Automatic LQCD Code Generation

Claude Tadonki

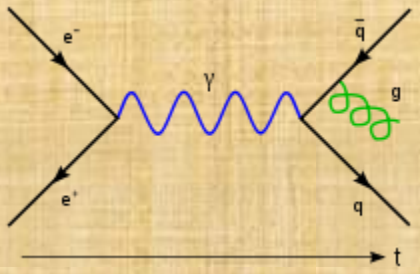
Mines ParisTech – CRI (Centre de Recherche en Informatique) **- Mathématiques et Systèmes**

LAL (Laboratoire Accélérateur Linéaire)**/IN2P3/IN2P3**

Joint work with

D. Barthou, C. Eisenbeis, G. Grosdidier, M. Kruse, L. Morin, O. Pène, and K. Petrov,

PetaQCD Project



Quatrièmes Rencontres de la Communauté Française de Compilation

December 5-7, 2011

Saint-Hippolyte (France)

Background

Quantum Chromodynamics is the theory of strong interactions, whose ambition is to explain nuclei cohesion as well as neutron and proton structure, i.e. most of the visible matter in the Universe.

The only systematic and rigorous method to solve this theory is **Lattice QCD** (**LQCD**), which can be numerically simulated on massively parallel supercomputers.

LQCD simulations are based on the *Monte Carlo* paradigm. The main ingredient of the computation is the resolution of a linear system based on the *Dirac matrix*, which is an abstract representation of the (local) *Dirac operator*.

The *Dirac operator* applied on a site x of the lattice can be expressed as follows:

$$D\psi(x) = A\psi(x) - \frac{1}{2}\sum_{\mu=0}^{4}\{ [(I_4 - \gamma_\mu) \otimes U_{x,\mu}]\psi(x+\hat{\mu}) + [(I_4 + \gamma_\mu) \otimes U^{\dagger}_{x-\hat{\mu},\mu}]\psi(x-\hat{\mu})\}$$

The *Dirac operator* involves a stencil computation, which applies on a large number of sites.

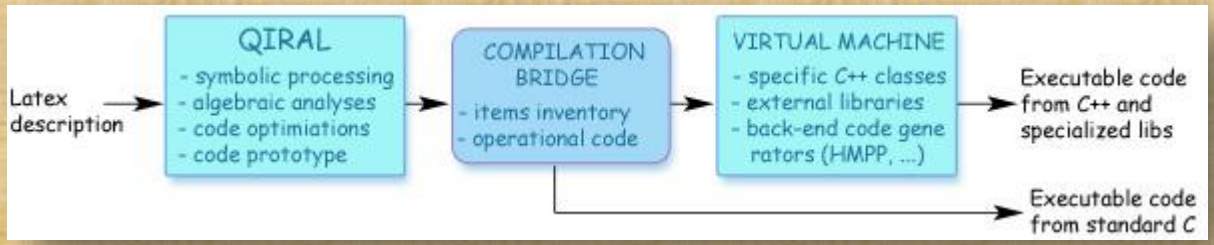
Key Computation Issues

- Large volume of data (disk / memory / network)
- Significant number of solvers iterations due to numerical intractability
- Redundant memory accesses coming from interleaving data dependencies
- Use of double precision because of accuracy need (hardware penalty)
- Misaligned data (inherent to specific data structures)
  - Exacerbates cache misses (depending on cache size)
  - Becomes a serious problem when consider accelarators
  - Leads to « false sharing » with Shared-Memory paradigm (Posix, OpenMP)
  - Padding is one solution but would dramatically increase memory requirement
- Memory/Computation compromise in data organization (e.g. gauge replication)

Why an Automatic Code Generation System

- As the formulae could be frequently changed or adapted, a push-button system to get the corresponding code is certainly comfortable.
- One application of the Dirac operator involves more than thousand floating point instructions, thus it is hard to plan low level optimization by hand.
- There are different variants of the Dirac operator and several way of expression the calculation depending on the data layout.
- There are different target architectures which can be considered. Thus, generating the code for each of them manually can be tedious and error-prone.
- One way to seek optimal implementation is to filter from an exhaustive search over all possible (or reasonable) .

