QDP++ Talk

Bálint Joó,
Scientific Computing Group
Jefferson Lab

QDP++ Basics

- QDP++ Is the Foundation of Chroma
- It provides
  - way to write the maths of lattice QCD without looping over site/spin/color indices (expressions)
  - Custom memory allocation possibilities
  - I/O facilities (XML and Binary)
- You can do Lattice QCD just in QDP++ without Chroma
  - See: Lectures from the 2007 INT Lattice Summer School
  - but you'd need to write a whole lot of infrastructure that comes for free with Chroma
- In terms of parallel computing, QDP++ is an implementation
  - of the data parallel expression model in C++
  - is domain specific (it is specialized to QCD)

QDP Templated Types

- QDP++ captures the tensor index structure of lattice QCD types

	Lattice	Spin	Colour	Reality	BaseType
Real	Scalar	Scalar	Scalar	Real	REAL
LatticeColorMatrix	Lattice	Scalar	Matrix(Nc,Nc)	Complex	REAL
LatticePropagator	Lattice	Matrix(Ns,Ns)	Matrix(Nc,Nc)	Complex	REAL
LatticeFermionF	Lattice	Vector(Ns)	Vector(Nc)	Complex	REAL32
DComplex	Scalar	Scalar	Scalar	Complex	REAL64

- To do this we use C++ templated types

```
typedef OScalar < PScalar     < PScalar<       RScalar <REAL>     >     > > Real;
typedef OLattice< PScalar     < PColorMatrix< RComplex<REAL>, Nc>     > > LatticeColorMatrix;
typedef OLattice< PSpinMatrix< PColorMatrix< RComplex<REAL>, Nc>, Ns> > LatticePropagator;
```

- Heavy lifting: Portable Expression Template Engine(PETE)

QDP++ and Expressions

- The idea is to try and capture the maths ...
- ... while hiding details of the machine, parallelism etc

```
LatticeFermion x,y,z;
Real a = Real(1);
gaussian(x);
gaussian(y);
z = a*x + y;
int mu, nu;
multi1d<LatticeColorMatrix> u(Nd);
Double re_plaq =  sum( real( trace(  u[mu]
                     * shift(u[nu],FORWARD,mu)
                     * adj( shift(u[mu],FORWARD,nu) )
                     * adj(u[nu])
                                              ) ) );
```

