Efficiency on multi-core CPUs: the Wilson Dirac operator on Aurora

Michele Brambilla[*]
*ECT**
E-mail: `brambilla@ectstar.eu`

Francesco Di Renzo
University of Parma and INFN
E-mail: `francesco.direnzo@fis.unipr.it`

Marco Grossi
*ECT**
E-mail: `mgrossi@ectstar.eu`

An optimized code has to be tuned to the CPU architecture: a current trend in modern CPUs is the increasing number of cores per socket, with different levels of cache. It turns out to be natural to have different parallelization "granularities" (multithreading and multiprocessing) characterized by completely different bandwidth and latencies.
We present different strategies for the implementation of the Wilson Dirac operator which aim at maximizing the performance on the Aurora architecture.

The XXIX International Symposium on Lattice Field Theory - Lattice 2011
July 10-16, 2011
Squaw Valley, Lake Tahoe, California

[*]Speaker.

1. Introduction

Numerical computations in unquenched lattice QCD require the inversion of the Dirac operator, i.e. a large, sparse matrix. The usual approach is to perform such an inversion via an iterative algorithm, usually some variant of conjugate gradient. This requires to apply many and many times the Dirac operator, whose main operation is a (complex) matrix-vector multiplication (the Dslash operator).

The traditional approach to high performance LQCD computations entails a few steps. One first of all looks for a clever algorithm which reduces the number of computations; then one implements it exploiting the features of the machine on which the program will run (e.g. vector operations); eventually one devises a parallel version. In all these steps, optimization is a machine dependent task.

In recent years the CPU architecture has been quickly changing. A single CPU is made of more than one core, whose resources can be partially doubled in order to allow hyperthreading. The number of cache levels has been increasing (from 2 to 3, with the cores on the same socket possibly sharing L3). Boards are usually provided with more than one CPU (SMP architecture). Are traditional approaches to LQCD computations still effective, or do we have to take something else into account to improve efficiency of our programs?

In the first part of our work we try to explore how one can exploit the new architectural features in order to improve the performances of LQCD codes. We perform such a study on the Aurora machine prototype [1] based in Trento, which we will briefly describe in the next section. The second part of our work involves the merging of this new understanding with a key feature of Aurora, a 3-D toroidal network based on FPGA.

2. The Aurora system

The Aurora machine is based on commodity Intel Xeon 5600 series (Westmere) processors running at 3.33GHz. The building block of the machine is a nodecard consisting of two processors connected to each other via the QPI bus (Intel QuickPath Interconnect) with a bandwidth of 6.4 GT/s. Each processor is made of 6 cores and is connected to 6 GB of DDR3 memory@1333MHz. The 6 cores share 12MB level 3 (L3) cache. Each core has 256KB level 2 (L2) and 32KB level 1 (L1) private cache. Caches are inclusive: what is contained in L1 is also present in L2 and L3 (the same holds for L2). The architecture supports 2-way hyperthreading: in hyperthreading mode one has 24 cores per nodecard. In addition to scalar units, each core has 16 SSE registers which allow to perform a broad range of SIMD computations, namely SSE 4.2 instructions [2]. In each clock cycle the core can perform 2 (SIMD instruction)$\times$2 (execution unit) double precision operations, resulting in $\sim$ 13.3 GFlops/core and $\sim$ 160 GFlops/board.

Nodecards are interconnected by a high speed FPGA based torus network [3]. Each direction on the torus network has a bandwidth of 1 GB/s and a latency of $\sim 1\ \mu$s. We studied the strong scaling behaviour of the machine on a $48^3 \times 96$ lattice, ranging from 4 to 16 boards.

