QIRAL: A High Level Language for Lattice QCD Code Generation

Denis Barthou
University of Bordeaux,
Bordeaux, France
denis.barthou@labri.fr

Gilbert Grosdidier
Laboratoire de l'Accélerateur Linéaire,
Orsay, France
grosdid@in2p3.fr

Michael Kruse
INRIA Saclay / ALCHEMY,
Orsay, France
michael.kruse@inria.fr

Olivier Pene
Laboratoire de Physique Théorique,
Orsay, France
olivier.pene@th.u-psud.fr

Claude Tadonki
Mines ParisTech/CRI - Centre de Recherche en Informatique,
Fontainebleau, France
claude.tadonki@mines-paristech.fr

Abstract

Quantum chromodynamics (QCD) is the theory of subnuclear physics, aiming at modeling the strong nuclear force, which is responsible for the interactions of nuclear particles. Lattice QCD (LQCD) is the corresponding discrete formulation, widely used for simulations. The computational demand for the LQCD is tremendous. It has played a role in the history of supercomputers, and has also helped defining their future. Designing efficient LQCD codes that scale well on large (probably hybrid) supercomputers requires to express many levels of parallelism, and then to explore different algorithmic solutions. While algorithmic exploration is the key for efficient parallel codes, the process is hampered by the necessary coding effort.

We present in this paper a domain-specific language, QIRAL, for a high level expression of parallel algorithms in LQCD. Parallelism is expressed through the mathematical structure of the sparse matrices defining the problem. We show that from these expressions and from algorithmic and preconditioning formulations, a parallel code can be automatically generated. This separates algorithms and mathematical formulations for LQCD (that belong to the field of physics) from the effective orchestration of parallelism, mainly related to compilation and optimization for parallel architectures.

1 Introduction

Quantum Chromodynamics (QCD) is the theory of strong subnuclear interactions[13]. Lattice QCD (LQCD) is the numerical approach to solve QCD equations. LQCD simulations are extremely demanding in terms of computing power, and require large parallel and distributed machines. At the heart of the simulation, there is an inversion problem:

$$Ax = b, \tag{1}$$

where A is a large sparse (also sparse and implicit) matrix, called Dirac matrix, b is a known vector and x is the unknown. To model reality accurately, A should be of size around $2^{32} \times 2^{32}$ elements at least. Current simulations on supercomputers handle sizes up to $2^{24} \times 2^{24}$ elements. High performance LQCD codes on multicores, multinode and hybrid (using GPUs) architectures are quite complex to design: performance results from the interplay between the algorithms chosen to solve the inversion and the parallelism orchestration on the target architecture. Exploring the space of algorithms able to solve the inversion, such as mixed precision or communication-avoiding algorithms, aggressive preconditioning, deflation techniques or any combination of these is essential in order to reach higher levels of performance. Many hand-tuned, parallel libraries dedicated to LQCD, such as Chroma [5] or QUDA [2] propose building blocks for these algorithms, so as to ease their development. Parallelism does not stem from the algorithm themselves but from the structure of the sparse matrices involved in the computation. However, there are a number of shortcomings to the library approach: First, any significant evolution of the hardware requires to tune the library to the new architecture. This can lead to change the grain of parallelism (as an adaptation to cache sizes for instance) or change the data layout. Then, sparse matrices for LQCD, in particular the matrix A, have a very regular structure and library functions take advantage of it. Combining different algorithms, such as preconditioners, leads to structural changes in matrices that are not supported by these libraries. These two limitations hinder considerably the time necessary to develop a parallel/distributed code.

We propose in this paper a high-level Domain-Specific Language, QIRAL, to model LQCD problems and enable automatic parallel code generation. This novel approach makes possible the exploration and test of new algorithms for LQCD. Sparse matrices in LQCD can be structured using algebra operators on dense matrices and the key idea is to use this structure to express parallelism. This extends ideas proposed in SPIRAL [8], the library generator for DSP algorithms to more complex codes and matrices. The problem formulation is first presented in Section 2, then the language is described in Section 3 and its implementation in Section 4.

2 Lattice QCD Description

The principle of LQCD is to discretize space-time and describe the theory on the resulting lattice. The lattice has to contain a very large number of sites since it has to be fine grained and describe large enough volumes. LQCD exists since 1974. Huge progress in hardware, software and algorithms have been achieved, but it is not enough to really sit on the parameters of nature. The light quarks, "u" and "d" are still described by quarks heavier than in nature while the heavy "b" quark is described as lighter. This implies systematic errors in our results. To break this limitation, several orders of magnitudes will be needed in the computing power and related resources. It will demand new hardware with several level of parallelism, and make coding more and more complicated. Our goal is to provide tools helping to face this new situation.

