

QCD-API

Level 3

Dirac Operators, CG Routines etc.
C, C++, etc.

(Organized by high level codes etc.)

Level 2

Data Parallel QCD Lattice Operations
(overlapping Algebra and Messaging)

$A = \text{SHIFT}(B, \mu) * C$; Global sums, etc

Lattice Wide Lin Alg
(No Communication)

e.g. $A = B * C$

Lattice Wide Data Movement
(Blocking Communication)

e.g. $A_{temp} = \text{SHIFT}(A, \mu)$

I/O
(XML)

Serialization/
Marshalling

Level 1

QLA: Linear Algebra API

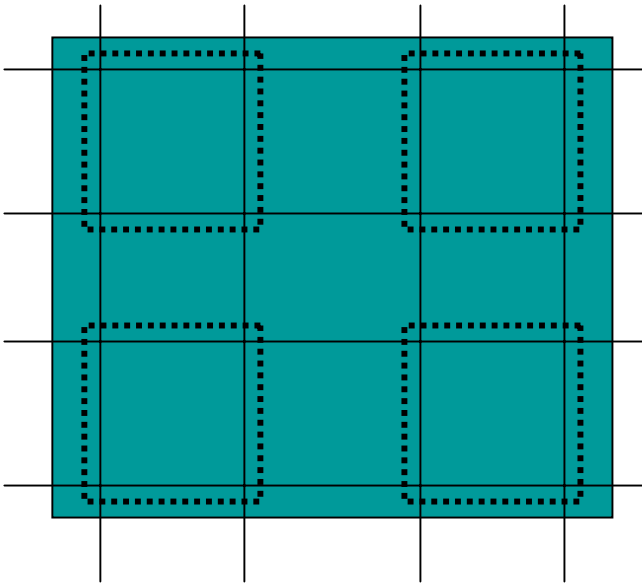
SU(3), gamma algebra etc.

QMP: Message Passing API

Communications

Data-Parallel Programming Model

Data layout over processors



- Basic uniform operations across lattice:
 $C(x) = A(x)*B(x)$
- Distribute problem grid across a machine grid
- Want API to hide subgrid layout and communications (bundling subgrid faces) between nodes
- Data-parallel - basic idea something like Fortran 90 with more complex types
- Implement via a programming interface (API), e.g. a C interface

Why C++?

- **Many OOP's benefits**
 - Strong type checking
 - Can create expressions!
 - Can eliminate lattice temporaries (Expression Templates)
 - Straightforwardly automate memory allocation/destruction
- **Maintenance and performance**
 - Existence of compilers meeting standards - GNU g++
 - Templates important for optimized code
 - Simple to replace crucial sections (instantiations) with optimized code using templates
 - g++ generating decent assembly!
- **User acceptance**
 - Language in common use

QCD Data Types

- Fields have various types (indices):

Color: $U^{ij}(x)$, Spin: $\Gamma_{\alpha\beta}$, $\psi_{\alpha}^i(x)$, $Q_{\alpha\beta}^{ij}(x)$

- Lattice Fields or scalar (same on all sites)
- Index type (at a site)
 - Gauge fields: Product(Matrix(Nc), Scalar)
 - Fermions: Product(Vector(Nc), Vector(Ns))
 - Scalars: Scalar
 - Propagators: Product(Matrix(Nc), Matrix(Ns))
- Support compatible operations on types:

$$U^{ij}(x) * \Gamma_{\alpha\beta} * \psi_{\alpha}^i(x) \longrightarrow \text{Matrix(color)} * \text{Matrix(spin)} * \text{Vector(color, spin)}$$

Example

```
// Declarations and construction - no default initialization
```

```
LatticeGauge u, tmp0, tmp1;
```

```
// Initialization
```

```
gaussian(u);           // fills all real/imag elems with gaussian
```

```
zero(tmp0);           // zero works on vectors, matrices, etc...
```

```
tmp1 = tmp2 = 1.0;    // legal, but not legal for LatticeFermion
```

```
// Expressions - no lattice temporaries. One big site expression
```

```
//  $tmp_{\{i,k\}}(x) = u_{\{i,j\}}(x) tmp1^{dag}_{\{j,k\}}(x) + tmp2_{\{i,k\}}(x)$  forall x
```

```
tmp0 = u * conj(tmp1) + tmp2;           // can merge * and conj
```

```
// Shifts - no lattice temporaries or internal lattice copies
```

```
tmp1 = u * shift(conj(tmp1)*tmp0, FORWARD, mu);
```

```
// Alternative subset syntax?
```

```
tmp0 = forall(Even_subset, conj(shift(tmp0, BACKWARD, mu)));
```

Operations

Unary operators: *operate on one source and return a result*

Infix: operator-, +, !, ~

Functions: cos, sin, exp, ..., conj, trace, spinProject,

Void functions: random, gaussian

Binary operators: *operate on two sources and return a result*

Math: operator*, +, -, /, &, <<, ...

