Scalable NUMA-Aware Wilson-Dirac on Supercomputers

Claude Tadonki

Scalable NUMA-Aware Wilson-Dirac on Supercomputers

Claude Tadonki
Mines ParisTech - PSL Research University
Centre de Recherche en Informatique (CRI)
35, rue Saint-Honoré, 77305, Fontainebleau Cedex (France)
Email: claude.tadonki@mines-paristech.fr

***Abstract*—We revisit the Wilson-Dirac operator, also referred as *Dslash*, on NUMA manycore vector machines and thereby seek an efficient supercomputing implementation. Quantum ChromoDynamics (QCD) is the theory of the strong nuclear force and its discrete formalism is the so-called Lattice Quantum ChromoDynamics (LQCD). Wilson-Dirac is the major computing kernel in LQCD, where a special attention is paid to large scale simulations. The corresponding computing demand is tremendous at various levels from storage to floating-point operations, thus the crucial need for powerful supercomputers. Designing efficient LQCD codes on modern (mostly hybrid) supercomputers requires to efficiently exploit all available levels of parallelism including accelerators. Since Wilson-Dirac is a coarse-grain stencil computation performed on a huge volume of data, any performance and scalability related investigation should skillfully address memory accesses and interprocessor communication overheads. In order to lower the latter, explicit shared memory implementations should be considered at the level of a compute node, since this will lead to a less complex data communication graph and thus (at least intuitively) reduce the overall communication latency. We focus on this aspect and propose a novel efficient NUMA-aware scheduling, together with a combination of the major HPC strategies for large-scale LQCD. We reach nearly optimal performances on a single core and a significant scalability improvement on several NUMA nodes. Then, using a classical domain decomposition approach, we extend our scheduling to a large cluster of many-core nodes, thus illustrating the global efficiency of our hybrid implementation.**

I. Introduction

Quantum ChromoDynamics (QCD) [23], the theory of the strong nuclear force which is responsible for the interactions of nuclear particles, can be numerically simulated on massively parallel supercomputers using the Monte Carlo paradigm and the lattice gauge theory (LQCD) approach (see Vranas et al. [22]).

A typical LQCD simulation workflow applies basic linear algebra computations on a huge number of variables. The major LQCD kernel is the inversion of the *Dirac operator*, which is an important step during the synthesis of a statistical gauge configuration sample. Indeed, in the Hybrid Monte Carlo (HMC) algorithm [20], it appears in the expression of the *fermionic force*, used to update the momenta associated with the gauge fields along a trajectory. The *Wilson-Dirac matrix* is sparse and implicit (i.e. not in the form (a_{ij})), thus iterative solvers are the main option for its inversion. In addition, some sensitive scenarios bring up *almost null eigenvalues*, which exacerbates numerical instability and pushes far away the required number of iterations to reach convergence. Moreover, such a numerical sensitivity justifies the importance of a double precision computation. Some authors consider so-called mixed-precision [5], which sacrifices the precision of the core computation, while keeping double precision for the convergence criterion. In the presence of very small eigenvalues, thus a ill-conditioned *Wilson-Dirac matrix*, the iteration process will be likely to diverge or the way to convergence will be noticeably longer (too many iterations). Mixed precision is mainly motivated by the desire to use single precision, which yields the best performances with GPUs, and also with CPU through larger vector registers and lower memory bandwidth. However, the penalty from the loss of numerical robustness might not be affordable when its comes to sensitive LQCD scenarios like the ones related to very small pion masses. For all the aforementioned reasons, the need for efficient high-precision implementations of the *Dirac operator* is on the spotlight of both the HPC and the LQCD communities.

A common way to parallelize LQCD applications is to follow the *domain decomposition paradigm*, which means to partition the lattice into sublattices and then assign each sublattice to a computing node (see [5], [14]). This yields a standard SPMD model, which is then mapped onto a given parallel machine. Thus, providing an efficient single node implementation of kernel computations in LQCD is a valuable contribution. In case of multicore or manycore nodes, the impact of optimizing the computation following a shared memory approach is that the communication graph related to data exchanges (mostly through MPI) will be smaller with less connections. Thereby, the interprocessor communication overhead should be significantly lowered. This is very important for large scale LQCD on supercomputers, where each node has to communicate with its 8 "neighbors" (stencil computation), thus the unacceptable communication overhead usually observed in that context. Number of authors have studied LQCD implementation on various kinds of supercomputer [22], [4], [18]. However, the efficiency of LQCD frameworks on large clusters is likely to be mixed, sometimes unacceptable. The main reason is that, current and future supercomputers are potentially powerful, but all levels of parallelism need to be skillfully harnessed in order to harvest a significant fraction of this noticeable potential. In addition, memory accesses

and data exchanges, never counted on the theoretical peak performance, are dominant in LQCD computations.

For the critical case of solving *Wilson-Dirac system*, a domain decomposition approach associated with the deflation technique (related to small eigenvalues) is studied by Luscher in [14]. A mixed-precision solution accelerated with GPUs is proposed by Clark et al. [5]. A hybrid threaded-MPI approach is presented in [18] by Smelyanskiy et al. QCD implementations on the IBM-CELL are reported and discussed in [3], [9], [19], an a dedicated cluster of CELLs is presented in [15] by Pleiter. An implementation on Intel Xeon Phi by Joo et al. can be found in [10]. This panorama will be extended and detailed in the related work section.

The main argument of this paper is that, the way to get the maximum efficiency of a supercomputer is to seriously focus on the compute node and harness all performance related units and mechanisms. In addition to lower data communication overhead because of less complex interprocessor exchanges, data redundancy is also reduced by an explicit shared memory implementation on local nodes. This is the basis of our main contribution from this work, where we provide efficient strategies for *memory and data management*, *vector computing*, and *multithreading*, all illustrated by very promising experimental results. Then, we propose a novel efficient NUMA-aware scheduling in order to improve the scalability on NUMA systems. We focus on a single evaluation of the *Wilson-Dirac* operator, also called *Dslah*. Since Wilson-Dirac inversion is exclusively done through iterative approaches, making each iteration faster should certainly improve the overall performance, beside those approaches which try to reduce the number of iterations through purely numerical techniques (not our concern here). In addition to our factual achievements, this paper aims at providing a pedagogical and instructive HPC material related to high performance LQCD.

