Algebraic temporal blocking for sparse iterative solvers on multi-core CPUs

Alappat, Christie; Thies, Jonas; Hager, Georg; Fehske, Holger; Wellein, Gerhard

DOI
10.1177/10943420241283828

Publication date
2024

Document Version
Final published version

Published in
International Journal of High Performance Computing Applications

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Research Paper

Algebraic temporal blocking for sparse iterative solvers on multi-core CPUs

The International Journal of High Performance Computing Applications
2024, Vol. 0(0) 1–21

DOI: 10.1177/10943420241283828
journals.sagepub.com/home/hpc
Sage

Christie Alappat[1], Jonas Thies[2], Georg Hager[1], Holger Fehske[1] and Gerhard Wellein[1,2,3]

Abstract

Sparse linear iterative solvers are essential for many large-scale simulations. Much of the runtime of these solvers is often spent in the implicit evaluation of matrix polynomials via a sequence of sparse matrix-vector products. A variety of approaches has been proposed to make these polynomial evaluations explicit (i.e., fix the coefficients), e.g., polynomial preconditioners or s-step Krylov methods. Furthermore, it is nowadays a popular practice to approximate triangular solves by a matrix polynomial to increase parallelism. Such algorithms allow to evaluate the polynomial using a so-called matrix power kernel (MPK), which computes the product between a power of a sparse matrix A and a dense vector x, i.e., $A^p x$, or a related operation. Recently we have shown that using the level-based formulation of sparse matrix-vector multiplications in the Recursive Algebraic Coloring Engine (RACE) framework we can perform temporal cache blocking of MPK to increase its performance. In this work, we demonstrate the application of this cache-blocking optimization in sparse iterative solvers. By integrating the RACE library into the Trilinos framework, we demonstrate the speedups achieved in (preconditioned) s-step GMRES, polynomial preconditioners, and algebraic multigrid (AMG). For MPK-dominated algorithms we achieve speedups of up to 3× on modern multi-core compute nodes. For algorithms with moderate contributions from subspace orthogonalization, the gain reduces significantly, which is often caused by the insufficient quality of the orthogonalization routines. Finally, we showcase the application of RACE-accelerated solvers in a real-world wind turbine simulation (Nalu-Wind) and highlight the new opportunities and perspectives opened up by RACE as a cache-blocking technique for MPK-enabled sparse solvers.

Keywords

Sparse matrices, iterative solvers, matrix polynomial, cache blocking, performance

1. Introduction and related work

The solution of linear systems involving large sparse matrices is at the core of many computational workflows. Apart from application-specific approaches like domain decomposition methods and geometric multigrid, the most popular classes of solvers are Krylov subspace methods (often combined with preconditioning) or algebraic multigrid. These algorithms are key components in open-source parallel simulation frameworks like Trilinos, see Trilinos Project Team (2020).

Krylov subspace methods perform a sequence of sparse matrix-vector multiplications (SpMV), vector updates (axpy) and inner products to construct some basis of the Krylov subspace $\mathcal{K}_k(A, v) = \{v, Av, A^2v, \ldots, A^{k-1}v\}$, and then extract an approximate solution by solving a much smaller problem involving A projected onto that subspace. In general, maintaining some orthogonality property of the basis is essential for stability, which leads to other vector operations being required in between SpMVs. If a preconditioner is used, the SpMVs may also be alternated with other operators, e.g., approximations of A^{-1} or triangular factors $A^{-1} \approx U^{-1}L^{-1}$. Preconditioning is a broad field of research; for an overview of methods, see Wathen (2015). For sufficiently large matrices A, the SpMVs and preconditioners typically dominate the runtime, and it is known that these operations are main-memory bound for

[1]Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
[2]Institute of Applied Mathematics, Delft University of Technology, Delft, The Netherlands
[3]Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

Corresponding author:
Christie Alappat, Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg, Martensstraße 1, Erlangen 91058, Germany.
Email: christie.alappat@fau.de

appropriately chosen sparse data layouts and may achieve high spatial locality when accessing the data elements of the matrix (Kreutzer et al., 2014).

In the early days of parallel computing, s-step methods were developed to improve data locality (i.e., reduce communication) in Krylov methods (Erhel, 1995; Chronopoulos and Gear, 1989; Chronopoulos, 1991; Chronopoulos and Kim, 2020). They break up the data dependency by first computing a sequence of SpMVs and then using a sequence of scalar/vector operations to approximate the basis produced by, e.g., a Conjugate Gradient (CG) or Generalized Minimum Residual (GMRES) method. These variants have recently received attention as they may use fast 'kernels' like the 'Matrix-Power Kernel' (MPK) and the 'Tall-Skinny QR' (TSQR), see Demmel et al. (2008), Hoemmen (2010). Recent work focuses on distributed-memory systems, i.e., reducing the number of messages and synchronization points in MPI implementations (e.g., Yamazaki et al., 2014b; Dongarra et al., 2017; Yamazaki et al., 2017).

However, the performance potential of the MPK for modern cache-based multicore architectures has not been exploited so far in any solver frameworks. MPK involves the successive application of SpMV with the same matrix and offers the opportunity to exploit temporal locality by reusing the matrix elements from cache instead of repeatedly loading them from main memory. For regular stencil algorithms it is well known how to improve temporal locality by temporal blocking (Datta, 2009); on the other hand, for irregular sparse matrices such geometrical blocking approaches are generally not applicable. Instead, an algebraic formulation of the problem needs to be considered to control the data dependencies and cache-access locality between successive SpMVs. In Alappat et al. (2023), we have shown that this can be realized by a cache-aware traversal of the levels obtained from a breadth-first search (BFS) on the graph underlying the matrix. Our implementation of the MPK achieves good scalability and high performance on modern multicore architectures for a broad range of matrices. Compared to state-of-the-art implementations, RACE provides speedups in the range of 2–4 ×. We refer to Alappat et al. (2023) for an overview of related work on optimizing MPK. Besides s-step Krylov algorithms there are other classes of methods like polynomial preconditioning, smoothers in multigrid, Chebyshev time propagation, and power methods for eigenvalue solvers, which may also benefit from cache blocking of MPK.

1.1. Contributions

In this paper we address the integration of the cache-blocked RACE MPK (Alappat, 2019) into a number of representative iterative methods and evaluate the overall performance benefit on various solvers. The cache-blocking strategy does not change the numerical behavior of the methods. Thus, the purpose of this paper is not to compare different iterative schemes or identify the most efficient preconditioners. Instead, we focus on a broad range of numerical algorithms including several preconditioners and investigate the performance gains achieved through optimized MPK. Our specific contributions can be summarized as follows:

- Demonstration of the use of RACE MPK to accelerate s-step GMRES on modern multi-core CPUs,
- incorporation of diagonal and triangular preconditioners into the MPK, where the triangular solves are approximated using Jacobi-Richardson iterations,
- application of RACE MPK to GMRES polynomial preconditioning, demonstrating substantial performance improvements for high matrix powers,
- introduction of strategies to accelerate algebraic multigrid (AMG) methods using RACE's cache blocking technique, and
- showcasing the impact of highly efficient MPK on algorithmic choices using a case study from wind turbine simulation (Nalu-Wind, Sprague et al., 2020).

In all cases a thorough performance analysis is conducted and the speedup obtained by RACE for different solvers is quantified.

1.2. Outline

Throughout the paper we use the same representative hardware and matrices for demonstration purposes; these are introduced in Section 2. We start by briefly recapitulating the idea of cache-blocking MPK using RACE in Section 3. Section 4 discusses the hardware-efficient integration of RACE MPK into *s*-step GMRES methods. Section 5 addresses the integration of preconditioners into the MPK for *s*-step GMRES methods. We choose Jacobi and Gauss-Seidel sweeps as representative examples for diagonal and triangular preconditioners, where the triangular systems are solved approximately using Jacobi-Richardson iterations. Further in the section we discuss the application of RACE to advanced polynomial and AMG preconditioners. In Section 6 we bring together the ideas developed in the paper to accelerate the solution of a momentum equation arising in the Nalu-Wind wind turbine simulation. Finally, we summarize our findings in Section 7.

2. Hardware and software environment

2.1. Hardware testbed

The experiments presented in this paper were performed on single Intel Ice Lake (ICL) and AMD Epyc Rome (ROME) multicore processors. These processors or similar ones are

Table 1. Key specification of test bed machines.

Architecture	ICL	ROME
Chip model	Xeon Platinum 8368	AMD EPYC 7662
Microarchitecture	Sunny cove	Zen-2
Cores per socket	38	64
Max. SIMD width	512 bits	256 bits
L1D cache capacity	38 × 48 KiB	64 × 32 KiB
L2 cache capacity	38 × 1.25 MiB	64 × 512 KiB
L3 cache capacity	57 MiB	16 × 16 MiB
L3 bandwidth	420 GB/s	2700 GB/s
Mem. Configuration	8 ch. DDR4-3200	8 ch. DDR4-3200
Mem. Bandwidth	170 GB/s	146 GB/s

used in the majority of Top500[1] systems today. Key features of the chips are listed in Table 1. Both architectures implement an x86 instruction set. The 10 nm ICL processor supports the AVX-512, while the 7 nm ROME processor supports AVX2. The systems are capable of sustaining more than 2 GHz clock frequency and the turbo mode was active for all our experiments. The AMD system has a higher core count (64 per socket) compared to its Intel counterpart (38 per socket). Both systems have three levels of cache: private, inclusive L1 and L2 caches, and shared victim L3 cache. The L3 cache on ICL is shared among all the cores within a socket, while on ROME the L3 cache is shared within one core complex (CCX) unit comprising four cores. Due to ROME's hierarchical "chiplet" design, the L3 cache is highly scalable and it can sustain an aggregate L3 load only bandwidth of 2700 Gbyte/s, while ICL achieves only 420 Gbyte/s. The total L3 cache size of ROME is also much larger compared to ICL. Both systems have eight-channel DDR memory and sustain similar memory bandwidth. Both are configured with one ccNUMA domain per socket configuration, i.e., Sub-NUMA Clustering (SNC) was disabled on ICL and one NUMA node per socket (NPS1) mode was used on ROME.

2.2. Software environment

The ICL system runs Red Hat Enterprise Linux (RHEL) version 8.4 while ROME runs Ubuntu 20.04.4 LTS. For best performance, the OS setting "Transparent Huge Pages" (THP) was set to "always" on both the systems, see Alappat et al. (2020b) for details. For compilation we used the Intel compiler version 2021.5.0 and 19.0.5 on ICL and ROME, respectively, at the highest optimization level `-O3`. Machine-specific code generation was employed via `-xHOST` on ICL and `-march=core-avx2 -mtune=core-avx2` on ROME. All floating-point computations were performed in double precision, and integers were 32 bits wide. The linear solvers from the Trilinos framework used in this work were adapted to use the RACE MPK. Both RACE and the modified Trilinos solvers are available through GitHub repositories; the exact versions used for the experiments are available at RACE [2] and Trilinos[3]. For BLAS computations, Intel MKL version 2022.0 (see Intel, 2022) was used on ICL. On ROME we used MKL version 2020.0.4 unless otherwise stated. It is well known that MKL sometimes exhibits low performance when it detects AMD hardware. In order to make results comparable, we overwrite the `mkl_serv_intel_cpu_true` symbol with a function that always returns true.[4] On ROME we occasionally use the AOCL BLIS library (see AMD, 2022; Van Zee and Van de Geijn, 2015) version 3.2 as an alternative. This is clearly indicated in the text. Thread affinity was enforced by setting `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. The run-to-run fluctuations in the experiments were less than 5% and therefore we do not present any error bars.

Table 2. Details of the benchmark matrices. See Davis and Hu (2011) for details.

ID	Matrix name	N_r	N_{nz}	N_{nzr}
1	G3_circuit	1,585,478	7,660,826	4.83
2	thermal2	1,228,045	8,580,313	6.99
3	Transport	1,602,111	23,487,281	14.66
4	Fault_639	638,802	28,614,564	44.79
5	Emilia_923	923,136	41,005,206	44.42
6	af_shell10	1,508,065	52,672,325	34.93
7	ML_Geer	1,504,002	110,879,972	73.72
8	Flan_1565	1,564,794	117,406,044	75.03

2.3. Benchmarking

For our experiments we choose matrices from two prior publications (Loe et al., 2020; Berger-Vergiat et al., 2021) that are relevant to our work and are also available in the SuiteSparse Matrix Collection by Davis and Hu (2011). We selected only square matrices with a memory footprint beyond 100 MB as our optimization targets big matrices that have to be loaded from main memory. Table 2 lists the matrices together with some relevant parameters: number of rows (N_r), number of non-zeros (N_{nz}), and average number of non-zeros per row (N_{nzr}). In the following discussion we refer to the matrices by their IDs (first column of Table 2). The compressed row storage (CRS) format was used throughout.

