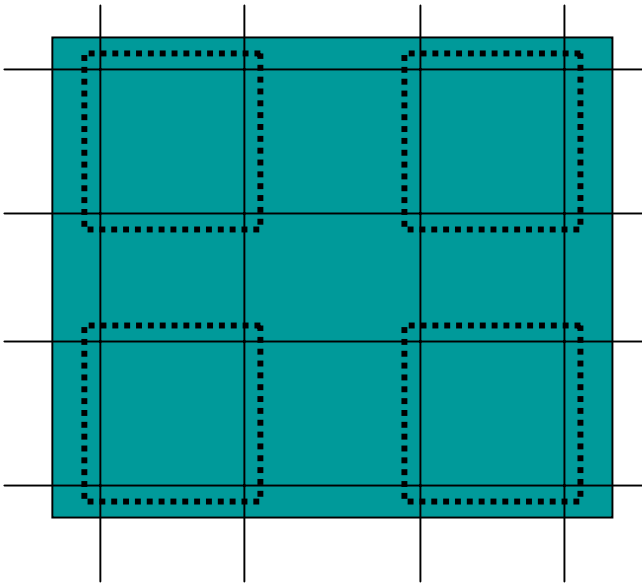


Data-Parallel Programming Model

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

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:

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

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

I/O
(XML)

Serialization/
Marshalling

Level 1

QLA: Linear Algebra API

SU(3), gamma algebra etc.

QMP: Message Passing API

Communications

Level 2 Functionality

Object types/declarations:

LatticeGaugeF, ReadF, ComplexF, ...

Unary operations: operate on one source and return (or into) a target

```
void Shift(LatticeGaugeF target, LatticeGaugeF source, int sign, int direction, Subset s);  
void Copy(LatticeFermionF dest, LatticeFermionF source, Subset s);  
void Trace(LatticeComplexF dest, LatticeGaugeF source, Subset s);
```

Binary operations: operate on two sources and return into a target

```
void Multiply(LatticePropagatorF dest, LatticePropagatorF src1, LatticeGaugeF src2, Subset s);  
void Gt(LatticeBooleanF dest, LatticeRealF src1, LatticeRealF src2, Subset s);
```

Broadcasts: broadcast throughout lattice

```
void Fill(LatticeGaugeF dest, RealF val, Subset s);
```

Reductions: reduce through the lattice

```
void Norm2(RealD dest, LatticeFermionF source, Subset s);
```

C Interface for Level 2

Binary:

Multiplication has many varieties

```
void QDP_F_T3_eqop_T1op1_mult_T2op2 (restrict Type3 *r, Type1 *a, Type2 *b, const Subset s)
```

Multiply operations where T1, T2, T3 are shortened type names for the types Type1, Type2 and Type3

LatticeGaugeF, LatticeDiracFermionF, LatticeHalfFermionF, LatticePropagatorF, ComplexF, LatticeComplexF, RealF, LatticeRealF, ...

and eqop,op1,op2 considered as a whole imply operations like

eq,, $r = a*b$

eq,,A $r = a*\text{conj}(b)$

eq,A, $r = \text{conj}(a)*b$

eq,A,A $r = \text{conj}(a)*\text{conj}(b)$

peq,, $r = r + a*b$

peq,,A $r = r + a*\text{conj}(b)$

peq,A, $r = r + \text{conj}(a)*b$

peq,A,A $r = r + \text{conj}(a)*\text{conj}(b)$

eqm,, $r = -a*b$

eqm,,A $r = -a*\text{conj}(b)$

eqm,A, $r = -\text{conj}(a)*b$

eqm,A,A $r = -\text{conj}(a)*\text{conj}(b)$

meq,, $r = r - a*b$

meq,,A $r = r - a*\text{conj}(b)$

meq,A, $r = r - \text{conj}(a)*b$

meq,A,A $r = r - \text{conj}(a)*\text{conj}(b)$

QDP++

- Why C++ ?

