Lecture 6: Untold Tales

Bálint Joó

Jefferson Lab

INT Summer School – Lattice QCD and its Applications. Aug 2007

Nucleons

Working from: W. Wilcox, T. Draper, K.-F. Liu, Phys. Rev. D46(1992) 1109-1122 (hep-lat/9205015) – notation changed slightly. Also they use a different (Dirac) spin basis from us.

The interpolating operators for the nucleon are:

$$\chi_\alpha(x) = \epsilon^{abc} u^a_\alpha(x) u^b_\beta(x) \tilde{C}_{\beta\gamma} d^c_\gamma(x)$$
$$\bar{\chi}_{\alpha'}(x) = -\epsilon^{a'b'c'} \bar{d}^{c'}_{\gamma'}(x) \tilde{C}_{\gamma'\beta'} \bar{u}^{b'}_{\beta'}(x) \bar{u}^{a'}_{\alpha'}(x)$$

Where:

$$\tilde{C} = C\gamma_5$$

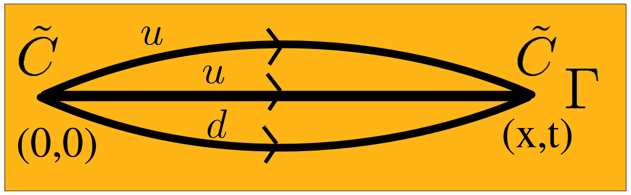
is the charge conjugation matrix. In our spin basis for QDP we have

$$C = \gamma_0\gamma_2 \equiv \texttt{Gamma(10)}$$

Two point function:

$$C(t;\vec{p},\Gamma) = \sum_{\vec{x}} e^{i\vec{p}.\vec{x}}\Gamma_{\alpha',\alpha}\langle 0|T\left\{\chi_\alpha(x)\bar{\chi}_{\alpha'}(0)\right\}|0\rangle$$

$$= \sum_{\vec{x}} e^{i\vec{p}.\vec{x}}\mathrm{Tr}\ \left[\Gamma\langle 0|T\left\{\chi(x)\bar{\chi}(0)\right\}|0\rangle\right]$$



Three quark piece

diquark piece

$$C(t;\vec{p},\Gamma) = \sum_{\vec{x}} e^{i\vec{p}.\vec{x}}\epsilon^{abc}\epsilon^{a'b'c'}$$

$$\times \left(\mathrm{Tr}\left[\Gamma U^{aa'}(x,0)\tilde{C}D^{bb'}(x,0)\tilde{C}^{-1}U^{cc'}(x,0)\right]\right.$$

$$\left. + \mathrm{Tr}\left[\Gamma U^{aa'}(x,0)\right]\mathrm{Tr}\left[\tilde{C}D^{bb'}(x,0)\tilde{C}^{-1}U^{cc'}(x,0)\right]\right)$$

The Di-Quark Piece

$$\begin{aligned}
\text{di-quark piece} &= \mathrm{Tr}\ \left[\tilde{C} D^{bb'}(x,0)\tilde{C}^{-1} U^{cc'}(x,0)\right] \\
&= \mathrm{Tr}\left[D^{bb'}(x,0)\tilde{C}^{-1} U^{cc'}(x,0)\tilde{C}\right] \\
&= \mathrm{Tr}\left[\left(\tilde{C} D^{bb'}(x,0)\right)^T \left(U^{cc'}(x,0)\tilde{C}\right)\right] \\
&= \mathrm{Tr}\left[\left(Q^{bb'}_{\alpha\beta}\right)^T R^{cc'}_{\beta\gamma}\right] \\
&= \mathrm{Tr}\left[Q^{bb'}_{\beta\alpha} R^{cc'}_{\beta\gamma}\right]
\end{aligned}$$

Having Contractions...

$$\epsilon^{abc}\epsilon^{a'b'c'}\left[Q^{aa'}_{\beta\alpha}R^{bb'}_{\beta\gamma}\right]$$

index1 index2 index3 index4

In QDP++:

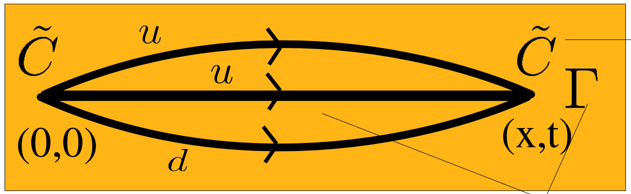
```
quarkContractXY( Q , R )
```

contracts spin indices X and Y as well as the epsilons e.g: above

```
quarkContract13( Q, R )
```