The rest of the paper is organized as follows. The next section provides fundamental LQCD background and computing considerations. Sections III discusses basic computing considerations, followed by key HPC facts related to large-scale LQCD in section IV. Related works are presented in section V. Our methodology and implementation efforts, together with performance results, are presented in section VI for the basic vector multithreaded implementation, in section VII for NUMA-aware scheduling, and in section VIII for MPI extension. Experimental results on a supercomputer are provided in section IX. Section X outlines some future works and concludes the paper.

II. LQCD Background and Computation

LQCD models the time-space universe as a four dimensional grid. In practice, a regular bounded grid is considered through a subset of $\mathbb{N}^4$, which can be represented as $\{0,1,\cdots,L_t-1\}\times\{0,1,\cdots,L_x-1\}\times\{0,1,\cdots,L_y-1\}\times\{0,1,\cdots,L_z-1\}$, where L_t, L_x, L_y, and L_z are the length of each dimension respectively. The size of the lattice for a given scenario, commonly written in the form $L_t\times L_x\times L_y\times L_z$, is somehow correlated with the underlying space density. That is why large-scale LQCD is a serious target for cutting-edge investigations in particle physics. Each point x of the lattice, commonly referred as a *site*, is connected to its eight neighbors $x\pm e_i$, $i=1,2,3,4$, where e_i are the vector of the canonical basis of $\mathbb{N}^4$, and each $\pm e_i$ operation is performed modulo L_i on the i^{th} component. This yields a regular symmetric graph.

Five 4×4 special matrices, called *Dirac γ-matrices*, are defined below

$$\gamma_0=\begin{pmatrix}0&0&-1&0\\0&0&0&-1\\-1&0&0&0\\0&-1&0&0\end{pmatrix}\quad \gamma_1=\begin{pmatrix}0&0&0&-i\\0&0&-i&0\\0&i&0&0\\i&0&0&0\end{pmatrix} \tag{1}$$

$$\gamma_2=\begin{pmatrix}0&0&0&-1\\0&0&1&0\\0&1&0&0\\-1&0&0&0\end{pmatrix}\quad \gamma_3=\begin{pmatrix}0&0&-i&0\\0&0&0&i\\i&0&0&0\\0&i&0&0\end{pmatrix} \tag{2}$$

$$\gamma_5=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&0\\0&0&0&-1\end{pmatrix} \tag{3}$$

The *Wilson-Dirac operator* can be expressed as follow:

$$\begin{aligned}D\psi(x) = A\psi(x)\ -\\ \frac{1}{2}\sum_{\mu=0}^{3}\{\ [(I_4-\gamma_\mu)\otimes U_{x,\mu}]\psi(x+e_\mu)+\\ [(I_4+\gamma_\mu)\otimes U^{\dagger}_{x-\hat{\mu},\mu}]\psi(x-e_\mu)\}\end{aligned} \tag{4}$$

where

- I_n is the identity matrix of order n,
- e_μ is the μ^{th} vector of the canonical basis of $\{0,1\}^4$,
- A is a 12×12 complex matrix of the form $\alpha I_{12}+\beta(\nu\otimes\gamma_5)$, where α,β are complex coefficients and ν a 3×3 complex matrix,
- x is a given point of the lattice (a *site*), which is a finite subset of $\mathbb{N}^4$,
- ψ is an array over the lattice (called *quark field* or *Wilson vector*), and for each site x, $\psi(x)$ is a 12-components complex vector (called *spinor*) ,
- $U_{x,\mu}$ is a 3×3 complex matrix (called *gluon field matrix*, *gauge matrix*, or *SU3 matrix*), which is associated to the link $(x,x+e_\mu)$, and also to the reverse link $(x+e_\mu,x)$,
- $\otimes$ is the matrix tensor product
- $A^\dagger=\bar{A}^T$ (i.e. transpose of the conjugate matrix)

For a given *quark field* ψ, $D\psi$ is obtained by computing $D\psi(x)$ for all sites of the lattice, and the result is also a *quark field* of the same length. Equation (4) shows that $D\psi(x)$ is a linear combination of the components of $\psi(x)$. Thus, it is consistent to see $D\psi$ as a matrix-vector product, and thereby consider D as an implicit square matrix (the *Wilson-Dirac matrix*). This matrix-product, sometimes referred in the literature as *Wilson Dslash operation* (WD), is the most time consuming kernel, as it involves a significant amount of floating point operations on large lattices and is performed very frequently.

Solving a linear system, whose the principal matrix is the *Wilson-Dirac matrix* is an important LQCD operation that is performed several times along a trajectory. Since the *Wilson-Dirac matrix* is implicit and sparse from its specification, iterative solvers are so far the only considered approaches to solve the corresponding linear system, called the *Wilson-Dirac equation*. In addition, for some specific but important physics parameters, the matrix is ill-conditioned, which severely increases the number of iterations to reach an acceptable level of convergence. This noticeable repetition of the *Dslash* operation clearly justifies the need for very fast implementations.

III. Basic and Typical Computing Considerations

There are number of fundamentals to know or consider when it comes to LQCD programs. We describe some of them.

A. Data complexity

All data structures are based on the type *complex*, which means a structure composed with two floating point numbers. Then, all arithmetic operations follow their corresponding specifications on complex numbers. The update of one spinor involves eight input spinors and the eight SU(3) matrices of the corresponding links. This yields, for the computation of one spinor, a volume of data (in bytes) given by

$$8(12 \times 2 \times p + 9 \times 2 \times p) = 336p, \tag{5}$$

where p is the size of the actual floating point number, which is typically 4 bytes (resp. 8 bytes) for single (resp. double) precision, $2 \times p$ stands for the derived complex type. We will later see that the choice between single precision and double precision is not only a matter of volume. For a given lattice of size $L = L_t \times L_x \times L_y \times L_z$, we see that $336 \times L \times p$ bytes of data will be moving within the computing system. The PetaQCD project [1], for instance, was targeting 256×128^3 double precision simulations, which means $336 \times 256 \times 128^3 \times 8$ bytes = 1.45×10^{12} bytes = 1.45 Terabytes of effective data transfer at various levels. This aspect sometimes appears as the main reason for using large clusters, since the aggregation of available (distributed) memories should be sufficient to house all working data.