In all solver experiments we iterated until a convergence threshold $(\|b - Ax\|/\|b\| \leq 10^{-12})$ or algorithm-specific maximum iteration count was reached. Since cache blocking does not alter convergence properties, this is a valid method of performance comparison.

3. Accelerating MPK using RACE

The central theme of this work revolves around speeding up various iterative solvers by using cache-blocked MPK.

MPK computes the application of powers of a sparse matrix to a dense vector. For a given sparse square matrix A and input vector x, MPK computes the matrix powers $A^p x$ up to a maximum power of p_m, i.e., $p = 1, \ldots, p_m$, and stores the result into p_m vectors ($y_p = A^p x$). Usually this is done by performing back-to-back SpMVs as shown in Alg. 1. The input vector x is stored in y_0 and each SpMV computation promotes the power by one. Therefore, we reach $A^{p_m} x$ after p_m SpMV computations.

SpMV being the central kernel in Alg. 1, it is clear that its performance will be similar to that of SpMV. For most of the matrices encountered in computational science and engineering, the latter is limited by main memory bandwidth on modern CPUs. In Alg. 1, if A is larger than any cache it will be loaded p_m times from memory. However, since MPK uses the same matrix A for every SpMV, cache blocking of matrix accesses across successive matrix power calculations may reduce main memory traffic and thus improve performance. This blocking is not straightforward due to dependencies among the SpMV computations, i.e., $y_p = A^p x = A y_{p-1}$ depends on the results of the previous $y_{p-1} = A^{p-1} x = A y_{p-2}$ computation; we denote this by $A^{p-1} \rightarrow A^p$. The dependency, however, is not necessarily an all-to-all dependency, i.e., to calculate $A^p x$ on a subset of rows there is no need to finish the full $A^{p-1} x$ computation first. The structure (or graph) of the matrix determines the dependencies between successive power calculations; the RACE library exploits it to enable cache blocking.

RACE uses a fundamental concept called *levels* to achieve cache blocking. Levels are formed by performing a BFS traversal on the graph of the matrix, where the vertices within each frontier of the BFS are assigned to a *level*. These levels have the property that the neighborhood $\mathcal{N}(L(i))$ of all the vertices in a level $L(i)$ is clearly confined to the vertices within the previous, current, and next levels, i.e.:

$$\mathcal{N}(L(i)) \in \{L(i-1) \cup L(i) \cup L(i+1)\}, \text{ for } i > 0. \quad (1)$$

In terms of MPK, this means performing an $A^p x$ computation on $L(i)$ requires the computations of $A^{p-1} x$ to be complete only on the neighboring levels $\mathcal{N}(L(i))$. Due to the nearest-neighbor dependency of the levels, the entire dependency structure simplifies and is similar to a one-dimensional tri-diagonal system, the only difference being that instead of a vertex we use a set of vertices called a level. Therefore, with the level concept, the entire temporal cache blocking schemes existing for the tri-diagonal system can be applied to any general sparse matrix; see Muranushi and Makino (2015) for an overview. RACE uses a parallelotope tiling scheme to achieve cache blocking for MPK. In contrast to the tri-diagonal system, the irregular structure of sparse matrices present some performance challenges. RACE employs various optimization strategies like grouping of levels according to available cache size, point-to-point synchronization, and recursion to obtain a highly efficient implementation of MPK. More details on cache blocking via RACE can be found in Alappat et al. (2023).

Algorithm 1 Computing $A^{p_m} x$ using back-to-back SpMVs. The arrays val, col, and $rowPtr$ hold the CRS data structure of A. The input and output vectors are stored in the y matrix.

Input:
  $double\ A.val[N_{nz}]$ //store values of nonzeros in A
  $double\ A.col[N_{nz}]$ //column index of A
  $int\ A.rowPtr[N_r+1]$ //row pointer of A
  $double :: x[N_r]$ //input vector x
Output:
  $double :: y[0{:}p_m, N_r]$ //to store results of $A^p x$

  $y[0,:] = x[:]$ //starting vector x
  //Perform p_m SpMVs
  for $p = 1 : p_m$ **do**
    $y[p,:]$=SpMV(A, $y[p-1,:]$) //$y_p = A y_{p-1}$
  end for

Algorithm 2 A prototype of SpMV callback function that can be passed to RACE for cache blocking MPK computation. The function is based on CRS data format.

```
function SpMV_callback(int row_s, int row_e, int p,
arg_type kernel_args)
  A = kernel_args.A
  y = kernel_args.y
  //Loop over rows
  #pragma omp parallel for schedule(static)
  for row = row_s : row_e do
    double tmp = 0
    int idx_start = A.rowPtr[row]
    int idx_end = A.rowPtr[row + 1] − 1
    //Loop over nonzeros in row
    for idx = idx_start : idx_end do
      tmp += A.val[idx] ∗ y[p − 1, A.col[idx]]
    end for
    y[p, row] = tmp
  end for
end function
```

On the user side, RACE follows a two-phase approach comprising preprocessing and execution. In the preprocessing phase, RACE performs a BFS and uses the information from the level structure and the cache size of the hardware to determine an execution order that enables cache blocking. The routine to be blocked (here SpMV) is passed to RACE via a user-defined callback function. The callback function takes the range of rows, the current power p, and any input required by the kernel as arguments. In the execution phase, RACE supplies the values to these arguments and executes the kernel in a cache-blocked manner according to the internally created execution order. The user has to write a generic SpMV function computing

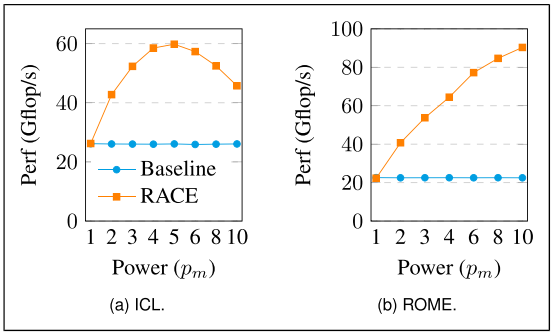

Figure 1. Performance as a function of maximum power p_m for RACE and the baseline implementation of MPK. The experiment was conducted on `Flan_1565` matrix and on ICL (a) and ROME (b). Figure reprinted from Alappat et al. (2023).

$y_p = Ay_{p-1}$ on a range of rows. Algorithm 2 shows a prototype of such a function.

In Alappat et al. (2023), we have shown that RACE's level-based cache blocking achieves significant performance speedup (up to 5×) on MPK computations. The maximum power p_m has a substantial influence on the performance of the cache-blocked MPK. Figure 1(a) shows the MPK performance of the `Flan_1565` matrix as a function of p_m on one socket of ICL compared with the baseline (non-blocked) kernel. As a baseline for comparison we also show the performance of the naive kernel from Alg. 1. Both the RACE and baseline variants are parallelized using OpenMP (Dagum and Menon, 1998). At $p_m = 1$, both the variants are on par as expected. As p_m increases, the RACE variant ideally needs to load the matrix only once from the main memory and the remaining $p_m - 1$ accesses can be served from the caches. Therefore, performance increases with p_m (see Figure 1(a)) until a maximum is reached. Larger p_m has a detrimental effect due to overhead from the blocking. Hence, for maximum performance it is recommended to run the MPK kernel with the optimal power value, which we denote by p_{opt}. If the required p_m of an application is greater than p_{opt} we execute multiple MPKs with p_{opt} until p_m is reached. Of course the last MPK computation (the "remainder loop") might only operate up to a $p < p_{\text{opt}}$. The p_{opt} value depends on the matrix structure and the hardware and needs to be determined once for a given setting. Comparing Figure 1(a) and (b) demonstrates the qualitative impact of the hardware on p_{opt}. Due to its larger cache and massive cache bandwidth, ROME has a higher p_{opt} value and achieves substantially better performance.

In the next sections we will discuss various applications of the MPK and similar kernels in various iterative solvers. Application-specific details and how to reformulate the algorithms to use RACE's cache blocking will be discussed. Via thorough performance analysis we will observe the speedup achieved by cache blocking and also detail some optimization strategies.

4. *s*-step GMRES solver

The GMRES method developed by Saad and Schultz (1986) is a linear solver algorithm that computes an approximation x to the unknown solution x of the linear system of equation $Ax = b$. It constructs x within $x_0 \cup \mathcal{K}_n(A, r) = \{r, Ar, A^2r, \ldots, A^{n-1}r\}$, where $r = b - Ax_0$ is the initial residual vector, $\mathcal{K}_n(A, v)$ is the n-th Krylov subspace with respect to the vector v, and $A^k r$ $(\forall k \in [0, n-1])$ are the Krylov vectors. The size n of the subspace is expanded iteratively to improve the approximation quality of x. Algorithm 3(b) shows the subspace generation routine of the GMRES solver. Within each iteration, the algorithm generates a new vector $v[j+1]$ by performing an SpMV operation with the previous Krylov vector $v[j]$. The newly generated vector is then orthonormalized against all previously generated Krylov basis vectors and added to the subspace. Theoretically, the procedure can be repeated until the system converges. However, within each iteration the memory and computational requirement grows as the subspace is expanded. Therefore, the procedure is restarted every m iterations. The parameter m is commonly known as the restart length of the GMRES solver. The pseudocode in Alg. 3(a) shows the wrapper around the subspace generation routine that restarts the GMRES solver after every m iterations.

As the result of each SpMV operation is fed to the orthogonalization procedure, this dependency prohibits the idea of calling an MPK, and thus the temporal blocking optimizations provided by RACE can not be applied to the subspace generation as presented in Alg. 3(b). However, the alternative s-step formulation of GMRES (see Chronopoulos, 1991; Chronopoulos and Kim, 2020) allows for MPK computations. The basic structure of the s-step GMRES solver remains the same as that of the standard GMRES solver (Alg. 3(a)). However, the Krylov subspace generation is modified to compute blocks of s orthonormal vectors together (see Alg. 3(c)). In the first step, the construction of the Krylov vectors in Alg. 3(c) is done as a sequence of s back-to-back SpMV operations (line 5-7), which can be replaced by an MPK. The subsequent orthonormalization procedure is split into two routines: BOrtho and TSQR. The BOrtho routine orthogonalizes the newly generated block of vectors with the previously generated Krylov basis vectors, and TSQR orthonormalizes the vectors within the block. The main advantage of the s-step variant is that it can use highly efficient BLAS kernels in the orthogonalization routines and effectively reduce the frequency of MPI communications in distributed MPI-parallel setting due to the block-wise computations. This results in a performance speedup over the standard GMRES solver (cf. Yamazaki et al., 2014a). Such implementations have therefore been called "communication-avoiding GMRES" (CA-GMRES) in the literature, see Mohiyuddin et al. (2009), Hoemmen (2010).

We use the s-step GMRES method as implemented in the Belos package (Bavier et al., 2012) of the Trilinos framework. Belos performs back-to-back SpMVs as shown in Alg. 3(c) (lines 5–7) to generate the new Krylov vectors. This part is replaced with our cache-blocked MPK from RACE. Note that, for stability reasons, the actual implementation of the s-step GMRES solver uses a Newton basis instead of the monomial basis (Hoemmen, 2010). This means that the MPK routine computes $[v, (A-\lambda_1 I)v, (A-\lambda_2 I)^2 v, (A-\lambda_3 I)^3 v, \dots]$ instead of $[v, Av, A^2v, A^3v, \dots]$, where the λ_i are just constant shifts. As the shifts only change the matrix diagonal, the RACE adaptation is straightforward and we pass the shifted SpMV callback function to RACE. The λ_i are computed from the eigenvalue information gathered by running a few steps of standard GMRES in Trilinos, see Yamazaki et al. (2014a).

Figure 2 shows the performance advantage of the RACE-accelerated s-step GMRES solver on the ICL and ROME systems (see Sec. 2 for details). The numbers on the x-axis represent the matrix IDs from Table 2; the matrices are ordered by increasing size (N_{nz}). Unless mentioned otherwise, we set the cache size parameter C of RACE to 85 MB and 200 MB for ICL and ROME, respectively. The restart length m of the solver was set to 50. Typically the step size s of the s-step GMRES solver is kept under eight for stability reasons, cf. Hoemmen (2010). In our experiment we used $s = 4$, which limits the maximum matrix power p_m. The numbers above bars in Figure 2 denote the optimal power p_{opt} at which RACE executed the kernel. As this value is the same as p_m, i.e. s, for most matrices, an increase in s will lead to higher speedups. However, even with $s = 4$ we manage to achieve a significant fraction of the maximum MPK speedup. This is in line with the discussion on Figure 1, where we observe substantial performance gains already at low/moderate matrix power values. On our test matrices, RACE accelerates the MPK computation by an average factor of 1.8× and 2.1× over the baseline method on ICL and ROME, respectively. Note that this baseline uses the SpMV provided by the Trilinos package but modified by us to achieve a performance in line with the roofline model (see the discussion in Appendix for details). Otherwise the RACE MPK speedup would be even higher.