The heaviest part in the LQCD calculation is to generate a large Monte-Carlo sample of very large files (field configurations) according to an algorithm named "Hybrid Monte-Carlo". It is a Markovian process, every step of which takes several hours on the most powerful computers. The algorithm is complex but it spends most of its time in inverting very large linear systems which depend on the field configuration. Typically one will deal with matrices with billions of lines and columns. The second heaviest part in the calculation is to compute "quark propagators" which again boils down to *solving large linear systems* of the same type[7]. Therefore we will *concentre on this task.* The matrices involved in these computation represent 4D stencil computations, where each vertex is updated by the value of its 8 neighbours. Their structure is therefore very regular, statically known, and it is used in the following section.

3 A High-Level Domain-Specific Language

QIRAL is a high level language for the description of algorithms and for the definition of matrices, vectors and equations specific to LQCD. The objective of the algorithmic part is to define algorithms independently of the expression of sparse matrices used in LQCD. The

$$
\begin{aligned}
Dirac = {} & I_{L\otimes C\otimes S} \\
& + 2 * i * \kappa * \mu * I_{L\otimes C} \otimes \gamma 5 \\
& + \kappa * \sum_{d\in D} ((J_L^{-d} \otimes I_C) * \bigoplus_{s\in L} U(d)[s]) \otimes (I_S + \gamma[d]) \\
& + \kappa * \sum_{d\in D} ((J_L^{d} \otimes I_C) * \bigoplus_{s\in L} U(-d)[s]) \otimes (I_S - \gamma[d])
\end{aligned}
$$

Figure 1: Definition of Dirac matrix on a Lattice L in QIRAL.

objective of the system of definitions and equations is to define properties and structure sparse matrices in order to be able to find parallelism. While the algorithmic part uses straightforward operational semantics, the equational part defines a rewriting system.

For the sake of simplicity, QIRAL is a subset of LaTeX, meaning that the QIRAL input can be either compiled into a *pdf* file, for rendering purposes, or compiled into executable codes. Algorithms and definitions correspond to different predefined LaTeX environments.

Variables, Types: Variables and constants used in QIRAL are vectors (denoted by the type `V`) of any length, matrices of any size (`M`), complex ($\mathbb{C}$) or real numbers ($\mathbb{R}$). Besides, counted loops are indexed by variables of type `index`, iterating over a domain (denoted `indexset`). Index variables are potentially multidimensional, and integers are a particular case of index value. Functions of any number of argument of these types can be also defined. The size of a vector is defined through the size of its index set. The size of an index set can be left undefined. If `V1` is the vector and `IS` is an index set, `V1[IS]` denotes the subvector indexed only by `IS`.

Matrices and vectors cannot be manipulated or defined element-wise. Instead, matrices and vectors are built using either constant, predefined values such as identity I_{IS} (for the identity on an index set IS), or operators such as $+, -, *$ and the transposition, conjugate, direct sum and tensor product. The tensor product $\otimes$ and direct sum $\oplus$ are defined by:

$$
A \otimes B = [a_{ij}B]_{ij}, \qquad A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}.
$$

These two operators define parallel operations and help defining the sparse matrices of LQCD.

Definition and Equations: All LQCD knowledge is given as a system of definition (for matrices and vectors) and a system of equations on these values. In particular, the structure of sparse matrices, explicitly defining where non-zero elements are located, is given in these definitions. This structure is then propagated in the algorithms. Figure 1 defines the sparse Dirac matrix. Declarations of index sets L, C, S, D are not shown here, as well as the declaration of the constant matrices used here.

The direct sums indexed by the vertices of the lattice (L) define block-diagonal matrices. These diagonals are then shifted by d or $-d$ columns by a permutation matrix J_L^d (predefined).

Algorithms, Preconditioners: Many algorithms proposed for solving Equation 1 are Krylov methods. Figure 2 presents two algorithms in QIRAL. The first is a variant of the conjugate gradient and is representative of iterative methods. The second one, Schur complement method, is a preconditioner: Computing x, solution of Equation 1 is achieved by computing two solutions to smaller problems.