B. Organization of the Computation

Computing *Wilson-Dslash* is typically done by visiting the whole lattice while updating the corresponding spinor at each site. This yields one dependence-free main loop, whose the body implements equation (4). The effective scheduling of this main loop, if different from the natural 4D lexicographic order, should be managed with the aim of addressing explicit data reuse or content sharing among the caches, following a skillful analysis of the data dependence graph. Unfortunately, the coarse granularity of the computation makes the potential of this effort rather marginal in practice, unless it goes along with an explicit mechanism that implements a specific data management strategy [18].

Concerning the calculation of $D\psi(x)$ following equation (4), some factorizations should be applied in order to put in common the major floating point operations. Indeed, let first observe that a relation of the form $v = (I_4 - s\gamma_\mu)u$, where u, v are two 4-components complex vectors and $s = \pm 1$, can be computed as follows (for $s = 1$, the case $s = -1$ is similar):

$\mu = 0$	$\mu = 1$
$v_1 = u_1 + u_3$	$v_1 = u_1 + iu_4$
$v_2 = u_2 + u_4$	$v_2 = u_2 + iu_3$
$v_3 = v_1$	$v_3 = -iv_2$
$v_4 = v_2$	$v_4 = -iv_1$
$\mu = 2$	$\mu = 3$
$v_1 = u_1 + u_4$	$v_1 = u_1 + iu_3$
$v_2 = u_2 - u_3$	$v_2 = u_2 - iu_4$
$v_3 = -v_2$	$v_3 = -iv_1$
$v_4 = v_1$	$v_4 = iv_2$

(6)

We see from (6) that only the first two components v_1 and v_2 need to be computed, and afterwards the remaining two components v_3 and v_4 are derived by considering a factor in $\{1, -1, i, -i\}$. This saving is the major computing benefit of scheme (6), especially when a matrix-vector product is involved in between as we are going to illustrate in our main calculation. Considering the so-called *normal factors decomposition* of the tensor product and the associated commutativity [21], we get

$$(I_4 - s\gamma_\mu) \otimes U = ((I_4 - s\gamma_\mu) \otimes I_3)(I_4 \otimes U) \tag{7}$$

$$= (I_4 \otimes U)((I_4 - s\gamma_\mu) \otimes I_3) \tag{8}$$

In (4), each term of the form $[(I_4 - s\gamma_\mu) \otimes U]\psi(x)$ becomes

$$[(I_4 - s\gamma_\mu) \otimes U]\psi(x) = (I_4 \otimes U)\overbrace{((I_4 - s\gamma_\mu) \otimes I_3)\psi(x)} \tag{9}$$

By (virtually) block partitioning the 12-components vector $\psi(x)$ (we use ψ for simplicity) into 3-components vectors (sometimes referred as *su3 vectors*) $\psi^{(k)} = (\psi_k, \psi_{k+1}, \psi_{k+2})$, for $k = 1, \cdots, 4$, we get a 4-components block vector expressed by $\tilde{\psi} = (\psi^{(1)}, \psi^{(2)}, \psi^{(3)}, \psi^{(4)})$. Using this block representation, we get $((I_4 - s\gamma_\mu) \otimes I_3)\psi = (I_4 - s\gamma_\mu)\tilde{\psi}$, which can then be calculated using (6), provided we replace u_k by $\psi^{(k)}$. Having thereby evaluated $\omega = (I_4 - s\gamma_\mu)\tilde{\psi}$, we finally have to compute $\varphi = (I_4 \otimes U)\omega$, which is equivalent to $(\varphi^{(1)}, \varphi^{(2)}, \varphi^{(3)}, \varphi^{(4)}) = (U\omega^{(1)}, U\omega^{(2)}, U\omega^{(3)}, U\omega^{(4)})$.

C. Even-odd Partitioning

Noticing that the update of a given site (t, x, y, z) involves eight sites $(t, x, y, z) \pm e_i$, $i = 0, 1, 2, 3$, we see that summing up all the components of two dependent sites respectively yields a difference of 1. Therefore, we might think of partitioning the lattice into two subsets, based on the parity of the sum of their components, thus the *odd* (resp. *even*) sublattice. The main advantage of this partitioning is that data dependencies are only between the odd sublattice and the even sublattice, with the gauge matrices seated on the corresponding links. This organization simplifies the macroscopic data exchanges and improves read/write data locality. There is less attention in the literature for gauge matrices sharing, we address this in our work as we will see.

D. Gauge Matrices Storage and Management

There is one gauge matrix per link in the lattice, thus $4L$ gauge matrices for a lattice of length L, since each site has 8 neighbors and the graph is symmetric. This yields a huge amount of data that need to be stored and managed efficiently since there is poor reuse of gauge matrices. Indeed, each of them is used only two times (one time in some cases), whereas each spinor is used eight times. Thus, gauge matrices are serious source of (compulsory) cache misses, waste of memory bandwidth and cache pollution. The later is due to the fact a gauge matrix, whose size is $9 \times 2 \times \{4, 8\}$, does not fully covers typical 64-bits cache lines. In order to reduce the impact of this, and maybe to simplify the indexing, the so-called *gauge copy* is applied. The idea is to store contiguously the 8 gauge matrices of each site, which explicitly doubles the volume of data but yields a significant (memory) performance improvement. Moreover, since memory accesses are dominant in any case, the so-called *SU(3)-reconstruct* or *2-rows gauge field compression* [5] might be considered. Indeed, the third row of a SU(3) matrix can be reconstructed (on the fly) by taking the complex conjugate of the *cross product* of the first two rows (i.e. $u_3 = \overline{u_1 \wedge u_2}$).

E. Important Numerical Aspects

High-precision LQCD simulations require a special attention regarding numerical issues, we point out two of them. First, the *reversibility* property, which can be seen as a kind of *numerical determinism*, aims at ensuring that the calculations made along a trajectory are predictable, and the consistency of the computed results remains whether flowing forward or backward. Thus, every computation scheme should preserve this reversibility, which restriction might prevent from considering whatever efficient but too specific or "black-box"-like subroutines.

For several reasons including the reversibility and the quality of the results for better estimates of the targeted physical quantities, the need for highly accurate calculations is relevant, thus the use of *double precision* computations, which is strictly the case for our investigations in this paper. The temptation of *single precision* looks strong, as it reduces (by half) the volume of data and leads to higher processor performance as we will detailed later. A mixed precision [5] approach might be an acceptable compromise.

As previously mentioned, for some particular physics parameters, the *Wilson-Dirac* matrix is known to have *almost null eigenvalues*, which seriously complicates its inversion. The case would be certainly worst with single or mixed precision. Thus, using double precision (or higher if possible), even if more computationally challenging, is the price for robustness, accuracy and stability.