So the Full Nucleon

- Prior to Fourier Transform for momenta - from chroma:
  - file: lib/meas/hadron/baryon_w.cc



```
 LatticeComplex nucl2pt(const LatticePropagator& quark_propagator,
                        const SpinMatrix& T, const SpinMatrix& sp)
{
    LatticePropagator di_quark = quarkContract13(quark_propagator * sp,
                                                 sp * quark_propagator);
   return LatticeComplex(trace(T * traceColor(quark_propagator *
                                              traceSpin(di_quark)))
                        + trace(T * traceColor(quark_propagator * di_quark)));
}
```

di-quark piece

three quark piece

I/O

- I will discuss 2 kinds of I/O here:
  - Reading/Writing Parameters (and simple data)
  - Reading/Writing Lattice Quantities (QIO)

Reading and Writing Parameters

- It is generally useful, to not have to rewrite our main-programs every time we need to change parameters.
- Preferable to read the parameters from a file
- Parameter file can be
  - Flat: e.g.: <parameter>=<value> pairs
  - Structured:
    - somehow the parameters follow the 'structure' of the classes they parameterize...
      - Interface Description Languages like Sun RPC
      - Structured Markup:
        - » Fortran Namelists (old...)
        - » XML (currently fashionable)

A tale of two parameter files

Flat File:

```
Lx=4
Ly=4
Lz=4
Lt=8
StartType=DISORDERED
beta=5.4
Mass=0.02
MaxCG=500
RsdCG=1.0e-8
n_steps=16
traj_length=1
```

Structured Markup

```
<?xml version='1,0' encoding='UTF-8'?>
<Params>
  <lattSize>
      <Lx>4</Lx>
      <Ly>4</Ly>
      <Lz>4</Lz>
      <Lt>8</Lt>
  </lattSize>
  <config>
      <cfgType>DISORDERED</cfgType>
 </config>
 <HMC>
   <monomials>
     <wilsonGauge>
             <beta>5.4</beta>
     </wilsonGauge>
     <TwoFlavorWilsonFermion>
          <mass>0.02</mass>
          ...
```

Advantages/Disadvantages

- Advantages of Flat Files
  - Easy to write your own parser
  - Quite Dense files
- Disadvantages
  - Can be pretty inflexible
    - Suppose you add something completely new
    - Are there name clashes?
    - Do you have to introduce rules for the meanings of names?

- Advantages of Markup
  - Follow your class structure
  - More flexible if well designed
    - A new class is just a new section of markup
  - Can do flat files if you really want..
- Disadvantages
  - Quite Verbose
    - whats with the <> anyway?
  - You probably don't want to write the parser
    - force use of 3rd party libraries like libxml2

Once you go down the XML path...

- In QDP++ for various considerations, we opted to use XML
- Our experiences:
  - XML is great for parameter files
  - XML is OK for writing simple data
    - Plaquettes, HMC energies, etc.
  - XML is good in principle for writing out highly structured ouput data
    - eg: Momentum Channels, Operators, etc
    - In practice this highly structured output data was very difficult to process for analysis:
      - HUMONGOUS FILES
      - Takes a long time to extract desired subset

Reading / Processing XML in QDP++

- The main input XML handling class in QDP++ is:

  `XMLReader`

- An XMLReader will
  - read and parse an XML document
  - allow the interrogation of the document via XPath

root node → `<foo>`

from root: `/foo/bar/fred`
from foo/bar: `./fred`
→ `<fred>6</fred>`

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
   <bar>
      <fred>6</fred>
      <jim>7 8 9</jim>
  </bar>
</foo>
```

Reading/Processing XML in QDP++

- Opening and reading the file:

```
XMLReader  xml_in(“./my_xml_file.xml”);
```

- Reading Elements:

```
int fred;
multi1d<int> jim;
read( xml_in, “/foo/bar/fred”, fred);
read( xml_in, “/foo/bar/jim”, jim);
```

- Switching Context (cd – ing) :

```
XMLReader new_reader(xml_in, “/foo/bar”);
read( new_reader, “./jim”, jim);
```

Except for the Exceptions

- Suppose you ask for a file/path that does not exist?
- XMLReader will “throw a C++ exception”
  - All XML Reader exceptions are of type std::string
  - It is your responsibility to “catch” these.
  - So typically the code for XML processing will look more like:

```
int fred;
try {
        XMLReader xml_in(“./my_xml_file.xml”);
        read( xml_in, “/foo/bar/fred”, fred);
}
catch( const std::string& e) {
    QDPIO::cerr << “Caught Exception: “ << e << endl;
    QDP_abort(1);
}
```

Exercise/Homework:

- Change the qcd.cc code in Example 3 so that
  - It reads the run time parameters from a file
    - Try the following flat approach first:
    - Then try adding some structure

Remember:
The XMLReader is part of QDP++. Make sure you've called QDP_initialize() before you use it...

An XML Flat File

```
<?xml version='1,0' encoding='UTF-8'?>
<Params>
    <Lx>4</Lx>
    <Ly>4</Ly>
    <Lz>4</Lz>
    <Lt>8</Lt>
    <cfgType>DISORDERED</cfgType>
    <beta>5.4</beta>
    <mass>0.02</mass>
    <RsdCG>1.0e-7</RsdCG>
    <MaxCG>1000</MaxCG>
    <nSteps>16</nSteps>
    <trajLength>1.0</trajLength>
    <numHMCTrj>500</numHMCTrj>
</Params>
```

Writing XML

- We have an XML Writer Class XMLWriter
  - This class is abstract. There are 2 derivations:
    - **`XMLFileWriter`** - write to a file
    - **`XMLBufferWriter`** – write a documet "in memory"
  - A writer can do three things:
    - Open / Close Tags for a Group
    - Write a leaf element
    - "Paste in" the contents of an XMLReader

Writing XML

- Creating a writer

```
XMLFileWriter xml_out(“./output_file);
```

```
<?xml version='1.0'?>
```

- Open a group tag

```
push(xml_out, “foo”);          <foo>
```

- Write an element

```
Real t=Real(0.5);                  <t>0.5</t>
write(xml_out, “t”, t);
```

- Close a group tag

