Level-Based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication

Christie Alappat, Georg Hager, Olaf Schenk, *Senior Member, IEEE*, and Gerhard Wellein

Abstract—The multiplication of a sparse matrix with a dense vector (SpMV) is a key component in many numerical schemes and its performance is known to be severely limited by main memory access. Several numerical schemes require the multiplication of a sparse matrix polynomial with a dense vector which is typically implemented as a sequence of SpMVs. This results in low performance and ignores the potential to increase the arithmetic intensity by reusing the matrix data from cache. In this work we use the recursive algebraic coloring engine (RACE) to enable blocking of sparse matrix data across the polynomial computations. In the graph representing the sparse matrix we form levels using a breadth-first search. Locality relations of these levels are then used to improve spatial and temporal locality when accessing the matrix data and to implement an efficient multithreaded parallelization. Our approach is independent of the matrix structure and avoids shortcomings of existing "blocking" strategies in terms of hardware efficiency and parallelization overhead. We quantify the quality of our implementation using performance modelling and demonstrate speedups of up to 3× and 5× compared to an optimal SpMV-based baseline on a single multicore chip of recent Intel and AMD architectures. Various numerical schemes like s-step Krylov solvers, polynomial preconditioners and power clustering algorithms will benefit from our development.

Index Terms—Algorithm design and analysis, computer architecture, graph algorithms, kernel optimization, memory hierarchies, performance evaluation, sparse matrices

1 Introduction and Related Work

Sparse matrix-vector multiplication (SpMV) is a critical building block for a wide variety of computational algorithms used in science, engineering, and data analytics. The SpMV kernel is known to perform poorly on modern compute devices due to its low arithmetic intensity and often irregular memory access pattern. Most performance optimization efforts target a single SpMV invocation. To minimize the data access costs to the matrix entries, a plethora of data layout choices have been proposed for GPGPUs [1] and CPUs [2], [3], [4], including hardware-agnostic formats [5]. These formats typically ensure linear access to matrix data, but the input vector is always accessed indirectly and therefore potentially in an irregular way. Optimization strategies like matrix reordering or partitioning techniques [6] aim to reduce the reuse distances in the vector accesses and thus improve the performance. Finally, at the kernel implementation level, automatic performance optimization for SpMV has been a subject of research for decades. These approaches mainly account for the complexity of cache-based microprocessors, where SpMV performance maybe extremely sensitive to the spatial/temporal data access locality, out-of-order instruction capability, register scheduling, and SIMD vectorization. Choosing parameters for these code optimizations and choosing among alternative implementations is critical for efficient hardware utilization. It has been demonstrated [3], [7], [8], [9] that it is possible to build an automatic tuning system capable of generating implementations that are on par with or even outperform the best manually tuned code.

In this work, we extend SpMV performance tuning research towards automatic data reuse optimization across several SpMV invocations in the *sparse matrix power kernel* (MPK), which computes $Ax, A^2x, A^3x, \cdots, A^kx$ for matrix A, vector x, and a small constant k. Our focus is on thread-level parallel and efficient CPU implementation of MPK using the popular compressed row storage (CRS) sparse matrix format. To this end we extend the recursive algebraic coloring engine (RACE) framework [10] to tackle the dependencies between several SpMV invocations in the MPK. The algebraic formulation used in RACE is general in the sense that it does not assume any special structure in the underlying matrix.

The need for software implementations and structures for MPK is exemplified by communication-avoiding algorithms [11], [12], [13], [14], which have been proposed to improve performance by reducing the memory and network traffic. In these algorithms, independent SpMV invocations are replaced by the MPK to compute A^kx. Once the computation has been performed, the next k steps of the solver can proceed without further memory accesses to A by combining vectors from this set.

