An Introduction to Computational Lattice QCD

Bálint Joó,

Scientific Computing Group

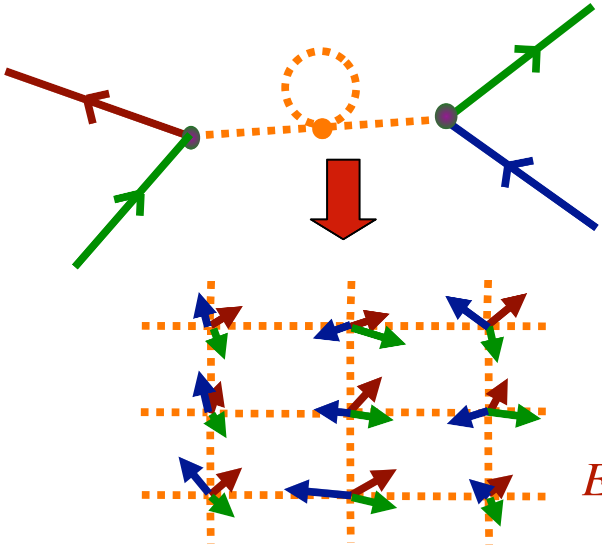
Jefferson Lab

Contents

- Introductory Lecture
- A lecture on Solvers (we’ll write a solver)
- A lecture on ‘optimization’
- A lecture on Hybrid Monte Carlo (we’ll write an HMC)
- A lecture on data analysis
- There will also be exercises

Lattice QCD

- Lattice QCD is the only known model independent, non-perturbative technique for carrying out QCD calculations.
  - Move to Euclidean Space, Replace space-time with lattice
  - Move from Lie Algebra su(3) to group SU(3) for gluons
  - Gluons live on links (Wilson Lines) as SU(3) matrices
  - Quarks live on sites as 3-vectors.
  - Produce Lattice Versions of the Action



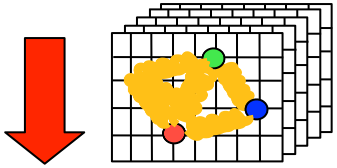
$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}A \; \mathcal{D}\bar{\psi} \; \mathcal{D}\psi \; \mathcal{O} \; e^{-S(A,\bar{\psi},\psi)}$$

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\text{all links}} dU \prod_{\text{all sites}} d[\bar{\psi},\psi] \; \mathcal{O} \; e^{-S(U,\bar{\psi},\psi)}$$

Evaluate Path Integral Using Markov Chain Monte Carlo Method

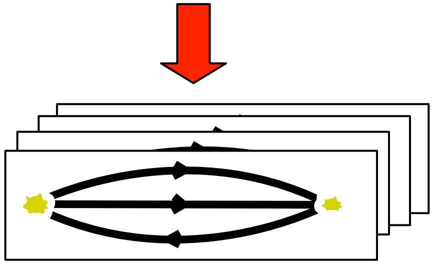
Large Scale LQCD Simulations Today

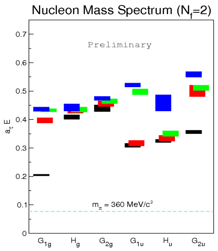
- Stage 1: Generate Configurations
  - snapshots of QCD vacuum
  - configurations generated in sequence
  - capability computing needed for large lattices and light quarks



- Stage 2a: Compute quark propagators
  - task parallelizable (per configuration)
  - capacity workload (but can also use capability h/w)

- Stage 2b: Contract propagators into Correlation Functions
  - determines the physics you'll see
  - complicated multi-index tensor contractions





- Stage 3: Extract Physics
  - on workstations, small cluster partitions

Monte Carlo Method

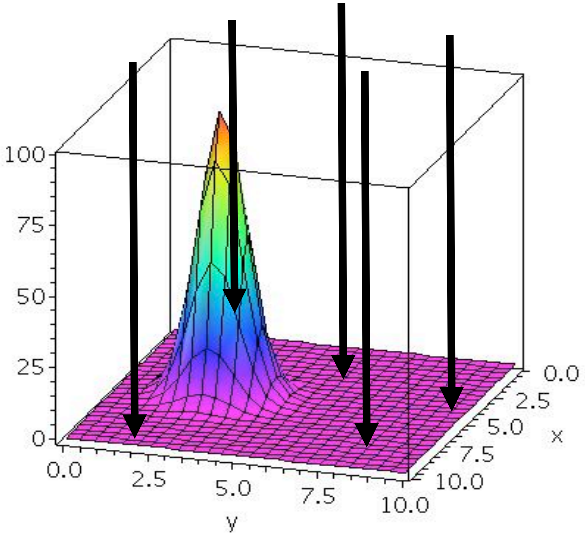
Evaluating the Path Integral:

- There are 4V links. V~32^3x256 → 4V = **~ 33M links**
- Direct evaluation unfeasible. Turn to Monte Carlo methods

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\text{all links}} dU_i \; \mathcal{O} \;\; e^{-S(U)} \longrightarrow \bar{O} = \frac{1}{Z} \sum_{\text{configuration}} \mathcal{O}(U) \; P(U)$$

- Basic Monte Carlo Recipe
  - Generate some configurations U
  - Evaluate Observable on each one
  - Form the estimator.

Problem with uniform random sampling: most configurations have P(U) ~ 0



Importance Sampling

- Pick U, with probability P(U) if possible
- Integral reduces to straight average, errors decrease with statistics

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\text{all links}} dU_i \; \mathcal{O} \; e^{-S(U)} \longrightarrow \bar{O} = \frac{1}{N} \sum_{\text{N}} \mathcal{O}(U) \qquad \sigma(\bar{\mathcal{O}}) \propto \frac{1}{\sqrt{N}}$$

Metropolis Method:

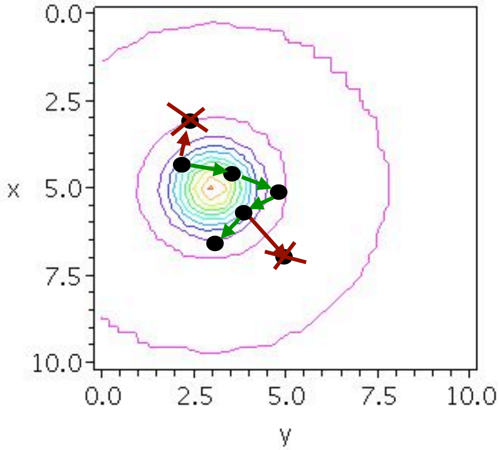
Start from some initial configuration.