Lattice Wide Types: e.g. for fermions

Fill fermion with gaussian random numbers

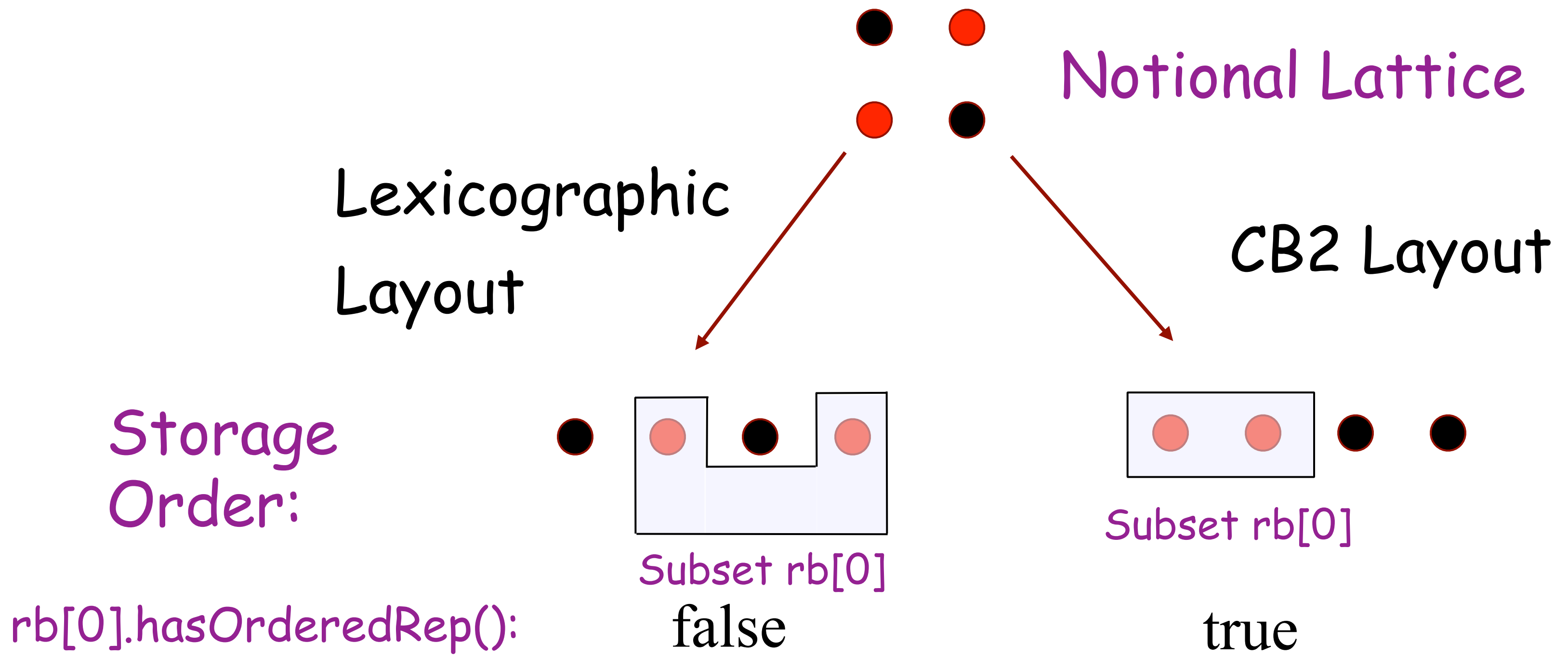
BLAS 1: AXPY like operation all indices hidden

multi1d<T> : 1D array of T (explicitly indexed)

shift() = nearest neighbour comms
this one gets $u_\nu(x+\mu)$

Subsets and Layouts

- Subset: Object that identifies a subset of sites
- Can be predefined: eg rb is “red-black” colouring
- Can be contiguous or not (s.hasOrderedRep() == true or not)
- Layout is an ordering of sites in memory (compile time choice)
- Same subset may be contiguous in one layout and not in another



Using Subsets

- In QDP++ expressions, subset index is always on the target

```
bar[ rb[1] ] = foo; // Copy foo's rb[1] subset
```

- Users can define new sets
- Layout is chosen at configure time, and fixed at compile time.
  - default is CB2 (2 color checkerboard, each checkerboard contiguous)
- The geometry of the layout is set at run-time on entry to QDP++

```
multi1d<int> nrow(4); nrow[0]=nrow[1]=nrow
[2]=4; nrow[3]=8;
Layout::setLattSize(nrow);
Layout::create();
```

QDP++ and XML

- XML is a great way to read parameters
  - turns out, its not such a good way to write lots of data
- QDP++ supports XML reading and simple XML writing
- Reading is done by reading XML documents using XPath
  - XML parsing etc is done by libxml2 - a dependent library

root node →

From root: /foo/bar/fred

from /bar: ./fred

```xml
<?xml version="1.0" encoding="UTF-8"?>
<foo>
   <bar>
      <fred>6</fred>
      <jim>7 8 9</jim>
   </bar>
</foo>
```

Reading XML from QDP++

```
XMLReader r(“filename”);

Double y;
multi1d<Int> int_array;
multi1d<Complex> cmp_array;

try {
 read(r, “/foo/cmp_array”, cmp_array);

 XMLReader new_r(r, “/foo/bar”);

 read(new_r, “./int_array”, int_array);
 read(new_r, “./double”, y);
}
catch( const std::string& e) {
 QDPIO::cerr << “Caught exception: ”
              << e <<endl;
 QDP_abort(1);
}
```