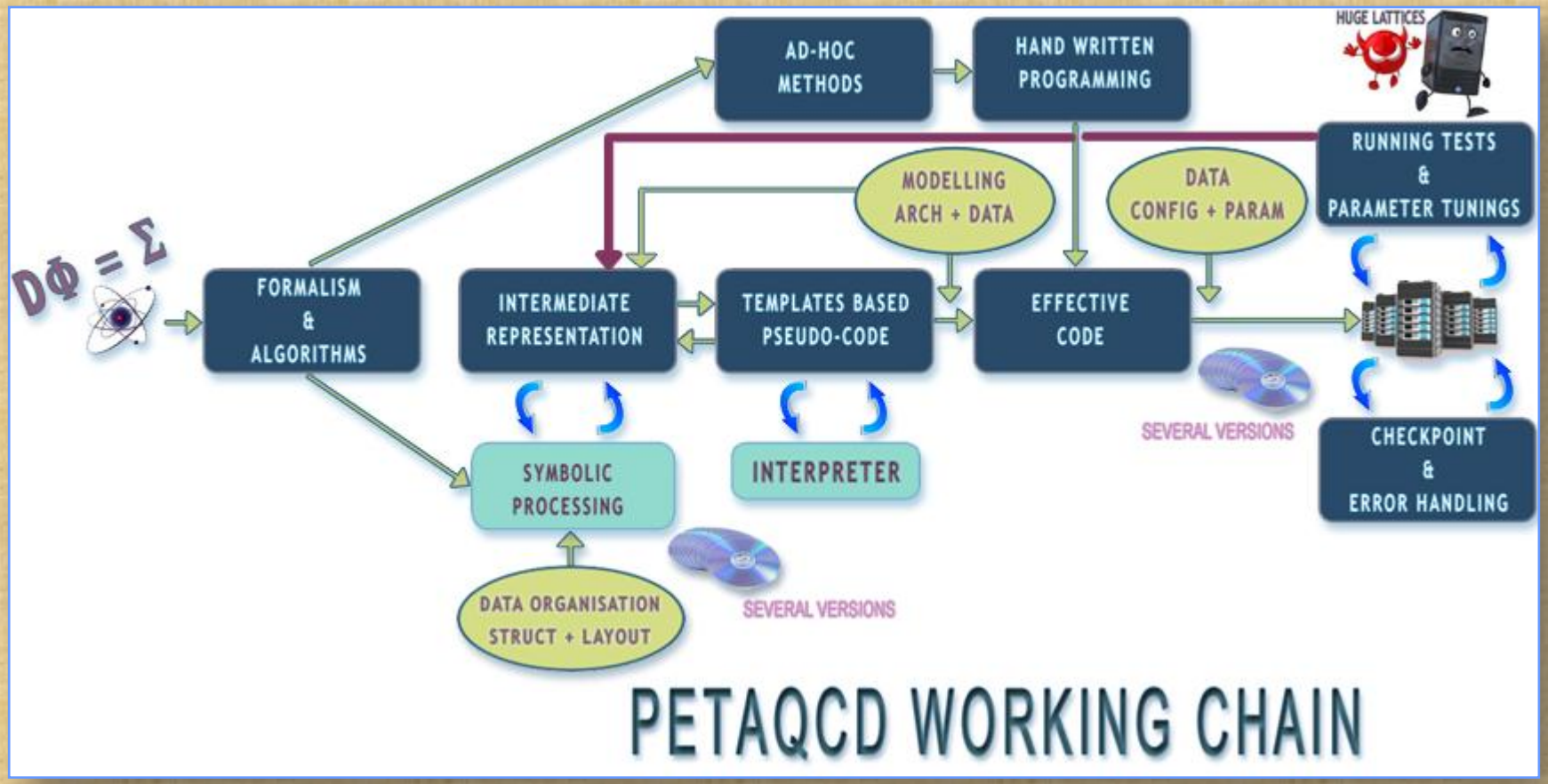
How the chain is implemented



- From an input file containing the formalism (formulae + algorithm), the QIRAL module generates a code prototype (called pseudo-code)
- QIRAL is composed with a set of rewriting rules and predefined datatypes. The input, even written in latex, has to follow some syntaxic guidelines
- Experienced user can modify the kernel of QIRAL in order for instance to enhance the rewriting rules basis or introduce new datatypes.
- The output of QIRAL (the pseudo-code) is clearly a C-like prototype, which therefore needs to be retreated in order to obtain a valid (C or C++) code.

