Multiple right-hand-side setup for the DD-αAMG[*]

Daniel Richtmann[†a]**, Simon Heybrock**[a,b]**, Tilo Wettig**[a]

[a]*Department of Physics, University of Regensburg, 93040 Regensburg, Germany*
[b]*Data Management and Software Centre, European Spallation Source, Universitetsparken 5,*
2100 Copenhagen, Denmark
E-mail: daniel.richtmann@ur.de

The setup cost of a modern solver such as DD-αAMG (Wuppertal Multigrid) is a significant contribution to the total time spent on solving the Dirac equation, and in HMC it can even be dominant. We present an improved implementation of this algorithm with modified computation order in the setup procedure. By processing multiple right-hand sides simultaneously we can alleviate many of the performance issues of the default single right-hand-side setup. The main improvements are as follows:

By combining multiple right-hand sides the message size for off-chip communication is larger, which leads to better utilization of the network bandwidth. Many matrix-vector products are replaced by matrix-matrix products, leading to better cache reuse. The synchronization overhead inflicted by on-chip parallelization (threading), which is becoming crucial on many-core architectures such as the Intel Xeon Phi, is effectively reduced. In the parts implemented so far, we observe a speedup of roughly 3x compared to the optimized version of the single right-hand-side setup on realistic lattices.

[†]Speaker.

1. Introduction and motivation

Conventional iterative Krylov subspace solvers for the Dirac equation share a common behavior when going to small quark masses: Their iteration number and time to solution (wall-clock time) increases drastically, which basically renders them unusable. Therefore a lot of effort has been put into developing efficient preconditioning algorithms that aim at tackling this problem, such as domain decomposition [1], inexact deflation [2], and multigrid approaches [3, 4]. While these methods significantly reduce the iteration number, the latter two introduce an additional overhead compared to standard solvers since they require an initial setup phase before one can start solving the Dirac equation. In HMC the setup cost can even be the dominant contribution to the total wall-clock time spent in the solver since only a few solves can be done before the setup has to be updated. Thus an optimization of the setup code potentially has a large impact on the overall HMC performance.

Based on the attractive theoretical properties and the performance of the DD-αAMG algorithm, the Regensburg group (RQCD) recently decided to port the implementation of this algorithm by the Wuppertal group [4], which is C-MPI code aimed at standard CPUs, to SIMD architectures, with a special focus on the Intel Xeon Phi architecture (KNC) used in QPACE 2 [5]. This involved threading the code using OpenMP, optimizing it for the wide SIMD registers of the KNC, and reducing memory-bandwidth requirements by enabling the use of half precision on the coarse grid. For a detailed description of this effort see [6].

Even with the improvements achieved in [6], there is still optimization potential in the setup of DD-αAMG, as it remains expensive. In this contribution we document our work on an improved implementation that modifies the computation order in the setup phase to process multiple right-hand sides simultaneously.

2. Description of the algorithm

The DD-αAMG algorithm uses FGMRES as the outer Krylov subspace solver for the Dirac equation, preconditioned by a multigrid method that consists of two parts: a smoother working on the fine grid that reduces the error contribution of eigenvectors with large eigenvalues (high modes), and a coarse-grid correction (CGC) that reduces the error contribution of low modes.[1] To this end projection operators between the grids and the Dirac operator on the coarse grid need to be defined in an initial setup phase.

The setup procedure of DD-αAMG is based on a set of N_{tv} random test vectors (each of dimension $12V$, where V is the lattice volume) that are used to construct restriction R, prolongation $P = R^{\dagger}$, and coarse-grid operator D_c. The setup is split into two parts: an initial phase and an iterative refinement phase. In the initial phase, a domain-decomposition (DD) smoother based on the Schwarz alternating procedure is run on each of the test vectors for a few iterations with starting guess 0. Then the initial operators are constructed from the updated test vectors. This completes the initial phase. The operators are then updated in the iterative refinement phase (Alg. 1) that makes use of the full V-cycle of the multigrid algorithm. For a more detailed description see [4, 6].