QDP++ error “stream”

```
<?xml version=”1.0”
      encoding=”UTF-8”?>
<foo>
<cmp_array>
  <elem>
    <re>1</re>
    <im>-2.0</im>
  </elem>
  <elem>
    <re>2</re>
   <im>3</im>
  </elem>
</cmp_array>
 <bar>
   <int_array>2 3 4 5</int_array>
  <double>1.0e-7</double>
 </bar>
</foo>
```

Array of complex-es

Array element

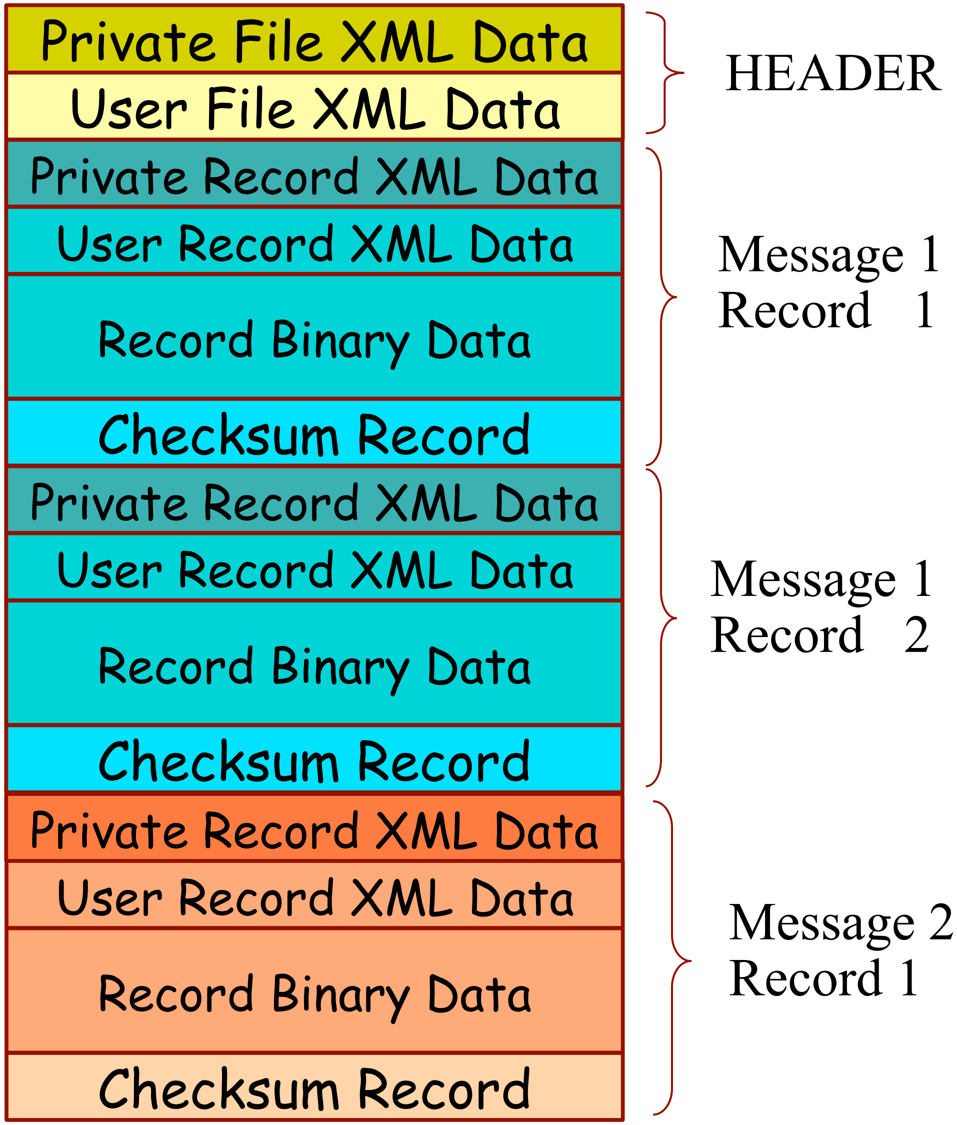
Writing XML

```
// Write to file
XMLFileWriter foo(“./out.xml”);
push(foo, “rootTag”);
int x=5;
Real y=Real(2.0e-7);
write(foo, “xTag”, x);
write(foo, “yTag”, y);
pop(foo);
```

```
<?xml version=”1.0”?>
<rootTag>
  <xTag>5</xTag>
  <yTag>2.0e-7</yTag>
</rootTag>
```

```
// Write to Buffer
XMLBufferWriter foo_buf;
push(foo_buf, “rootTag”);
int x = 5;
Real y = Real(2.0e-7);
write(foo_buf, “xTag”,  x);
write(foo_buf, “yTag”, y);
pop(foo_buf);
QDPIO::cout << “Buffer contains” << foo_buf.str()
            << endl;
```