3. Wilson Dirac operator

Our study refers to the Wilson Dirac operator

$$\phi'_x = (m_0 + 4r)\phi_x - \frac{1}{2}\sum_{\mu=1}^{4}\left\{U_{x,\mu}(1-\gamma_\mu)\phi_{x+\hat\mu} + U^\dagger_{x-\hat\mu,\mu}(1-\gamma_\mu)\phi_{x-\hat\mu}\right\} \tag{3.1}$$

The computational effort required by the Dirac operator can be reduced observing [4] that $P^\pm_\mu = 1 \pm \gamma_\mu$ is a spin projector; this reduces the effective number of spin components from 4 to 2. Only two components have to be multiplied by the corresponding U matrix; the other two components are then reconstructed. To update each lattice site one has to collect the 8 neighbour spinors (each made of 24 double). All in all, the algorithm performs 1396 flops operating on $\sim$ 1.5 KB (double precision) data for each site.

Data layout is SIMD friendly: space-time indices run slower while spin, color and real/imaginary run faster. Such a layout allows to keep data required for the update of a site in contiguous memory regions.

We exploited SIMD programming via SSE intrinsics, which appear more natural to implement than assembly code. The resulting code reads something like

```
__m128d x2 = _mm_mul_pd(R2,(v+i)->whr[0].m);
v2 = _mm_addsub_pd(_mm_shuffle_pd(x2,x2,1), ...
```

Westmere processors introduce the new standard SSE4.2. In particular a new instruction turns out to be very useful for our purposes: `_mm_dp_pd` computes the dot product between two SSE registers. Exploiting this feature the dot product of two double precision complex numbers is one instruction cheaper. However, the results presented in this proceeding does not exploit this feature: if not carefully tuned it can give rise to bottlenecks in the pipeline.

4. Single board optimization

In the case of a single board we consider multithread parallelization via *pthread*. We made our tests on different lattice sizes and considered different approaches, which we will describe later in this section. We make use of the hyperthreading feature of Westmere processors: in this way it is natural to accommodate up to 24 computing threads. Since not all the physical resources of the CPU are doubled in the core, the expected speedup is less than 2: the measured value is $\sim$1.2.

The most relevant issue in multithreading is to find out how to split the lattice between threads in order to achieve the best performances. Partitioning is enforced taking care of core affinity and memory binding. One makes sure that each thread will be executed on a precise core (physical or virtual) and that all the memory resources will be allocated on the proper socket (inter-socket communications are quite slow).

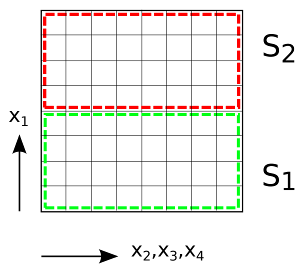


Figure 1: Along the slowest direction the lattice is sliced into two big chunks across the two sockets: this reduces the number of inter-socket communications.

All in all, we want to reduce the number of times one core has to retrieve data accessing memory resident on the other socket. In order to do that, we slice the lattice into two large blocks along the slowest direction (x_1) and bind each block to a different socket. All the threads resident on socket 0 will take care of one slice, while the threads resident on socket 1 will take care of the second slice (see the figure). Only at the boundaries (with respect to the slowest direction) one needs to access memory resident on the other socket.

To further improve efficiency, one also needs to take into account accesses to different cache levels (even though L3 and L2 latency is low and bandwidth high). Our approach is to fit the fastest running direction (x_4) inside L2 cache. To perform this we tried two different approaches: a first case named “interleaved”, the latter named “traditional”. Refer to Fig. 2:

- in the “interleaved” approach a thread takes many chunks, each of the minimum (1) size along the second fastest direction (x_3); in the figure (where 4 threads are at work) the resulting stride is 4.
- in the “traditional” case each thread takes care of a single chunk, resulting in a non-trivial size along the second fastest direction (x_3).