The full working chain

https://www.petaqcd.org/chain/



The pseudo-code from Latex

PetaQCD
Automatic code generation chain
Contact: claude.tadonki@u-psud.fr

```
spinor *ID2;
spinor *ID22;
spinor *ID41;
spinor *ID45;
spinor *ID48;
spinor *ID49;
spinor *ID5;
spinor *ID6;
spinor *ID65;
spinor *ID82;
spinor *ID85;
forall(iLt = 0; iLt < Lt; iLt ++)
 {
  forall(iLz = 0; iLz < Lz; iLz ++)
   {
    forall(iLy = 0; iLy < Ly; iLy ++)
     {
      forall(iLx = 0; iLx < Lx; iLx ++)
       {
        r[iLt][iLz][iLy][iLx] = bb[iLt][iLz][iLy][iLx] ;
        p[iLt][iLz][iLy][iLx] = r[iLt][iLz][iLy][iLx] ;
        x1[iLt][iLz][iLy][iLx] = 0 ;
       }
     }
   }
 }
nr = r[Lt][Lz][Ly][Lx] . r[Lt][Lz][Ly][Lx] ;
forall(iLt = 0; iLt < Lt; iLt ++)
 {
  forall(iLz = 0; iLz < Lz; iLz ++)
   {
    forall(iLy = 0; iLy < Ly; iLy ++)
     {
      forall(iLx = 0; iLx < Lx; iLx ++)
       {
        ID2 = id(C) x (gamma5 + (i * mu * kappa * 2) * id(S)) * p[iLt][iLz][iLy][iLx] ;
        ID1 = i * ID2 ;
        ID6 = U(- (dx))[iLt][iLz][iLy][(1 + iLx)] x (gamma5 + gamma(dx) * gamma5) * p[iLt][iLz][iLy][1 + iLx] + U(-
(dx))[iLt][iLz][(1 + iLy)][iLx] x (gamma5 + gamma(dy) * gamma5) * p[iLt][iLz][1 + iLy][iLx] + U(- (dz))[iLt][(1 + iLz)]
```

Note: Copy/paste your code here and click on [proceed].

Erase | Sample input | Proceed

Output type: C operational

Sample input: Current

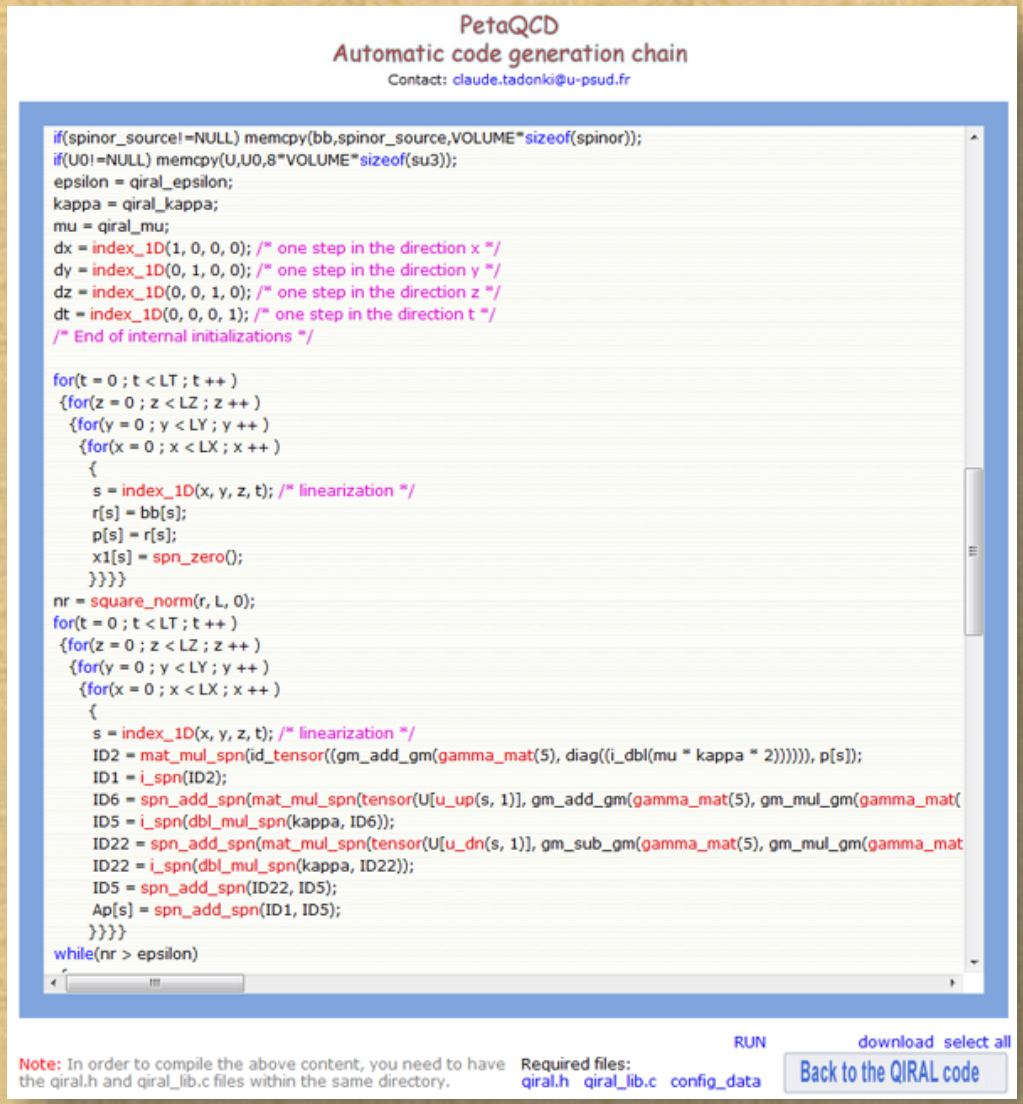
The needs from the pseudo-code to a valid C code