QIO and LIME Files



- QIO works with record oriented LIME files
- LIME files made up of messages
- messages are composed of
  - File XML records
  - Record XML records
  - Record Binary data
- SciDAC mandates checksum records
- ILDG mandates certain records

QDP++ interface to QIO

- Write with QDPFileWriter
- Must supply user file and user record XML as XMLBufferWriter-s

- Read with QDPFileReader
- User File XML and User Record XML returned in XML Readers
- Checksum/ILDG details checked internally to QIO

File XML

```
LatticeFermion my_lattice_fermion;
XMLBufferWriter file_metadata;
push(file_metadata, "file_metadata");
write(file_metadata, "annotation", "File Info");
pop(file_metadata);

QDPFileWriter out(file_metadata,
                  file_name,
                  QDPIO_SINGLEFILE,
                  QDPIO_SERIAL);

XMLBufferWriter record_metadata;
push(record_metadata, "record_metadata");
write(record_metadata, "annotation", "Rec Info");
pop(record_metadata);
out.write( record_metadata, my_lattice_fermion);
out.close();
```

QIO Write Mode Flags

Record XML

```
XMLReader file_in_xml;
XMLReader record_in_xml;
QDPFileReader in(file_in_xml,
                 file_name,
                 QDPIO_SERIAL);

LatticeFermion my_lattice_fermion;
in.read(record_in_xml, my_lattice_fermion);
in.close();
```

Custom Memory Allocation

- Occasionally need to allocate/free memory explicitly – e.g. to provide memory to external library.
- Memory may need custom attributes (eg fast/communicable etc)
- Memory may need to be suitably aligned.
- May want to monitor memory usage

Allocate memory from desired pool if possible, with alignment suitable to pool

```
pointer=QDP::Allocator::theQDPAllocator::Instance().allocate( size,
                                                        QDP::Allocator::FAST);

QDP::Allocator::theQDPAllocator::Instance()::free(pointer);
```

Namespace | Get reference to allocator | MemoryPoolHint (attribute)

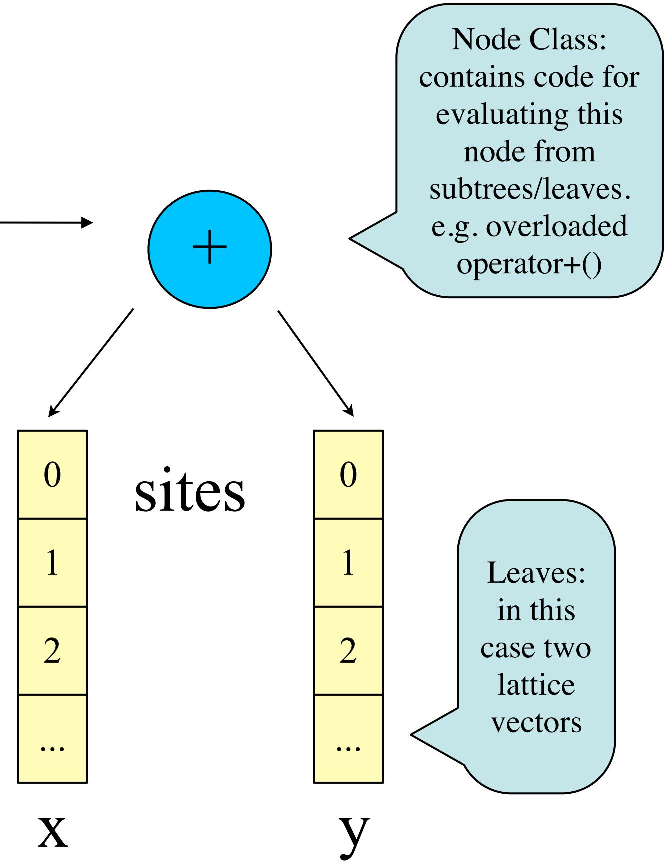
How do expressions work?