The keyword `Match` helps to define the algorithm as a rewriting rule: Whenever the statement in the `Match` condition is found, it can be rewritten by the algorithm. For the `SCHUR` preconditioner, this rewriting can be performed at the condition that the requirement is fulfilled: $P_1 A P_1^t$ has to be invertible. Using identities defined through the equation system, the rewriting system proves here that indeed, the requirement is valid, when matrix A is Dirac matrix and P_1 is a projection matrix keeping only vertices of the lattice with even coordinates (even-odd preconditioning). QIRAL is expressive enough to represent different variants of con-

Algo 1: Conjugate Gradient, Normal Resolution [CGNR]

Input : $A \in M, b \in V, \epsilon \in \mathbb{R}$
Output : $x \in V$
Match : $x = A^{-1} * b$
Var : $r, p, Ap, z \in V, \alpha, \beta, n_r, n_z, n_{z1} \in \mathbb{R}$

$r = b$;
$z = A^{\dagger} * r$;
$p = z$;
$x = 0$;
$n_z = (z \mid z)$;
$n_r = (r \mid r)$;
while $(n_r > \epsilon)$ **do**
  $Ap = A * p$;
  $\alpha = n_z / (Ap \mid Ap)$;
  $x = x + \alpha * p$;
  $r = r - \alpha * Ap$;
  $z = A^{\dagger} * r$;
  $n_{z1} = (z \mid z)$;
  $\beta = n_{z1} / (n_z)$;
  $p = z + \beta * p$;
  $n_z = n_{z1}$;
  $n_r = (r \mid r)$;

Algo 2: Definition of Schur complement method [SCHUR]

Input : $A, P_1, P_2 \in M, b \in V$
Output : $x \in V$
Match : $x = A^{-1} * b$
Var : $v_1, v_2, x_1, x_2 \in V, D_{11}, D_{12}, D_{21}, D_{22} \in M$
Require : invertible$(P_1 * A * P_1^t)$

$D_{21} = P_2 * A * P_1^t$;
$D_{11} = P_1 * A * P_1^t$;
$D_{22} = P_2 * A * P_2^t$;
$D_{12} = P_1 * A * P_2^t$;
$v_1 = P_1 * b$;
$v_2 = P_2 * b$;
$x_2 = (D_{22} - D_{21} * D_{11}^{-1} * D_{12})^{-1} * (v_2 - D_{21} * D_{11}^{-1} * v_1)$;
$x_1 = D_{11}^{-1} * (v_1 - D_{12} * x_2)$;
$x = P_1^t * x_1 + P_2^t * x_2$;

Figure 2: Two algorithms. On the left: A variant of the conjugate gradient method; on the right: A preconditioner, the Schur complement method.

jugate gradient, BiCGSTAB, methods with restart (such as GCR), and other methods proposed by physicists.

4 From QIRAL to parallel code

The QIRAL compiler takes as input a file describing algorithms and equations and generates parallel code. The phases are described in 3 and presented in detail in the following.

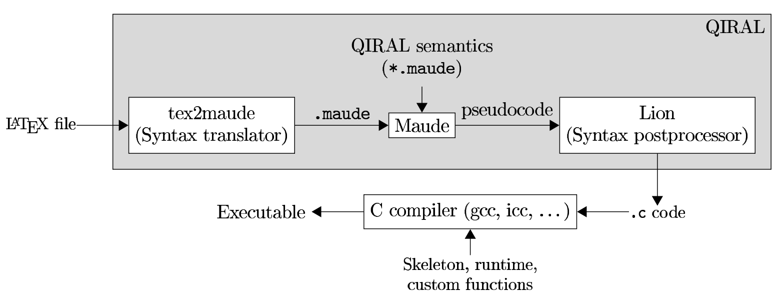

Figure 3: Overview of QIRAL compilation chain

Rewriting System Generation: From its current LaTeX implementation, the QIRAL input is translated into a set of rewriting rules and equations, using Maude framework [3]. Maude is a multi-purpose rewriting system, handling equational rewriting rules and reflection. More precisely, definitions are translated by a first phase, `tex2maude` into equations (or conditional equations) and algorithms into rewriting rules. The output file contains domain specific definitions (e.g. the Dirac operator from Figure 1), the desired program (in our case $x = Dirac^{-1}b$), and all algorithms and preconditioners we want to apply.

Definitions have a semantic similar to a rewriting rule: the left hand side is rewritten into the right hand side. To ensure the existence of a normal form, the system of equation has to be convergent and confluent. So far, our implementation does not use a tool such as an automatic Church-Rosser Checker [4] . Algorithms are translated into conditional rules, applied is their prerequisite are checked. The left hand side of the rule correspond to the `Match` clause and the algorithm itself is the right hand side.