Repeat until set of configs. is large enough:

- From config U, pick U' (reversibly)
- Accept with Metropolis probability:

$$P(U' \leftarrow U) = \min\left(1, \frac{e^{-S(U')}}{e^{-S(U)}}\right)$$

- If we reject, next config is U (again)



Generates a Markov Chain of configurations. Errors in observables fall as the number of samples grows

Global Updating

- Imagine changing 'link by link'
- For each change one needs to evaluate the fermion action twice: before and after

$$S_f = \phi^\dagger \left(M^\dagger M\right)^{-1} \phi = \langle \phi | X \rangle$$

where

$$\left(M^\dagger M\right) X = \phi$$

Two Degenerate Flavors of fermion (eg: u & d). Guaranteed
- **Hermitean**
- **Positive Definite**

Use Sparse Krylov Subspace Solver: eg: Conjugate Gradients

Linear system needs to be solved on entire lattice.
- **Dimension: ~ O(10M)**
- **Condition number: O(1-10M)**

- 1 Sweep: 2x4V solves, with 4V ~ O(1M-33M) is prohibitive
- Need a Global Update Method

Hybrid Monte Carlo

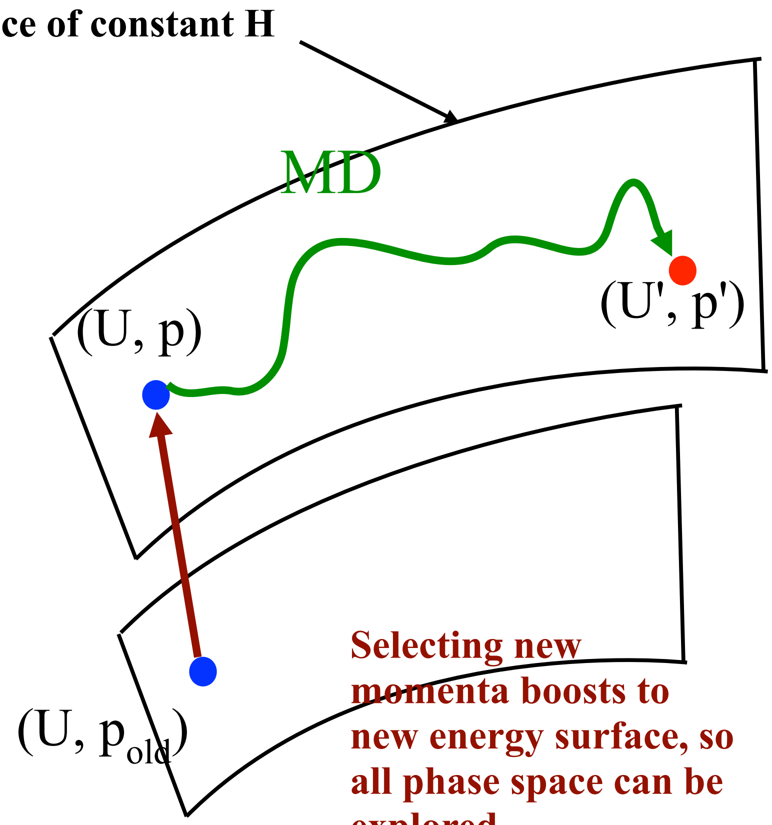
- **Big Trick: Go from config U to U' doing Hamiltonian Molecular Dynamics in Fictitious Time**

- start from config U
- generate momenta p
- evaluate H(U,p)
- perform MD in fictitious time t
- evaluate H(U', p')
- accept with Metropolis probability

$$P = \min\left(1, e^{-H(U',p')+H(U,p)}\right)$$

- if accepted new config is U', otherwise it is U

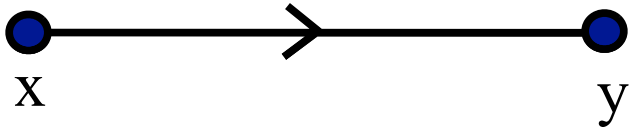
MD Conserves Energy
If done exactly P = 1 (always accept)
Otherwise dH depends on the error from the integrator



Selecting new momenta boosts to new energy surface, so all phase space can be explored

After the Gauge Generation

Quark Propagator: $G(x,y) = M^{-1}_{x,y} S(x)$



Correlation Functions:

Mesons:

$$C(\vec{p},t) = \sum e^{i\vec{p}.\vec{x}}\ \mathrm{Tr}\ \Gamma\ G^{\dagger}(\vec{x},t;0,0)\ \Gamma\ G(\vec{x},t;0,0)$$

G projects onto correct spin-parity quantum numbers

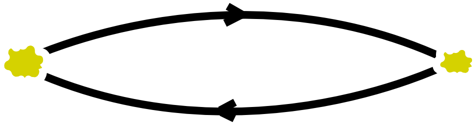
Fourier Transform in space, transforms to Momentum Space.

antiquark

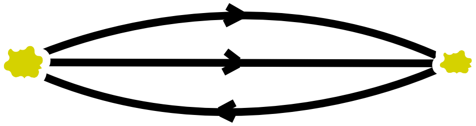
quark

Translation invariance:
G(x,0) <=> G(z+x, y)

Meson:



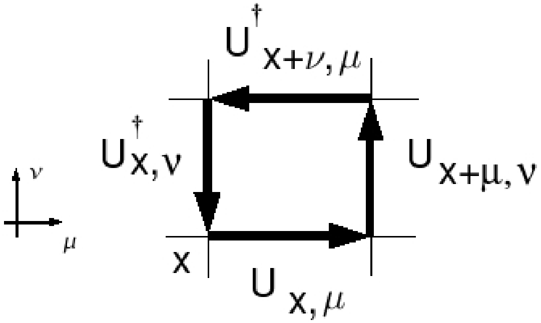
Baryon:



- Measure on each configuration, but only the 'average' is 'physical.
- Baryons also need color antisymmetrization
- Fourier transform fixes definite momenta, but loses volumetric info
  - Not much in the way of pretty visualizations – mostly 2D plots

Lattice QCD and Parallel Computing

- We have two basic patterns in LQCD computations:
  - ***do the same thing*** at every site
    - either independently or
    - depending on other nearby sites



$$P_{\mu\nu}(x) = U_{\mu}(x)\ U_{\nu}(x+\mu)U^{\dagger}_{\mu}(x+\nu)U^{\dagger}_{\nu}(x)$$