```
pop(xm_out);                   </foo>
```

- Close writer

```
xml_out.close();
```

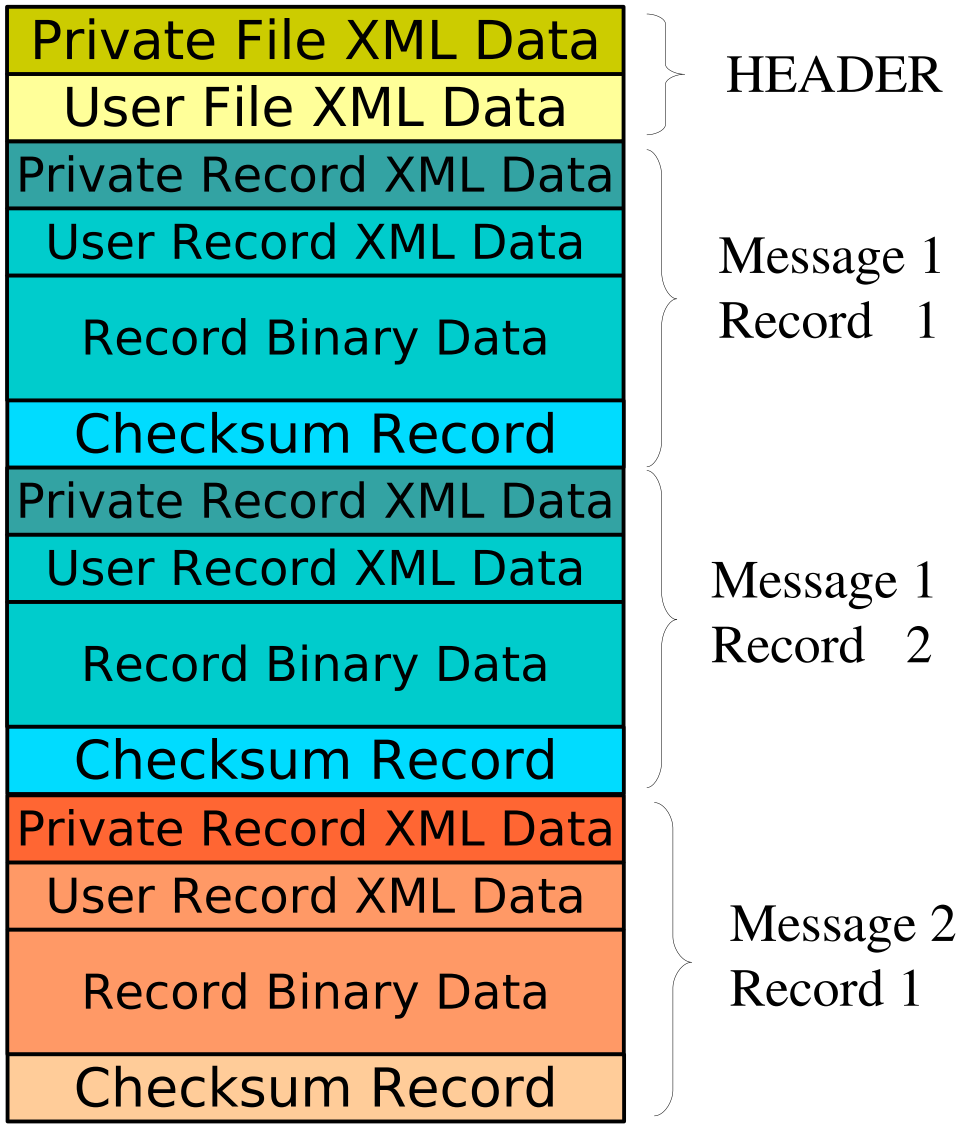
Exercise: Logging the HMC...

- Use XML to log the plaquette measurement in the HMC code from exercise 3.
- Advanced:
  – Use XML to log the energies, acceptance probabilities, etc from the HMC code.
  – Hints:
    - You may need to change the operator() in HMCTrj to allow the passing of an XMLWriter
    - Here you may consider using a global variable for the HMC log, to leave the interfaces unchanged.
    - What happens to the log if your HMC crashes? (You can simulate this by hitting Ctrl^C while the code is running. Is XML Ideal for logging?)

Reading/Writing Configurations/Props

- Clearly XML is not a good way to write out really big things
- Binary is preferable
- However, it can be useful, to attach some information to the binary to tell us about its contents
  - Lattice Size, Byte order, datatype (system stuff)
  - How the contents were computed (user stuff)
- The SciDAC software committee developed a file format to do just this
    - Along with the format came a library called QIO to manipulate the files.
    - Here I will present the QDP++ interface to QIO,

LIME



- Lime files contain:
  - A header
  - One or more messages
- Each Message contains
  - One or more Logical records
- Each record contains
  - Private XML Data
  - User XML Data
  - Binary data