IV. Key HPC facts related to Large-scale LQCD

Here we point out a number of important facts that should be carefully considered in order to harvest an increasing fraction of the available computing power.

Let start by pointing this performance of 0.5 GFlops/core reported by G. Grosdidier [8] when running tmLQCD [11] on 10,000 cores of the CURIE-FAT machine [7]. The machine is based on Xeon X7560 8C 2.26GHz processor, thus a peak of 9 GFlops per core. We then see that each core is running at 5% if its theoretical peak performance, which is unacceptable.

Among the reasons why large-scale LQCD might show some inefficiencies with standard codes, first there is a lack of low-level parallelism, which thereby reduces the theoretical performance expectation by a factor 4, since most of modern processors now have at least 256-bit vector registers (4 double precision components).

Memory performance is also a bottleneck. Indeed, as we have previously explained, computing *Wilson Dslash* implies a noticeable memory activity with lot of redundant accesses and waste of memory bandwidth. Indeed, the volume of a single spinor (resp. SU(3) matrix) is 192 bytes (resp. 144 bytes). Thus, regarding the L1 cache with its typical 64-bytes cache line, there is no waste coming from spinors use since each of them perfectly fits into 3 cache lines, whereas for SU(3) matrices there is a waste of $192 - 144 = 48$ bytes per access (unless we are in the *gauge copy* mode). Considering other levels of the cache, which implies wider cache lines for some architectures, the situation gets worse. We later explain how our data packing, primarily designed for vector computing, also improves the memory efficiency. Another memory issue is *cache pollution*. Indeed, SU(3) matrices, which are heavily loaded during the computation, have a poor or no reuse. This is not the case, at least by specification, for the spinors, since each of them is used to compute 8 spinors. The benefit from this spinor reuse is likely to be hampered by the aforementioned SU(3) pollution.

Another important source of performance penalty is the interprocessor communication overhead when running on distributed memory parallel machines. Indeed, in addition to the natural cost of data transfers, there is a strong gap between the ideal 8D-torus topology required for LQCD computations and the physical topology of existing supercomputers. Moreover, most of the time, there is less attention in providing a suitable virtual topology that will reduce this gap. Hybrid implementations are certainly a relevant approach to reduce the need for explicit data exchanges, but this requires to have an efficient intranode implementation, which is the essence of this paper. With the advent of multi-socket processors, thus with a significant number of cores, designing efficient scalable LQCD code is challenging because of NUMA side effects, whose illustrative case studies can be found in [12], [13].

V. Related Work

LQCD is a major in both QCD and HPC communities. For the reasons previously explained, LQCD simulations can be computationally challenging for some interesting scenarios. Thus, this hot topic is so far being investigated in various directions.

The basis of LQCD computation are explained by Luscher in [14]. The paper also discussed the so-called *delfation technique*, whose main aim is to overcome the hindering

numerical impact of almost null eigenvalues. Urbach describes in [20] the hybrid Monte-Carlo algorithm with multiple time scale integration and mass preconditioning.

General implementations and experimentations on large computing clusters are discussed by Vranas in [22], and also by Grosdidier [8] within the scope of the PETAQCD project [1]. In [15], Pleiter presents the QPACE cluster based on IBM PowerXCell 8i and dedicated to LQCD. A hybrid threaded-MPI approach on multi-core based parallel systems is studied by Smelyanskiy et al. in [18]. On-chip multiprocessing for LQCD is studied by Bilardi et al. in [4].

Accelerators-based solutions are provided for the IBM CELL by Belletti et al. [3], Ibrahim and Bodin [9], and Tadonki et al. [19]. The case of GPUs is studied by Clark et al. in [5], where a mixed precision is considered and analyzed.

A complete and operational LQCD framework named `tmLQCD` is provided by Urbach in [11]. Since LQCD computation kernels are built up from basic linear algebra routines with special data structures, dedicated computing libraries are released for generic use like QDP++ [16], which provides a data-parallel programming environment suitable for Lattice QCD, and Chroma [6], an open source LQCD toolbox.

A systematic DSL code generation approach is provided by Barthou et al. in [2]. The corresponding framework, named QIRAL, provides a high level language for LQCD code generation together with the associated engine.

In this work, we explore all levels of parallelism in order to derive an efficient high-performance implementation for large-scale LQCD scenarios. The major novelty of our intranode parallelisation is a NUMA aware scheduling, which yields a significant impact on scalability across several nodes. We now describe the main steps of our achievement.

VI. Optimal Multithread Vector Implementation

In this section, we provide a synthetic view of the main techniques that we used to design our basic implementation on a multicore vector machine. Most of these techniques are mentioned in the (general HPC of specific) literature, our merit here is to have skillfully combined them into an efficient single implementation, then illustrate and discuss their effects.

A. Gauge Matrices Management

We consider the *gauge copy* organization together with the *2-row gauge field compression*, thus our data structure for SU(3) matrices does no longer include the third row, which is then reconstructed on the fly whenever needed. While the first strategy helps for data alignment, the second saves memory bandwidth and reduces cache pollution. Indeed, there is no data reuse with the SU(3) matrices in the *gauge copy* configuration, so the less we load the best. In addition, contention on memory buses is also thereby reduced, which acts in favor of a better performance scalability.

B. Dynamic AoS to SoA for Efficient Vectorization

AoS-SoA is a well-know data layout transformation aiming at creating regular data accesses depending on the target and the computation paradigm. Vector computing is the typical beneficiary for this approach, since data to be processed should be prepared accordingly for vector accesses. In our case, considering double precision and 256-bit-wide vector registers, we just replace the original complex data type
typedef struct { *double re, im;* } *complex;*
by
typedef struct { *__m256d re, im;* } *complex_simd;*
within all of our original data structures. This is followed by a vector implementation (using AVX intrinsics) of our linear algebra kernels. Now comes the explicit data shuffling that is needed in order to have the vector operands ready for the computation. For instance, if we consider four spinor structures $s^{(j)} = [s_1^{(j)}, s_2^{(j)}, \cdots]$, $j = 1, 2, 3, 4$, the corresponding vector structure would be $s = [s_1^{(1)} s_1^{(2)} s_1^{(3)} s_1^{(4)}, s_2^{(1)} s_2^{(2)} s_2^{(3)} s_2^{(4)}, \cdots]$. The spinors $s^{(j)}$ are not required to be consecutive in memory, thus one might expect a penalty from the extra memory cost due to this dynamic packing (at load time) and unpacking (at store time). The stencil nature of LQCD computation exacerbates this fear. Since SU(3) matrices are constant, they are packed once before the computation, thus no extra cost should be considered at runtime. For the spinors, there is no choice other than doing it on the fly.