Of course only the MPK routine is accelerated by RACE; the runtime of the other routines in the solver will be almost the same for both variants. This reduces the average speedup for the complete solver to 1.3× and 1.2× as seen in Figure 2(a) and (b). On the other hand, the overall speedup stems purely from the performance gain, while numerically both s-step solver variants are identical. The reduced overall performance impact of RACE is mainly due to the significant cost of the orthogonalization procedure (Ortho). In our experiments, one sweep of classical Gram-Schmidt (CGS) was performed in the BOrtho step of the Ortho routine (line nine in Alg. 3(c)) and tall-skinny QR decomposition was used in the TSQR step (line 11). Both of these routines are accounted for in the Ortho time. In order to achieve orthogonality to machine precision, at least two sweeps of CGS + TSQR are required (exactly two for $s = 1$). However, for all examples in this paper the above choice of one sweep proved to be sufficiently stable due to the relatively short restart length. The cost of Ortho is especially high for matrices with low N_{NZR} values because here the runtime complexity for both SpMV and Ortho approaches $\mathcal{O}(N_{\text{R}})$.

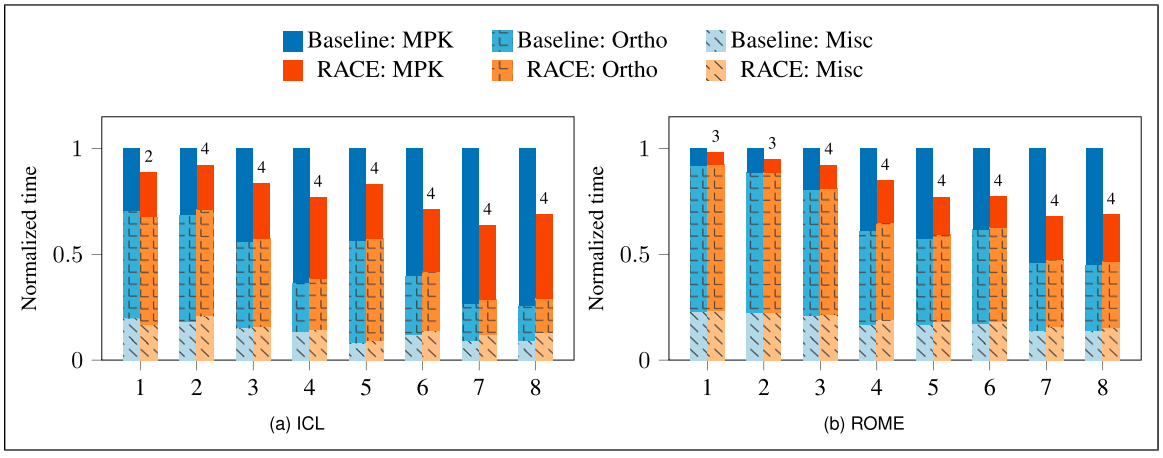

Figure 2. Normalized execution time for 1000 iterations of the s-step GMRES method with (orange bars) and without (blue bars) RACE MPK for the eight matrices (x-axis) shown in Table 2 on ICL (a) and ROME (b). The absolute execution time is normalized to the baseline variant for each matrix separately. The stacked bar plot shows the time contributions of orthonormalization (Ortho) kernels, SpMV kernel and other miscellaneous (Misc) routines. The numbers on top of the orange bars indicate the tuned power value p_{opt} of RACE MPK operation.

Although RACE MPK attains higher speedup on ROME compared to ICL, the total solver speedup on ROME is lower than on ICL. Again the Ortho routine is the culprit as it takes substantially longer on ROME for the same matrices. An in-depth performance analysis revealed that the BLAS calls associated with the Ortho routines performed poorly on ROME (see Appendix). Overall it could be said that the speedup of the s-step GMRES solver achieved by cache blocking is limited by the Ortho routines. As a result, implementing cache blocking on Krylov methods with short recurrence such as the s-step conjugate gradients (CG), where the Ortho cost is minimal, might lead to a solver speedup that approaches the RACE MPK speedup. However, Trilinos currently does not have an s-step CG implementation and we would perform this analysis in the future once the solver becomes available.

The runtime contribution of the remaining kernels (apart from MPK and Ortho) is minor (indicated as "Misc" in Figure 2). Typically this contribution is slightly higher in the RACE variant because it includes the pre-processing cost of RACE (usually 30–50 SpMVs). However, the total number of solver iterations is very large in most of the applications, and the extra cost can be easily amortized.

5. Preconditioners

A GMRES solver is rarely used without a preconditioner because its long recurrence makes it infeasible if the number of iterations increases. Restarting, on the other hand, may drastically increase the total number of iterations and even cause stagnation. A preconditioner transforms the linear system to an equivalent system by formally multiplying the system matrix with another linear operator from the left or right, or both. The goal of this transformation is to improve the condition number of the overall operator, or more specifically to decrease the number of iterations required to solve the system by GMRES. Throughout the paper we will apply the preconditioner from the right, but other choices can be implemented analogously. Hence, the system $Ax = b$ is transformed to $AP^{-1}y = b$, where $y = Px$ is solved for x by applying the preconditioner a final time in the end. The preconditioner P^{-1} is chosen to be some approximation of A^{-1} which is cheap to construct and to apply to a vector. In

Algorithm 3 Pseudocode of the GMRES and s-step GMRES solvers. (a) shows the general algorithmic structure of the solvers. (b) and (c) show the specialized algorithms of the Krylov subspace generation routine for GMRES and s-step GMRES solvers, respectively.

Input:
A, b, x_0 //LSE to solve $Ax = b$, x_0 initial guess
n, tol //n max. iterations, tol convergence tolerance

Output:
x //solution vector
$iter$ //iterations to converge

```
1: x = x_0, iter = 0
2: while iter < n and !converged do
3:    r = b − Ax
4:    v[0 : m − 1] = GenerateKrylovSubspace(A,r)
5:    iter = iter + m
6:    Update x with the vector from the subspace v[0 : m − 1]
      that minimizes the residual
7: end while
```

(a) (s-step) GMRES solver

```
1: function GenerateKrylovSubspace(A,r)
2:    v[0] = r/||r||
3:    for j = 0 : m − 1 do
4:       v[j + 1] = SpMV(A, v[j])
5:       Orthonormalize v[j + 1] against v[0 : j]
6:       //Check for convergence
7:       converged = checkConvergence(tol)
8:       if converged then
9:          break
10:      end if
11:   end for
12:   return v[0 : m − 1]
```

(b) subspace generation routine of GMRES solver

```
1: function GenerateKrylovSubspace(A,r)
2:    v[0] = r/||r||
3:    for j = 0 : s : m − 1 do
4:       //MPK kernel
5:       for p = 0 : 1 : s − 1 do
6:          v[j + p + 1] = SpMV(A, v[j + p])
7:       end for
8:       //BOrtho
9:       Orthogonalize v[j + 1 : j + s] against v[0 : j]
10:      //TSQR
11:      Orthonormalize vectors within v[j + 1 : j + s]
12:      //Check for convergence
13:      converged = checkConvergence(tol)
14:      if converged then
15:         break
16:      end if
17:   end for
18:   return v[0 : m − 1]
```

(c) subspace generation routine of s-step GMRES solver

practice, we then perform the operation $AP^{-1}v$ instead of the SpMV routine computing Av.

In case of s-step GMRES, the introduction of a preconditioner requires us to compute the vectors $[v, AP^{-1}v, (AP^{-1})^2v, (AP^{-1})^3v, \dots]$ instead of $[v, Av, A^2v, A^3v, \dots]$ in the MPK. The main challenge here is that the preconditioner results in an additional dependency between P^{-1} and A, and, using the dependency notation introduced in Section 3, we can denote the MPK dependencies as $P^{-1} \rightarrow AP^{-1} \rightarrow P^{-1}AP^{-1} \rightarrow (AP^{-1})^2 \dots$. We will show that, despite these additional dependencies, it is possible to cache block the preconditioned s-step GMRES using RACE and achieve significant speedups on modern multicore CPUs.

5.1. Relaxation preconditioners

Relaxation preconditioners use iterations from a stationary iterative splitting method. In the following we will investigate two popular choices in this category, Jacobi and Gauss-Seidel, and demonstrate how RACE can be used to accelerate the preconditioned s-step GMRES solvers.

5.1.1. Jacobi. The application of a Jacobi preconditioner to a vector v follows the Jacobi iteration:

$$z^{k+1} = D^{-1}v - D^{-1}(L+U)z^k. \tag{2}$$

Here z^{k+1} and z^k denote the new and old iterate of $P^{-1}v$. The matrices L and U are the strictly lower and upper triangular part of matrix A and matrix D is the diagonal. In many use cases only one Jacobi iteration is applied for the preconditioner. If the initial guess z^0 is also chosen to be zero, the entire Jacobi preconditioner simplifies to $z^1 = D^{-1}v$, which is just a diagonal scaling of the input vector v. Consequently this type of Jacobi preconditioner is also commonly known as diagonal preconditioner.

An advantage of the diagonal preconditioner is that the diagonal scaling $P^{-1} = D^{-1}$ does not introduce any additional dependency between D^{-1} and A and therefore AD^{-1} can be fused to a single kernel. Hence, the actual MPK dependency shown in Section 5 boils down to $AD^{-1} \rightarrow (AD^{-1})^2 \rightarrow (AD^{-1})^3 \dots$ and is analogous to the one seen for the plain MPK routine without preconditioners in Section 3. The diagonally preconditioned SpMV routine computing $AD^{-1}v$ is straightforward and similar to SpMV as shown in Alg. 2, except that it requires an extra diagonal scaling along the columns. The baseline s-step GMRES solver using a Jacobi preconditioner calls this routine s times on the whole matrix to compute the MPK. The RACE variant, however, blocks the matrices A and D^{-1} in cache across the s iterations.[5] To achieve this, we pass the diagonally scaled SpMV callback routine to RACE, which then performs cache blocking based on the internally created execution order, similar to the plain unpreconditioned MPK computations seen in Section 4. Due to the similarities between the plain MPK and the Jacobi-preconditioned MPK, the performance characteristics of s-step GMRES solver remain unchanged (compare Figures 2 and 3). In this case, too, the RACE-accelerated solver achieves an average speedup of almost 1.25× (see Figure 3).

In practice, on CPUs more than one Jacobi iteration is rarely used as a preconditioner as it requires additional SpMVs ($z^k \neq 0$ in (2)). In fact, performing k Jacobi iterations is equivalent to a simple matrix polynomial preconditioner based on the Neumann series $(I-B)^{-1} \approx \sum_{j=0}^{k} B^k$ for $B = -D^{-1}(L+U)$. The cache-blocking approach in RACE MPK offers the opportunity to reduce the computational cost of these additional SpMVs to the point that this approach may be competitive. Cache-blocked polynomial preconditioners can even accelerate a standard Krylov method, as we will show in Section 6. Combining it with an s-step method allows to cache block for higher powers, which may be even more efficient.

5.1.2. Gauss-Seidel. The Gauss-Seidel (GS) preconditioner is derived from the GS iteration:

$$(L+D)z^{k+1} = v - Uz^k. \tag{3}$$

For many linear systems, GS is considered to be superior to Jacobi since it uses the new iterate z^{k+1} whenever available (L is applied to z^{k+1} in (3)). However, shared-memory parallelization is a challenge as a thread does not know when other threads have updated their z entries. Two well-known solutions to this triangular solver problem are multicoloring (Evans (1984)) and level scheduling (Anderson and Saad, 1989). Reordering via multicoloring often degrades the data locality and the convergence rate, resulting in performance loss. On the other hand, level scheduling maintains the convergence rate but often exhibits limited parallelism. Another promising solution, especially for preconditioners, is the two-stage Gauss-Seidel (GS2) iteration, see Lanzkron et al. (1990). Here a fixed number of Jacobi-Richardson iterations is used to solve (3). This means that, within each iteration of GS, we have inner iterations of Jacobi-Richardson. The benefit with this approach is that we can solve the system using simple SpMVs and BLAS-1 operations. This technique has been used to increase parallelism for GPUs, Chow et al. (2018), Berger-Vergiat et al. (2021), but not for cache blocking on multi-core CPUs. Of course, in contrast to level scheduling, the system is not solved exactly with the Jacobi-Richardson iterations; however, it was shown in Berger-Vergiat et al. (2021) that for preconditioners, where A^{-1} is already approximated, this method produces similar convergence rates for many matrices.

