matrix operation

$$C_{ij} = \sum_k A_{ik} B_{kj}.$$

We can extend this concept to multiplication of a vector by a matrix, or even to the multiplication of objects with indices in different spaces. As a simple example, consider an SU(3) gauge field U, a gamma matrix Γ and some Dirac spinors $\psi_{\alpha,i}$ and $(\phi, \psi)_{\alpha,i}$ where the Greek and Latin indices refer to spin and colour respectively. Then we would want to define what we mean by multiplication of a spinor according to the matrix

$$\psi_{\alpha i} = U_{ij} \Gamma_{\alpha\beta} \phi_{\beta j}$$

In QDP++, this is expression can be simply written as

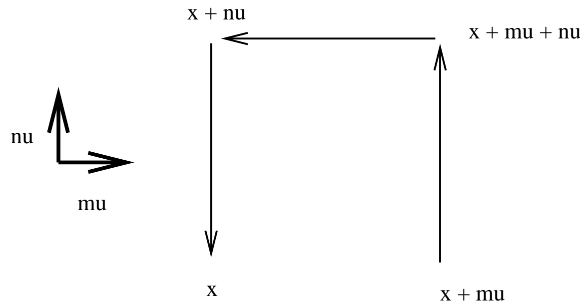
```
LatticeFermion psi
LatticeFermion phi
LatticeColorMatrix u

psi = u*Gamma(1)*phi
```

Note that the operation of the gamma matrices are defined in qdp++ by the action on spinors, and in most cases you need not worry about the precise gamma matrix convention. Furthermore, because the quantities are *lattice variables*, this is a data parallel operation performed on all the sites. All the usual binary operations are provided, as detailed in the QDP++ manual - but, of course, not divides in the case of matrices. Similarly, a set of *unary* operations such as `sin`, `cos` etc which can act both on *scalar reals* and on *lattice reals*. Certain unitary operators, for example `adj` and `trace`, have the obvious interpretation of a *Hermitian Conjugate* and *trace* respectively when applied to matrices, but the operations only apply to the "internal" indices as opposed to the space-time indices. Thus `trace(u)` when applied to a `LatticeColorMatrix` returns a `LatticeComplex`. Finally, there are the global-reduction operations, such as `sum`, that reduce a Lattice object to a scalar.

3.3 Shifts and communication

We are all familiar with the evaluation of *staples* in lattice calculations, that enter into the calculation of plaquettes, link smearing etc.. Let us consider

the case of the evaluation of the upper plaquettes



In mathematical language, we want

$$S(x) = U_\nu(x+\mu) * U_\mu(x+nu)^\dagger * U_{nu}(x)^\dagger$$

and of course we want to evaluate this at every site - or a subset, say on the even sites. In QDP++, this has the compact notation

```
LatticeColorMatrix S
multi1d<LatticeColorMatrix> u(Nd)
int mu,nu
...
...
...

S = u[nu]*adj(shift(u[mu], FORWARD, nu))*adj(u[nu]);
```

`shift` is identical to that of Fortran 90, and the "FORWARD" refers to the direction of the source relative to the destination. Note that the above operation involves no temporaries, the bugbear of C++ programming.

3.4 Subsets

Efficient algorithms in both the pure-gauge and fermion sectors make use of *Chequerboards* (to use the British spelling!), in which operators are performed in parallel on a *subset* of the lattice sites. The simplest is a simple red/black checkerboard, in which the sites are either even ($rb = 0$) or odd ($rb = 1$) according to the value of

$$rb = \mathrm{mod}(x + y + z + t, 2).$$

Some algorithms, e.g. those for improved gauge actions, use more involved checkerboarding with more than two checkerboards. Thus we need a flexible means of restricting operations to a subset of sites, i.e. those on a particular checkerboard. *N.B.* QDP++ specifies that "Lattice" types are defined on *all* the sites; "subsets", the QCD-API terminology we will be using, restrict the arithmetical operators to a subset of sites, but do not alter the allocation of memory.

Because we wish subsets to be as general as possible, we will make no assumptions about whether a given checkerboarding has subsets of equal size, or indeed whether the sum of subsets contains all the lattice sites. This has the important consequence that, in an arithmetic operator restricted to a subset, it is only necessary, and indeed only makes sense, to specify the subset of the *target*. Thus for the case of the calculation of a generalised Wilson line

$$A(x) = U(x)U(x + \mu)U(x + \mu + \nu)U(x + \mu + \nu + \rho)$$

restricted to the case where x is *even* this is written

```
cb = 0;
a[rb[cb]] = u*shift(u*shift(u*shift(u,FORWARD, rho),FORWARD, nu),
                       FORWARD, mu);
```

The index `rb[cb]` specfies the subset `cb` of the checkerboarding `rb`, and the subsets of the sources are completely specified by the shift operations.

4 Simple examples

We will now put the above to use by some simple examples, beginning with the measurement of the mean plaquette, the true *hello, world* of the lattice world.

4.1 Mean Plaquette

The top-level program is `t_mesplq.cc`; the first part of the program is taken from `hello_world`, apart from the additional header file, but then it proceeds as follows:

```
#include "qdp.h"
#include "tutorial.h"

using namespace QDP;

int main(int argc, char *argv[])
{
  ....
  ....

  multi1d<LatticeColorMatrix> u(Nd);  // Define a lattice gauge field

  Double w_plaq, s_plaq, t_plaq, link;   // Measurement variables

  for(int m=0; m < u.size(); ++m)
    gaussian(u[m]); // Fill gauge field with Gaussians

  MesPlq(u, w_plaq, s_plaq, t_plaq, link);  // Measure the plaquette

  cerr << "w_plaq = " << w_plaq << endl;
  cerr << "link = " << link << endl;  // Print some results

  ...
  ...
}
```

`multi1d<LatticeColorMatrix> u(Nd)` creates the gauge field, a one-dimensional array of length `Nd` and of type `LatticeColorMatrix`. Note that, in contrast to C, we do not have to `malloc` memory for the gauge field, since there is a constructor that allocates the memory, and a destructor that releases the memory outside the scope of the variable; memory management is thus greatly simplified. For this simple example, we will just fill the gauge field with Gaussian random numbers, performed in `gaussian(u[mu])`.