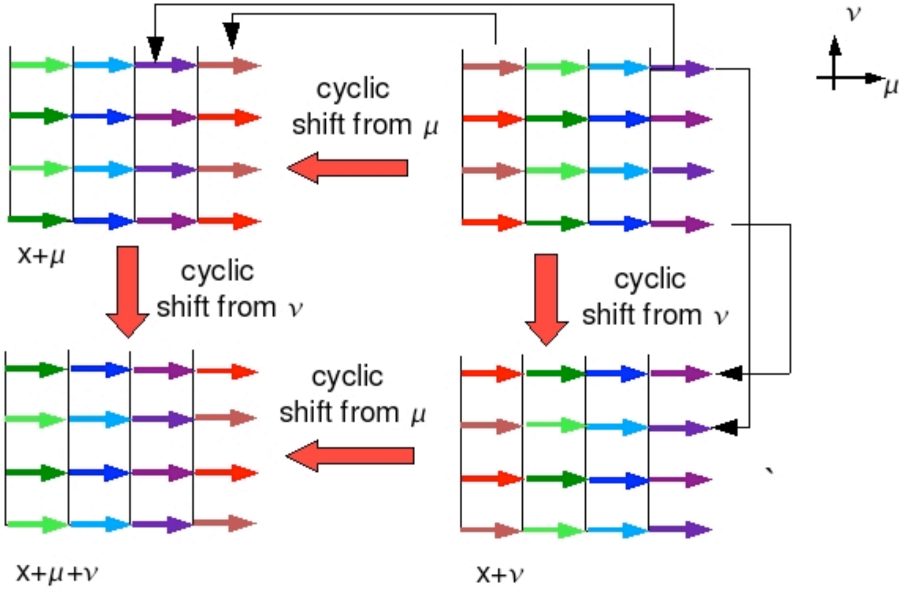
  - perform a ***global reduction*** (sum, inner product)

$$\sum_{x}\sum_{\mu\neq\nu} \mathrm{Re}\ \mathrm{Tr}\ P_{\mu\nu} \qquad \langle\psi|\chi\rangle = \sum_{x}\psi^{\dagger}(x)\chi(x)$$

- This is a classic ‘data parallel’ pattern

Expressing Data Parallelism: 1

- Data Parallel Expressions (QDP++, CM-Fortran, etc)
  - Work on lattice wide objects : **Global View**
  - Hide indices where possible
  - Nearest neighbour => shift whole lattice
  - Reductions: functions like sum(), norm2() etc



```
LatticeColorMatrix plaq = zero;
for(int mu=0; mu < Nd; mu++) {
  for(int nu=mu+1; nu < Nd; nu++) {
     LatticeColorMatrix tmp, tmp2,tmp3;
     // U_nu(x + mu)
     tmp = shift( u[nu] , FORWARD, mu);
     tmp2 = u[mu]*tmp;
     // U_mu(x + nu)
     tmp = shift( u[mu], FORWARD, nu);
     tmp3 = u[nu]*tmp;
     plaq += tmp2*adj(tmp3);
  }
}
Double w_plaq = sum(real(trace(plaq)));
```

Expressing Data Parallelism: 2

- ‘Map-Reduce’ like: CUDA/Thurst/TBB
  - define “kernel” to execute per site: **Local View** (+reductions)

```
class PlaqKernel :public Kernel2Arg<const GaugeField&,LatticeColorMatrix&> {
public:
   PlaqKernel(GaugeField& u,LatticeColorMatrix& p_):u(u_),plaq(p_) {}

 void operator(int site) {
    plaq[site] = 0;
    for(int mu=0; mu < Nd; mu++) {
    for(int nu=mu+1; nu < Nd; nu++) {
      Matrix m1= u[mu][site];
       Matrix m2= getPlus(u[nu],mu,site);
       Matrix m3= getPlus(u[mu],nu,site);
       Matrix m4= u[nu][site];
       plaq[site] += m1*m2*adj(m3)*adj(m4);
     }
    }
   }
private:
   const GaugeField& u;  LatticeColorMatrix& plaq;
};
```

Expressing Data Parallelism: 2

```
// Use
GaugeField u=...;  // Get U somewhow
LatticeColorMatrix plaq;

// Call the kernel
map_2arg<PlaqKernel,GaugeField, LatticeColorMatrix>(u,plaq);



// Underneath in the framework:
template<class K, class T1, class T2>
map_2arg(T1& in1, T2& in2)
{
   K foo(in1, in2); // create kernel

   // Implement this in OpenMP/TBB/CUDA etc
   parallel_forall(sites) {

       // Call the kernel once for each
       // site. Uses the operator()
       foo(site);
   }
}
```