A GS2 iteration algorithm based on the non-compact form of the GS iteration (see Berger-Vergiat et al., 2021 for details) is shown in Alg. 4(a). As in the Jacobi case, typically only one outer iteration ($K = 1$) of GS2 is employed as a preconditioner and frequently combined with one ($\gamma = 1$) or two ($\gamma = 2$) inner

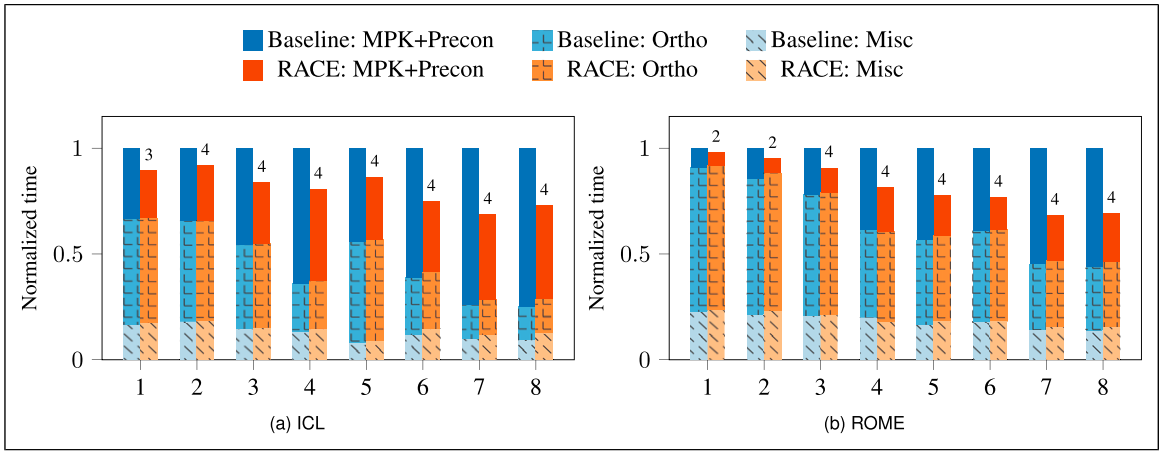

Figure 3. Time taken by baseline and RACE accelerated variant of s-step GMRES solver using Jacobi preconditioner. The stacked bar plot displays the time contributions by orthonormalization (Ortho) kernels, SpMV kernel and other small miscellaneous (Misc) routines.

Jacobi-Richardson iterations. The cache-blocking of GS2 preconditioner with RACE is more involved than the Jacobi counterpart. Here, the preconditioner P^{-1} itself involves many interdependent steps. The dependency within P^{-1} of GS2 with $\gamma = 2$ can be expressed as $U \rightarrow LU \rightarrow L^2U$. Finally, when applying the preconditioner to the matrix in the s-step GMRES solver, there is an additional dependency $P^{-1} \rightarrow A$. This means that even in a single step $(s = 1)$ of the s-step GMRES solver we have a dependency chain of length four. To allow for an easy integration of RACE into the such preconditioners, we divide each power computation into a fixed number of *sub-powers*. In case of GS2 as shown above we would have four sub-powers within a power, and the power loop in RACE will map to the power loop along the MPK computations of the s-step GMRES solver (e.g., line five in Alg. 3(c)). Algorithm 4(b) shows the MPK routine with the GS2 $(\gamma = 2)$ preconditioner that is passed to RACE for cache blocking. Here we distinguish each stage of the dependency chain using the sub-power (j in Alg. 4(b)), which goes from zero to $\gamma + 1$ (three in this case) for each power computation. The first three sub-powers compute the application of the GS2 preconditioner on a vector v, and the result is stored in vector z^1. The last sub-power, i.e., at $j = 3$ in case of Alg. 4(b), calculates Az^1. Note that "sub-power" is just a convenient abstraction on top of the power loop in RACE; it satisfies all the BFS level dependencies mentioned in Section 3, ensuring that the dependencies within P^{-1} (i.e., $U \rightarrow LU \rightarrow L^2U$) are met. Of course in this case the callback function passed to RACE will have an extra input argument for

Algorithm 4 (a) Pseudocode of a two-stage Gauss-Seidel (GS2) iteration with γ inner Jacobi-Richardson iterations. The k-loop is the outer Gauss-Seidel iteration and j is the inner iteration. (b) Unrolled implementation of MPK with GS2 preconditioner ($K = 1$, $\gamma = 2$) passed to RACE for cache blocking. The implementation takes an input vector $v[p]$ and performs the computation $v[p+1] = AP^{-1}v[p]$, where P^{-1} is the GS2 preconditioner.

Input:
L, D, U //$A = L + D + U$
$v[p]$, z_0 //$v[p]$ input vector, z_0 initial guess
K, γ //K outer iterations, γ inner iterations

1: **for** $k = 0 : K - 1$ **do**
2: $g_k^0 = D^{-1}(v[p] - Uz^k)$
3: **for** $j = 1 : \gamma$ **do**
4: $g_j^k = g_0^k - D^{-1}Lg_{j-1}^k$;
5: **end for**
6: //update
7: $z^{k+1} = z^k + g_\gamma^k$
8: **end for**
9: $v[p+1] = Az^K$;

(a) GS2 pseudocode

Output:
$v[p+1]$ //output vector storing $AP^{-1}v[p]$

1: **if** j == 0 **then**
2: $g_0^0 = D^{-1}(v[p] - Uz^0)$
3: **else if** j == 1 **then**
4: $g_j^0 = g_0^0 - D^{-1}Lg_{j-1}^0$
5: **else if** j == 2 **then**
6: $g_j^0 = g_0^0 - D^{-1}Lg_{j-1}^0$
7: $z^1 = z^0 + g_j^0$
8: **else if** j == 3 **then**
9: $v[p+1] = Az^1$
10: **end if**

(b) MPK with GS2 of $\gamma = 2$

the sub-power j, which will be imported by RACE internally during the execution phase. The levels in RACE are still generated from matrix A only, as the matrices U and L have a subset of the sparsity pattern of A.

In contrast to the plain unpreconditioned or Jacobi-preconditioned s-step GMRES solver, a GS2 preconditioned solver has the benefit that even with $s = 1$ (normal GMRES solver) some performance advantage is possible since we reuse the matrices within P^{-1} and between P^{-1} and A. For example, with two inner iterations $(\gamma = 2)$ a straightforward implementation requires to load the matrices A and U once and L twice. However, with the cache-blocked variant ideally we need to load the matrix L only once. We can further save the traffic from matrix A if we perform the SpMV with a split form, i.e., $A = L + D + U$, leading to reuse in matrices L and U. This optimization is applied in our RACE implementation. For s-step GMRES solvers with $s > 1$, this benefit adds to the advantage of blocking the matrices to higher powers.

The GS2 preconditioner is implemented in the Ifpack2 package (Prokopenko et al., 2016) of Trilinos. In this section, we use this preconditioner for the s-step GMRES solver as a baseline for comparison. Similar to the Jacobi preconditioner, it is common practice to choose the starting vector z^0 to be zero. This allows for short-circuiting some computations in the GS2 iteration, i.e., line two in Alg. 4(a) simplifies to $g_0^k = D^{-1}v[p]$ and the update step (line 7) to $z^{k+1} = g_\gamma^k$. The Ifpack2 implementation currently initializes the vector with zero by default but it does not short circuit the computations although the Kokkos backend supports this. This mainly results in an additional overhead of half an SpMV (Uz^k). In the interest of a fair comparison we modified the Ifpack2 code to allow for short-circuiting the unnecessary computations as well.

Figure 4 shows the performance of the GS2-preconditioned s-step GMRES solver with $\gamma = 1$ on ICL and ROME. On ICL, the power value at which RACE operates is lower compared to the previously discussed s-step solvers because each power step (p) contains multiple sub-power computations. Thus, the effective total power to which RACE applies cache blocking is the product of the two power computations and the maximum performance is achieved at lower p_{opt} (see discussion of Figure 1(a)). However, on ROME most matrices reach the maximum power value of four $(p_{\text{opt}} = s)$ due to its large cache size. Increasing the γ from one to two improves the speedup slightly from 1.22× to 1.3× on ICL (not shown in figure), as more reuse can be applied within the inner iterations.

The results for the GS2 preconditioner demonstrate the applicability of RACE to a wider range of preconditioners that require chaining of multiple routines. This includes sparse approximate inverse preconditioners where the P^{-1} is explicitly computed and can be chained effectively with the matrix A to implement the s-step GMRES solver. Similarly, factorization-based preconditioners like ILU can be implemented with RACE using the chaining idea and applying Jacobi-Richardson iterations to solve the triangular systems Chow et al. (2018). In this paper we do not investigate further on this class of preconditioners but rather demonstrate the applicability of RACE to two different classes of preconditioners which show significant performance improvement even on standard $(s = 1)$ GMRES solvers.

5.2. Polynomial preconditioners

A polynomial preconditioner has the form $P^{-1} = \mathcal{P}(A)$, where $\mathcal{P}$ is a polynomial. Such preconditioners have been extensively studied in the context of Krylov-based solvers, see Johnson et al. (1983), Saad (1987). Stability and setup costs of polynomial preconditioners, such as extreme eigenvalue calculations, were major concerns for a long time. However, many recent studies, e.g. Loe and Morgan (2022), Ye et al. (2021), have stimulated renewed interest in these methods. In particular, the lack of global communication in the evaluation of the polynomial make them attractive for large-scale computing.

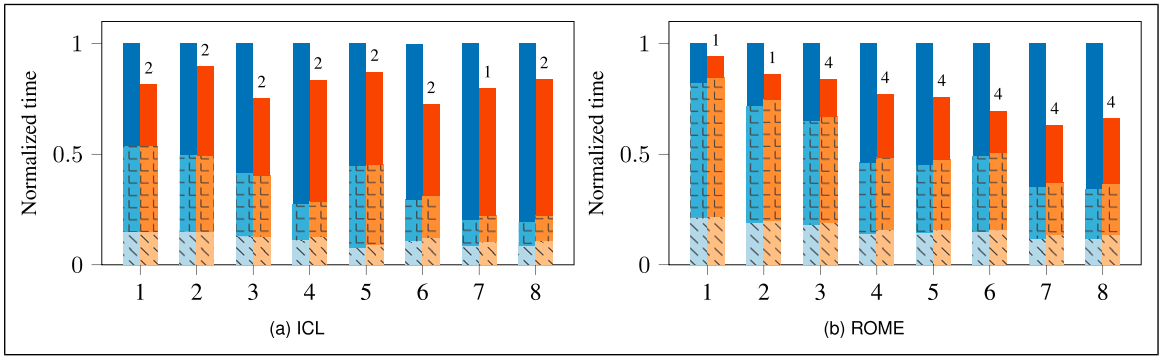

Figure 4. Time taken by baseline and RACE-accelerated variant of the s-step GMRES solver using the GS2 preconditioner with one inner Jacobi-Richardson iteration. The legend is the same as in Figure 3.

In each preconditioning step, a polynomial of degree d is applied. The application of P^{-1} to a vector x can be expressed as

$$P^{-1}x = \mathcal{P}(A)x = \lambda_0 x + \lambda_1 Ax + \lambda_2 A^2 x + \cdots + \lambda_d A^d x, \quad (4)$$

where the $\{\lambda_i\}$ are scalar coefficients that determine the type of polynomial, with Chebyshev (Johnson et al., 1983) and GMRES polynomials (Abdel-Rehim et al., 2014) being the most popular ones. Here we focus on the GMRES polynomial, where the scalar constants are generated by running d iterations of a GMRES solver in a pre-processing step.

The optimal degree d for GMRES polynomial preconditioners is rather high (in the range of 40–100), thus applying the preconditioner requires many back-to-back SpMVs, making this operation frequently the dominant part of the solver.[6] This is a very attractive scenario for the MPK of RACE as potentially large speedups are achievable and it directly applies to the hotspot of the solver. Thus, we exclusively focus on accelerating the polynomial preconditioner, which is called in each iteration of the GMRES solver. In the following the parameters C and m remain similar to the previous experiments with the s-step GMRES solver (see Section 4). We use the Belos package of Trilinos as the implementation baseline (see Loe et al., 2020 for details) and choose a polynomial of degree 80.