- the variables need to be explicitly declared (QIRAL declares part of the main ones)
- some simplification still need to be done (id(C) x id(S) = id(C x S) and id(n)*u = u)
- special statements need to be appropriately expanded (sum(d in {dx,dy,dz,dt})
- libraries calls are required for macroscopic operations (Ap[L].Ap[L]) ; (Ap[L].r[L])
- specific routines should be called for special operations like (id(S) + gamma(d))*u)
- 4D indexation should be correctly handled/matched with its 1D correspondence
- some profiling and monitoring instructions should be inserted for user convenience
- I/O routines are required for user parameters and data files
- data types need to be correctly captured for further semantic purposes

The last point is particularly important since it will determine the routines to be called (in standard C) or the method execute (C++ / ad-hoc polymorphism).

The C code from the pseudo-code

- The module is based on *Lex* and *Yacc*
- Its generated a valid C code
- The output is associated to an external library for special routines
- A C++ output is planned, will be associated with QDP++ or QUDA through a specific interface
- A web based interface is available



Running the code (mCR solver)

PetaQCD
Automatic code generation chain
Contact: claude.tadonki@u-psud.fr

```
  >> READING USER INPUT FILE AND DATA
     Input  1: qiral_Epsilon = 0.001
     Input  2: qiral_Kappa = 0.160856
     Input  3: qiral_Mu = 0.012
     Input  4: qiral_GaugeType = 0
     Input  5: qiral_GaugeFile = config_data.bin
     Input  6: qiral_SourceType = 1
     Input  7: qiral_SourceFile = _
     Input  8: qiral_LX = 4
     Input  9: qiral_LY = 4
     Input 10: qiral_LZ = 4
     Input 11: qiral_LT = 4
  >> Reading user gauge file config_data.bin
  >> Done
  >> END READING USER INPUT FILE AND DATA
Execution started
> Initialization started ...
> Initialization ended
> Reading data files ...
> Data importation ended
> User program execution started
Iteration 0 residue = 1.000000
Iteration 1 residue = 0.292781
Iteration 2 residue = 0.266481
Iteration 3 residue = 0.118587
Iteration 4 residue = 0.116612
Iteration 5 residue = 0.062480
Iteration 6 residue = 0.062210
Iteration 7 residue = 0.036884
Iteration 8 residue = 0.036866
Iteration 9 residue = 0.023275
Iteration 10 residue = 0.023259
Iteration 11 residue = 0.015474
Iteration 12 residue = 0.015471
Iteration 13 residue = 0.010680
Iteration 14 residue = 0.010679
Iteration 15 residue = 0.007670
Iteration 16 residue = 0.007668
```

download select all

Note: In order to compile the above content, you need to have the qiral.h and qiral_lib.c files within the same directory.

Required files:
qiral.h qiral_lib.c config_data

Back to the QIRAL code

Concluding remarks and peserspecives

- Try to generate the final code in one step
- Insert pragmas (OpenMP, HMPP, …) for special targets or automatic parallelization
- Add a complexity evaluation module (useful for automatic code optimization)
- Build the global system including the searching mecanism to reach the optimal code
- Generalize the framework (LQCD should not remains the main target)

THANKS FOR YOUR ATTENTION

END