Session 1: Hello world!

Bálint Joó

Jefferson Lab

INT Summer School on Lattice QCD and its Applications,

Aug 8, 2007

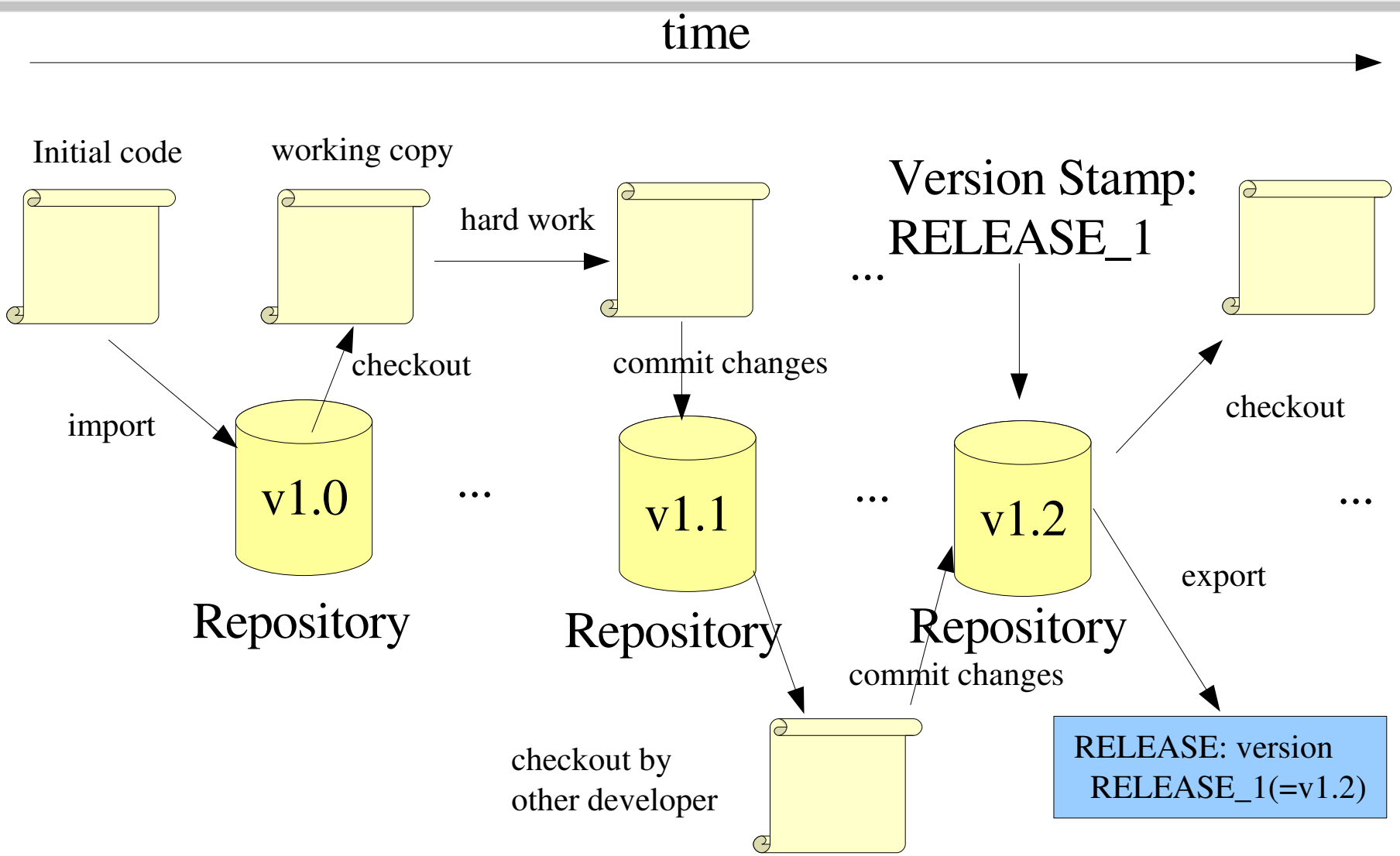
Goals

- Basic:
  - Compute the plaquette of a random configuration
- Advanced:
  - Compute a Polyakov loop on the configuration
- Topics Touched on:
  - CVS
  - 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)

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 (eg: Chroma Change Log)
    - http://usqcd.jlab.org/usqcd-software/chroma/chroma/ChangeLog.html

Revision Control In this Tutorial

- We will use CVS to check out the codes for the tutorials
- Detailed discussion of revision control systems is beyond this course, but you should read about them. Well known ones are
  - SCCS – The granddaddy of them all.
  - RCS – Awkward to use because of its locking
  - CVS – Good all arounder – We'll use this
    - Network Access, via SSE
    - Anonymous checkout via :pserver: server interface
    - poor support for changes in directory structures
  - Subversion – One of the best (free) out there currently
    - Free book at: http://svnbook.red-bean.com

Get the code

- Use CVS To Get the code

bash$ **export CVSROOT=:pserver:anonymous@cvs.jlab.org:/group/lattice/cvsroot**
bash$ **cvs checkout seattle_tut/example1**
cvs checkout: CVS password file /home/bjoo/.cvspass does not exist - creating a new file
cvs server: Updating seattle_tut/example1
U seattle_tut/example1/Makefile
U seattle_tut/example1/example1.cc
U seattle_tut/example1/example1_model.cc
cvs server: Updating seattle_tut/example1/include
U seattle_tut/example1/include/reunit.h
cvs server: Updating seattle_tut/example1/lib
U seattle_tut/example1/lib/Makefile
U seattle_tut/example1/lib/reunit.cc

Note:

`export CVSROOT=...`

is a bash-ism. For tcsh use:

`setenv CVSROOT ...`

- Backup (if CVS doesn't work: Download Tarball from:

bash$ wget http://www.jlab.org/~bjoo/Seattle/example1.tar.gz
bash$ tar xvf example1.tar.gz

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++
    - I set the install script to `$HOME/install/qdp++`
    - On my machine `$HOME` expands to

      `/home/bjoo`

    - On your machine it may expand to something else
    - Try typing `echo $HOME` into your shell
- Do this also in `seattle_tut/example1/lib/Makefile`

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
	$(CXX) -o $@ $(CXXFLAGS) $< $(LDFLAGS) $(LIBS)
```

Makefile Macros

use macros as $(macro)

Makefile Targets

Makefile Dependencies

Makefile action

TAB

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 f
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,x
     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

Collectives: alltoall (sum)/ local (trace)

QDP++ utility function

Print Result

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=0.0012776317811989 8
```

- 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.

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

Next Session: “Dances with Solvers”

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