Figure 5 shows the performance benefit when using RACE to accelerate the polynomial preconditioner in the GMRES solver. The striking observation is that the speedup obtained by RACE is significantly higher than previously observed with s-step GMRES solvers. There are two reasons for this: First, the polynomial application $P^{-1}x$ consumes a significant fraction of the entire solver runtime (more than 95% for most matrices; see Figure 5). Second, high powers in the MPK computations (80 in our case) allow RACE to block for higher power values and thus operate at high performance levels. This is also the reason why most of the matrices on ROME operates at $p = 8$, where eight is the highest power value in our tuning space of $p \in [1:8]$. Higher p did not prove to be significantly faster. On ICL, the optimal power values for RACE are lower due to the smaller cache size and therefore most of the matrices have $p_{\text{opt}} < 8$. Overall, RACE improves the MPK performance on ROME (ICL) by an average factor of almost 3× (2×), which translates to an average 2.7× (1.9×) speedup on the entire GMRES solver.

GMRES polynomial preconditioners can be further combined with other preconditioners. In such scenarios, the polynomial is formulated in terms of the combined matrix AM^{-1}, where M^{-1} is another preconditioner. The application of the preconditioner then reads:

$$\begin{aligned} P^{-1}x &= \mathcal{P}(AM^{-1})x \\ &= \lambda_0 x + \lambda_1 AM^{-1}x + \lambda_2 (AM^{-1})^2 x + \ldots \end{aligned} \quad (5)$$

The dependencies caused by the new matrix M^{-1} have to be taken into account in the cache blocking, and a similar approach to the one described in Section 5.1 is in order. In case of a Jacobi preconditioner, only matrix diagonal scaling is required and we attain almost the same speedup as with plain polynomial preconditioning as shown in Figure 5. The speedups obtained by RACE when using the GS2 $(\gamma = 1)$ preconditioner on top of the polynomial preconditioner are shown in Figure 6. Although the computational kernels remain similar to the ones discussed in Section 5.1.2 above, the speedup is much higher (1.5× and 2.1× on ICL and ROME) in the case of polynomial preconditioners due to the two reasons discussed in the previous paragraph.

5.3. Algebraic multigrid preconditioners

Algebraic Multigrid (AMG) preconditioners are among the most widely used preconditioners for Krylov solvers (Wathen, 2015). AMG preconditioners are particularly

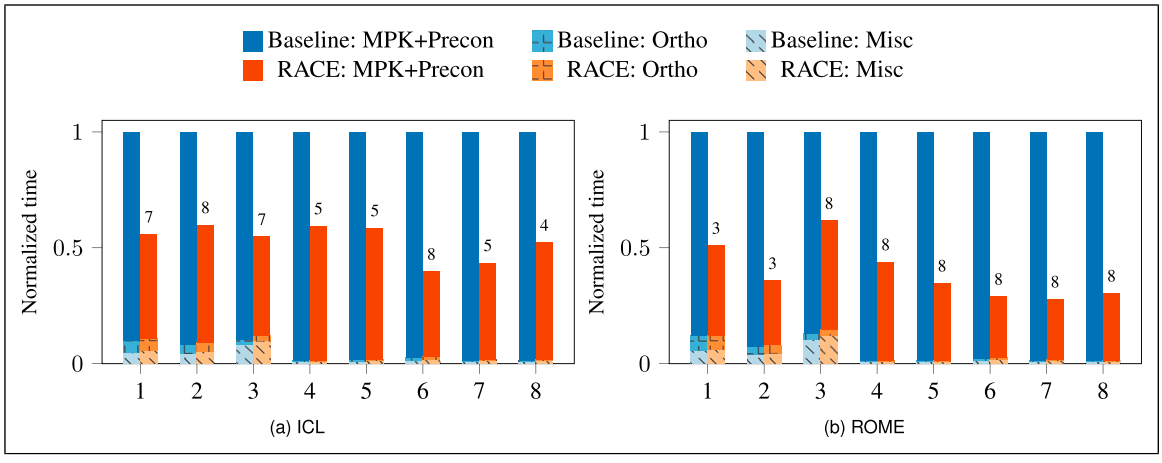

Figure 5. Time taken by the baseline and RACE-accelerated variants of the GMRES solver using a polynomial preconditioner of degree 80.

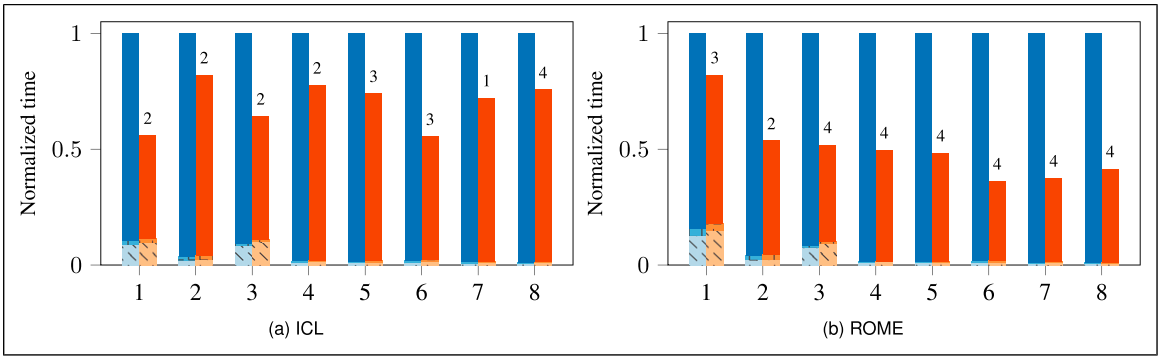

Figure 6. Time taken by baseline and RACE-accelerated variants of the GMRES solver using GS2 with one inner iteration on top of the polynomial preconditioner of degree 80. Colors have the same meaning as in Fig. 5.

effective for solving large 3D problems. Similar to any multilevel or geometrical multigrid schemes (Brandt, 1977), the AMG preconditioner uses a hierarchy of grids with various refinement (discretization) levels. Inter-grid transfer operators are used to transfer information between the grid levels. The restriction and prolongation operators are the two inter-grid transfer operators used to transfer information from a fine to coarse grid and vice versa. Within each grid level a smoothing operator is applied to reduce the error within the level. In contrast to geometrical multigrid, AMG algebraically determines the coarse grids and the inter-grid transfer operators, both of which are based on the matrix entries, see Falgout (2006) for example. As AMG does not require any explicit knowledge of the problem geometry, it is particularly useful for problems having a complicated or even unknown geometry.

Algorithm 5 Pseudocode of a single AMG V-cycle, adapted from Thomas et al. (2019). The letter k denotes the grid level.

Input:
A, v //LSE to solve $Az = v$
max_levels //max. levels of refinement
Output:
z //solution vector

1: $z = 0$
2: AMG(A,v,z,0)
3: **function** AMG(A_k,b,x,k)
4: $x = S_k^{\text{pre}}(A_k, b, x)$ //Pre-smoothing
5: **if** $k \neq max\_levels - 1$ **then**
6: $r_k = b - A_k x$ //Residual
7: $r_{k+1} = R_k(r_k)$ //Restriction on the residual
8: $c_{k+1} = 0$
9: //Call AMG with next coarser matrix A_{k+1}
10: AMG(A_{k+1}, r_{k+1}, c_{k+1}, $k + 1$)
11: $c_k = P_k c_{k+1}$ //Prolongation on the correction
12: $x = x + c_k$ //Add correction
13: $x = S_k^{\text{post}}(A_k, b, x)$ //Post-smoothing
14: **end if**

A single iteration of AMG starts with a smoothing operation (pre-smoothing) performed on the finest grid level with an initial guess of zero. The residual is then calculated on the finest level and transferred to the next coarser level using the restriction operator. Then, smoothing is performed on the next coarser level. The same procedure (grid transfer to the next, coarser level and smoothing) is performed throughout all levels in the hierarchy until the coarsest level is reached. The linear system on the coarsest level is usually solved by a direct solver. This solution then serves as a correction to the next finer level and is transferred using the prolongation operator. The smoothing operation is again performed (post-smoothing) on the finer level using the correction as the initial guess. The grid then transfers the solution to next finer level and the process repeats until we reach the finest level. Due to the manner in which the grids are traversed, i.e., finest to coarsest and then back to finest, this is called *V-cycle AMG*. Although there are many other types of cycles, we will concentrate on the V-cycle in this paper. When using AMG as a preconditioner, a single V-cycle of AMG is typically used to compute $P^{-1}v$. Algorithm 5 shows the corresponding high-level algorithm computing $z = P^{-1}v$, where P^{-1} is an approximation to A^{-1}.

For large problems, most of the solver's runtime is spent in the smoothing operation on the finest grid level. Typically, a few sweeps of simple iterative methods like Jacobi or GS are used for this. As the matrix does not change between the sweeps, RACE can cache block the matrix entries. For example, the GS2 sweeps introduced in Section 5.1.2 can be used as a smoother and RACE can block both for inner Jacobi-Richardson iterations within GS2 and outer sweeps of the smoother. Figure 7(a) and (b) demonstrate the speedup attained by RACE over the baseline method when using AMG with the GS2 smoother ($\gamma = 1$). The baseline performs a single V-cycle of smoothed-aggregation AMG (Míka and Vaněk, 1992) provided by the MueLu package (Berger-Vergiat et al., 2019) in Trilinos as the preconditioner to the GMRES solver from Belos package. The pre-smoothing employs two forward sweeps of GS2 while the

post-smoothing uses two backward sweeps (L and U are exchanged in Alg. 4). The GS2 smoother baseline is provided by the Ifpack2 package. For the RACE variant we modified the MueLu code such that the cache-blocked variant of GS2 is used as the smoother for the finest level. The only difference from the previous implementation shown in Alg. 4(b) is that we do not need the computation of Az^1 in the last sub-power, i.e., at $j = 3$ in Alg. 4(b). In case of pre-smoothing, at the last sub-power we instead compute the residual that is required in the next step of AMG (line six in Alg. 5). This allows us to reuse the matrix A_k from the cache when computing the residual. Note that this reuse is on top of the reuse in the GS2 sweeps. As the number of sweeps in the smoother is typically small (in the range of 1–4) we do not tune the power value at which RACE operates but set it to account for all the sweeps and the residual computation. The post-smoother performance can also theoretically benefit by fusing it to the next kernel. Post-smoothing is the last step of AMG preconditioner; the next step of the Krylov solver (in case of right preconditioning) involves an SpMV of the matrix with the preconditioned vector. By integrating this SpMV as the last sub-power computation, the matrix can be served from the cache. However, in the current implementation we do not do this as the SpMV is performed not by the MueLu package but by the Belos package and therefore would involve fusing kernels from two different packages, which is possible but requires significant changes.

From Figure 7(a) and (b) we see that the cache blocking of the GS2 smoother by RACE achieves a moderate overall speedup of 17% and 37% in the solver time compared to the baseline on ICL and ROME, respectively. For the first time a slowdown of the RACE variant is encountered: With the `thermal2` matrix (matrix ID = 2) the overall RACE runtime is 19% and 6% higher than the baseline on ICL and ROME, respectively. This is due to the extra cost of RACE's pre-processing (see miscellaneous contributions in Figure 7), which is typically in the range of 30–50 SpMVs (see Alappat et al., 2023 for more details). As the number of solver iterations can be relatively small for AMG-preconditioned solvers (28 in case of the `thermal2` matrix), this cost cannot always be amortized.

Figure 7(c) and (d) show how the number of outer sweeps for the GS2 smoother with one and two inner iterations (γ) influences the speedup. Increasing these parameters improves the speedup as higher effective powers in RACE can be used. This effect is most pronounced on ROME where we achieve 40% improvement when applying three outer sweeps instead of a single one.

Good smoothers have the general property that they dampen the error component orthogonal to the coarse grid correction step, which typically means damping high-frequency errors (Adams et al., 2003). GS-based smoothers enjoy this property. Another very attractive and commonly used smoother in this regard is the Chebyshev polynomial smoother. The polynomial is tailored to dampen the high-frequency errors and is constructed using spectral information and Chebyshev recursion. Similar to the polynomial preconditioners described in Section 5.2, the Chebyshev polynomial has the property that it can be solely implemented with MPKs as it takes the form shown in (4). In contrast to polynomial preconditioners, the degree d of the polynomial smoother is typically low (less than 10). When using Chebyshev polynomials, each smoothing step of AMG computes the application of a Chebyshev polynomial to a vector using the MPK, which is subject to cache blocking via RACE. Note that in case of pre-smoothing we also cache-block the residual computation similar to the GS2 pre-smoothing seen above. As a baseline for comparison we use the Chebyshev smoother from the Ifpack2 package, which implements specialized (with appropriate scales and shifts) SpMV-based MPK kernels in Kokkos; its performance is impacted by the same dynamic scheduling problem for larger matrices as discussed in Section 4. Again we modified the code to always use static scheduling in the baseline variant.