- Many OOP's benefits:

- Simplify naming with operator overloading
- Strong type checking - hard to call wrong routine
- Can create expressions!
- Closures straightforward - return types easily
- Can straightforwardly automate memory allocation/destruction
- Good support tools (e.g., *doxygen* - documentation generator)

- Maintenance:

- Generate large numbers of variant routines via templates
- Contrary to popular opinion, there are compilers that meet standards. Minimum level is GNU g++. Expect to use full 3.0 standards
- Simple to replace crucial sections (instantiations) with optimized code
- Initial tests - g++ generating decent assembly!

- User acceptance:

- Language in common use

Flavors

Operation syntax comes in two variants:

Global functional form (optimization straightforward):

```
void Multiply_replace(Type3& dest, const Type1& src1, const Type2& src2);
```

Operator form (optimizations require delayed evaluation):

```
Type3 operator*(const Type1& src1, const Type2& src2);  
void operator=(Type3& dest, const Type3& src1);
```

Were the types Type1, Type2 and Type3 are the names again
LatticeGaugeF, LatticeDiracFermionF, ...

Declarations:

Objects declared and allocated across the lattice (not initialized)

```
LatticeGaugeF a, b;
```

Templates for Operations

Without templates, multiple overloaded declarations required:

```
void Multiply_op3(LatticeComplexF& dest, const RealF& src1, const LatticeComplexF& src2);  
void Multiply_op3(LatticeDiracFermionF& dest, const LatticeGaugeF& src1, const  
LatticeDiracFermionF& src2);
```

Can merge similar operations with templates:

```
template<class T3, class T1, class T2> void Multiply_op3(T3& dest, const T1& src1, const  
T2& src2);
```

Optimizations easily provided via specializations:

```
template<class LatticeDiracFermionF, class LatticeGaugeF, class LatticeDiracFermionF>  
void Multiply_op3(LatticeDiracFermionF& dest, const LatticeGaugeF& src1, const  
LatticeDiracFermionF& src2) { /* Call your favorite QLA or QDP routine here */ }
```

Templates for Declarations

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

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

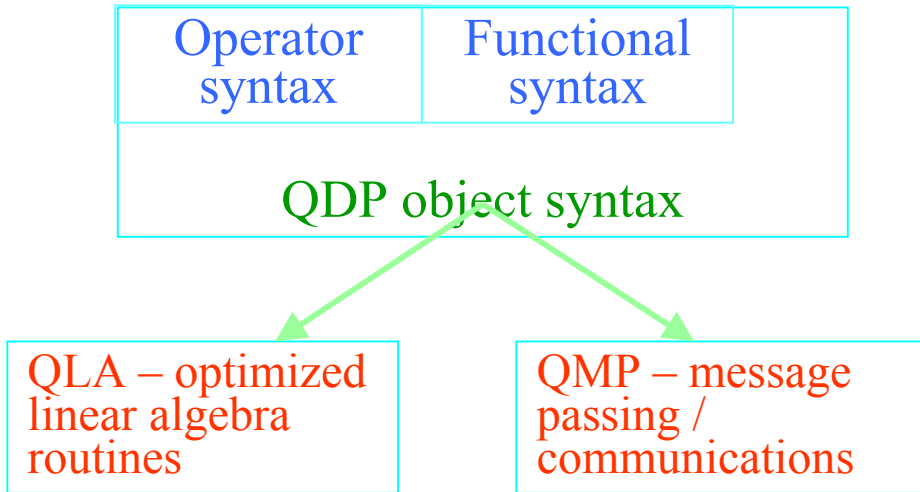
With templates:

```
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)

Implementation Example



- Use a container class to hold info on object
- Easily separate functional and operator syntactic forms
- Closures implemented at this container level
- Underlying object class does real work
- No additional overhead – use inlining

// Functional form

```
inline void Multiply_replace(QDP<Type3>& dest, const QDP<Type1>& src1, const QDP<Type2>& src2)
{ dest.obj().mult_rep(src1.obj(), src2.obj()); }
```

// Operator form without closure optimizations