- *Christie Alappat and Georg Hager are with the Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany. E-mail: {christie.alappat, georg.hager}@fau.de.*
- *Olaf Schenk is with the Institute of Computing at Faculty of Informatics, Università della Svizzera italiana, 6962 Lugano, Switzerland. E-mail: olaf.schenk@usi.ch.*
- *Gerhard Wellein is with the Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany, and also with the Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany. E-mail: gerhard.wellein@fau.de.*

Manuscript received 3 May 2022; revised 12 September 2022; accepted 8 November 2022. Date of publication 21 November 2022; date of current version 19 December 2022.
This work was supported by in part by NHR@FAU, in part by the State of Bavaria, and in part by the German Federal Ministry of Education and Research. (Corresponding author: Christie Alappat.)
Recommended for acceptance by P. D'Ambra.
Digital Object Identifier no. 10.1109/TPDS.2022.3223512

There has been some research in exploiting data locality in MPK, mostly motivated by classic blocking strategies well established in stencil computations. In particular, in [15] blocking schemes for MPK have been developed that first partition the graph of a matrix A into p blocks of almost equal size, where p is the number of cores. For cache reuse, the blocks assigned to each core are further partitioned. Within each block, an orthotrope-style [16] temporal blocking is used to perform MPK computation locally for the block. This requires to find neighbors of each block that are involved in an MPK computation with power k. However, these neighbors end up in nonconsecutive spots. Therefore, those schemes require either redundant computations (explicit schemes) and/or irregular accesses to the matrix entries with bookkeeping (implicit schemes), resulting in performance bottlenecks. In [17] a runtime auto-tuning was introduced for the MPK scheme described above to choose the appropriate parameters (e.g., explicit versus implicit schemes) for a given matrix. This was generalized to various kernels like Jacobi and serial Gauss-Seidel iterative solvers and automated using a sparse tiling algorithm via the power of loop chain abstraction [18], [19]. In [20], MPK kernels were studied on modern multicore architectures for banded sparse matrices that arise from stencil discretization. Following classic stencil blocking approaches, a geometrical blocking method was proposed. For matrices arising from two-dimensional discretization the method achieved decent speedup. However, for matrices from three-dimensional discretization it yielded very limited performance gains due to high matrix bandwidth. Most of the other works [11], [21], [22], [23] on MPK schemes focused on reducing the MPI communication overhead. A recent work [24] in this direction presents a theoretical study on the benefit of diamond tiling for reducing communication.

Contribution and Outline

Our work bridges the gap between temporal blocking of stencil algorithms [25], [26], [27], which can be considered as an MPK on structured grids, and recursive spatial blocking strategies for SpMV [28]. In addition we reduce the need to manually set up the blocks. We cover full thread-level parallelization and focus on a single multicore processor. Our contributions are as follows:

- We generalize temporal tiling strategies known from stencil computations on structured grids to MPK computations on structured and unstructured sparse matrices using the levels of the graph of the matrix.
- We present an efficient, multi-threaded implementation of our level-based blocking method for sparse MPK on modern multicore processors. Our solution aims to reduce the main memory traffic and to avoid scalability bottlenecks such as synchronization overhead or load imbalance.
- We conduct a detailed performance analysis of our approach as implemented in RACE on various CPU architectures.
- For a broad set of sparse matrices we demonstrate full threading functionality and excellent multicore performance achieving speedups of 3× to 5× compared to a standard baseline implementation.
- We validate the performance improvements using the roofline model and the phenomenological Execution-Cache-Memory (ECM) model. These models corroborate the optimality of both our level-blocking approach and the baseline implementation to which we compare.

TABLE 1
Key Specification of Test Bed Machines

Architecture	CLX	ICL	ROME
Chip Model	Xeon Gold 6248	Xeon Platinum 8368	AMD EPYC 7662
Microarchitecture	Cascade Lake	Sunny Cove	Zen-2
Release year	2019	2021	2020
Cores per socket	20	38	64
Max. SIMD width	512 bits	512 bits	256 bits
L1D cache capacity	20×32 KiB	38×48 KiB	64×32 KiB
L2 cache capacity	20×1 MiB	38×1.25 MiB	64×512 KiB
L3 cache capacity	27.5 MiB	57 MiB	16×16 MiB
Memory Configuration	6 ch. DDR4-2933	8 ch. DDR4-3200	8 ch. DDR4-3200
Mem. Bandwidth (b_{Mem})	116 GB/s	170 GB/s	146 GB/s
Operating system	Ubuntu 20.04.4	RHEL 8.4	Ubuntu 20.04.4
Compiler	Intel 19.1 update 2	Intel 2021.5 update 0	Intel 19.0 update 5