Figure 8(a) and (b) compare the solver time of baseline and RACE variants using AMG with the Chebyshev smoother of degree three. On average, the RACE variant achieves a speedup of 24% and 32% on ICL and ROME, respectively. Interestingly, the performance benefit of RACE tends to increase with matrix size[7], e.g., on ROME a 1.7× speedup is attained for the largest matrix (`Flan_1565`). This is mainly due to three reasons: First, for small problems with low iteration counts, the Misc contribution (including RACE preprocessing) to the runtime is significant. Second, the small test matrices that we have considered (see Table 2) also tend to have a low N_{NZR}. This makes the smoothing operation less prominent compared to the BLAS-1 type (vector-only) operations. Third, RACE cache blocking is currently only implemented on the finest grid level and the share of this level in overall AMG runtime increases as matrix size increases (see Figure 8(c)). Another observation in line with our previous results is the positive correlation between the polynomial degree d and RACE's speedup (see Figure 8(d)).

In summary, above experiments have demonstrated that RACE provides moderate speedups on the AMG-preconditioned GMRES solver. Two main factors that currently prevent larger speedups on some matrices are the large time contribution from coarser grid levels and the low number of solver iterations.

6. Case study: Momentum equation in the Nalu-Wind solver

To demonstrate a practical use case of RACE's cache-blocking technique we consider the dominant sparse

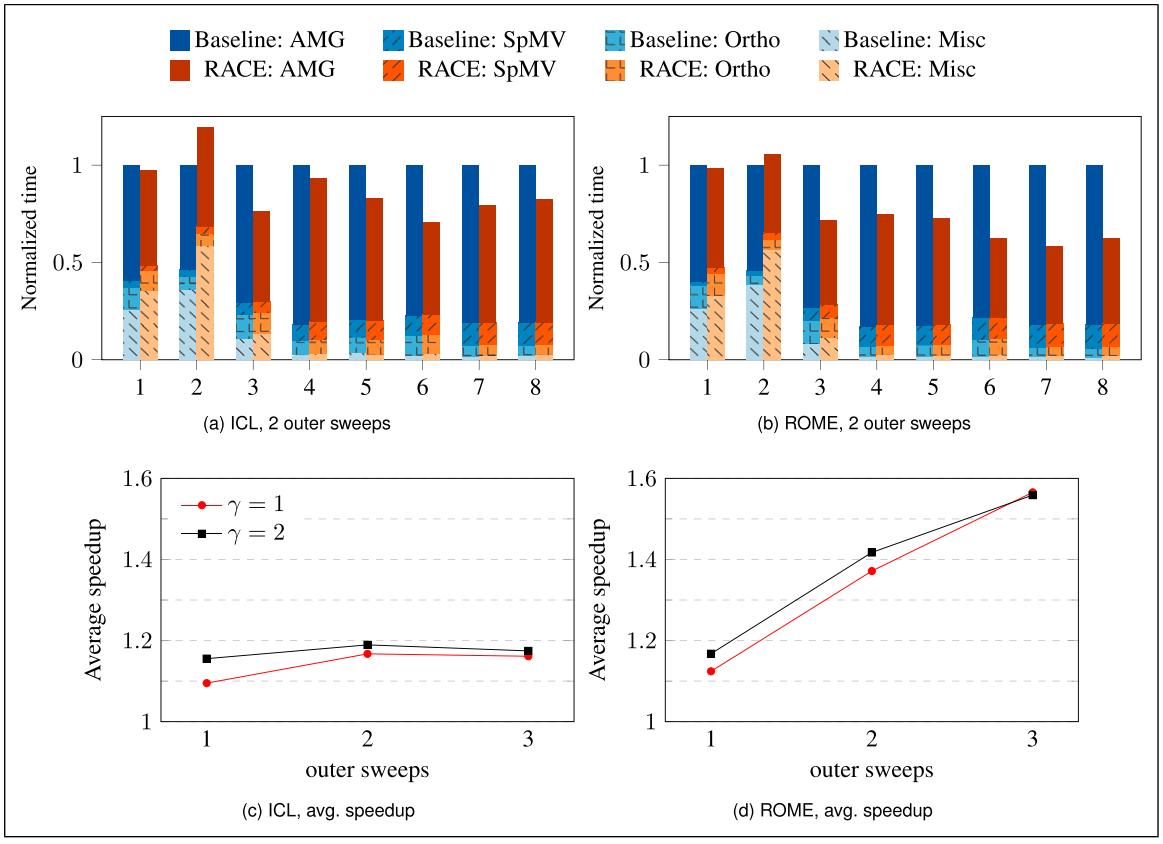

Figure 7. (a), (b) Comparison of time taken by the baseline and the RACE-accelerated variant of the GMRES solver preconditioned by the algebraic multigrid preconditioner using the GS2 smoother with one inner iteration (γ = 1) and two outer sweeps. (c), (d) Improvement in speedup (averaged across the eight benchmark matrices) as the number of outer sweeps is increased on ICL and ROME.

linear system of equations (LSE) in the Nalu-Wind simulation code (Sprague et al. (2020)). The LSE arises when solving the unsteady compressible Navier-Stokes equations for the velocities in the simulation of wind turbines. Specifically, we focus on the case of a large-eddy simulation of two aligned wind turbines under uniform flow, where the turbine blades are modeled using the actuator line model (example 1.3.4 in Nalu-Wind documentation[8]). We use a mesh with 256×256 horizontal cells and 64 layers, which translates to a momentum matrix with 12 million rows and almost 300 million nonzeros. In this scenario the numerical behavior of the linear systems is very similar between time steps after a short start-up phase. The purpose of this case study is not to claim or find an optimal solver but to demonstrate that cache blocking techniques should be taken into account when selecting and tuning linear solvers for best time to solution.

This application employs an established approach in computational fluid dynamics (CFD), where the model state is propagated forward in time using an ODE (ordinary differential equation) time-stepping method, and discretized PDEs (partial differential equations) in space are solved in each time step for momentum and conservation, which leads to the sparse linear systems. Nalu-Wind uses the Trilinos library for solving these time-consuming sparse linear systems. While the time-integration scheme is implicit and thus unconditionally stable, a small time step is used to cover the relevant physical scales as wind turbines rotate at high velocities. Consequently, the LSE arising from the PDEs is diagonally dominant and GMRES converges quickly: For the matrix studied here, the default method (GMRES with GS preconditioning) achieves a residual norm of 10^{-12} in 13 iterations. This results in a total runtime of 1.83 s on ROME. We will use this a baseline for comparison when investigating cache blocking in combination with different preconditioning strategies.

Due to the small number of iterations, s-step GMRES is unsuitable as it needs a few steps of standard GMRES to calculate the Newton shifts (see Section 4). However, using a polynomial preconditioner instead of GS may enable accelerating the polynomial application using RACE (see Section 5.2). Figure 9(a) compares the time required to solve the LSE with and without RACE for different polynomial degrees. The polynomial preconditioner was

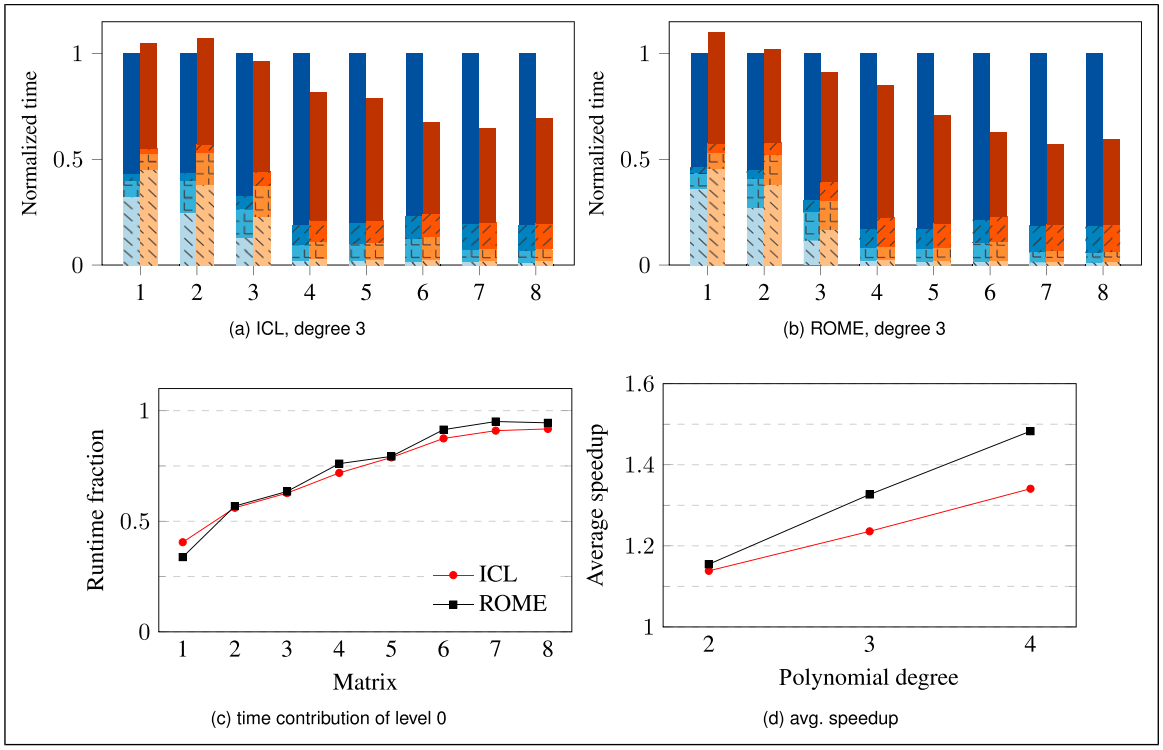

Figure 8. (a), (b) Comparison of time taken by the baseline and the RACE-accelerated variant of the GMRES solver preconditioned by algebraic multigrid using Chebyshev smoothers of degree three (same legend as in Fig. 7). (c) Fraction of AMG runtime spent on the finest level (i.e., level 0) for the different matrices. (d) Average speedup of the solver as a function of Chebyshev polynomial degree.

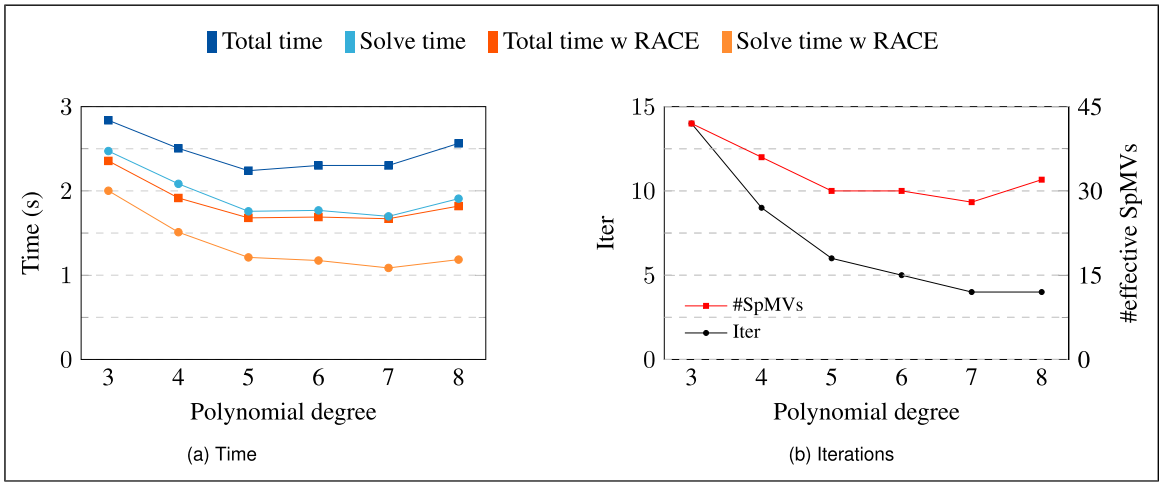

Figure 9. (a) Time required to solve the momentum equation using different degrees of the polynomial + Jacobi preconditioner with and without RACE on ROME. (b) Number of solver iterations and the effective number of SpMVs performed.