---

[1]As in [6] we restrict ourselves to two grid levels.

Algorithm 1: Iterative part of MG setup (standard implementation)

---

1 **for** $i = 1$ **to** N_{setup} **do**

2   // apply V-cycle to test vectors

3   **for** $j = 1$ **to** N_{tv} **do**

4   *// coarse-grid correction*

5   restrict test vector v_j to coarse grid: $v_{c,j} = R v_j$

6   solve coarse system to low accuracy: $u_{c,j} \approx D_c^{-1} v_{c,j}$

7   prolongate result of coarse-grid solve to fine grid: $u_j = P u_{c,j}$

8   *// fine grid*

9   apply smoother to test vector v_j, with result from CGC as starting guess

10   replace test vector v_j by result of smoother

11   **end for**

12   setup of restriction R and coarse-grid operator D_c

13 **end for**

---

3. Basic idea

After the optimizations described in [6], we identified the application of the V-cycle to the test vectors in the iterative refinement phase to be the dominant contribution to the setup time. Therefore our work focuses on this part of the code exclusively.

In the implementations of [4, 6] the V-cycle is applied to the test vectors in a loop sequentially, i.e., to a single right-hand side (SRHS) at a time. The basic idea of our improvements is simple. We modify the computation order of the code by blocking the loop over the test vectors (Alg. 2) with a block length of N_{b} and apply the V-cycle to multiple right-hand sides (MRHS), i.e., all vectors inside such a block, simultaneously. Choosing $N_{\text{b}} = N_{\text{SIMD}}$ and moving the loop inside a block to the lowest level functions of the code enables us to use this loop for SIMD vectorization.

Algorithm 2: Iterative part of MG setup (improved implementation)[2]

---

1 **for** $i = 1$ **to** N_{setup} **do**

2   **for** $j = 1$ **to** $N_{\text{tv}}/N_{\text{b}}$ **do**

3 $k = 1 + (j-1) \cdot N_{\text{b}}, \ \ell = j \cdot N_{\text{b}}$

4   apply coarse-grid correction (CGC) to test vectors $v_k, \ldots, v_\ell$

5   apply smoother to test vectors $v_k, \ldots, v_\ell$, with result from CGC as starting guess

6   replace test vectors $v_k, \ldots, v_\ell$ by result of smoother

7   **end for**

8   setup of restriction R and coarse-grid operator D_c

9 **end for**

---

[2]In the description of the algorithm we assume that N_{tv} is an integer multiple of N_{b}. If it is not the algorithm gets modified in a straightforward way, but then part of the SIMD unit is wasted in the last iteration, see also [6].

4. Communication bandwidth

The effective network bandwidth for off-chip communication via MPI depends on the message size (cf. Fig. 1). For small messages latency effects are dominant, resulting in low bandwidth. For larger messages the effective bandwidth increases since latency effects become negligible.
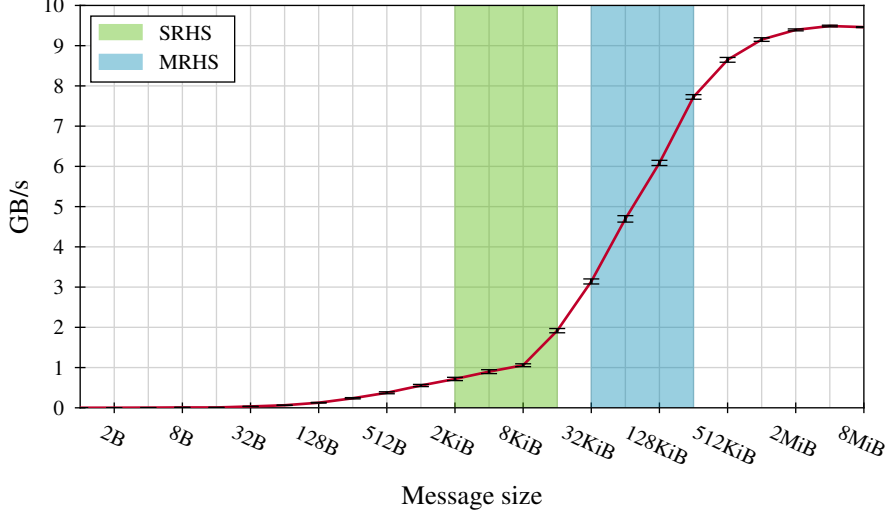