Generic 2 arg map function

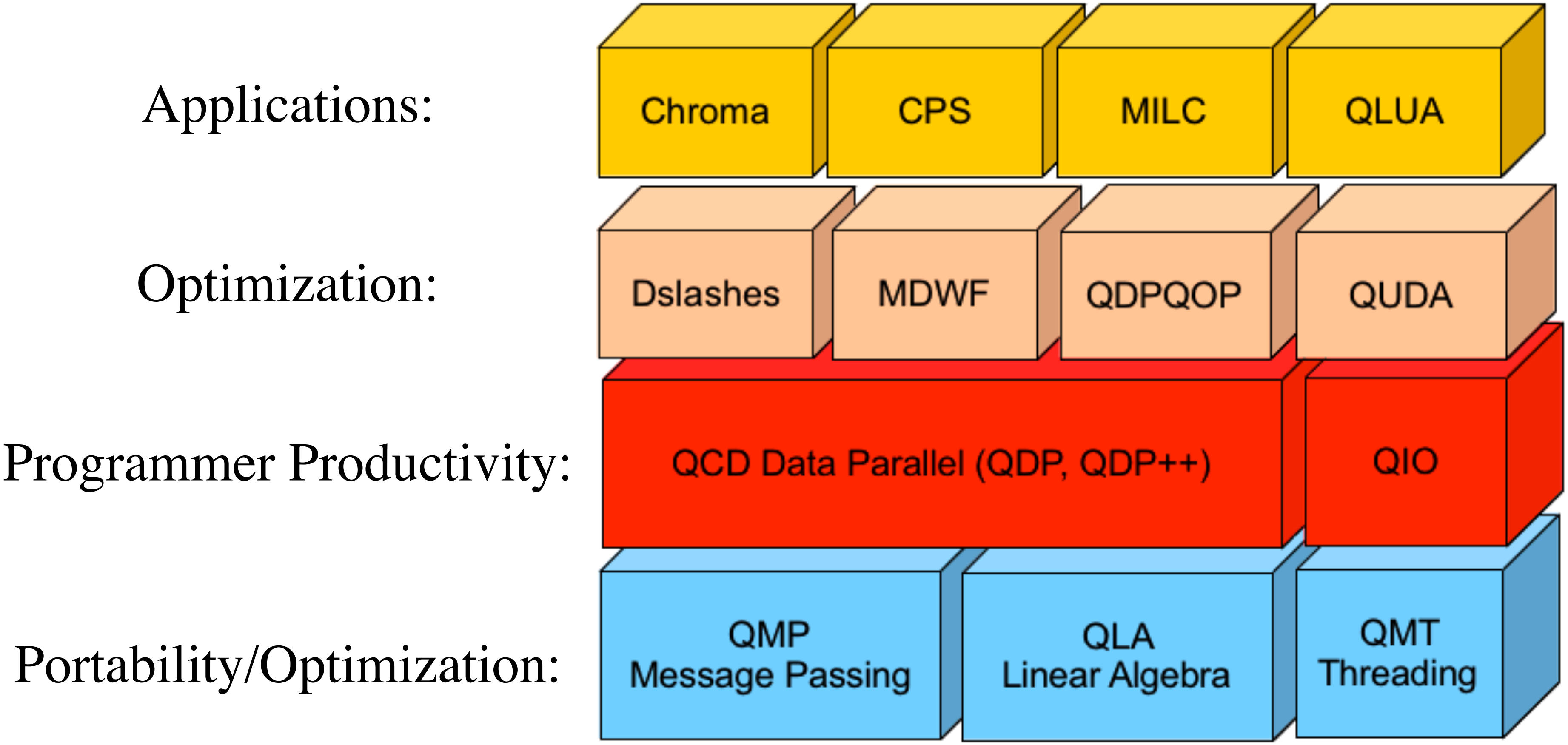
Trade-offs

- Trade offs come in terms of where you want to focus:
  - expressions express maths better
    - at the expense of expressing data re-use
  - ‘Kernels’ can express data re-use/locality better
    - at the risk of losing the expressiveness of the maths
- Mapping to underlying hardware
  - CUDA and OpenCL organized around ‘Kernel’ approach
  - Compile kernels to execute on the ‘device’.
  - Provide Compiler/Language/Driver support for this.
  - See Mike Clark’s lectures on GPUs for more.
- Can mix and match
  - Can implement expressions, as kernels

What are QDP++ and Chroma

- QDP++ and Chroma are software packages for numerical simulations of Lattice QCD (mostly)
- QDP++
  - provides data parallel expressions for QCD
    - ‘embedded domain specific language’,
    - ‘virtual data parallel machine’
  - plus I/O
  - configure time: Nd, Nc, Ns (dimensions, colors, spins)
- Chroma
  - provides the application on top of QDP++
  - propagators, HMC, measurements
  - also link to external libraries for dslash-es/solvers etc.

Place in USQCD Software Stack



Applications: Chroma, CPS, MILC, QLUA

Optimization: Dslashes, MDWF, QDPQOP, QUDA

Programmer Productivity: QCD Data Parallel (QDP, QDP++), QIO

Portability/Optimization: QMP Message Passing, QLA Linear Algebra, QMT Threading

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)

Using QDP++ and Chroma

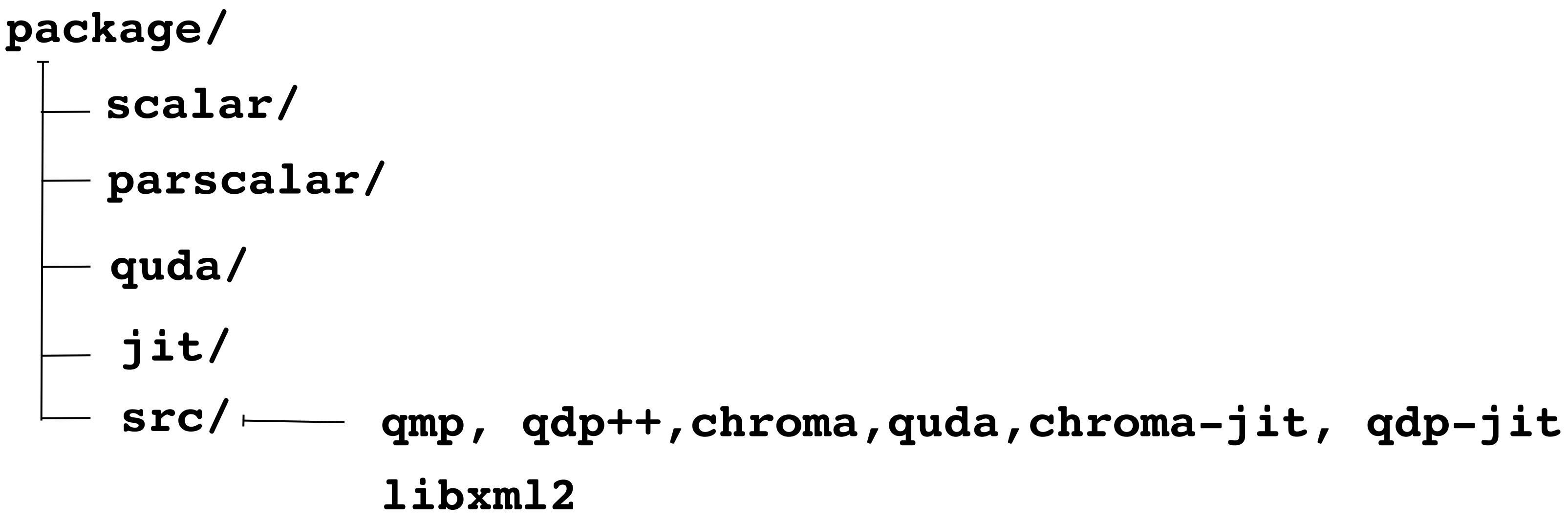
- Our experience:
  - a large number of users use the ‘chroma’/‘hmc’ executables with a XML input files
  - relatively few users write QDP++/Chroma programs or interface with QDP++/Chroma
  - a small subset of users check code back in or send us patches
- These lectures will focus mostly on QDP++
  - Chroma is very large and the ‘trees obscure the woods’
  - I provide a software package which includes Chroma too.
  - You should be able to build using the build scripts (possibly modified to suit your system)

Code package

- package-int.tar.gz contains:
  - sources for QDP++, Chroma, QUDA and support libraries
  - build directories for
    - scalar -- for use on your laptops
    - parscalar -- a build with MPI
    - quda - a parscalar build combined with QUDA for GPUs
    - jit - a parscalar build over the JIT version of QDP++ for GPUs as well as QUDA
  - Builds from scalar -> jit require increasing amount of intrepidity
  - QUDA version is older, but reasonably stable
  - JIT branch of QDP++ is current
  - As with all free and developing software: ‘Caveat Emptor!’