This rewriting system is merged with another one defining general algebraic properties, code generation rewriting rules and code optimizations. The main phase of the QIRAL compiler is therefore described through rewriting system, following works such as Stratego [11]. QIRAL is a static strongly type language, vectors and matrices are defined by their index sets. Type checking is the first analysis achieved by this main phase.

Applying Algorithms and Simplifications: The user provides the list of algorithms to compose and the initial program (here, we focused on the equation $x = Dirac^{-1}b$). Exploring different algorithms and sequences of algorithms boils down to change the list given to the QIRAL compiler. Checking that algorithm requirements is automatically achieved by the rewriting system, using the equational theory provided through the system of definitions. The equational system, using both equations coming from LQCD definitions and algebraic properties involving the different operators, simplifies terms that are equal to zero.

The result is a program where the matrix A of the algorithms has been replaced either by the Dirac matrix, or by a matrix obtained through transformation by preconditioners.

Loop generation and parallelization: At this step, the initial statement has been replaced by the algorithm statements, directly using the Dirac matrix or a preconditioned version. Assignment statements are vector assignments: Values for all vertices of the lattice can be modified in one statement. Matrices are still described using tensor products and direct sums.

Loops are obtained by transforming all indexed sums, products into loops or sequences. For instance, the direct sum operator indexed by the lattice L in the definition of Dirac matrix, in Figure 1 is transformed into a parallel loop over all elements of the lattice. This loop is parallel, due to the meaning of $\oplus$. As the lattice is 4D, either 4 nested loops are created, or one single, linearized loop is created. The choice depends on a parameter in the QIRAL compiler.

Some usual compiler transformations are then used, computing dependences, fusioning loops, applying scalar promotion and other optimizations. At the end of this phase, an OpenMP code is produced, essentially by identifying for parallel loops the set of private variables. These transformations are driven by rewriting strategies, using reflection in Maude.

Matching Library Calls: The resulting code still uses some high level operators on dense matrices, such as tensor products on dense matrices, matrix-vector product. These operators are then replaced by library calls. For this step, it is sufficient to define for each library the expression it computes, as a rewriting rule. We defined LION, a set of hand-written library functions used for validation purposes.

Finally, post-processing phase, mostly syntactic rewriting, transforms this output into a C function. This function is compiled with a program skeleton and a runtime that initializes data, calls the generated code and stores the result. Custom functions can also be defined here

```
...
while(nr  > epsilon) {
 #pragma omp parallel for private(ID30,ID31,ID43,ID44,ID56)
 for(iL = 0; iL < L; iL ++) {
  ID31 = spnaddspn(matmulspn(tensor(U[uup(iL, dt)], gmsubgm(gmdiag(c1), G(dt))), p[sup(IDX, dt)]),
   spnaddspn(matmulspn(tensor(U[uup(iL, dz)], gmsubgm(gmdiag(c1), G(dz))), p[sup(IDX, dz)]),
   spnaddspn(matmulspn(tensor(U[uup(iL, dy)], gmsubgm(gmdiag(c1), G(dy))), p[sup(IDX, dy)]),
   matmulspn(tensor(U[uup(iL, dx)], gmsubgm(gmdiag(c1), G(dx))),
   p[sup(IDX, dx)])))) ;
  ID30 = cplmulspn(kappa, ID31) ;
  ID44 = spnaddspn(matmulspn(tensor(U[udn(iL, dt)], gmaddgm(gmdiag(c1), G(dt))), p[sdn(IDX, dt)]),
   spnaddspn(matmulspn(tensor(U[udn(iL, dz)], gmaddgm(gmdiag(c1), G(dz))), p[sdn(IDX, dz)]),
   spnaddspn(matmulspn(tensor(U[udn(iL, dy)], gmaddgm(gmdiag(c1), G(dy))), p[sdn(IDX, dy)]),
   matmulspn(tensor(U[udn(iL, dx)], gmaddgm(gmdiag(c1), G(dx))), p[sdn(IDX, dx)])))) ;
  ID43 = cplmulspn(kappa, ID44) ;
   ...
```

Figure 4: Sample output C code produced for CGNR algorithm

since undefined operations from the QIRAL input appear as functions calls. One application is to call BLAS routines instead of letting QIRAL implement them. For instance one defines a rule that says $(C = A * B) = \texttt{dgemm}(A, B, C)$. At this stage, we are more concerned about the correctness of the output rather than the efficiency of the code, which will be the purpose of further steps. Automatically generated codes for different algorithms and preconditionings show different convergence speed, as shown in Figure 5.