C. Threads Design and Management

The way the threads are managed is very important in a context where they are potentially numerous. Common issues will act on scalability or global performance degradation due to the cumulative overhead of boarding effects. In order to preclude some of the predictable hindering effects, we apply the following on our *Pthread* implementation:

- **Pool of tasks:** Instead of a static distribution, we create a pool of tasks and let the threads dynamically provision themselves according to their respective throughput. There are several reasons which can create execution time unbalance among threads, especially when there are several of them operating on the same arrays with interleaved data dependencies. Cutting the main loop into a large number of small chunks (tasks) restores a reasonable balance, hence improves scalability. Our set of threads is grouped following the *even-odd partitioning*, each group is assigned to a partition and has its own mutual exclusion mechanism. This prevents unnecessary interference among groups, since the partitions are computationally independent.
- **Active threads:** Since *Wilson-Dirac* is repetitively computed within an iterative process, it sounds better to create the working threads once just let them idle between two consecutive *Wilson-Dirac* computation requests. We do this by means of synchronization mechanisms (signaling mechanism could be used too). In any case, we get ride of the overhead of threads creation and destruction, which is significant in a performance sensitive context like ours.
- **Explicit threads binding and hyperthreading:** Although the operating system has its standard and likely satisfactory way of managing the allocation of threads

into the CPU-cores, which is not always a one-to-one correspondence and might trigger some dynamic thread migrations, explicit threads binding is required when a specific scheduling applies. This combines well with the pool of tasks organization in order to yield a relatively good load balance among the CPU-cores. This is noticeably crucial in a NUMA-aware context as we will see. Still considering thread binding, we choose to allocate two threads on each CPU-core, in the sole intent of getting some benefit from hyperthreading. We use the explicit allocation routines provided by the *Pthread* library.

- **Tasks assignment:** Each thread is assigned to a fixed set of the *even-odd partition*, then it uses the corresponding *id* to select associated tasks from the main pool. So, a given thread gets executed on a fix CPU-core and operates on a fix set of the *even-odd partitioning*. Once again, this strategy will be much important in the NUMA-aware context. In this case, the main benefit is a good high-level cache sharing regarding the inputs (only spinors).

We now show and discuss our performance results at the current stage on a NUMA manycore machine.

D. Basic Vector-Multithread Performance Results

We consider the recently released Intel Broadwell-based configuration described in figure Fig. 1.

Hardware

CPU Name:	Intel Xeon E5-2699 v4
CPU Characteristics:	Intel Turbo Boost Technology up to 3.60 GHz
CPU MHz:	2200
FPU:	Integrated
CPU(s) enabled:	44 cores, 2 chips, 22 cores/chip, 2 threads/core
CPU(s) orderable:	1,2 chip
Primary Cache:	32 KB I + 32 KB D on chip per core
Secondary Cache:	256 KB I+D on chip per core
L3 Cache:	55 MB I+D on chip per chip
Other Cache:	None
Memory:	256 GB (16 x 16 GB 2Rx4 PC4-2400T)
Disk Subsystem:	1 x SATA, 500 GB, 7200 RPM
Other Hardware:	None

Fig. 1. Our Intel Broadwell Charateristics

Considering the available 256-bit-wide vector registers and the corresponding vector processing capability using AVX2, we obtain a peak performance of $2.2 \times 4 = 8.8$ GFlops/core in double precision, which might be higher with the Turbo Boost. We do not count FMA or double FPU, as any of these two features are considered in our implementation.

The `numactl --hardware` command gives the information displayed in figure Fig. 2.