```
inline QDP<Type3> operator*(const QDP<Type1>& src1, const QDP<Type2>& src2)
{ QDP<Type3> tmp;
  tmp.obj().mult_rep(src1.obj(), src2.obj());
  return tmp; }
```

Example

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

```
LatticeGaugeF u, tmp0, tmp1;
```

```
// Initialization
```

```
Gaussian(u);
```

```
Zero(tmp0);
```

```
tmp1 = 1.0;
```

```
// Two equivalent examples
```

```
Multiply_replace(tmp0, u, tmp1);
```

```
tmp0 = u * tmp1;
```

```
// Three equivalent examples
```

```
Multiply_conj2_shift2_replace(tmp1, u, tmp0, FORWARD, mu);
```

```
Multiply_replace(tmp1, u, Conj(Shift(tmp0,FORWARD,mu)));
```

```
tmp1 = u * Conj(Shift(tmp0,FORWARD,mu));
```

```
{
```

```
    // Change default subset as a side-effect of a context declaration
```

```
    Context foo(Even_subset);
```

```
    tmp1 = u * Conj(Shift(tmp0,FORWARD,mu)); // Only on even subset
```

```
    // Context popped on destruction
```

```
}
```

SZIN

- SZIN and the Art of Software Maintenance
 - M4 preprocessed object-oriented data-parallel programming interface
 - Base code is C
 - High level code implemented over architectural dependent level 2 layer
 - Supports scalar nodes, multi-process SMP, multi-threaded SMP, vector, CM-2, CM-5, clusters of SMP's
 - The Level 2 routines are generated at M4 time into a repository
 - Supports overlapping communications/computations
 - No exposed Level 1 layer
 - Can/does use (transparent to user) any external Level 1 or 2 routines
 - Optimized Level 3 routines (Dirac operator) available
 - Code publicly available by tar and CVS with many contributors – no instances of separate code camps
- Data parallel interface
 - SZIN is really the data-parallel interface and not the high level code
 - Some projects have only used the low level code and library
 - Could easily produce a standalone C implementation of the Level 2 API

Problems with SZIN

Why change current system? Cons:

– *Maintenance:*

- M4 has no scope. Preprocessor run over a file - no knowledge of C syntax or scope – bizarre errors can result
- M4 has no programming support for holding type info. Must invent it
- Interface problem – awkward to hook onto level 1 like optimizations

– *Extensibility:*

- Problems with C – awkward to write generic algorithms, e.g. CG for different Dirac operators (like Clover requiring `sigma.F` but not Wilson), or algorithms on lattices or just single sites.
- No real language support for closures
- C very limited on type checking
- No automated memory allocation/free-ing. No simple way to write expressions.

– *User acceptance:*

- M4 not familiar (and limited) to most users

Current Development Efforts

- QDP++ development
 - Level-2 C/C++ API not yet finalized by SciDAC – documentation underway
 - Currently testing the basic object model code needed for the single node version
 - Shortly start the parallel version – uses object model code
 - Will incorporate QLA library routines once agreed upon by SciDAC
- Stand-alone demonstration/prototyping suite
 - Some test routines and drivers for physics routines live in the QDP++ test directory
 - David/Richard/Andrew/others writing some simple stand-alone routines (like pure gauge) as a prototyping effort
 - Expect to produce a user's guide
- SZIN port
 - Porting existing SZIN code with merging of MIT and Maryland code bases
 - The M4 architectural support part is completely removed and only uses QDP
 - Will become another high level code base using implementation dependent QDP

Hardware Acquisitions

- FY02
 - **NOW:** 128 single processor P4, 2Ghz, Myrinet
 - Expect \sim 130 Gflops
 - **Late summer:** probably 256 single processor P4, 2Ghz, 3-dimensional Gigabit-ethernet
 - Expect \gtrsim 200 Gflops
- FY03
 - **Late summer:** probably 256 single processor P4, 2.4Ghz, 3-dimensional Gigabit-ethernet
 - Expect \gtrsim 240 Gflops