Figure 1: Network bandwidth vs. message size between two KNCs (bi-directional) in QPACE 2 via FDR InfiniBand. Typical message sizes in the DD-αAMG setup are shown for SRHS (green) and MRHS (blue) setup.

The message size (S_μ in direction μ) on the coarse grid of the DD-αAMG setup depends on the local volume of one MPI rank and the degrees of freedom per site ($2N_{tv}$):

$$S_\mu = \prod_{\nu=0,\nu\neq\mu}^{3} \frac{\text{(local lattice)}_\nu}{\text{(domain size)}_\nu} \cdot \frac{2N_{tv}}{2} \cdot 8\,\text{Byte}, \tag{4.1}$$

where the factor 2 in the denominator accounts for even-odd preconditioning and 8 Byte is the size of a complex number in single precision. From Eq. (4.1) we obtain message sizes of order 1 KiB for the default setup, which is just the point where the effective bandwidth starts to increase significantly. By processing multiple right-hand sides simultaneously we are able to perform all halo exchanges and global sums, respectively, in the same call to MPI. Thus the message size increases by a factor of $N_b = N_{SIMD} = 16 = 2^4$. Fig. 1 suggests an estimated increase in effective bandwidth of a factor $3 \sim 4$, which might translate directly to the wall-clock time spent in communication.

5. Mapping to SIMD registers

The basic layout for data structures based on complex numbers in the original Wuppertal implementation did not take vectorization into account and uses the *complex* data type in C. This way, a vector-like object v of length ℓ is stored such that the real and imaginary parts alternate in memory:

Re v_1	Im v_1	Re v_2	Im v_2	$\cdots$	Re v_ℓ	Im v_ℓ

This is known as Array-of-Structs (AoS) layout. In the code parts relevant for us, the implementation in [6] works with this layout by de-interleaving two registers using swizzle intrinsics before and after doing a SIMD computation. This introduces additional overhead that could be avoided with a data layout more suitable to vectorization.

Our implementation uses another index for vectorization, i.e., the index of the different right-hand sides inside a block. For each vector index i we store $N_{\mathrm{SIMD}}\ (=16)$ real parts of the right-hand sides followed by the corresponding imaginary parts:

Re $v_i^{(1)}$	Re $v_i^{(2)}$	$\cdots$	Re $v_i^{(16)}$	Im $v_i^{(1)}$	Im $v_i^{(2)}$	$\cdots$	Im $v_i^{(16)}$

This is known as Array-of-Structs-of-Short-Vectors (AoSoSV) layout. While the conversion required non-trivial programming effort, this layout yields a more natural mapping to SIMD. The de-interleaving overhead is gone, and the individual entries in the registers contain data independent of one another, which eliminates the need for reduction operations over the elements in the register.

With our modifications to the data layout, matrix-vector multiplications become matrix-matrix multiplications, which enables us to use a different vectorization scheme. In contrast to [6] we vectorize the restriction of a vector from the fine to the coarse grid (as an example) by broadcasting the elements of the projection operator R as shown in Alg. 3 (see [6] for the definition of N_{block}, V_{block}, and y_c). The same vectorization scheme is used for the application of D_c in the coarse-grid solve. BLAS-like linear algebra (e.g., vector adds) is vectorized trivially with this data layout.