Comparisons: operator<, <=, ==, &&, ... [returns boolean]

Assignments: operator=, +=, *=, ...

Broadcasts: *broadcast throughout lattice*

Assignments: operator=, +=, ... [e.g., LatticeComplex a = 3]

Reductions: *reduce through the lattice*

Sums: sum(a), norm2(a), innerproduct(a,b)

Multi-sums: sumMulti(a,subset), ... [returns array of results]

Features

Expressions: *full expression construction*

```
LatticeFermion psi;
```

```
LatticeFermion chi = 17 * Gamma(3) * psi;
```

Shifts: *general permutation map (shift) constructor*

```
MakeShift(int some_func(x0,x1,x2,x3)) my_shift [not yet implemented]
```

```
shift(a, sign, dir) [nearest neighbor shift]
```

Subsets: *narrow which sites participate in operation*

```
Subset(int some_func(x0,x1,x2,x3)) my_subset [constructor]
```

```
Double tmp = sum(a*b, my_subset) [only sites in my_subset]
```

```
// Subset expression syntax?
```

```
tmp0 += forall(Even_subset, conj(shift(tmp0, BACKWARD, mu)));
```

Templates for Declarations

Expect to have data driven (and not method driven) class types:

```
class LatticeDiracFermionF {  
private:  
    DiracFermionF sites[layout.Volume()];  
};
```

With templates:

```
template<class T> class Lattice {  
private:  
    T sites[layout.Volume()];  
};  
typedef Lattice<DiracFermionF> LatticeDiracFermionF;
```

Type composition:

The fiber (site) types can themselves be made from templates
(implementation choice)

Data Types

Fields have various types (indices):

Color: $U^{ij}(x)$, Spin: $\Gamma_{\alpha\beta}$, $\psi_{\alpha}^i(x)$, $Q_{\alpha\beta}^{ij}(x)$

Tensor product of indices forms type

- Gauge fields: Product(Matrix(Nc), Scalar, Complex)
- Fermions: Product(Vector(Nc), Vector(Ns), Complex)
- Scalars: Product(Scalar, Scalar, Scalar)
- Propagators: Product(Matrix(Nc), Matrix(Ns), Complex)

Some types

Real, Double, Complex, Integer, Boolean, ColorMatrix
LatticeReal, LatticeFermion, LatticePropagator

Expression Templates - Transmogrification

// Lattice operation

```
A = -B + 2 * C;
```

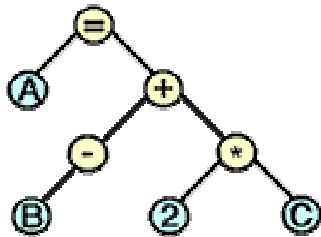
// Lattice loop

```
for (i = ... ; ... ; ...) {  
    A[i] = -B[i] + 2 * C[i];  
}
```

- Naïve ops involve lattice temps - inefficient

- Express RHS in a parse tree - PETE

Parse tree



// Generated code

```
BinaryNode<OpAssign,  
    ArrayA,  
    BinaryNode<OpAdd,  
        UnaryNode<OpUnaryMinus, ArrayB>,  
        BinaryNode<OpMultiply, Scalar<int>, ArrayC>>
```

// Evaluate expression at each site

```
void evaluate(const Expression& expr) {  
    for(int site=0; site < vol; ++site)  
        eval(expr[site]);    // At each site ops in tree executed on leafs  
}
```

Current Development Efforts

- QDP++ development
 - Level-2 C/C++ API not yet finalized by SciDAC – documentation underway
 - Nearly ``complete'' single node version – tests underway
 - Shortly start the parallel version
 - Incorporate QLA library routines when agreed upon by SciDAC
- Stand-alone demonstration/prototyping suite
 - Some demos of physics routines in the QDP++ test directory
 - Prototyping effort underway
 - Expect to produce a user's guide
- Software ports
 - Porting some critical portions of SZIN code with merging of MIT and Maryland code bases
 - Need more software efforts!

I/O and Data Formats

- Currently using Namelist (from Fortran) data format. Supports nested groups and resources (tags) within groups.
- Switching now to new nested header format for configurations, propagators, input files, etc.
- Will propose to use XML for ascii formatted headers
 - These headers can be used for a user extensible replica(ted) catalog of *locations* of configurations and propagators.
 - Search user data using SQL to find locations.
- Use XML input and output data files
 - Parameter input for program execution
 - Output for cooked data can be sent over net to receiver
 - Want SQL-like data base for storing data and retrieving
- Problems:
 - Long computation time for typical SQL commits or rollbacks
 - SQL data types too primitive – want multi-D arrays, complex numbers.