Untarring the package

- Download package-int.tar.gz
- After unzipping:

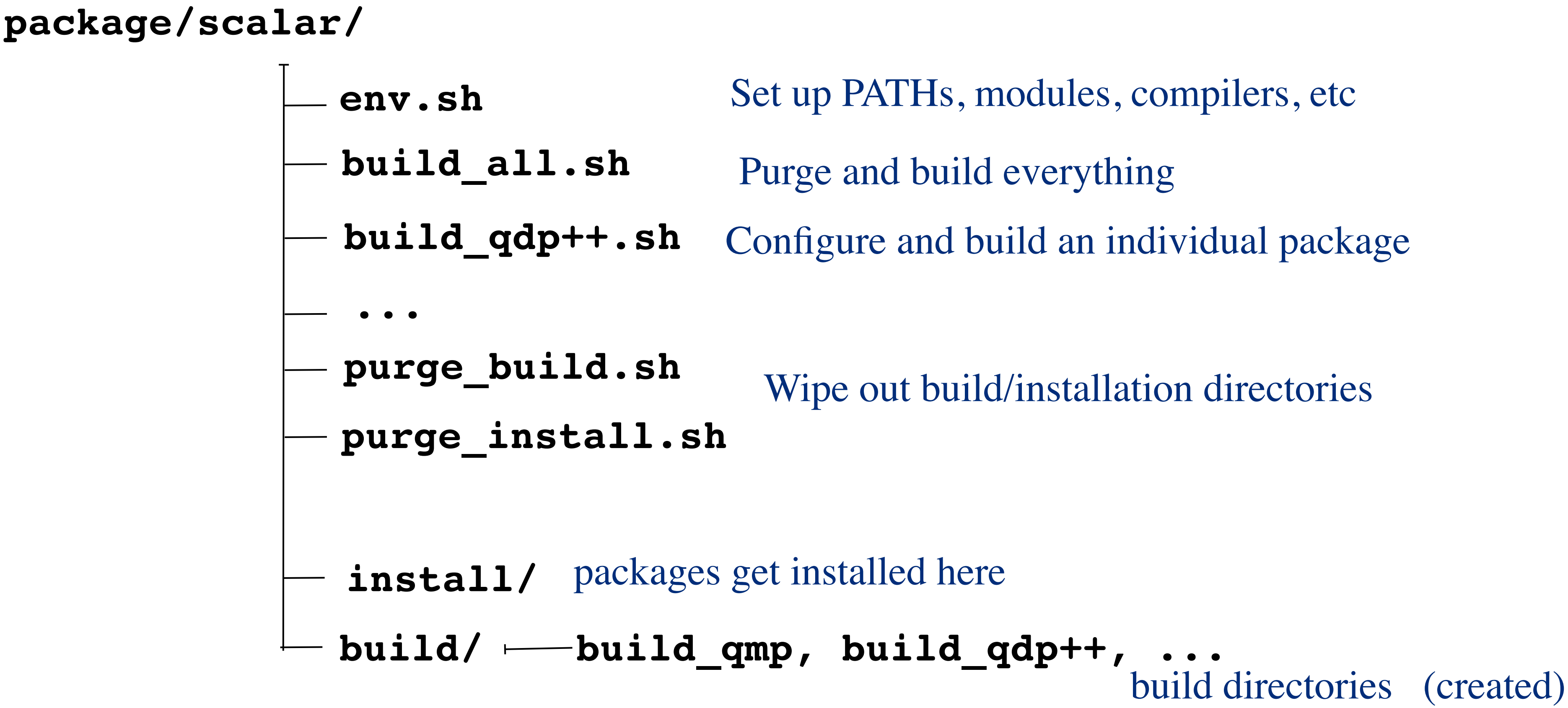


```
package/
├── scalar/
├── parscalar/
├── quda/
├── jit/
└── src/ ── qmp, qdp++,chroma,quda,chroma-jit, qdp-jit
            libxml2
```

package sources

Structure of build directories

- Build directories have scripts to build and install packages
  - filenames may change but the scheme is as below
  - env.sh sets the environment. Tailor this to your system



`package/scalar/`

`env.sh` Set up PATHs, modules, compilers, etc

`build_all.sh` Purge and build everything

`build_qdp++.sh` Configure and build an individual package

`...`

`purge_build.sh`

`purge_install.sh`

Wipe out build/installation directories

`install/` packages get installed here

`build/` — `build_qmp, build_qdp++, ...`

build directories (created)

Setting up the environment

- env.sh (or env-jit.sh) sets up build environment
  - sets up paths, compiler flags, copiler commands, parallel make etc
  - e.g. on my Mac, the user servicable parts of scalar/env.sh look like:

```
OMPFLAGS=""
OMPENABLE=""
```

don’t use OpenMP for now

CFLAGS/ CXXFLAGS to use

```
### COMPILER FLAGS
PK_CXXFLAGS=${OMPFLAGS}" -O3 -finline-limit=50000 -march=core2  -fargument-noali
as-global"

PK_CFLAGS=${OMPFLAGS}" -O3 -march=core2 -fargument-noalias-global -std=gnu99"

### Make
MAKE="make -j 2"
```

use parallel make with 2 processes

```
### MPI
PK_CC=gcc
PK_CXX=g++
```

compiler commands

Performing the builds

- Usually a script that looks like build_all-xxx.sh invokes the build steps.
- E.g. for scalar build. Builds QDP++ only for the exercises
- commands to build chroma + DP versions commented out

```
#!/bin/bash

#BUILD QDP++ AND CHROMA IN PARALLEL WITHOUT QUDA
./purge_build.sh
./purge_install.sh

./build_libxml2.sh

# BUILD Single Prec QDP++ -- sufficient for tutorials
./build_qdp++-scalar.sh

# IF you feel brave you can build chroma too
#./build_chroma-scalar.sh
#
#./build_qdp++-double-scalar.sh
#./build_chroma-double-scalar.sh
```

Invokes: configure/make/make install chain for package

Running Chroma

- Main applications
  - chroma - for measurements
  - hmc - for gauge generation
- Typical command line (after the MPI options)
  - ./chroma -i in.xml -o out.xml -geom Px Py Pz Pt
  - in.xml - Input Parameter File
  - out.xml - Output XML file
  - Px Py Pz Pt are the dimensions of a virtual processor grid: e.g.: -geom 4 4 8 8 implies 4x4x8x8 grid of MPI processes
  - for threaded builds need also OMP_NUM_THREADS/ QMT_NUM_THREADS env variables set
  - env vars/thread binding etc are system specific