- Header has
  - Private XML Data
  - User XML Data

What goes in the Records?

- Private records
  - Datatype, Byte order etc
  - YOU DON'T WRITE THESE. QIO does it automatically
- Checksum records
  - QIO Writes these.
- User records
  - You do write these
    - User XML comes from an XMLBufferWriter
    - Binary comes from your QDP++ Lattice Datatype
    - Writing non Lattice Types is possible (global data feature of QIO) but awkward...

QIO Interface

- Write with **`QDPFileWriter`**
- Must supply user file and user record XML as **`XMLBufferWriter`**

- Read with **`QDPFileReader`**
- User File XML and User Record XML returned in **`XMLReader`**
- Checksum/ILDG details checked internally

```
LatticeFermion my_lattice_fermion;
XMLBufferWriter file_metadata;
push(file_metadata, "file_metadata");
write(file_metadata, "annotation", "File Info");
pop(file_metadata);

QDPFileWriter out(file_metadata,
                  file_name,
                  QDPIO_SINGLEFILE,
                  QDPIO_SERIAL);

XMLBufferWriter record_metadata;
push(record_metadata, "record_metadata");
write(record_metadata, "annotation", "Record Info");
pop(record_metadata);
out.write( record_metadata, my_lattice_fermion);
out.close();
```

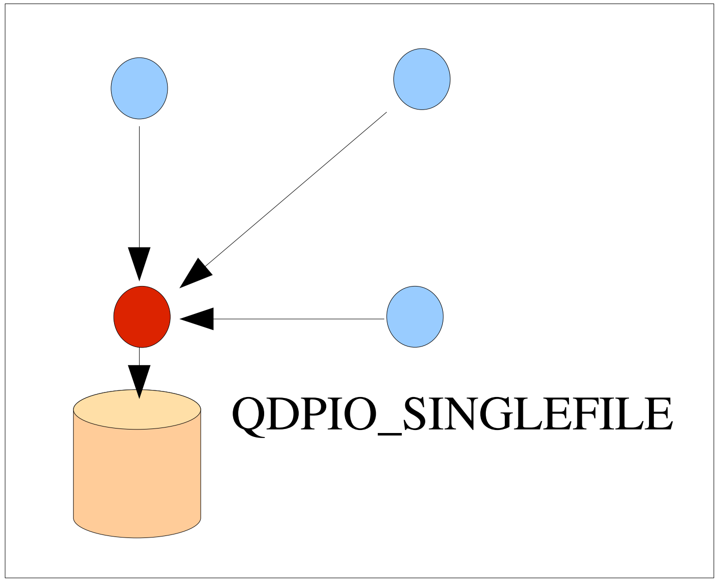
```
XMLReader file_in_xml;
XMLReader record_in_xml;
QDPFileReader in(file_in_xml,
                 file_name,
                 QDPIO_SERIAL);