combined with Jacobi in this case as this proved to be most effective. Clearly the time to solution can be reduced by increasing the polynomial degree (blue lines in Figure 9(a)). This is correlated with a decrease in the number of iterations (see Figure 9(b)), which has two positive effects: First, the total number of SpMVs, the product of iterations and polynomial degree, goes down up to a certain point (red line in Figure 9(b)). Second, the number of orthogonalization steps decreases. The combination of theses effects leads to the decrease in solver time as the polynomial degree

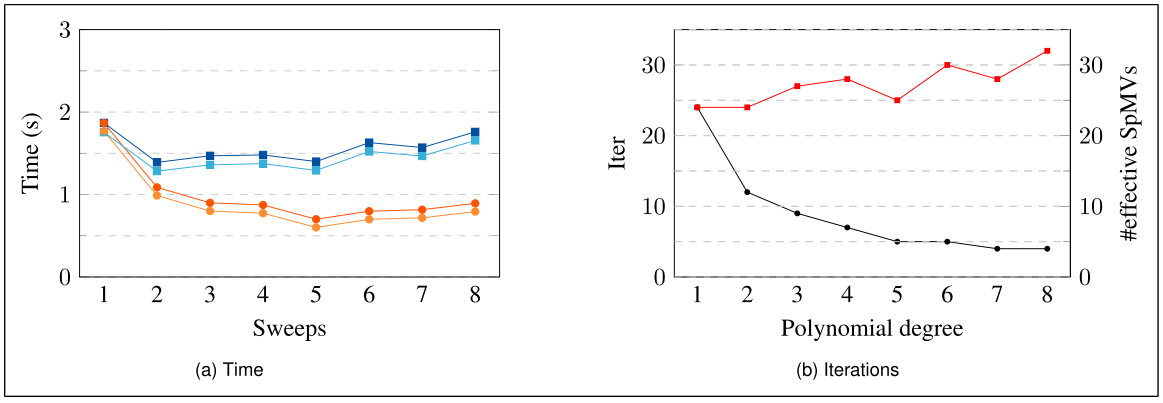

Figure 10. (a) Time required to solve the momentum equation using multiple sweeps of the Jacobi preconditioner with and without RACE on ROME. (b) Number of solver iterations and effective number of SpMVs performed. The same legends as in Figure 9 apply here.

Table 3. Overview of the effectiveness of different preconditioners on the momentum equation in Nalu-Wind. The rows show the number of solver iterations, the effective number of SpMV-like operations, pure solve time, and the total time including the setup cost for the preconditioner.

Preconditioner	None	Jacobi	GS	Poly, d = 5	RACE + poly, d = 5	Jacobi, d = 5	RACE + Jacobi, d = 5
Iter	43	24	13	6	6	5	5
#Eff. SpMVs	43	24	26	30	30	25	25
Solve time (s)	3.72	1.73	1.73	1.76	1.16	1.29	0.60
Time (s)	3.83	1.83	1.83	2.24	1.64	1.40	0.70

increases up to seven. However, the total time, which includes solver and setup time, is lowest at degree five because the setup cost, which primarily includes d SpMVs where d is the polynomial degree, increases linearly with the degree. Nevertheless, the default Trilinos implementation of the polynomial preconditioner (without RACE) achieves a minimum runtime (at $d = 5$) of 2.24 s, which is approximately 20% slower than the above specified baseline (GMRES and GS preconditioner). This picture changes if the polynomial preconditioner uses RACE's cache blocking (orange lines in Figure 9(a)). Time to solution reduces to 1.68 s, which is a 9% improvement over the baseline. Again, the main reason for the limited speedup is the relatively high setup cost of the polynomial preconditioner, which incurs a huge overhead for the small number of iterations at hand. We may conclude that any sophisticated preconditioner with high setup cost would not be effective in this context because the setup has to be repeated in every time step due to the updated momentum matrix. On the other hand, the RACE setup cost of 30–50 SpMVs for creating the graph traversal scheme may be neglected in applications where the matrix sparsity pattern stays constant for all time steps, as then the RACE setup is done only once for the entire simulation.

Given these observations, a viable strategy may be increasing the sweeps of basic relaxation preconditioners, which have negligible setup cost. Figure 10(a) shows the benefit of increasing the number of Jacobi sweeps in the standard GMRES preconditioner. In this case the number of iterations also decreases linearly while the SpMV counts remain almost constant up to a certain sweep count (see Figure 10(b)). Remember that the cost of every Jacobi sweep is similar to an SpMV (see (2)). Although the number of SpMVs remains the same, we reduce the orthogonalization cost in GMRES, which effectively reduces the solver time (blue lines). In this case, Jacobi with five sweeps converges in 1.4 s, achieving a speedup of 1.3× over the baseline. Multiple sweeps of Jacobi imply that the same SpMV-like operator is applied back-to-back, which can be accelerated via RACE. This further reduces the time to solution to 0.7 s, which is an additional improvement of a factor of two. Table 3 summarizes the results, showing that a total speedup of 2.6× over the default solver is possible.

This study does not make any claims on the optimality of the chosen preconditioners but demonstrates the runtime impact of RACE's cache-blocking technique. The above findings indicate that RACE may be particularly useful in situations where the sparsity pattern of matrix is constant (over a long time), but the values of matrix elements change too frequently to afford the cost of computing a strong preconditioner repeatedly.

7. Conclusion and outlook

In this article, we demonstrated that the node-level performance of various sparse iterative solvers can be boosted by performing temporal cache blocking using the Recursive Algebraic Coloring Engine (RACE). The key is to identify steps in the solver and/or preconditioner which can be (re) formulated into matrix polynomials and then replace the related back-to-back sparse matrix-vector operations with RACE's cache-blocked matrix powers kernel.

First we investigated s-step GMRES as a method representative of the broad class of s-step Krylov methods. Their basic structure allows to easily benefit from cache-blocked MPKs. The raw performance improvement of the MPK can be utilized in parts of these solvers, leading to speedups up to 1.5× for the full solver. For short-recurrence Krylov methods like conjugate gradients (CG), where the orthogonalization cost is low, the overall improvement may be substantially higher. Second, we showed how to apply cache blocking when combining the s-step method with preconditioners. Here we addressed the additional problem of calculating polynomials on parts of the sparse matrix, e.g., the triangular factors of the matrix. Using a two-stage Gauss-Seidel preconditioner, we further illustrated that cache blocking can be performed across multiple chained operators. Third, the challenges and benefits of using RACE in the context of polynomial and algebraic multigrid preconditioners were evaluated. These preconditioners are suitable even for standard Krylov methods, which broadens the scope of our work. Fourth, we showed that a thorough performance analysis is required to perform fair comparisons with baseline implementations. Performance problems in the baseline libraries have been identified (e.g., a scheduling issue in Trilinos SpMV Kernels) and fixed, if possible. Furthermore we showed that the efficiency of cache blocking can be improved by combining it with inter-kernel optimizations (e.g., fusing the pre-smoother with the residual computation of AMG). Finally, using a case study from wind turbine simulation we illustrated the potential impact of our approach on a real-world application.

RACE's optimizations can be applied to accelerate a variety of other applications like eigenvalue solvers, Chebyshev time propagation, and exponential time integration. In the future we plan to extend this work to multi-node distributed systems and GPUs.

Acknowledgements

The authors would like to thank NHR@KIT for providing access to the HoreKa supercomputer (ICL system), which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was partially supported by NHR@FAU, which is funded by the State of Bavaria and by the Federal Ministry of Education and Research and Ministry of Science, Research and the Arts Baden-Württemberg.

ORCID iDs

Christie Alappat https://orcid.org/0000-0003-4548-8727
Georg Hager https://orcid.org/0000-0002-8723-2781

Notes

1. Top 500 list: https://top500.org/lists/top500/2022/06/
2. Exact version of RACE used for experiments: https://github.com/RRZE-HPC/RACE/tree/v0.8.0
3. Modified Trilinos repository used for experiments: https://github.com/christiealappatt/TrilRACE/commit/119adc404d5c5d7f965970d86ec8a91205ab247a
4. See https://doc.zih.tu-dresden.de/jobs/_and_/resources/rome/_/nodes/for details on running Intel MKL code on AMD processors.
5. Note that the D^{-1} matrix has only one entry per row and is therefore stored as a vector.
6. Note that the high computational cost of the preconditioner is often amortized by a decrease in the total number of iterations, making the preconditioner effective.
7. Matrices are ordered according to increasing size; see Table 2.
8. Nalu-Wind documentation release 1.2.0: https://nalu-wind.readthedocs.io/_/downloads/en/latest/pdf/)
9. AOCL-BLIS was compiled with gcc v10.2.0 as the library did not support our de facto Intel compiler.

References

Abdel-Rehim AM, Morgan RB and Wilcox W (2014) Improved seed methods for symmetric positive definite linear equations with multiple right-hand sides. *Numerical Linear Algebra with Applications* 21(3): 453–471. DOI: 10.1002/nla.1892.

Adams M, Brezina M, Hu J, et al. (2003) Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics* 188(2): 593–610. DOI: 10.1016/S0021-9991(03)00194-3.

Alappat C (2019) *Recursive algebraic coloring engine library*. Available at: https://github.com/RRZE-HPC/RACE.

Alappat C, Basermann A, Bishop AR, et al. (2020a) A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication. *ACM Trans. Parallel Comput* 7(3): 1–37. DOI: 10.1145/3399732.

Alappat CL, Hofmann J, Hager G, et al. (2020b) Understanding HPC benchmark performance on intel broadwell and cascade lake processors. In: Sadayappan P, Chamberlain BL, Juckeland G, et al. (eds) *High Performance Computing*.

Cham: Springer International Publishing, 412–433. DOI: 10.1007/978-3-030-50743-5/_/21.

Alappat C, Hager G, Schenk O, et al. (2023) Level-based blocking for sparse matrices: sparse matrix-power-vector multiplication. *IEEE Transactions on Parallel & Distributed Systems* 34(2): 581–597. DOI: 10.1109/TPDS.2022.3223512.

AMD (2022) *AOCL-BLIS*. Available at: https://developer.amd.com/amd-aocl/blas-library/.

Anderson E and Saad Y (1989) Solving sparse triangular linear systems on parallel computers. *Int. J. High Speed Comput* 1(1): 73–95. DOI: 10.1142/S0129053389000056.

Bavier E, Hoemmen M, Rajamanickam S, et al. (2012) Amesos2 and Belos: direct and iterative solvers for large sparse linear systems. *Sci. Program* 20: 241–255. DOI: 10.3233/SPR-2012-0352.

Berger-Vergiat L, Glusa CA, Hu JJ, et al. (2019) *MueLu User's Guide*. Albuquerque, NM: Sandia National Laboratories. Technical Report SAND2019-0537.

Berger-Vergiat L, Kelley B, Rajamanickam S, et al. (2021) *Two-stage Gauss-Seidel preconditioners and smoothers for Krylov solvers on a GPU cluster*. Available at: https://doi.org/10.48550/arXiv.2104.01196.

Brandt A (1977) Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation* 31(138): 333–390. DOI: 10.2307/2006422.

Chow E, Anzt H, Scott J, et al. (2018) Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing* 119: 219–230. DOI: 10.1016/j.jpdc.2018.04.017.

Chronopoulos AT (1991) s-step iterative methods for (non)symmetric (in)definite linear systems. *SIAM Journal on Numerical Analysis* 28(6): 1776–1789. DOI: 10.1137/0728088.

Chronopoulos A and Gear C (1989) s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics* 25(2): 153–168. DOI: 10.1016/0377-0427(89)90045-9.

Chronopoulos AT and Kim SK (2020) *s-step Orthomin and GMRES implemented on parallel computers*. Available at: https://doi.org/10.48550/arXiv.2001.04886.

Dagum L and Menon R (1998) OpenMP: an industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng* 5(1): 46–55. DOI: 10.1109/99.660313.

Datta K (2009) *Auto-Tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD Thesis, USA.

Davis TA and Hu Y (2011) The university of Florida sparse matrix collection. *ACM Trans. Math. Softw* 38(1): 1:1–25. Available at: https://suitesparse-collection-website.herokuapp.com

Demmel J, Hoemmen M, Mohiyuddin M, et al. (2008) Avoiding communication in sparse matrix computations *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–12. DOI: 10.1109/IPDPS.2008.4536305.

Dongarra J, Tomov S, Luszczek P, et al. (2017) With extreme computing, the rules have changed. *Computing in Science Engineering* 19(3): 52–62. DOI: 10.1109/MCSE.2017.48.

Erhel J (1995) A parallel GMRES version for general sparse matrices. *Electronic Transactions on Numerical Analysis* 3: 160–176.