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10
node 0 size: 32651 MB
node 0 free: 13981 MB
node 1 cpus: 11 12 13 14 15 16 17 18 19 20 21
node 1 size: 32768 MB
node 1 free: 21098 MB
node 2 cpus: 22 23 24 25 26 27 28 29 30 31 32
node 2 size: 32768 MB
node 2 free: 24365 MB
node 3 cpus: 33 34 35 36 37 38 39 40 41 42 43
node 3 size: 32768 MB
node 3 free: 20300 MB
node distances:
node   0   1   2   3
  0:  10  11  21  21
  1:  11  10  21  21
  2:  21  21  10  11
  3:  21  21  11  10
```

Fig. 2. NUMA Specifications of our Machine

From this NUMA configuration overview, we see that we have 4 NUMA nodes grouped into 2 sockets. Considering a 32×16^3 lattice (thus numbers of cores that are multiple of 8), we obtain the performances displayed in table TABLE I. We remind the reader that each CPU-core runs 2 threads as previously explained and we use up to 8 cores within a NUMA node.

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.02552	9.98	1
2	4	0.01301	19.59	1.96
4	8	0.00679	37.50	3.76
8	16	0.00475	53.60	5.37
(2 nodes) 16	32	0.00476	53.53	5.36
(4 nodes) 32	64	0.00507	50.25	5.03

TABLE I
NUMA-UNAWARE VECTOR MULTITHREADED PERFORMANCES

We clearly see an optimal performance on a single core and a good scalability within one NUMA node (up to 8 cores). However, we can also observe a clear cut with the relative performances on several NUMA nodes. Indeed, the penalty of NUMA-unaware accesses, which are noticeably interleaved with *Wilson-Dirac*, are so important that it compensates the good performances on a single node. This yields a performance stagnation as we can see from the last two rows of table TABLE I. We now explain our method to overcome this severe scalability issue.

VII. NUMA-AWARE SCHEDULING AND IMPLEMENTATION

The first thought that comes in mind to address NUMA caprices is to replicate the inputs in all NUMA nodes. In our case, whatever it is implemented, the cost of the replication, even strictly restricted to the necessary data, appears to cost more time than the full execution of the NUMA-unaware implementation. So, this should be simply forgotten. Interleaved memory allocations is also a reasonable approach, but it efficiency heavily relies on favourable statistics. We now describe our explicitly NUMA-aware allocation strategy and explain how it clearly addresses contention and remote accesses in an efficient way.

A. NUMA-aware Allocation and Scheduling

We start by the following notation related to the *even-odd* partitioning.

$$P_k = \{(t, x, y, z) : (t + x + y + z) \bmod 2 = k\}. \quad (10)$$

Let first consider the case with 2 NUMA nodes. Since the dependence vectors are $(\pm 1, 0, 0, 0)$, $(0, \pm 1, 0, 0)$, $(0, 0, \pm 1, 0)$, and $(0, 0, 0, \pm 1)$, it follows that P_0 (resp. P_1) is computed using P_1 (resp. P_2). Therefore, we propose the data and tasks allocation illustrated in figure Fig. 3.

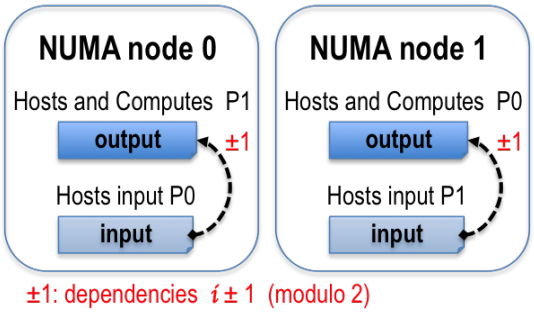

Fig. 3. NUMA-aware Scheduling with 2 Nodes

We obtain a configuration where there is no (compulsory) remote access. Now let take the case with 4 NUMA nodes. Our idea is to consider an extension of the *even-odd* partitioning to a 4-sets partition, so

$$P_k = \{(t, x, y, z) : (t + x + y + z) \bmod 4 = k\}. \quad (11)$$

From the ± 1 dependencies, we get that P_k requires $P_{k\pm 1}$, $k = 0, 1, 2, 3$ (± 1 is performed modulo 4). So, P_0 requires P_1 and P_3; P_1 requires P_2 and P_0; P_2 requires P_3 and P_1; P_3 requires P_0 and P_2. Therefore, we propose the data and tasks allocation illustrated in figure Fig. 4.

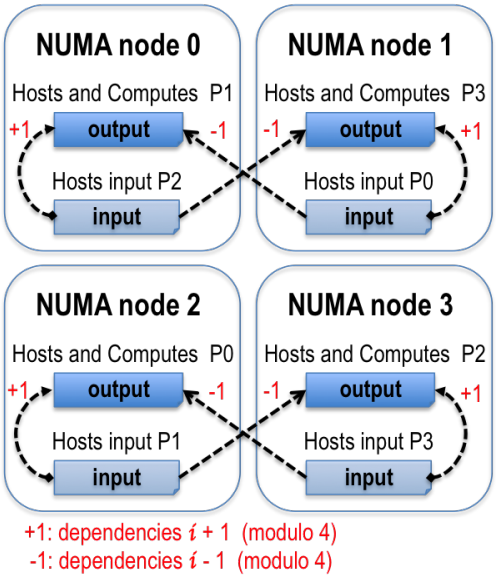

Fig. 4. NUMA-aware Scheduling with 4 Nodes

While all write accesses are local in any case of our allocation strategy, we now have half of the read accesses that are remote. For instance, NUMA node 0, which hosts and computes $P1$ needs $P2$ (local access) and $P0$ (remote access from NUMA node 1). However, figure Fig. 4 reveals that local and remote accesses could overlap in theory, thus yield a contention-free configuration related to memory buses. In order to force this to happen, we split the computation in two phases: the first one with $+1$ dependencies only and then the one with -1 dependencies. The first phase stores its results, which are afterwards loaded by the second phase to update for the final results. Figure Fig. 5 illustrates our *node splitting* transformation.

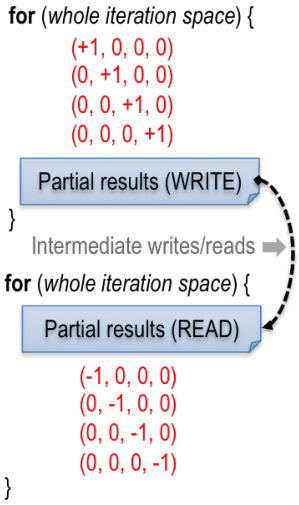

Fig. 5. Node Splitting in the Main Loop

The main concern that comes in mind about node splitting of figure Fig. 5 is that the cost of the intermediate writes/reads might hide the benefit of less memory contention. We now check this fact on our experimental results, while appreciating the impact of our NUMA-aware strategy.

B. NUMA-aware Vector-Multithread Performance Results

Using the same data and parameters on the same machine, we show our performance results without node splitting in table TABLE II.

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.02566	9.93	1
2	4	0.01313	19.40	1.95
4	8	0.00692	36.83	3.71
8	16	0.00443	57.48	5.79
(2 nodes) 16	32	0.00293	86.97	8.76
(4 nodes) 32	64	0.00248	102.79	10.35

TABLE II
NUMA-AWARE WITHOUT NODE SPLITTING

We clearly see a noticeable performance and scalability improvement when using several NUMA nodes (we use up

to 8 on each NUMA node). Table TABLE III displays the results with node splitting.

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.03025	8.42	1
2	4	0.01547	16.47	1.95
4	8	0.00825	30.87	3.66
8	16	0.00502	50.72	6.02
(2 nodes) 16	32	0.00305	83.65	9.33
(4 nodes) 32	64	0.00209	121.74	15.43

TABLE III
NUMA-AWARE WITH NODE SPLITTING

We see a 20% improvement when using the 4 NUMA nodes. Since node splitting was proposed to improve the scheduling on 4 NUMA nodes, it is useless to consider it for the other cases. It is interesting to note that the intermediate writes/loads is not hindering as it could be. This is probably due to the fact that each loop operates on a small chunk of the spinors array, thus the aforementioned writes/reads (Fig. 5) are performed on a higher level cache instead of main memory. This is another benefit of using a pool of (small) tasks. We now provide two other illustrative performance results.