Algorithm 3: SIMD implementation of restriction $Ry = y_c$ with N_{SIMD} right-hand sides

---

1   **for** $i = 1$ **to** N_{block} **do**
2   **foreach** $h = \ell, r$ **do**
3   **for** $n = 1$ **to** $6V_{\mathrm{block}}$ **do**
4   load real and imag. parts of N_{SIMD} rhs for entry $y_{i,n}^h$ into SIMD vectors
5   **for** $j = 1$ **to** N_{tv} **do**
6   load real and imag. parts of $(y_c)_{i,j}^h$ for N_{SIMD} rhs
7   broadcast real and imag. part of entry j in column n of R_i^h into SIMD vectors
8   increase $(y_c)_{i,j}^h$ by complex fused multiply-add and write to memory
9   **end for**
10   **end for**
11   **end for**
12 **end for**

---

6. Memory-bandwidth and cache-reuse considerations

A dense complex matrix-vector multiplication $c = A \cdot b$, where A, b, and c are of dimension $M \times K$, K, and M, respectively, requires transferring $(2M + M \cdot K + K) \cdot 8\,\mathrm{Byte}$ from and to memory in single precision. The computation needs a minimum of $4 \cdot M \cdot K / 16$ cycles, where a complex

fmadd consists of 4 real *fmadd* operations, of which a KNC core can perform 16 in one cycle. The ratio of these numbers yields the memory bandwidth per core required to avoid stalls, i.e., $32 \cdot (2/K + 1 + 1/M)$ Byte/cycle. For a typical working set of a core, K and M are large enough so that their contribution $2/K + 1/M$ is negligible compared to 1.[3] The resulting required memory bandwidth is then 32 Byte/cycle per core, or 2377 GB/s on 60 cores of a KNC with a clock speed of 1.238 GHz, which is well above the KNC's sustained memory bandwidth of $150 - 170$ GB/s, measured with the STREAM benchmark.

Performing the analogous calculation for the matrix-matrix multiplication with N_{SIMD} right-hand sides ($A = M \times K$, $B = K \times N_{\text{SIMD}}$, and $C = M \times N_{\text{SIMD}}$) yields $32 \cdot (2/K + 1/N_{\text{SIMD}} + 1/M)$ Byte/cycle ~ 2 Byte/cycle = 149 GB/s. Here, an element of A can stay in cache for N_{SIMD} right-hand sides, which results in the difference to the value above. Thus our method is able to reduce the memory bandwidth requirements of this code part significantly, and our estimate is now within reach of the KNC's sustained memory bandwidth.

7. Results

At the time of this writing we have finished the implementation of the coarse-grid solve and the projection operators, while the smoother [7] still works with the default data layout. This introduces some temporary copying overhead which will disappear as soon as we have a MRHS implementation of the smoother.

The results below are from runs on the CLS lattice C101 ($48^3 \times 96$, $\beta = 3.4$, $m_\pi = 220$ MeV, $a = 0.086$ fm) described in [8]. We use $N_{\text{SIMD}} = 16$ test vectors, a domain size of 4^4, and a relative coarse-grid tolerance of 0.05. The remaining solver parameters are tuned for minimal propagator wall-clock time with the default (SRHS) setup. To exclude algorithmic effects and allow for a direct comparison we use the same parameter combination also for the MRHS setup.
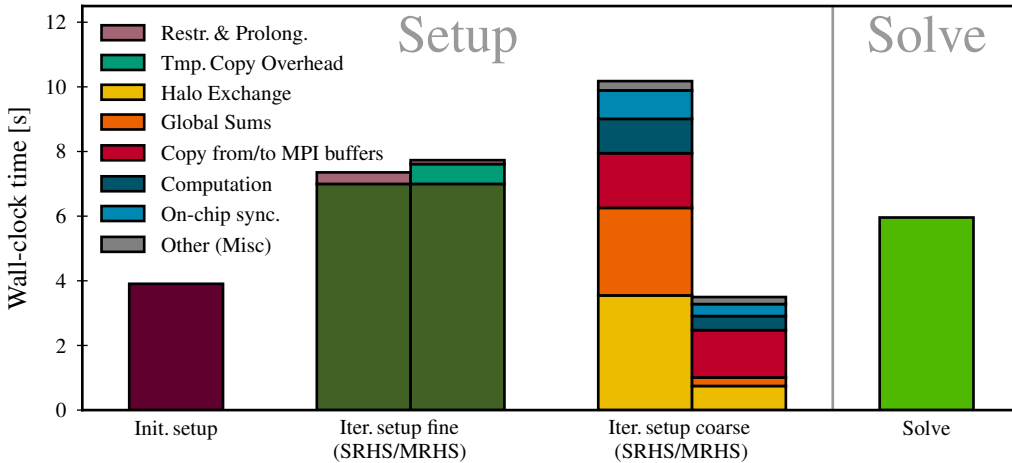