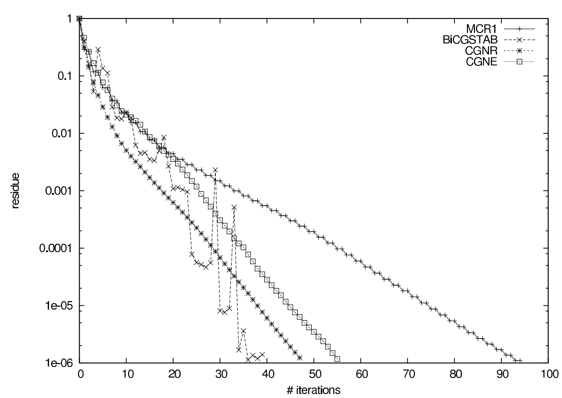

Figure 5: Convergence speed for a lattice of size 4^4 for different methods: Conjugate Gradient Normal Error (CGNE), Normal Resolution (CGNR), Biconjugate Gradient (BiCGSTAB), Modified conjugate gradient with preconditioning (MCR1).

5 Conclusion

This paper has presented an overview of QIRAL, a high level language for automatic parallel code generation of Lattice QCD codes. The language is based on algorithmic specifications and on mathematical definitions of mathematical objects used in the computation.

The contribution of this short paper is to show that a high-level representation makes possible the automatic generation of complex LQCD parallel code. Parallelism directly stems from the structure of sparse, regular matrices used in LQCD. While the initial matrix represents a stencil computation, QIRAL is able to manipulate more complex structures obtained through preconditioning for instance, unlike Pochoir [10]. The approach, similar to the one proposed by Ashby *et al.* [1] enables the user to define new equations and domain-specific definitions. The QIRAL compiler is able to keep such information through transformations resulting from preconditioners or algorithms. The tensor product and direct sum operators are translated into parallel loops and lead to OpenMP code generation. This way, algorithmic exploration, the key for higher levels of performance, can be freed from the constraints and costs of parallel tuning. Besides generation of distributed codes with communications is within reach. For heterogeneous architectures, such as Cell or GPUs, further work for automatic data-layout optimization is required, in order to reach levels of performance of previous works (such as [12, 6, 9] for the CELL BE and [2] for the GPU).

References

[1] T. Ashby, A. Kennedy, and M. O'Boyle. Cross Component Optimisation in a High Level Category-Based Language. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par Parallel Processing*, volume 3149 of *LNCS*, pages 654–661. Springer, 2004.

[2] M.A. Clark, R. Babich, K. Barros, R.C. Brower, and C. Rebbi. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):1517 – 1528, 2010.

[3] Manuel Clavel, Fransisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. The Maude System. In *Intl. Conf. on Rewriting Techniques and Applications*, pages 240–243, London, UK, 1999. Springer-Verlag.

[4] Francisco Durán and José Meseguer. A church-rosser checker tool for conditional order-sorted equational maude specifications. In *Rewriting Logic and Its Applications*, volume 6381 of *LNCS*, pages 69–85. Springer, 2010.

[5] Robert G. Edwards and Balint Joo. The Chroma Software System for Lattice QCD. *NUCL.PHYS.PROC.*, 140:832, 2005.

[6] Khaled Z. Ibrahim and Francois Bodin. Implementing Wilson-Dirac operator on the cell broadband engine. In *Intl. Conf. on Supercomputing*, pages 4–14, New York, NY, USA, 2008. ACM.

[7] Martin Lüscher. Local coherence and deflation of the low quark modes in lattice qcd. *J. of High Energy Physics*, 2007(07):081, 2007.

[8] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[9] Claude Tadonki, Gilbert Grodidier, and Olivier Pene. An efficient CELL library for lattice quantum chromodynamics. *SIGARCH Comput. Archit. News*, 38:60–65, January 2011.

[10] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. *Symp. on Parallelism in algorithms and architectures*, page 117, 2011.

[11] Eelco Visser and Zine el Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15(0):422 – 441, 1998. Rewriting Logic and its Applications.

[12] Pavlos Vranas, Matthias A. Blumrich, Dong Chen, Alan Gara, Mark Giampapa, Philip Heidelberger, Valentina Salapura, James C. Sexton, Ron Soltz, and Gyan Bhanot. Massively parallel quantum chromodynamics. *IBM J. of Research and Development*, pages 189–198, 2008.

[13] Frank Wilczek. What QCD Tells Us About Nature – and Why We Should Listen. *NUCL.PHYS.A*, 663:3, 2000.