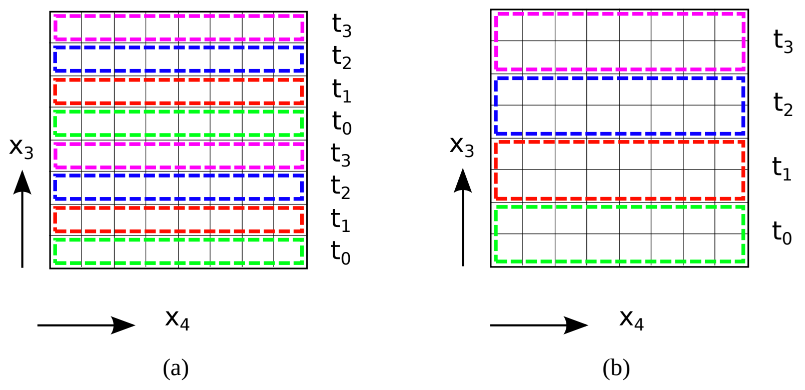


Figure 2: Different approaches to parallelization in the x_3 direction: either one thread con work on many smaller chunks or on a single bigger one.

Using hyperthreading, each couple of real+virtual cores take care of contiguous sublattices: this is expected to improve cache efficiency since L1 and L2 caches are shared between the real and virtual core (only registers are duplicated). The “traditional” approach shows the best performances, with a peak of 52.6 GFlops/board.

In both approaches the intra-node parallelization was along x_1 (the slowest) and x_3 (the second fastest) directions. We have already commented on the choice of x_1 (this optimizes accesses to

lattice size	GFlops
$8 \times 4 \times 24^2$	52.6
$12^3 \times 24$	32.2
$24^3 \times 48$	21.4
$24^2 \times 48^2$	20.2
24×48^3	20.3

Table 1: Single board performances at different lattice size. Once data is out-of-cache the performance reaches a *plateau*.

the other socket). The second choice reduces the amount of data needed in lower cache levels, implementing a cache blocking strategy. In this way each thread has a contiguous and aligned memory access to the fastest direction, with a chunk size as small as possible to fit into the L1 cache.

If the lattice does not fit into the L3 cache, performances can even go down (in out-of-cache execution) to $\sim$20 GFlops/board. Analyses performed with OProfile, monitoring hardware counters like CPU_CLOCK_UNHALTED and MEM_LOAD_RETIRED.LLC_MISS, show that L3 cache miss costs 25% of execution time. The latter does not come as a surprise, since each miss costs $\sim$180 clock cycles for a memory load in the local NUMA node. Performing the same analysis without hyperthreading we find that L3 cache misses are only a small fraction with respect to the former case: this is a limiting factor on hyperthreading approach, which nevertheless shows the best performances.

5. Parallelization

Parallelizing the execution on multiple boards requires to manage the communication and synchronization overhead on top of what we have already commented on single node execution. We can think of a computational and a communication sections:

- during the computational sections, all the threads (i.e. 24 threads) work together on proper computations;
- during the communication sections, only a few threads (8 threads, 4 per socket) are involved in the exchange of boundaries between nearest neighbour; all others threads (16 threads, 8 per socket) keep on going with proper computations; in this stage the lattice has a different partitioning (16 instead of 24 chunks).

This unbalancing (which has to be carefully tuned) allows to mask the communications time. Our parallelization fully exploits the custom toroidal network; MPI communications are only devoted to synchronization calls between boards (i.e. barrier).

We performed strong scaling tests on a $48^3 \times 96$ lattice using 4, 8 and 16 boards. One can roughly devise at least two parallelization strategies:

- keep fixed x_2, x_3 and x_4 and only scale x_1;

- scale sizes in more than one directions.

The latter performs better, as expected, because the sublattice becomes too small in the former case: it spends too much time in communication as compared to computation.