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.22265	7.17	1
2	4	0.11390	17.90	1.95
4	8	0.05948	34.27	3.74
8	16	0.04009	50.84	5.55
(2 nodes) 16	32	0.02365	86.18	9.41
(4 nodes) 32	64	0.01629	125.10	13.66

TABLE IV
PERFORMANCES WITH A 32×32^3 LATTICE

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.44756	9.11	1
2	4	0.22849	17.84	1.96
4	8	0.12016	33.93	3.72
8	16	0.08219	49.60	5.45
(2 nodes) 16	32	0.04648	87.70	9.63
(4 nodes) 32	64	0.03667	111.18	12.21

TABLE V
PERFORMANCES WITH A 64×32^3 LATTICE

Reaching such noticeable performances on a highly challenging case like the *Wilson-Dirac* is very encouraging. We need to stress the fact our computations are fully performed in *double precision*. In [10] for instance, where *single precision* is considered, a global performance of 250 GFlops on an Intel Xeon Phi 5110P is reported , whose peak performance is $1.053 \times 16 \times 2 \times 60 = 2021.66$ GFlops in single precision, thus a sustained efficiency of nearly 12% (25% if FMA is not considered). Taking into account the fact that upscaling this to double precision corresponds to at least factor 0.5 reduction (certainly more severe in practice because memory is highly dominant in *Wilson-Dirac*), and that Xeon Phi 5110P has bigger caches, more threads/core and does not suffer from NUMA effects, we see that our performance is better. As stated previously, most authors consider single precision, as this reduces memory bandwidth and increases the potential of vectorization, among other reasons. Showing good and scalable performances in double precision is thus a valuable achievement. Now, we describe our distributed memory extension.

VIII. MPI SCHEDULING AND IMPLEMENTATION

Assuming that our 4 NUMA nodes processor is now a single compute node in a distributed memory parallel machine (typical supercomputer), we propose to use our previously described implementation enhanced with an appropriate MPI communication mechanism. An overview of our MPI design at the manycore node level is displayed in figure Fig. 6.

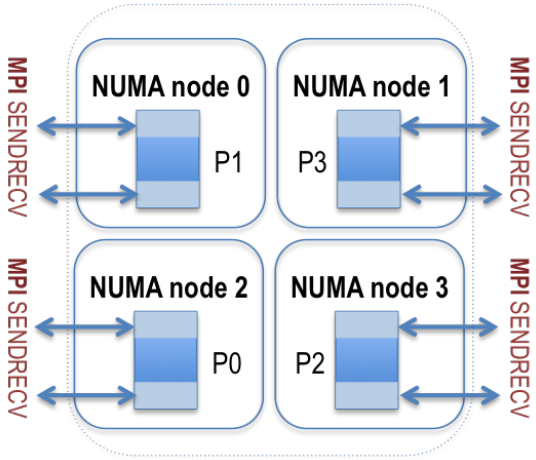

Fig. 6. MPI Layout on a Manycore Node

We recall that our threads are organized into groups, and each group is assigned one set of the (extended) *even-odd* partition. Within each group, only one thread will be in charge of MPI actions, thus sending and bringing data for the whole group. Note that we just need to exchange the surface (called *halo*) of the sublattices (classical in parallel domain decomposition). Thus, by first computing the inner part and then the halo, we get an opportunity to overlap communication and computation. Within a group, the first thread which picks up a task within their subpool (thus hold the mutex) issues MPI calls at that time. For the asynchronous case, MPI_Isend is issued for sending data. Then, later on, the thread which is the first to pick up a halo task issues an MPI_Recv. This blocks the thread and also the group because it holds the mutex. Note that this strategy requires to put the tasks related to the surface at the end of the queue (pool) (see figure Fig. 7).

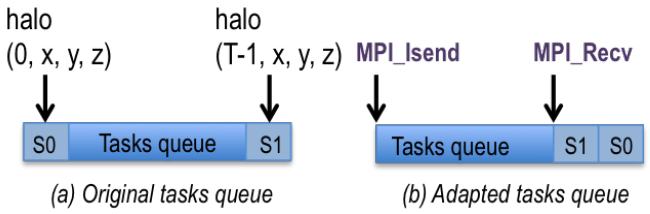

Fig. 7. Asynchronous MPI Organization

Figure Fig. 6 shows the case with 4 NUMA nodes, where each group of the threads runs on its exclusive NUMA nodes. Thus, even when we use 1 or 2 NUMA nodes only, we automatically get a similar organization. Consequently, when several MPI processes are assigned to a single cluster node, each of them will exclusively run on one NUMA node, assuming the number of processes is lower than the number of NUMA nodes. This is achieved from our thread binding mechanism by shifting the starting CPU-core to the first one within the target NUMA node. Such a flexible hybrid scheduling allows to efficiently consider the MPI granularity either at the level of the full NUMA cluster or at the level of the NUMA node. We now check the quality of our hybrid scheduling on the four NUMA nodes of our Broadwell cluster. Table TABLE. VI shows the results with our 64×32^3 scenario.

#MPI	#cores/MPI	t(s)	GFlops	Speedup
1	32	0.03942	103.42	$\times$ 1
2	16	0.05786	70.46	$\times$ 0.68
4	8	0.06533	62.41	$\times$ 0.60

TABLE VI
INTRANODE HYBRID WITH A 64×32^3 LATTICE

We clearly see that the overall performance decreases as the number of MPI tasks increases. Thus, our pure NUMA-aware multithreaded implementation (#MPI=1) is the way to go within a NUMA cluster. However, even with MPI, we still outperform the NUMA-unaware multithreaded version. The the impact of our contribution is thus twofold. We now present and discuss our experimental results on a supercomputer.

IX. Performance Results on a Supercomputer

We consider a supercomputer composed of 256 SMP nodes (IBM x3750-M4) interconnected by a high-speed InfiniBand network. Each IBM x3750-M4 compute node is a quadri-socket node of 4 Intel Sandy Bridge E5-4650, thus 4×8-core processors at 2.7 GHz. See [17] for more details about the supercomputer. The `numactl --hardware` command on one compute node gives the information displayed in figure Fig. 8.

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
node 0 size: 32614 MB
node 0 free: 30015 MB
node 1 cpus: 8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
node 1 size: 32768 MB
node 1 free: 26171 MB
node 2 cpus: 16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
node 2 size: 32768 MB
node 2 free: 31951 MB
node 3 cpus: 24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
node 3 size: 32768 MB
node 3 free: 32078 MB
node distances:
node   0   1   2   3
  0:  10  11  11  12
  1:  11  10  12  11
  2:  11  12  10  11
  3:  12  11  11  10
```