- Expression Template Technique
  - using Portable Expression Template Engine a.k.a PETE
  - Construct Expression Template Class representing the expression
  - Use C++ operator overloading:

Overload operator+()

x + y ; → QDPExpr<RHS, C>

C= container for return type for expression



How does it work?

- Operators =, += etc trigger evaluation

Overload operator=()

z = x + y ;

dst=z
Op=OpAssign
rhs is QDPExpr from op+()

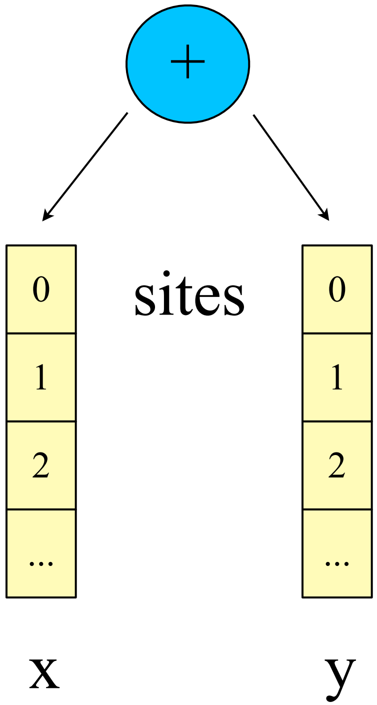
```
template<class T, class T1, class Op, class RHS>
void evaluate(OLattice<T>& dst, const Op& op,
               QDPExpr<RHS,OLattice<T1> >& rhs)
{
     forall sites i do:
        op( dst.elem(i),
            ForEach(rhs, EvalLeaf1(i), OpCombine()));
}
```

ForEach:
recursive tree traversal

EvalLeaf1 functor:
selects which site
to work with

OpCombine functor:
calls code in
node to evaluate its
subtrees

QDPExpr<RHS, C>



Parallelism

- "forall sites i do" can be implemented as you like:
  - for non-threaded architectures just a regular for loop

```
for(int i=all.begin(); i<= all.end(); i++) { ... };
```

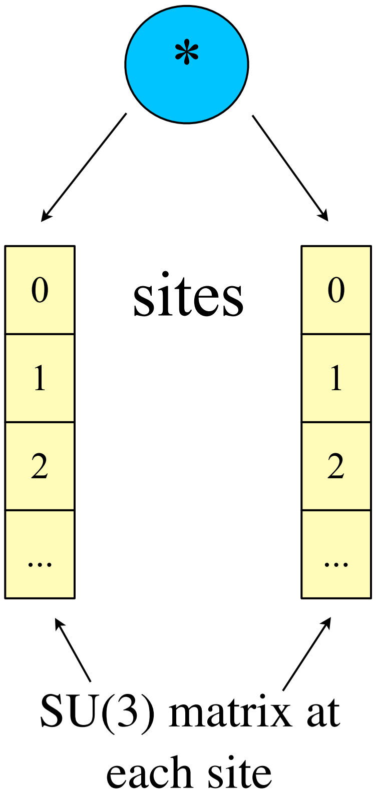
  - for threaded architectures one can employ e.g. OpenMP:

```
#pragma omp parallel for
  for(int i=all.begin(); i < all.end(); i++) { ... };
```