Evans D (1984) Parallel S.O.R. iterative methods. *Parallel Computing* 1(1): 3–18. DOI: 10.1016/S0167-8191(84)90380-6.

Falgout R (2006) An introduction to algebraic multigrid. *Computing in Science & Engineering* 8(6): 24–33. DOI: 10.1109/MCSE.2006.105.

Hoemmen M (2010) *Communication-Avoiding Krylov Subspace Methods*. PhD Thesis. USA, AAI3413388.

Intel (2022) Intel math kernel library. Available at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

Johnson OG, Micchelli CA and Paul G (1983) Polynomial preconditioners for conjugate gradient calculations. *SIAM Journal on Numerical Analysis* 20(2): 362–376. DOI: 10.1137/0720025.

Kreutzer M, Hager G, Wellein G, et al. (2014) A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36(5): C401–C423. DOI: 10.1137/130930352.

Lanzkron PJ, Rose DJ and Szyld DB (1990) Convergence of nested classical iterative methods for linear systems. *Numerische Mathematik* 58(1): 685–702. DOI: 10.1007/BF01385649.

Loe JA and Morgan RB (2022) Toward efficient polynomial preconditioning for GMRES. *Numerical Linear Algebra with Applications* 29(4): e2427. DOI: 10.1002/nla.2427.

Loe JA, Thornquist HK and Boman EG (2020) Polynomial preconditioned GMRES in trilinos: practical considerations for high-performance computing. *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. DOI: 10.1137/1.9781611976137.4.

Míka S and Vaněk P (1992) Acceleration of convergence of a two-level algebraic algorithm by aggregation in smoothing process. *Applications of Mathematics* 37(5): 343–356, Available at: https://eudml.org/doc/15720

Mohiyuddin M, Hoemmen M, Demmel J, et al. (2009) Minimizing communication in sparse matrix solvers *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*. New York, NY, USA: Association for Computing Machinery. DOI: 10.1145/1654059.1654096.

Muranushi T and Makino J (2015) Optimal temporal blocking for stencil computation. *Procedia Computer Science* 51: 1303–1312. DOI: 10.1016/j.procs.2015.05.315.

Olivier SL, Ellingwood ND, Berry J, et al. (2021) Performance portability of an SpMV kernel across scientific computing and data science applications *2021 IEEE High Performance*

Extreme Computing Conference (HPEC), 1–8. DOI: 10.1109/HPEC49654.2021.9622869.

Prokopenko A, Siefert CM, Hu JJ, et al. (2016) *Ifpack2 User's Guide 1.0*. Technical Report SAND2016-5338. Albuquerque, NM: Sandia National Labs.

Rajamanickam S, Acer S, Berger-Vergiat L, et al. (2021) Kokkos kernels: performance portable sparse/dense linear algebra and graph kernels. Available at: https://doi.org/10.48550/ARXIV.2103.11991.

Saad Y (1987) Least squares polynomials in the complex plane and their use for solving nonsymmetric linear systems. *SIAM Journal on Numerical Analysis* 24(1): 155–169. DOI: 10.1137/0724013.

Saad Y and Schultz MH (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 7(3): 856–869. DOI: 10.1137/0907058.

Sprague MA, Ananthan S, Vijayakumar G, et al. (2020) ExaWind: a multifidelity modeling and simulation environment for wind energy. *Journal of Physics: Conference Series* 1452(1): 012071. DOI: 10.1088/1742-6596/1452/1/012071.

Thomas SJ, Ananthan S, Yellapantula S, et al. (2019) A comparison of classical and aggregation-based algebraic multigrid preconditioners for high-fidelity simulation of wind turbine incompressible flows. *SIAM Journal on Scientific Computing* 41(5): S196–S219. DOI: 10.1137/18M1179018.

Trilinos Project Team T (2020) *The trilinos project website*. Available at: https://trilinos.github.io.

Van Zee FG and van de Geijn RA (2015) BLIS: a framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software* 41(3): 14:1–33. Available at: https://doi.acm.org/10.1145/2764454

Wathen AJ (2015) Preconditioning. *Acta Numerica* 24: 329–376. DOI: 10.1017/S0962492915000021.

Yamazaki I, Anzt H, Tomov S, et al. (2014a) Improving the performance of CA-GMRES on multicores with multiple GPUs *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 382–391. DOI: 10.1109/IPDPS.2014.48.

Yamazaki I, Rajamanickam S, Boman EG, et al. (2014b) Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 933–944. DOI: 10.1109/SC.2014.81.

Yamazaki I, Hoemmen M, Luszczek P, et al. (2017) Improving performance of GMRES by reducing communication and pipelining global collectives *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 1118–1127. DOI: 10.1109/IPDPSW.2017.65.

Ye X, Xi Y and Saad Y (2021) Proxy-GMRES: preconditioning via GMRES in polynomial space. *SIAM Journal on Matrix Analysis and Applications* 42(3): 1248–1267. DOI: 10.1137/20M1342562.

Author biographies

Christie Alappat received his master's degree with honors from the Bavarian Graduate School of Computational Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg. He is in the final stages of completing his doctoral studies under the guidance of Prof. Gerhard Wellein. His research interests include performance engineering, sparse matrix and graph algorithms, iterative linear solvers, and eigenvalue computation. He is the author of the RACE open-source software framework, which is used to accelerate challenging computations in sparse linear algebra on modern compute devices. He has won numerous awards including ACM Student Research Competition 2019 and SIAM Activity Group on Supercomputing (SIAG/SC) Best Paper Prize in 2024.

Jonas Thies received a bachelor's degree in computational engineering from FAU (2003), a master's degree in scientific computing from KTH Stockholm (2006), and a Ph.D. in applied mathematics from the University of Groningen (2010). He then spent 2 years as a postdoc at the Center for Interdisciplinary Mathematics in Uppsala. He worked at the German Aerospace Center in Cologne, where he led a research group on parallel numerics (2013-21). Since 2022, Dr. Theis has been an assistant professor in high-performance computing at the Delft Institute of Applied Mathematics, and the scientific advisor for users of the Delft University of Technology High Performance Computing Center.

Georg Hager holds a PhD and a Habilitation degree in Computational Physics from the University of Greifswald. He leads the Training & Support Division at Erlangen National High Performance Computing Center (NHR@FAU) and is an associate lecturer at the Institute of Physics at the University of Greifswald. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes on chip and system levels, and the analytic performance modeling of large-scale parallel codes. Georg Hager has authored and co-authored more than 100 peer-reviewed publications and was instrumental in developing and refining the Execution-Cache-Memory (ECM) performance model and energy consumption models for multicore processors. In 2018, he won the "ISC Gauss Award" (together with Johannes Hofmann and Dietmar Fey) for a paper on accurate performance and power modeling. He received the "2011 Informatics Europe Curriculum Best Practices Award" (together with Jan Treibig and Gerhard Wellein) for outstanding contributions to teaching in computer science. His textbook "Introduction to High Performance Computing for Scientists and Engineers" is recommended or required reading in many HPC-related lectures and courses worldwide. Together with colleagues

from FAU, HLRS Stuttgart, and TU Wien he develops and conducts successful international tutorials on node-level performance engineering and hybrid programming.

Holger Fehske received a Ph.D. in physics from the University of Leipzig, and a Habilitation degree and Venia Legendi in theoretical physics from the University of Bayreuth. In 2002, he became a full professor at the University of Greifswald. He currently holds the chair for complex quantum systems and works in the fields of solid-state theory, quantum statistical physics, light-matter interaction, quantum informatics, plasma physics, and computational physics. Dr. Fehske is a longstanding member of the steering committee of the High-Performance Computing Center Stuttgart and the Erlangen National High-Performance Computing Center.

Gerhard Wellein is a Professor for High Performance Computing at the Department for Computer Science of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and holds a PhD in theoretical physics from the University of Bayreuth. From 2015 to 2017 he was also a guest lecturer at the Faculty of Informatics at Università della Svizzera italiana (USI) Lugano. Since 2021 he is the director of the Erlangen National Center for High Performance Computing (NHR@FAU). He is a member of the board of directors of the German NHR-Alliance which coordinates the national HPC Tier-2 infrastructures at German universities. He has been serving for many years as the deputy speaker of the Bavarian HPC competence network KONWIHR. As a member of the scientific steering committees of the Leibniz Supercomputing Centre (LRZ) and the Gauss-Centre for Supercomputing (GCS) he is organizing and surveying the compute time application process for national HPC resources. Gerhard Wellein has more than 20 years of experience in teaching HPC techniques to students and scientists. He has contributed to numerous tutorials on node-level performance engineering in the past decade and received the "2011 Informatics Europe Curriculum Best Practices Award" (together with Jan Treibig and Georg Hager) for outstanding teaching contributions. His research interests focus on performance modeling and performance engineering, architecture-specific code optimization, novel parallelization approaches and hardware-efficient building blocks for sparse linear algebra and stencil solvers. He has been conducting and leading numerous national and international HPC research projects and has authored or co-authored more than 100 peer-reviewed publications.

Appendix

Performance of SpMV routine in Trilinos

Trilinos uses the Kokkos Kernels package for SpMV, which has been shown to achieve good performance on a wide range of architectures, see Olivier et al. (2021) and Rajamanickam et al. (2021). However, initial tests with some of our matrices showed inferior SpMV performance. For example, the performance on the `Transport` matrix was well below the Roofline prediction of 24 and 21 Gflop/s (see Alappat et al., 2020a for derivation of the performance model) on ICL and ROME, respectively (see Figure 11(a)). A closer investigation revealed that Trilinos (tested until version 13.4.1) by default calls a dynamically scheduled version of SpMV for matrices with $N_{nz} > 10^7$. Especially on ROME, the overhead associated with dynamic scheduling was too high, leading to inferior performance on our benchmark matrices. Therefore we manually modified the routine to always call a statically scheduled version of SpMV from the Kokkos Kernels. This led to a huge performance improvement and we ended up close to the memory-bound roofline model prediction (see Figure 11(a)). Note that on ROME the performance slightly exceeds the limit; this is because of a residual caching effect from ROME's large L3 cache, cf. Alappat et al. (2023).

Performance of Ortho routines in Trilinos

The Ortho routines in the s-step GMRES solver use tall-skinny `DGEMM` and `TRSM` computations, for which Trilinos employs BLAS libraries. Figure 11(b) reports the performance of the Ortho routine with the Intel MKL and AMD AOCL-BLIS BLAS libraries, respectively. On ICL, MKL achieves near-optimal performance of 190 orthogonalization steps per second (equivalent to 170 Gflop/s) as predicted by the roofline model. One would expect ROME to match this level due to its practically identical bandwidth and floating-point performance. However, the MKL version

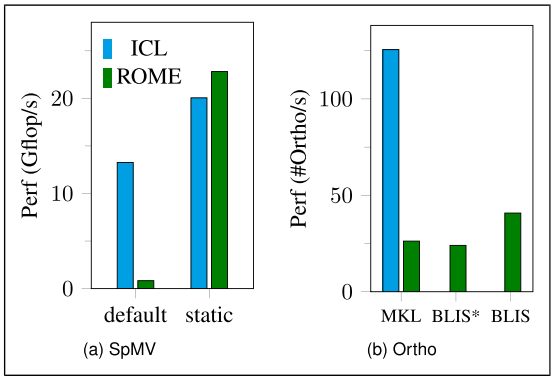

Figure 11. (a) Performance of the SpMV kernel in Gflop/s with the `Transport` matrix with default and static scheduling. Bars with blue and green colors show the result on ICL and ROME, respectively. (b) Performance of orthogonalization routine (in number of routines executed per second) with MKL and BLIS libraries. BLIS* represents the BLIS routine called with default setting, while the other one uses an optimized thread parallelization setting.

has 4.8× lower performance on ROME compared to ICL despite the use of the `LD_PRELOAD` trick mentioned in Section 2. Performance did not improve with the AOCL-BLIS library[9] with default configuration (BLIS* in Figure 11(b)). However, changing the OpenMP loop used for parallelism via an environment variable (`BLIS_JC_NT` = 64) yielded a 1.5 × speedup compared to MKL. Thus, we used the AOCL-BLIS library for our runs on ROME with the s-step GMRES solver. The attained performance of Ortho is still far from optimal but it is challenging to do further optimizations from the user level as the solver requires BLAS computations with various matrix shapes, and tuning environment variables globally will not fix the issue in all cases. We expect that the performance of the AOCL-BLIS library will improve in the future for tall-skinny matrices, leading to a performance boost for both the baseline and RACE-accelerated variants on AMD multi-core processors.