XML input files

Array of Measurements (Tasks)

```
<?xml version="1.0" encoding="UTF-8"?>
<chroma>
<annotation>Your annotation here</annotation>
<Param>
  <InlineMeasurements>
    <elem>
      <Name>MAKE_SOURCE</Name>
      <Frequency>1</Frequency>
      <Param/>
      <NamedObject>
        <gauge_id>default_gauge_field</gauge_id>
        <source_id>sh_source_0</source_id>
      </NamedObject>
    </elem>
    <elem>
      <Name>PROPAGATOR</Name>
      <Frequency>1</Frequency>
      <Param/>
      <NamedObject>
        <gauge_id>default_gauge_field<gauge_id>
        <source_id>sh_source_0</source_id>
        <prop_id>sh_prop_0</prop_id>
      </NamedObject>
      <xml_file>./prop_out.xml<xml_file>
    </elem>
  </InlineMeasurements>
 <nrow>4 4 4 8</nrow>
</Param>
<RNG/>
<Cfg>
 <cfg_type>SCIDAC</cfg_type>
 <cfg_file>foo.lime</cfg_file>
</Cfg>
</chroma>
```

XML Input Files

```
<?xml version=”1.0” encoding=”UTF-8”?>
<chroma>
<annotation>Your annotation here</annotation>
<Param>
  <InlineMeasurements>
    <elem>
      <Name>MAKE_SOURCE</Name>
      <Frequency>1</Frequency>
      <Param/>
      <NamedObject>
        <gauge_id>default_gauge_field</gauge_id>
        <source_id>sh_source_0</source_id>
      </NamedObject>
    </elem>
    <elem>
      <Name>PROPAGATOR</Name>
      <Frequency>1</Frequency>
      <Param/>
      <NamedObject>
        <gauge_id>default_gauge_field<gauge_id>
        <source_id>sh_source_0</source_id>
        <prop_id>sh_prop_0</prop_id>
      </NamedObject>
      <xml_file>./prop_out.xml<xml_file>
    </elem>
  </InlineMeasurements>
 <nrow>4 4 4 8</nrow>
</Param>
<RNG/>
<Cfg>
 <cfg_type>SCIDAC</cfg_type>
 <cfg_file>foo.lime</cfg_file>
</Cfg>
</chroma>
```

Task (array element)

Task name

Task Parameters

Named Objects (communicate between tasks -- like “in memory” files)

XML Input Files

```
<?xml version=”1.0” encoding=”UTF-8”?>
<chroma>
<annotation>Your annotation here</annotation>
<Param>
  <InlineMeasurements>
    <elem>
      <Name>MAKE_SOURCE</Name>
      <Frequency>1</Frequency>
      <Param/>
      <NamedObject>
        <gauge_id>default_gauge_field</gauge_id>
        <source_id>sh_source_0</source_id>
      </NamedObject>
    </elem>
    <elem>
      <Name>PROPAGATOR</Name>
      <Frequency>1</Frequency>
      <Param/>
      <NamedObject>
        <gauge_id>default_gauge_field<gauge_id>
        <source_id>sh_source_0</source_id>
        <prop_id>sh_prop_0</prop_id>
      </NamedObject>
      <xml_file>./prop_out.xml<xml_file>
    </elem>
  </InlineMeasurements>
 <nrow>4 4 4 8</nrow>
</Param>
<RNG/>
<Cfg>
 <cfg_type>SCIDAC</cfg_type>
 <cfg_file>foo.lime</cfg_file>
</Cfg>
</chroma>
```

Global Lattice Size

Input Configuration to use as default_gauge_field

Where to find XML Examples

- Most up to date place:
  - chroma/tests/
- All the regression tests inputs and outputs live here
- .ini.xml - input XML file
- .out.xml or .log.xml - expected output / log
- .metric.xml - metric file for XMLDIFF tool
- Typically suppose regression test produces foo.xml then we can check
  - xmldiff foo.xml expected.xml expected.metric.xml

Linking Against QDP++/Chroma

- Suppose QDP++ is installed in /foo/qdp++
- Use script qdp++-config in /foo/qdp++/bin
  - CXX=`/foo/qdp++/bin/qdp++-config --cxx`
  - CXXFLAGS=`/foo/qdp++/bin/qdp++-config --cxxflags`
  - LDFLAGS=`/foo/qdp++/bin/qdp++-config --ldflags`
  - LIBS=`/foo/qdp++/bin/qdp++-config --libs`
- Compile your program (prog.cc) with:
  - $(CXX) $(CXXFLAGS) prog.cc $(LDFLAGS) $(LIBS)
  - NB: Ordering of flags may be important.
- Linking against chroma:
  - Use install path of chroma (instead of QDP++) and
  - Use chroma-config (instead of qdp++-config)

Stopping point

- Covered high level view of numerical LQCD
- Considered parallel programming ‘models’
- Gave a brief overview of QDP++ and Chroma
- Discussed getting and building the packages
- Discussed running chroma, linking against chroma

- Exercises follow:
  - NB: The exercises are mostly using QDP++, rather than chroma
  - However, plenty of chroma exercises in existing tutorials for you to try:
    - http://usqcd.jlab.org/usqcd-docs/chroma/