Fig. 8. NUMA configuration of one node of the supercomputer

Each compute node is thus a manycore with $8\times4 = 32$ cores equally distributed over 4 NUMA nodes. Thus, our NUMA-aware scheduling fully applies here. Table TABLE. VII shows the results with a 256×64^3 scenario. We start with 16 compute nodes for the baseline because of the large amount of memory needed to process our big 256×64^3 lattice.

#nodes	#cores	t(s)	TFlops	GFLOPS per core	Sp
16	512	0.1047	1.25	2.43	$\times$ 1
32	1024	0.0674	1.94	1.89	$\times$ 1.56
64	2048	0.0465	2.81	1.37	$\times$ 2.25

TABLE VII
SUPERCOMPUTING PERFORMANCE RESULTS

We see that our implementation delivers a good absolute performance and scales well on large numbers of cores. The slight performance per core degradation is due to data communication and processes management. Every effort to reduce the data exchanges overhead is worth considering. We now conclude the paper.

X. Conclusion

We have presented our analyses, solutions and implementation efforts related to the *Wilson-Dirac* operator on a manycore processor and describe an efficient extension to a distributed memory supercomputer. Considering the challenging double precision case, we have been able to get a high fraction of the overall peak performance on a single core of the 4 NUMA nodes INTEL BROADWELL (based on Intel Xheon E5-2699 processors) and keep a good scalability on the overall processor when considering all available cores. Our NUMA-aware scheduling for the *Wilson-Dirac* operator is a novel and genuine contribution, which we think can be applied to similar stencil computation schemes like those related to *image procesing* and *partial differential equations*. Still concerning *Wilson-Dirac*, we think that memory accesses remain dominant, thus the need for further investigations, which could include considering more aggressive compression techniques similar to the *2-rows gauge fields compression*. Interprocessor communication also deserves a close inspection in order to reach a nearly perfect overlap with computations.

ACKNOWLEDGMENT

This work was initiated from the PetaQCD projet funded by ANR, the french national agency for research, through the program COSINUS. Thanks to Christine Einsenbeis from INRIA for our regular discussions about LQCD implementations, and to my PhD student Adilla Susungi for the same about NUMA considerations. I also express my sincere gratitude to Philippe Thierry from INTEL to have provided me an access to their Broadwell-based manycore machine. I also thank IDRIS for the grant to their supercomputer Ada.

REFERENCES

[1] https://www.petaqcd.org/?lang=en

[2] D. Barthou, G. Grosdidier, M. Kruse, O. Pène, and C. Tadonki, *QIRAL: A High Level Language for Lattice QCD Code Generation*, ETAPS 2012, Tallin - Estonia (2012)

[3] F. Belletti, G. Bilardi, M. Drochner, N. Eicker, Z. Fodor, D. Hierl, H. Kaldass, T. Lippert, T. Maurer, N. Meyer, A. Nobile, D. Pleiter, A. Schaefer, F. Schifano, H. Simma, S. Solbrig, T. Streuer, R. Tripiccione, and T. Wettig. *QCD on the Cell Broadband Engine*, Oct 2007.

[4] G. Bilardi1, A. Pietracaprina1, G. Pucci1, F. Schifano, and R. Tripiccione, *The Potential of On-Chip Multiprocessing for QCD Machines*, HiPC 2005, LNCS 3769, pp. 386397, 2005.

[5] Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. 181 (2010) 15171528.

[6] R. G. Edwards, Balint Jó, and T. Jefferson, *The Chroma Software System for Lattice QCD*
http://arxiv.org/pdf/hep-lat/0409003.pdf

[7] QDP++,
http://www.top500.org/system/177003.

[8] G.Grosdidier, *Scaling stories*, PetaQCD Final Review Meeting, Orsay,France, Sept. 27th 28th 2012.

[9] K. Z. Ibrahim and F. Bodin, *Implementing Wilson-Dirac operator on the cell broadband engine*, ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, pp. 4-14, Island of Kos, Greece, 2008.

[10] B. Joó, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. W Lee, P. Dubey, and W. Watson, *Lattice QCD on Intel Xeon Phi Coprocessors*, 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013.

[11] K. Jansen and C. Urbach, *tmLQCD: a program suite to simulate Wilson Twisted mass Lattice QCD*, Computer Physics Communications, vol. 180(12), p. 2717-2738, 2009.

[12] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman, *NUMA-aware algorithms: the case of data shuffling*
http://www.pandis.net/resources/cidr13numashuffling.pdf
2013.

[13] R. Al-Omairy, G. Miranda, H. Ltaief, R. M. Badia, X. Martorell, J. Labarta, and D. Keyes, *Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing*, Upercomputing Frontiers and Innovations, vol. 2(1), 2015.

[14] M. Luscher, *Implementation of the lattice Dirac operator*, 2006.

[15] D. Pleiter, *QPACE: Power-efficient parallel architecture based on IBM PowerXCell 8i*,EnA-HPC, Hamburg,17 September 2010.

[16] QDP++,
http://usqcd.jlab.org/usqcd-docs/qdp++/.

[17] Ada Supercomputer,
http://www.idris.fr/eng/ada/ada-presentation-eng.html.

[18] Smelyanskiy, M., Vaidyanathan, K., Choi, J., Joo, B., Chhugani, J., Clark, M.A., Dubey, P.: *High-performance lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach* In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC 11 (2011) 69:169:11

[19] C. Tadonki, G. Grosdidier, and O. Pene, *An efficient CELL library for Lattice Quantum Chromodynamics*, International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies (HEART) in conjunction with the 24th ACM International Conference on Supercomputing (ICS), pp. 67-71, Epochal Tsukuba, Tsukuba, Japan, June 1-4, 2010. ACM SIGARCH Computer Architecture News, vol 38(4) 2011.

[20] C. Urbach, K. Jansen, A. Shindler, and U. Wenger, *HMC Algorithm with Multiple Time Scale Integration and Mass Preconditioning*, Computer Physics Communications, vol. 174, p. 87, 2006.

[21] C. Van Loan, *Computational Framework for the Fast Fourier Transform*, SIAM, 1992.

[22] P. Vranas, M. A. Blumrich, D. Chen, A. Gara, M. E. Giampapa, P. Heidelberger, V. Salapura, J. C. Sexton, R. Soltz, G. Bhanot, *Massively parallel quantum chromodynamics*, IBM J. RES. & DEV. VOL. 52 NO. 1/2 JANUARY/MARCH 2008.

[23] F. Wilczek, *What QCD Tells Us About Nature and Why We Should Listen*, Nuc. Phys. A 663, 320, 2000.