- Complication: shift() operations, and message passing
  - Need to evaluate sub-expressions of shift() operation
  - Need to carry out shift() operation before finishing rest of expression

Optimization/Specialization

- native QDP++ expression templates may not necessarily be the most performant
- Consider SU(3) matrix multiply:




```
template<>
inline BinaryReturn<SU3Mat, SU3Mat, OpMultiply>::Type_t
operator*(const SU3Mat& l, const SU3Mat& r)
{
   BinaryReturn<SU3Mat, SU3Mat, OpMultiply>::Type_t ret;

   // Code for SU(3) xSU(3) multiply goes here
   // ...
   // Naively use complex types etc

   return ret;
}
```

Return Type (just SU3Mat in disguise, using a type trait)

returning SU3Mat on stack

Naive code may not be optimal. Sees only data for this site (inhibit prefetching)

Optimization/Specialization

- Two ways to optimize:
  - Way 1: optimize the site specific code in the nodes
    - e.g. SU(3) multiplies: replace code with SSE optimized code

```
template<>
inline BinaryReturn<PMatrix<RComplexFloat,3,PColorMatrix>,
  PMatrix<RComplexFloat,3,PColorMatrix>, OpMultiply>::Type_t
operator*(const PMatrix<RComplexFloat,3,PColorMatrix>& l,
          const PMatrix<RComplexFloat,3,PColorMatrix>& r)
{
  BinaryReturn<PMatrix<RComplexFloat,3,PColorMatrix>,
    PMatrix<RComplexFloat,3,PColorMatrix>, OpMultiply>::Type_t  d;

  // Unwrap pointers for leaves
  su3_matrixf* lm = (su3_matrixf *) &(l.elem(0,0).real());
  su3_matrixf* rm = (su3_matrixf *) &(r.elem(0,0).real());
  su3_matrixf* dm = (su3_matrixf *) &(d.elem(0,0).real());

  intrin_sse_mult_su3_nn(lm,rm,dm); // Call optimized routine

  return d;
}
```

Specialization: Matches op* only for SU(3) matrices at the leaves (no subtrees etc)

Optimization/Specialization

- Two ways to optimize: Way 2
  - specialize the whole evaluate() for this expression
    - remember: RHS in QDPExpr(RHS) is a type you can match

RHS type: mat. mult.

expression return type (C)

```
//   u = u1 * u2;
template<>
void evaluate(OLattice< SU3Mat >& d,
              const OpAssign& op,
              const QDPExpr<
                      BinaryNode<OpMultiply,
                          Reference<QDPType< SU3Mat, OLattice< SU3Mat > > >,
                          Reference<QDPType< SU3Mat, OLattice< SU3Mat > > >
                      >,
                      OLattice< SU3Mat >
                  >& rhs,
              const Subset& s)
{

      // Code here to loop over sites in subset s and
      // carry out matrix multiply. Can be optimized to the extreme
      // NB: Must feed parallelism (e.g. OpenMP pragmas) in here by hand...
}
```

Optimization

- One last optimization remains, which is much harder:
  - Currently expression blocks like this:

```
y = a*x + b;
z = q*x + y
norm2(z);
```

  - perform 3 site loops when one would do
  - this is wastes precious memory bandwidth
  - QDP++ cannot see through multiple expressions at this time
- Two solutions:
  - Work around: in performance critical code break out of QDP++
  - Heavy Handed: add some kind of compiler support for QDP++

Stopping Point

- Covered Basic QDP++ features
  - expressions, XML, I/O
  - the mechanics of the expression templates
  - how to optimize QDP++ with specializations
  - discussed some limitations (e.g. no expression fusion)

- Possible Continuations
  - Chroma
  - QDP++ and GPUs/future plans, Chroma and QUDA
  - Deeper dive into templates (traits etc) and generic programming