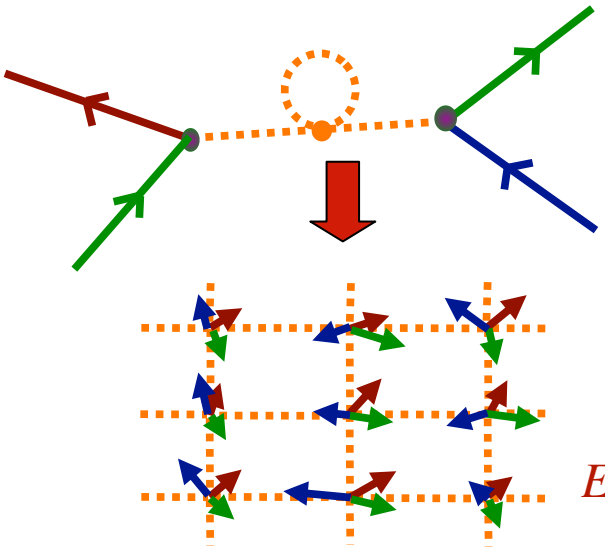
An Introduction

Bálint Joó,
Scientific Computing Group
Jefferson Lab

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

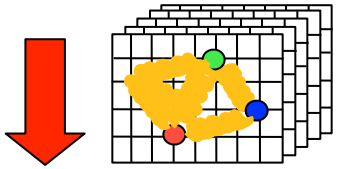


$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}A \; \mathcal{D}\bar{\psi} \; \mathcal{D}\psi \; \mathcal{O} \; e^{-S(A,\bar{\psi},\psi)}$$

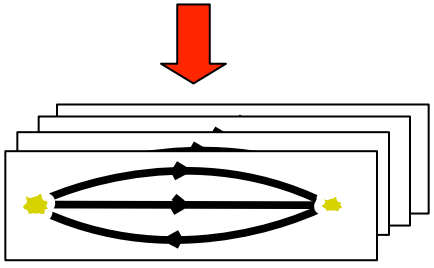
$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\text{all links}} dU \prod_{\text{all sites}} d[\bar{\psi},\psi] \; \mathcal{O} \; e^{-S(U,\bar{\psi},\psi)}$$

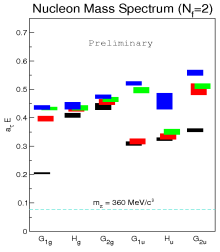
Evaluate Path Integral Using Markov Chain Monte Carlo Method

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

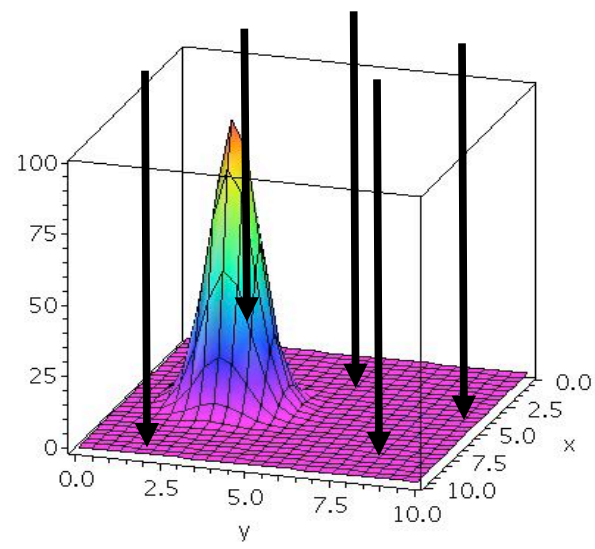
Evaluating the Path Integral:

- There are 4V links. V~16^3x64 - 32^3x256 → 4V = **1M ~ 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:

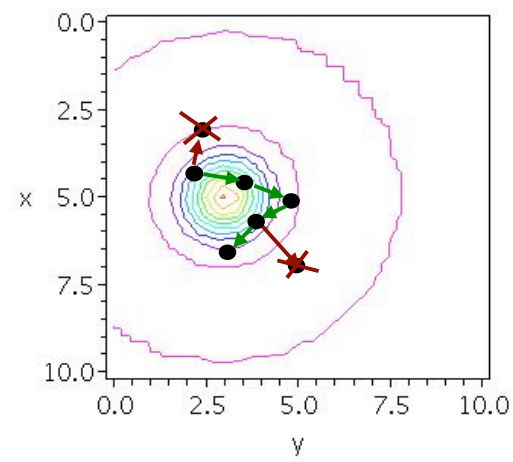
Start from some initial configuration.