Figure 2: Summary of contributions to the wall-clock time in the DD-αAMG setup on 64 KNCs in QPACE 2 (solve time for comparison). Improvements in parts affected by our modifications are given in detail.

---

[3]M needs to be multiplied by 2 for the spin-splitting of DD-αAMG.

With these parameters, a lattice vector on the coarse grid requires 1% of the memory of a vector on the fine grid. With the MRHS setup, we need 32 lattice vectors on the fine grid and $16 \cdot 32 = 512$ on the coarse grid. This is to be compared to 17 and 32 with the SRHS setup. Thus, our method needs roughly a factor of 2.2 more memory in total for the setup. In a realistic measurement run we typically also keep around several propagators on the fine grid (consisting of 12 vectors each), so the increase in total memory consumption is actually considerably smaller.

In Fig. 2 we show the improvements in wall-clock time we achieve with our method. We gain a factor of 2.9 in the projection operators and a factor of 2.4 in computation on the coarse grid. Our method needs fewer calls to barriers between threads, which yields an improvement of 2.7x in on-chip synchronization. However, the largest gains are in halo exchanges (4.7x) and global sums (10.3x), which were the dominant contributions previously. After our improvements, the wall-clock time is now dominated by copying data from and to MPI buffers. This is currently done by a single thread on a single core. In the future we will reduce the impact of these copy operations by threading them over cores, which will allow us to exploit a larger fraction of the KNC's sustained memory bandwidth.

In total, the time spent on the coarse grid is reduced by a factor of 2.9, which translates to a factor of 1.4 for the total setup time of DD-αAMG.

8. Conclusions and outlook

By combining multiple right-hand sides we were able to significantly reduce the wall-clock time of the previously dominant contribution (i.e., coarse-grid solve) to the setup of DD-αAMG, see Fig. 2 for details. Our biggest improvements are in communication, where we can send fewer messages that are larger and thus are able to reduce the impact of latency effects. Additional improvements were made in computation and on-chip synchronization.

As mentioned above, the impact of the red block in Fig. 2 (copy from/to MPI buffers) will be reduced in the future by threading these copy operations over cores. More importantly, Fig. 2 shows that the biggest optimization potential is now in the fine-grid part of the iterative setup. Therefore we will complete the multiple right-hand-side V-cycle by applying the techniques used in the present work also to the smoother [7], which should yield similar speedups.

References

[1] M. Lüscher, *Comput. Phys. Commun.* **156** (2004) 209 [hep-lat/0310048].

[2] M. Lüscher, *JHEP* **07** (2007) 081 [arXiv:0706.2298].

[3] R. Babich et al., *Phys. Rev. Lett.* **105** (2010) 201602 [arXiv:1005.3043].

[4] A. Frommer et al., *SIAM J. Sci. Comput.* **36** (2014) A1581 [arXiv:1303.1377].

[5] P. Arts et al., *PoS* **LATTICE2014** (2015) 021 [arXiv:1502.04025].

[6] S. Heybrock et al., *PoS* **LATTICE2015** (2016) 036 [arXiv:1512.04506].

[7] S. Heybrock et al., *Proceedings of SC '14* (2014) 69 [arXiv:1412.2629].

[8] M. Bruno et al., *JHEP* **02** (2015) 043 [arXiv:1411.3982].