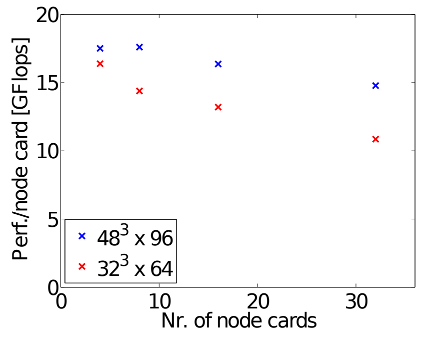
	Lattice size	Local sublattice		
x_1	48	24	12	6
x_2	96	48	48	48
x_3	48	48	48	48
x_4	48	48	48	48
	GFlops/board	18.7	15.4	11.1
	speedup	1	1.65	2.37

	Lattice size	Local sublattice		
x_1	48	24	24	12
x_2	96	48	24	24
x_3	48	48	48	48
x_4	48	48	48	48
	GFlops/board	18.7	17.6	16.4
	speedup	1	1 .88	3.51

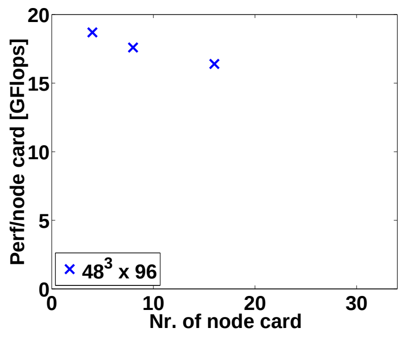
Table 2: Comparison between different strong scaling approaches. Results refers to 4, 8 and 16 board case.

The speedup in the 8 and 16 boards cases (computed with respect to the 4 boards case) is close to the ideal values of 2 and 4; since we are out-of-cache both in 4 and 16 boards cases, we cannot expect a superlinear speedup.

We compared strong scaling results obtained using our custom code with those obtained using a variant of ETMC code which improves performances on Aurora [5]. In Fig. 3 one can see that our approach gives better performances using a small number of boards. Increasing the number of node cards performances are roughly the same.



(a)



(b)

Figure 3: Strong scaling comparison between ETMC code *(a)* and our custom implementation *(b)*.

When comparing these results one has to keep in mind that there are deep differences between ETMC and our custom implementation. In the former case structures are optimized for computations, and in order to improve communications one need to gather and pack data to be sent, perform the communication and finally unpack and scatter the received message. In our case data structures are aligned for communications: we don't need to pack and unpack data, but this results in a slightly more fragmented memory image as far as computation requirements are concerned.

Moreover ETMC communications rely on MPI over INFINIBAND, while this case of study is based on the custom Aurora Torus Network (ATN): the latter is still under development, and at the time the expected communication performance was $\sim$50% peak.

6. Future directions

We expect to be able to further improve both single board and parallel implementations of the code.

In both cases efficiency can be increased improving the SSE implementations of the computational kernel: at the moment this is performed mainly using SSSE3, and we are studying a more performant based on SSE4.2. The bottleneck is that the single instruction which would reduce the number of flops introduces in turn higher latencies in the pipeline.

In the parallel implementation we expect that a rearrangement of data structures can improve communications, without loosing the alignment. This would allow a reduction of the number of threads involved in communications. At the same time this would allow a reduction in the total amount of data sent.

Exploiting prefetching is another candidate for improvement. A clever usage of profiling tools could result in a better understanding of the cache miss issue, which in turn could hint prefetching strategies.

Acknowledgments

The AuroraScience project is funded by the Provincia Autonoma di Trento and the Istituto Nazionale di Fisica Nucleare, in the framework of an agreement with the Fondazione Bruno Kessler.

References

[1] F. Di Renzo [AuroraScience Collaboration], "Status of the AuroraScience project," PoS **LATTICE2011** (2011) 031.

[2] Intel® SSE4 Programming Reference, 2007, http://www.intel.com/design/processor/manuals/253667.pdf.

[3] M. Pivanti, S. F. Schifano and H. Simma, "An FPGA-based Torus Communication Network," PoS **LATTICE2010** (2010) 038

[4] This is a standard trick: see for example K. Jansen and C. Urbach, "tmLQCD: A Program suite to simulate Wilson Twisted mass Lattice QCD," Comput. Phys. Commun. **180** (2009) 2717

[5] L. Scorzato [AuroraScience Collaboration], "AuroraScience," PoS **LATTICE2010** (2010) 039.