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

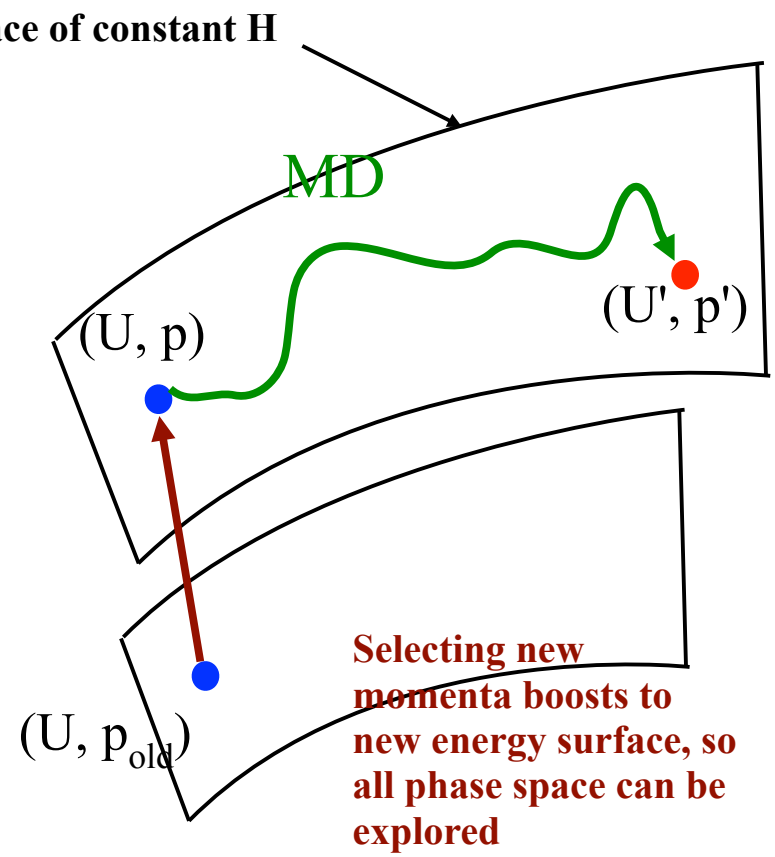
- **Big Trick: Go from config U to U' doing Hamiltonian Molecular Dynamics in Fictitious Time**

- 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

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

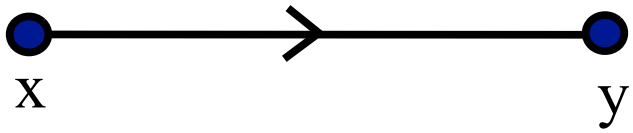
- if accepted new config is U', otherwise it is U

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



After the Gauge Generation

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



Correlation Functions:

G projects onto correct spin-parity quantum numbers

Mesons:

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

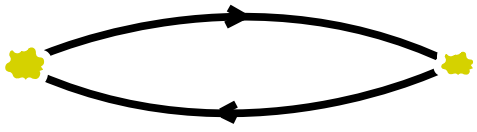
Fourier Transform in space, transforms to Momentum Space.

antiquark

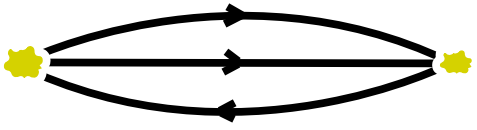
quark

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

Meson:



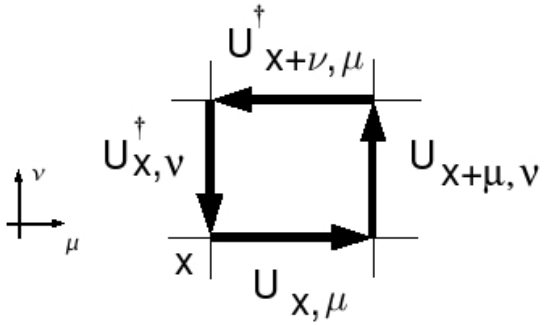
Baryon:



- 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



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

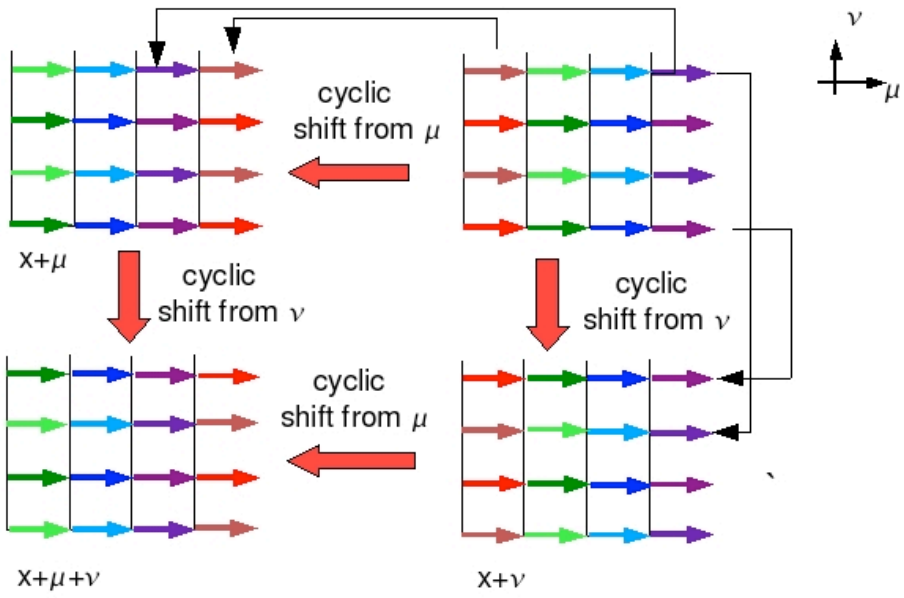
  - perform a ***global reduction*** (sum, inner product)

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

- 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.

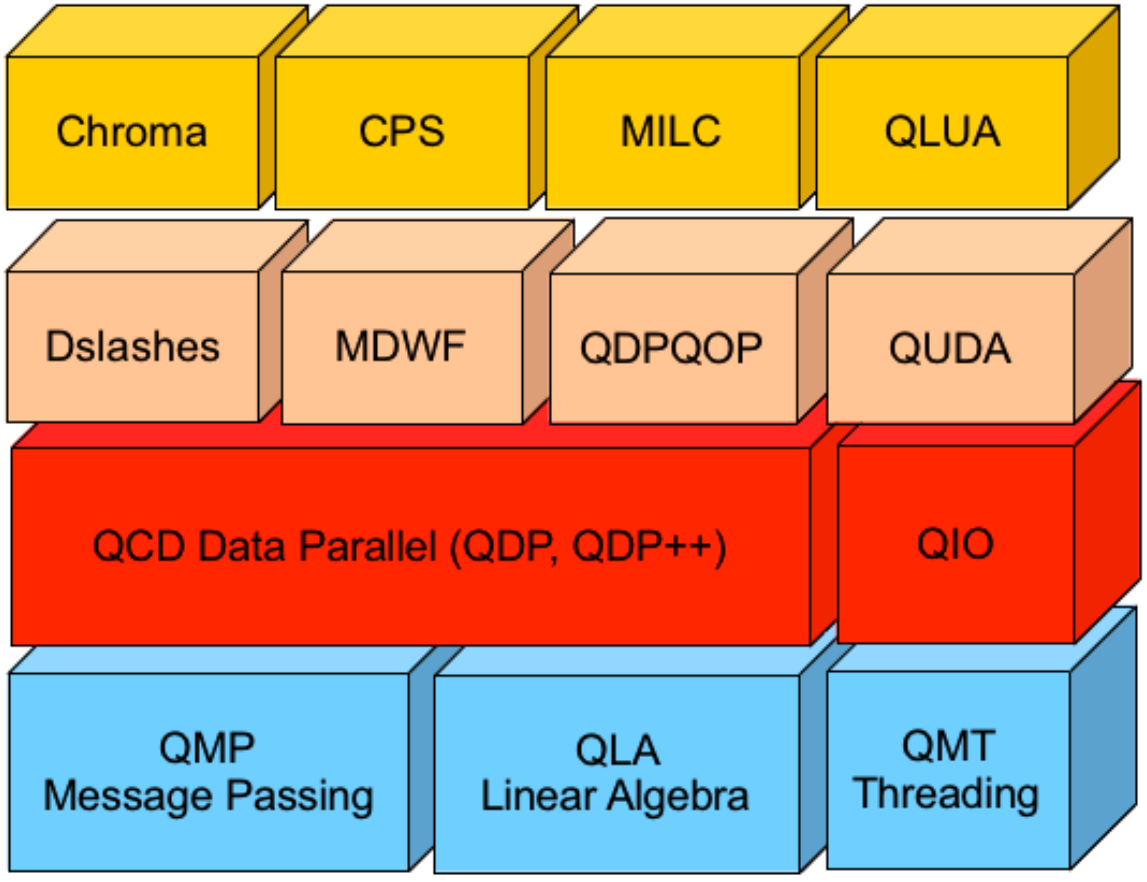
Place in USQCD Software Stack

Applications:

Optimization:

Programmer Productivity:

Portability/Optimization:



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
- Rest of this talk:
  - About getting/building/running chroma
- Later talks:
  - More in depth about Chroma/QDP++ internals
  - Future directions, including GPUs

Our Target Architectures

- x86 based systems
- Cray XE systems

- IBM Blue Gene Systems
  - BG/P systems using BAGEL Generated Dslash/BLAS
  - BG/Q systems - work in progress

- NVIDIA GPU Accelerated Architectures
  - Regular GPU Clusters
  - Cray XK systems
  - Porting QDP++ to GPUs has special challenges
    - QDP-JIT by Frank Winter addresses these

Getting the bits and pieces

- We have moved to using the GIT source code management system
- You can look at the repository online
  - http://git.jlab.org/
  - you'll see lots of projects
- Can get the code anonymously:

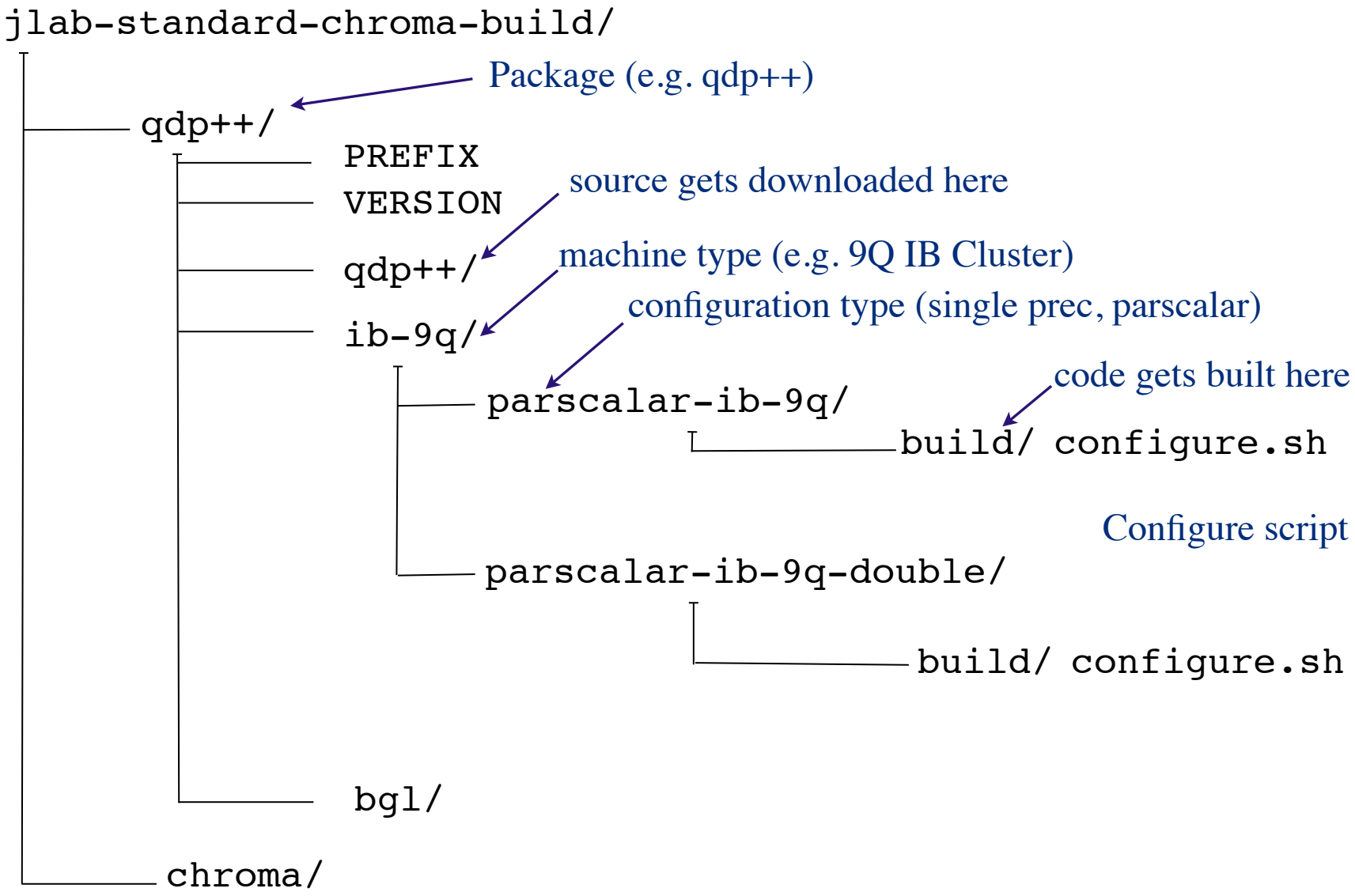
```
– git clone --recursive git://git.jlab.org/pub/lattice/usqcd/<module>
– <module> = qmp.git | qdp++.git | libxml2.git | chroma.git

– e.g. git clone --recursive git://git.jlab.org/pub/lattice/usqcd/chroma.git
```