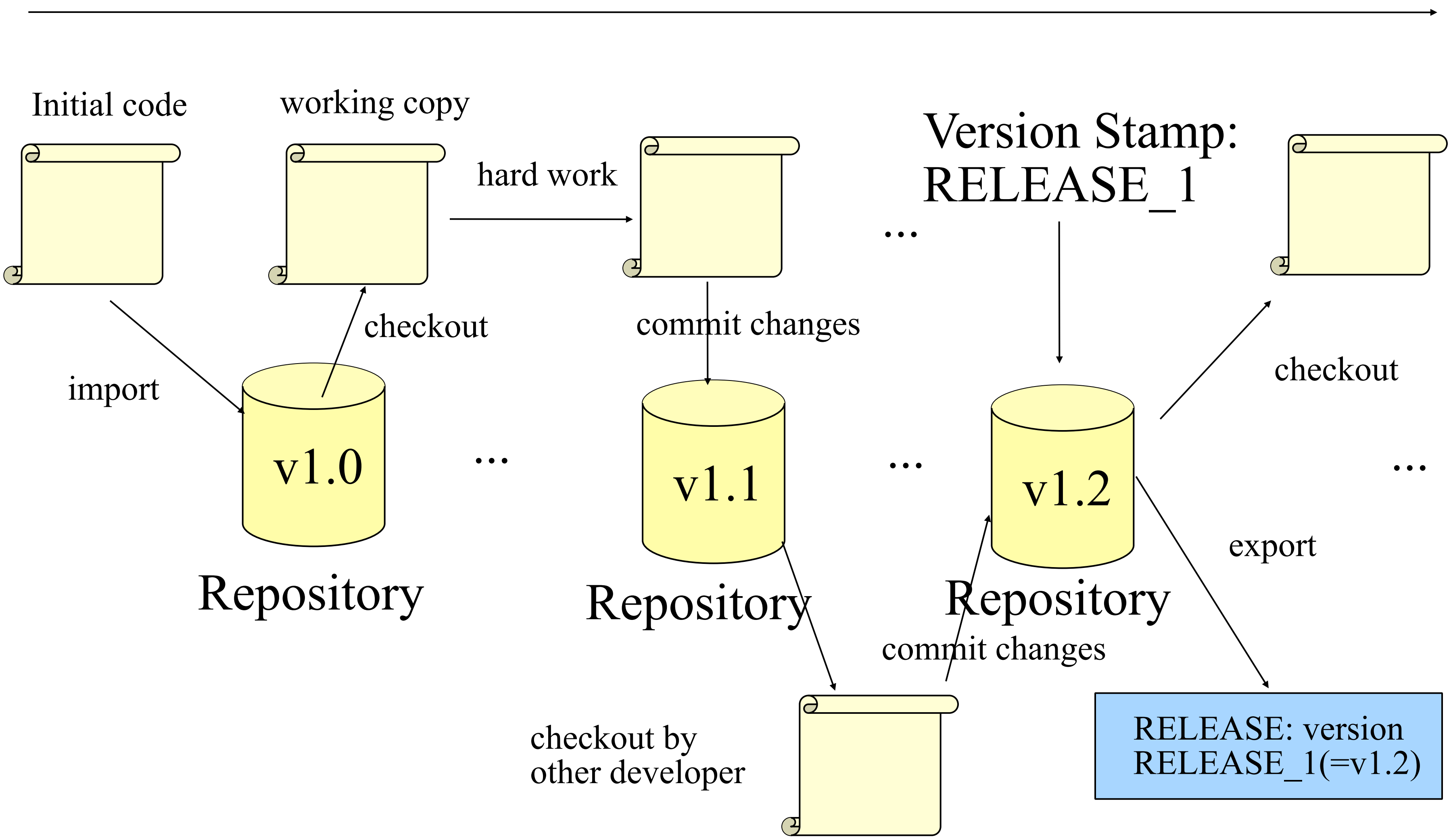
Exercises

- Basic:
  - Compute the plaquette of a random configuration
- Advanced:
  - Compute a Polyakov loop on the configuration
- Topics Touched on:
  - Makefiles
  - Basic QDP++ Boilerplate setup code
  - Shifts
  - Global Sums
  - Simple printing in a pseudo-parallel world

Revision Control (RC)

- RC systems track changes of your code over its lifetime
  - Lifecycle:
    - You import an initial code to a REPOSITORY
    - You check out a WORKING COPY of the files
    - You make some changes
    - You commit the changes
    - You can label versions at any point with a human readable label (eg: for releases)
    - You can create branches (eg: for bugfixes)
  - Which version control to use?
    - Currently I prefer Git
    - I cannot cover it in more detail here, but I recommend it to you: http://git-scm.org

Revision Control and software lifecycle



Why Should I use Revision Control

- A good revision control system provides the most important safety and convenience features
  - **IT IS YOUR PANIC BUTTON**
    - You can revert changes even if you've lost the original source in your working copy
  - **IT ALLOWS YOU TO DEVELOP ANYWHERE**
    - Most good Revision Control Systems allow you to check out over the network and anonymously too.
  - You can Branch off an existing revision to do maintenance (bug fixes etc). The RC system will help you merge changes back onto the main trunk
  - Many RC-s have web features: http://git.jlab.org

Get the Code

- Download the code tarball
- Actually this is a fully fledged GIT repository
- The tarball should uncompress into a directory called **`seattle_tut`**
- **`seattle_tut`** has 4 subdirectories
  - **`example1`**
  - **`example2`**
  - **`example3`**
  - **`example4`**

- We will work in example1 in this session.

Edit the Makefile

- Go to the example directory you've just checked out

```
bash$ cd seattle_tut/example1
```

- Edit the **`Makefile`** :
  - Replace the path in the **CONFIG** Makefile variable to reflect where you've installed qdp++
  - probably something like:
  - **`/.../package/scalar/install/qdp++-scalar/bin/qdp++-config`**
- Do this also in **`seattle_tut/example1/lib/Makefile`**
- You can now build the code by typing '**`make`**'

Run the example

- Run the executable:

```
bash$ ./example1
Finished init of RNG
Finished lattice layout
bash$
```

- NB: Cygwin Users should put .exe on the end of executables:

```
bash$ ./example1.exe
Finished init of RNG
Finished lattice layout
bash$
```

- Doesn't do much useful yet – just checking it works for now

Makefiles

- Makefile-s tell 'make' what to do
  - Three main parts (for our purposes)
    - MACROS (to make your life easier)
    - Rules (to tell make how to compile)
    - target/dependency pairs (tell make what to compile, and what depends on what else)

example1/Makefile:

```
# The config program of QDP++
CONFIG=/home/bjoo/install/qdp++/bin/qdp++-config

# Use the config program to set up compilation
CXX=$(shell $(CONFIG) --cxx)
QDP_CXXFLAGS=$(shell $(CONFIG) --cxxflags)
QDP_LDFLAGS=$(shell $(CONFIG) --ldflags)
QDP_LIBS=$(shell $(CONFIG) --libs)

# Some extra flags from us
CXXFLAGS=$(QDP_CXXFLAGS) -I./include
LDFLAGS=$(QDP_LDFLAGS) -L./lib
LIBS=-lexample $(QDP_LIBS)

all: example1

example1: example1.cc ex1_libs
TAB   $(CXX) -o $@ $(CXXFLAGS) $< $(LDFLAGS) $(LIBS)
```

Makefile Macros

use macros as $(macro)

Makefile Targets

Makefile Dependencies

Makefile action

example1/lib/Makefile

```
.SUFFIXES=.h .cc .o .a

# ... deleted some lines to save space

# A rule to make a .o file from a .cc file
%.o: %.cc
        $(CXX) $(CXXFLAGS) -c $<

# A rule that says:
# To make all our object files, compile the .cc files to .o
files
OBJS=$(SRCS:%.cc=%.o)

#deleted lines to save space
#dependencies
reunit.o: reunit.cc ../include/reunit.h
```

Compile Rule:
make a .o file from .cc

Special macro: $<
== name of input file

Rule: Make .o files from all .cc files in $SRCS

Special target/dependency pair:
Only enforces dependency. Rest done by compile rule.