LatticeFermion my_lattice_fermion;
in.read(record_in_xml,
my_lattice_fermion);
in.close();
```

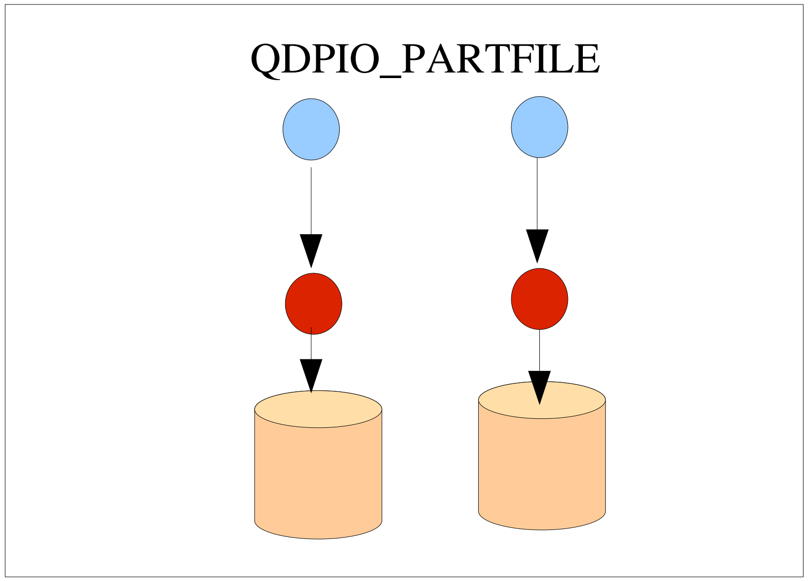
File XML

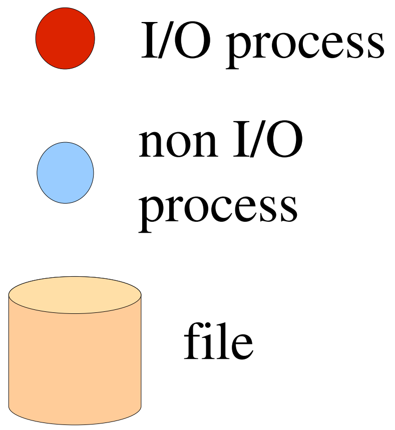
Record XML

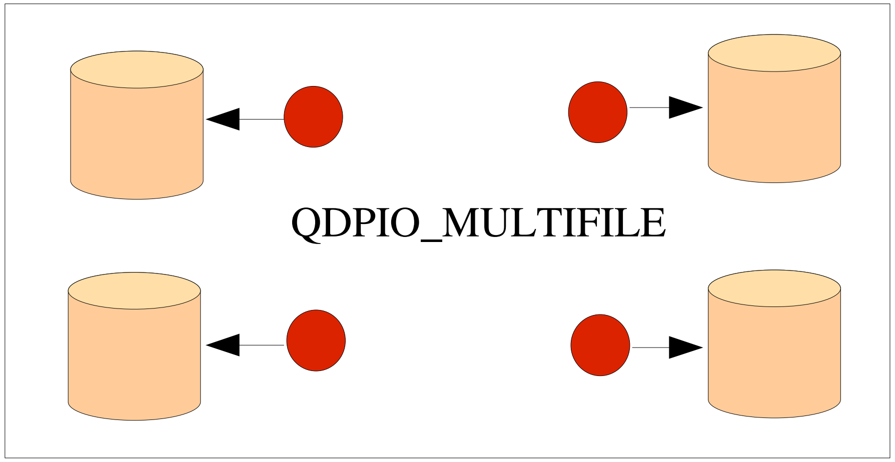
QIO Write Mode Flags

Three modes for reading/writing



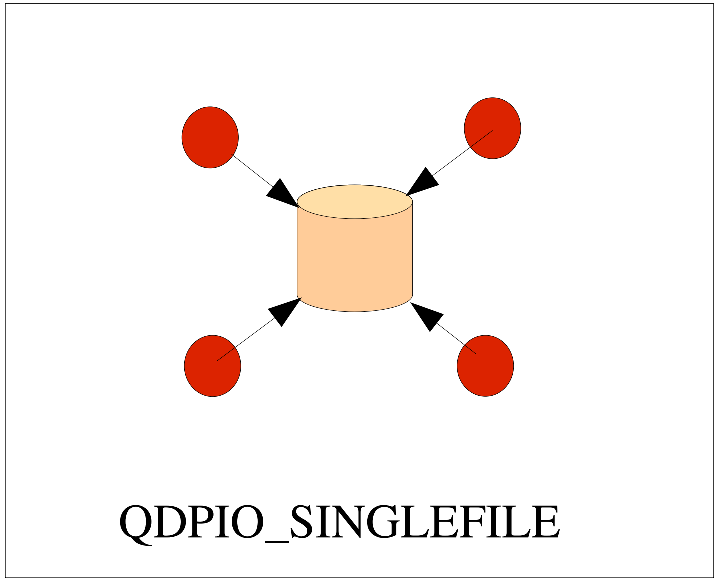


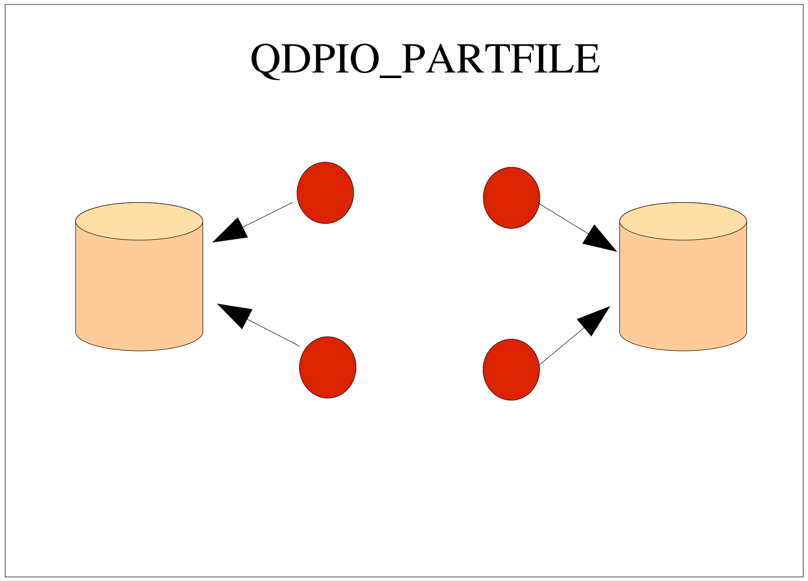




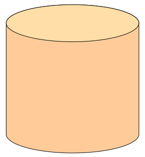
Writing/Reading patterns for QDPIO_SERIAL

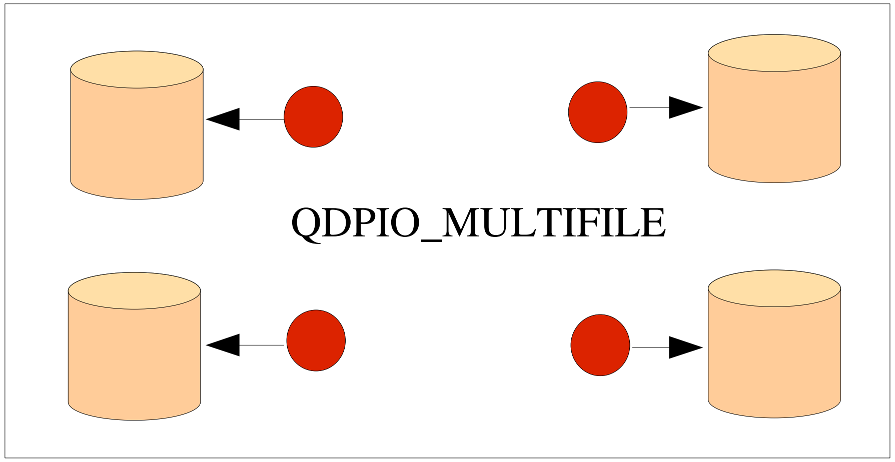
Three modes for reading/writing





● I/O process


file



Writing/Reading patterns for QDPIO_PARALLEL

QDP++ and QIO

- QDP++ Currently supports the SINGLEFILE and MULTIFILE formats for QDPIO_SERIAL
- PARTFILE has been tried occasionally but is not supported
- You can use QIO independently of QDP++ of course
- Recently the SciDAC committee has been trying to “standardize” the contents of User-Records for
  - Configurations
  - Propagators
  - BUT (thankfully) the software does not constrain you to comply with this.

Exercises/Homework:

- Change the QCD code of exercise 3 to dump the gauge configuration every now and again. You should then set up the parameter reader of the last exercise, to read the frequency of saving the configurations from the parameter file.
- Modify QCD code of exercise 3 so that it has the option for 3 kinds of gauge startup (driven from the parameter file):
  - ORDERED, DISORDERED, FROM_FILE
- Modify the propagator code of exercise 2, to read its input configurations from a file
- Generate 100 trajectories without saving, and then another 100 trajectories, saving every 10th trajectory
- Compute the pion correlator on the last 10 configs you saved

Final Words about I/O

- I guess the I/O sector of QDP++ still needs work
  - Interfacing to/testing more advanced QIO features
  - Better support for writing out non-lattice types
    - eg: Correlation functions.
  - Do we really need to write files at all???
    - Why not dump directly into some database?
      - Ah! But who will operate the database?
      - ... < Your future Technology trends go here >

“The Network is the Computer”

- Suppose you are working in a small collaboration and don't have the resources to produce your own configuration.
- Never fear! There are many configurations you can get for free.
- From
  - The Gauge Connection
  - The Brookhaven Configuration Archive
  - CP-PACS Archive
  - International Lattice Data Grid

The Gauge Connection

- Is a web based archive, based at NERSC :
  - http://qcd.nersc.gov
- You can browse the datasets and read about them, select the one you want and download it.
- You can perform mass downloads using the wget utility
- You need to register, to get a username and a password.
- Configurations come in "QCD Archive Format" a.k.a NERSC Format.
- QDP++ can read/write NERSC format through the functions:

```
void readArchiv(multi1d<LatticeColorMatrix>& u, const string& cfg_ file);
void writeArchiv(const multi1d<LatticeColorMatrix>& u, const string& cfg_file);
```

- Archive has code to read the files, in case you don't like QDP++

QCDOC Lattice Archive

- A similar archive to the NERSC one, hosted by Brookhaven National Lab, to share the Domain Wall Configurations produced by QCDOC
  - http://lattices.qcdoc.bnl.gov
- Some configurations completely freely available
- Others are restricted to members of USQCD and require a username and password
- Files are in the NERSC format

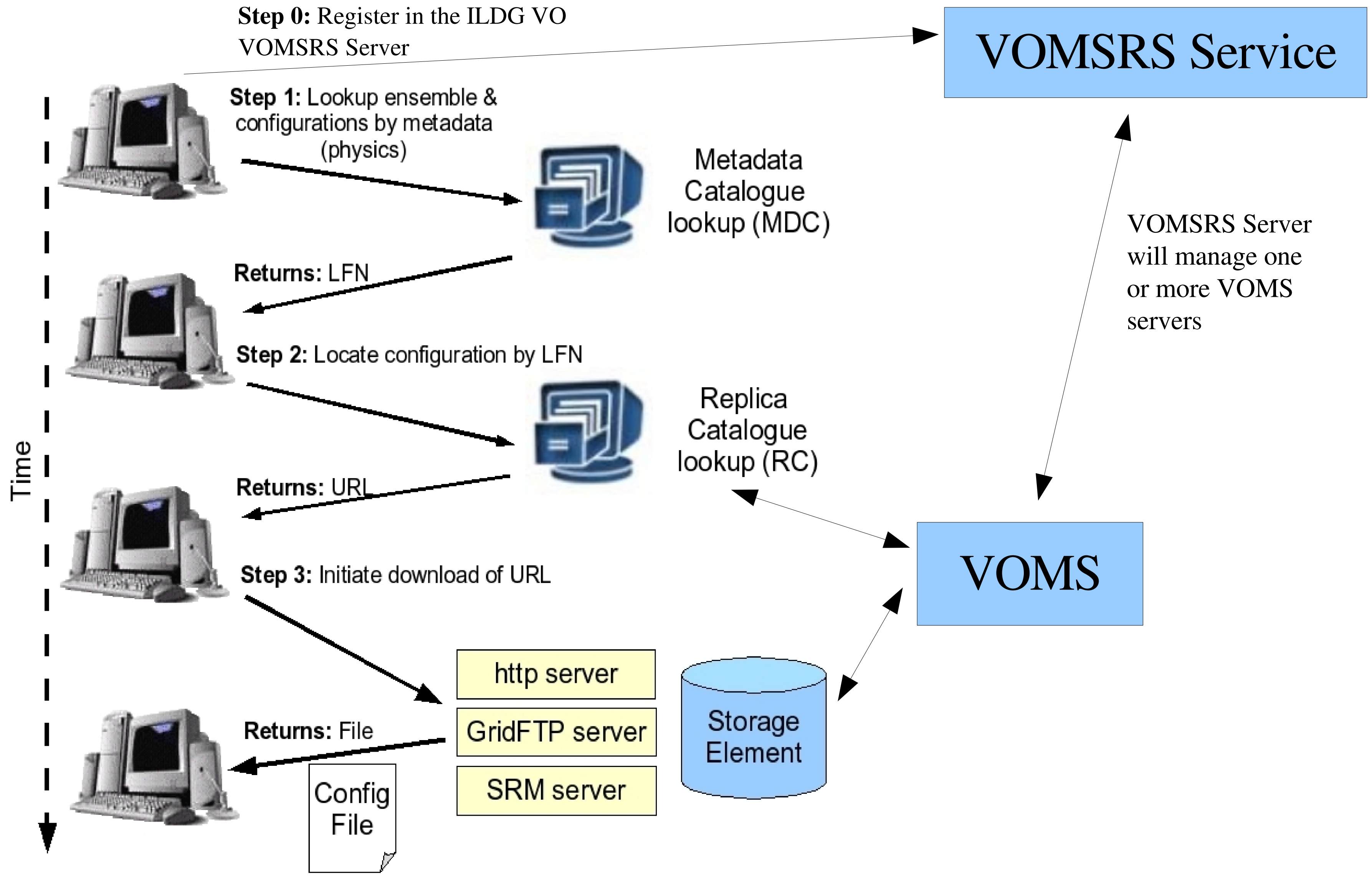
Lattice QCD Archive

- Hosted as part of the Japanese Lattice Data Grid
  - http://www.jldg.org/lqa provides a web front end
- Currently requires registration to use
- Files are formatted in the ILDG format
  - This is just the LIME format with a particular set of records. QDP++ should be able to read these.

The International Lattice Data Grid

- The International Lattice Data Grid (ILDG) is trying to define standards to make data sharing more easy (essentially it is an infrastructure group)
  - Standardized Metadata for information about configurations
  - Standardized File Format (LIME)
  - Standardized Middleware Architecture to assist:
    - Data Discovery (discovering the ensembles you want)
    - Data Location (finding the corresponding files)
    - Data Transfer (to download the data)
- Major Players: CSSM (Australia), LDG (Europe), QCDGrid/DiGS (UK), JLDG (Japan), USQCD, ...

ILDG Architecture


Step 0: Register in the ILDG VO VOMSRS Server
VOMSRS Service
Step 1: Lookup ensemble & configurations by metadata (physics)
Metadata Catalogue lookup (MDC)
VOMSRS Server will manage one or more VOMS servers
Returns: LFN
Step 2: Locate configuration by LFN
Replica Catalogue lookup (RC)
Time
Returns: URL
VOMS
Step 3: Initiate download of URL
http server
GridFTP server
SRM server
Storage Element
Returns: File
Config File

Using the ILDG

- You'll need
  - An acceptable X509 grid certificate
  - A whole bunch of Grid tools
    - LDG supplies a complete set (LTOOLS):
      - http://www-zeuthen.desy.de/latfor/ldg/doc/
- You'll need to register your Certificate in the ILDG Virtual Organisation at:
  - https://grid-voms2.desy.de:8443/vo/ildg/vomsrs
- You can browse the metadata catalogs online, via the web.
- You can download files with the ildg-get utility (in LTOOLS)

Metadata Catalogues

- Various implementations / Web portals should allow you to explore data in multiple catalogues.
  - USQCD: http://usqcd.jlab.org/mdc-web-client/index.jsp
  - LDG: http://www-zeuthen.desy.de/latfor/ldg/
  - CSSM: http://cssm.sasr.edu.au/ildg/
- Some MDCs may be slow (perhaps trying to contact non-existent databases) or 'partially functional' (in Development).

Comments on the ILDG

- Some parts of the ILDG are more mature than others
  - LDG: verbgy mature with many many configurations online. In production use.
  - Some other regional grids have not yet developed to the same extent (no names!)
  - But important milestones in interfaces and interoperability have been reached in the past year
  - We expect more maturation as time progresses.