The real work is done in `mesplq`:

```
#include "qdp.h"

using namespace QDP;

// Primitive way to indicate the time direction
static int tDir() {return Nd-1;}

void MesPlq(const multi1d<LatticeColorMatrix>& u,
            Double& w_plaq, Double& s_plaq,
            Double& t_plaq, Double& link)
{
  s_plaq = t_plaq = w_plaq = link = 0.0;

  for(int mu=1; mu < Nd; ++mu)
  {
    for(int nu=0; nu < mu; ++nu)
    {

      /* tmp_0 = u(x+mu,nu)*u_dag(x+nu,mu) */
      LatticeColorMatrix tmp_0 =
         shift(u[nu],FORWARD,mu) * adj(shift(u[mu],FORWARD,nu));

      /* tmp_1 = tmp_0*u_dag(x,nu)=u(x+mu,nu)*u_dag(x+nu,mu)*u_dag(x,nu) */
      LatticeColorMatrix tmp_1 = tmp_0 * adj(u[nu]);

      /* tmp =
       *    sum(tr(u(x,mu)*tmp_1=
       *    u(x,mu)*u(x+mu,nu)*u_dag(x+nu,mu)*u_dag(x,nu)))
       */
```

```
      Double tmp = sum(real(trace(u[mu]*tmp_1))); // Finally, take the trace

      w_plaq += tmp;

      if (mu == tDir() || nu == tDir())
        t_plaq += tmp;
      else
        s_plaq += tmp;
    }
  }

  // Normalize
  w_plaq *= 2.0 / double(Layout::vol()*Nd*(Nd-1)*Nc);

  if (Nd > 2)
     s_plaq *= 2.0 / double(Layout::vol()*(Nd-1)*(Nd-2)*Nc);

  t_plaq /= double(Layout::vol()*(Nd-1)*Nc);

  // Compute the average link
  for(int mu=0; mu < Nd; ++mu)
    link += sum(real(trace(u[mu])));

  link /= double(Layout::vol()*Nd*Nc);
}
```

The only argument passed to this routine is the gauge field; all the subprgrams already know all the information, such as the lattice sizes etc. contained in `Layout`. The returned arguments, `w_plaq`, `s_plaq`, `t_plaq` and `link`, are the mean plaquette, the space-space and space-time plaquettes, and the mean link variable respectively; the time direction is crudely specified at the top of the routine.

We discussed in the previous section how shifts are handled in QDP++, and here we can see it in action, using the temporaries `tmp_0` and `tmp_1`. We then see the unary operations `trace` (note that this does no include trace over sites!) and `real` that act on the lattice variables, yielding an object of type `LatticeReal` which is the argument to the global reduction operation

`sum`.

We can write this in the much more compact form

```
Double tmp =
  sum(real(trace(u[mu]*shift(u[nu],FORWARD,mu)*
   adj(shift(u[mu],FORWARD,nu))*adj(u[nu]))));
```

This has advantages both in memory through obviating the need for the extra lattice temporaries, and in performance through avoiding the need to write those temporaries to memory.

4.2 Further examples

The QDP++ release includes a suite of examples and test codes, in the `examples` subdirectory. The top-level codes are all named `t_<prog>.cc`. To see how to explore some of the features we have discussed, here is a sampling:

- `dslashm_w.cc` is the Wilson Dirac operator. It illustrates both the use of *subsets*, and some of the more complex routines optimised for Wilson fermions, such as those associated with spin decomposition and reconstruction.
- `mesons_w.cc` computes the diagonal meson correlators for Wilson fermions. It illustrates the creation of subsets associated with time-sliced sums.

Appendix: Getting Hold of QDP++

The QDP++ web page is at `http://www.jlab.org/~edwards/qdp`, accessible from the SciDAC web page `http://www.lqcd.org/scidac`. There are instructions there for acquiring the source, but the simplest way is via CVS which you should install on your favorite local linux/cygwin machine. For the moment, this method requires that you have an account on one of the JLAB central machines. To get the latest release of the code, login to your local workstation, and proceed as follows:

```
export CVS_RSH=ssh
cvs -d :ext:cvs.jlab.org:/group/lattice/cvsroot checkout qdp++
```

Note that JLAB only allows access with ssh2. This will create a directory on your machine called `qdp++`, containing the basic code and header files for QDP++ with several further subdirectories. Perhaps most important is `examples`, containing the various test programs that, say, compute the mean plaquettes etc.. Note that this tutorial assumes that QDP++ is created in `$HOME/qcd/src/qdp++`.

To compile QDP++, you need a sufficiently up-to-date version of `gcc`, say version 3.1 or later. To check that everything is installed correctly, go into the top-level qdp++ directory and type `make all`. This will create the library `libqdp.a`.

Documentation

A *reference manual* can be found at
`http://www.jlab.org/~edwards/qcdapi/qdp++/doc/html/`.

This is generated from the code using `doxygen`; it contains the basic definitions of the classes, and the prototypes for the QDP++ functions. However, it is for the moment strictly a reference manual rather than a user guide.