- Git will bring with it all the revision history
- To check back in, you need a local account for now
  - but you can email patches
- We really like git. See http://git-scm.org

Building

- This is typically the most baffling part. How to build everything?
- Large variation amongst target systems:
  - compilers/compiler flags/MPI wrappers
  - which libraries to link (e.g. QUDA/MDWF/QDPQOP) ?
  - are the libraries installed ? do we have to install them?
  - some libraries don't support things like: make install
- GIT module: `jlab-standard-chroma-build.git`
  - encapsulates our experiences with clusters, Cray XT, BG/L & P
  - used for our nightly builds and regressions.
- `package-XXX.tar.gz` tarballs -- distributed these as needed
  - turnkey builds for Cray/BlueGene/GPU resources
  - plan move to support this style more.

jlab-standard-chroma-build



jlab-standard-chroma-build

- Need to run ‘configure’ in toplevel directory to set versions:
  - ./configure --enable-install-root=<top dir for installs>
  - --enable-qdp-version=<qdp_version_tag> or “master”
  - --enable-chroma-version=<version_tag> or “master”
  - --enable-parallel-make=<N> - argument for parallel make
- Run: ./build-git.sh <package>/<mach-type>/configuration
  - e.g. ./build_git.sh qdp++/ib-9q/parscalar-ib-9q
  - Should:
  - Download QDP++ source, run configure, make, make install using the configure.sh file in the build directory
- You still need to remember which pacakges you want

jlab-standard-chroma-build

- E.g. for Cray XT systems we used:

```
./configure --enable-parallel-make=10 \
            --enable-install-root=<my directory> \
            --enable-qdp-version=master \
            --enable-chroma-version=master

./build-git.sh qmt/cray_xt/xt-barcelona-pat \
           libxml2/cray_xt3/xt-craypat \
           qmp/cray_xt3/xt-craypat \
           qdp++/cray_xt3/parscalar-xt-qmt-pat \
           qdp++/cray_xt3/parscalar-xt-double-qmt-pat \
           chroma/cray_xt3/parscalar-xt-qmt-pat \
           chroma/cray_xt3/parscalar-xt-double-qmt-pat
```

to build single, and double precision builds

package-XXX.tarballs

- New turnkey-builds, inspired partially by qinstall
- e.g. package-quda.tar.gz
- After unzipping:

```
package-quda/
 ├── 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                  Wipe out build/installation directories
 ├── purge_install.sh
 ├── src/ ──── qmp, qdp++,chroma,quda        package sources
 ├── install/── qmp, qdp++,chroma,quda       installed packages (created)
 └── build/ ────build_qmp, build_qdp++, ...  build directories  (created)
```

package-XXX.tar.gz Tarballs

- Unlike jlab-standard-chroma-build these packages
  - already contain a copy of the source codes for the architecture
  - env.sh allows more specific customization of paths
    - e.g. execute 'module load' commands for compilers
    - expicit place for setting PATH/LD_LIBRARY_PATHs
    - specify other things: e.g. CUDA Build version (sm20 etc)
- ./build_all.sh
  - will purge the build and install directories and invoke build scripts.
- ./build_yyy.sh will build package yyy only
- does not incorporate automatic running of regerssion checks (yet)

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 Chroma

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

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
- Possible continuations
  - Tutorial 1 (running chroma)
  - Mapping Data Parallel Model onto Computers
    - CUDA and QUDA, Thrust (Mike) QDP++ details
    - QDP++ Expression Templates (me)
    - Chroma Design Patterns
    - Chroma Class Structure