Now the code: example1/example1.cc

```
#include "qdp.h"      // The core QDP++ library header
#include "reunit.h"   // A reunitarizer I provide you with

using namespace std; // Import from STD namespace (io etc)
using namespace QDP; // Import from QDP namespace (QDP++ things)

// Here is our program
int main(int argc, char *argv[])
{
  // Set up QDP++
  QDP_initialize(&argc, &argv);
  multi1d<int> latt_size(Nd);
  latt_size[0] = 4; latt_size[1] = 4; latt_size[1]=4; latt_size[3]=8;

  Layout::setLattSize(latt_size);
  Layout::create();    // Setup the layout

  // QDP++ is now ready to rock

  // Clean up QDP++
  QDP_finalize();
  exit(0);  // Normal exit
}
```

The .h for qdp++
in Namespace QDP

multi1d<int>
- resizable 1d array of int-s
(for holding lattice size)

QDP++ Boiler plate setup
and finalization code

Program Body Goes in Here

Doing Stuff with QDP++

- Lattice Wide Types: eg a Lattice of SU(3) Color matrices
  - QDP++ Type: `LatticeColorMatrix`
  - Gauge field: Nd (ie: 4)length array of SU(3) lattices:
    - QDP Type: `multi1d<LatticeColorMatrix> u(Nd);`
    - Can index as `u[0]`, `u[1]` etc.
  - Filling a LatticeColorMatrix with gaussian noise:
    - QDP++ Function: `gaussian(u[i]);`
  - Projecting back into SU(3):
    - Function provided in the library in lib/
    - `void Example1::reunit(LatticeColorMatrix& u)`
      - in namespace Example1
      - need to #include “reunit.h” for definition

Starting Up a Gauge Field

- A Unit Gauge (Free Field):

```
multi1d<LatticeColorMatrix> u( Nd ); // Nd = 4 usually
for(int mu=0; mu < Nd; mu++) {
 u[mu] = Real(1);
}
```

- A Randomized Gauge Field (Disordered/Hot Start):

```
multi1d<LatticeColorMatrix> u( Nd ); // Nd = 4 usually
for(int mu=0; mu < Nd; mu++) {
 gaussian( u[mu] );          // Fill with gaussian Noise
 Example1::reunit( u[mu] );  // project back to reunitarize
}
```

Arithmetic and Shifts

- Can do 'normal' arithmetic: e.g.: Multiplies, adds, etc

```
LatticeColorMatrix x,y,z;
gaussian(x); gaussian(y);
z = x*y; // multiply x and y together on each site -> z
z = z*y; // This involves 'aliasing' of z.
         // It'll compile but may have wrong result, use *=
z += x;  // Add to
z = z + x; // This involes 'aliasing' again not recommended
           // use += in this case
z = x + y; // This is fine
```

- Shifts

```
LatticeColorMatrix x_x_plus_mu;
x_x_plus_mu = shift(x, FORWARD, mu); // get x from forward
                                     // mu direction
```

Utilities

- Things to know about the 'model computer' and the 'lattice'
  - in namespace **`QDP::Layout`**
    - **`Layout::sitesOnNode()`** - sites local to your Processing element (MPI process)
    - **`Layout::vol()`** - the global volume (sites)
- Text / IO to the screen:
  - **iostream** like **cout** and **cerr** streams (master node prints)
    - **`QDPIO::cout`**
    - **`QDPIO::cerr`**
  - C printf like routines (every node prints)
    - **`QDP_info(“fmt”, variables);`**

Computing the Plaquette

```
int n_planes = Nd*(Nd-1)/2;   // 6 in 4D
LatticeColorMatrix plaq = zero;
for(int mu=0; mu < Nd; mu++) {
  for(int nu=mu+1; nu < Nd; nu++) {
     LatticeColorMatrix tmp, tmp2,tmp3;
     tmp = shift( u[nu] , FORWARD, mu);  // U_nu, x+mu
     tmp2 = u[mu]*tmp;   // U_mu U_nu,x+mu
     tmp = shift( u[mu], FORWARD, nu);   // U_mu, x+nu
     tmp3 = u[nu]*tmp;   // U_x,nu U_mu,x+nu

     // U_mu U_nu,x+mu U^\dag_mu,x+nu U^\dag_nu
     plaq += tmp2*adj(tmp3);
  }
}
Double normalize = Real(3)*Real(n_planes)*Layout::vol();
Double w_plaq = (Double(1)/normalize)*sum(real(trace(plaq)));
QDPIO::cout << "Plaquette=" << w_plaq << endl;
```

Temporaries, disappear at end of {} scope

Use Shifts to get nearest neighbours

Print Result

Collectives: alltoall (sum)/ local (trace)

Some actual coding

- Add the code for starting up the random gauge field and computing the plaquette after the line

  ```
  // QDP++ is now ready to rock
  ```

  in the example1.cc file

- remake example1 (or example1.exe) by typing '**make**'
- rerun the example1 (or example1.exe)
  - Output should be something like:

    ```
    Finished init of RNG
    Finished lattice layout
    Plaquette=0.00127763178119898
    ```

- Replace the gauge startup code with the one for the free field (unit gauge). Remake and Rerun. Verify that the Plaquette=1.

Exercise 1: Random Gauge Transforms

- Can you write a routine to perform a random gauge transformation on u ?

$$U'_\mu(x) \leftarrow G(x)U_\mu(x)G^{-1}(x+\hat{\mu})$$

  - Hints:
    - You'll need a LatticeColorMatrix but not a **multi1d<>** one. (Gauge transform matrices - G- live on the sites.)
    - You'll need to randomize it and make it SU(3)
    - You'll need to shift and use the adj() function to get at

$$G^{-1}(x+\hat{\mu}) = G^\dagger(x+\hat{\mu})$$

    - Recompute the plaquette of the Random Gauge Transformed 'u' and check it is gauge invariance.
    - Compute the Link trace of the Random Gauge transformed 'u' and the original one. Should be different...

Exercise 2: Polyakov Loop

- Can you compute the Polyakov Loop?
  - This observable is an order parameter for the finite temperature phase transition.
  - This observable, modulo some normalization factor is the “sum of the (complex) trace of the product of matrices along the time direction of the lattice”
  - Hints:
    - You'll need to shift in the 't' direction
    - the rest is similar to the plaquette.

$$\boxed{P = \frac{1}{N_c V} \sum_x \mathrm{Tr} \left(\prod_t U_t(x) \right)}$$

Next Session: “Dances with Solvers”

- In the next session we'll play with Fermions, Fermion matrices, solvers, propagators and correlation functions.
  - See you then!