The remainder of the paper is structured as follows. Section 2 reviews our experimental setup, in particular hardware and software characteristics of the next generation of scalable processor, namely the Intel Cascade Lake and Intel Ice Lake, and the AMD EPYC architectures, and, additionally, the set of benchmark matrices. In Section 3 we review the computational workload of matrix-vector multiplications for sparse matrices. Section 4 is dedicated to the main contribution of the paper and describes in detail the algorithmic components of level-based blocking of MPK. Section 5 includes an assessment of performance parameters within our recursive level-based blocking engine (RACE MPK) method. In Section 6 we conduct a detailed performance analysis of our cache-aware implementation for matrix-power kernels and compare it to a state-of-the-art implementation. Finally, in Section 7 we conclude the paper and give an outlook on future directions.

2 Hardware and Software Environment

2.1 Hardware

The measurements in this paper were conducted on a single socket of Intel Cascade Lake (CLX), Intel Ice Lake (ICL), and AMD Epyc Zen2 (ROME), respectively. Key specifications of the three systems are summarized in Table 1.

These state-of-the-art processors power more than 50% of the top 100 ranking supercomputers [29]. The Intel CPUs support the AVX-512 instruction set, while the AMD CPU supports only AVX-2. Turbo mode was active for all the runs, and the systems were configured with one ccNUMA domain per socket, i.e., on Intel systems the Sub-NUMA Clustering (SNC) was disabled and on AMD the NPS1 mode was used.

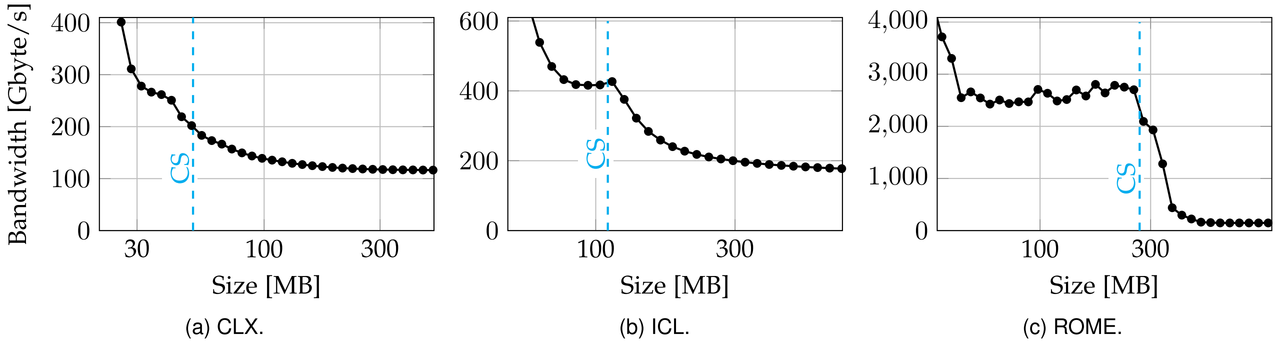


Fig. 1. Single socket L3 and memory bandwidth (load-only) of the three architectures under consideration. The dashed line represents the total available cache size (CS). Note the different scaling on the y-axis.

All CPUs have three levels of cache: private, inclusive L1 and L2, and a victim-type L3. The L3 cache on the Intel systems is shared by all cores of a socket, while on ROME it is shared only within a core complex (CCX), which comprises four cores. The aggregate L3 cache on ROME is 2.5× larger than on ICL and 5× larger than on CLX. This can be observed in the full-socket load-only bandwidth measurements in Fig. 1, where the combined L2 and L3 cache sizes are marked with dashed lines. This data also shows the L3 and main memory bandwidths of the three CPUs. CLX and ICL have a moderate L3 bandwidth of 300 Gbyte/s and 400 Gbyte/s, respectively, while ROME has a very high L3 bandwidth of more than 2500 Gbyte/s. It is worth noting that the transition from L3 to main memory is very sharp on ROME and occurs exactly where the data-set size exceeds the total cache size, while on the Intel systems the drop is gradual and there is a noticeable cache effect even when the working set exceeds the cache size by 2× or more, due to its dynamic cache replacement policy [30]. The main memory bandwidth (b_{Mem}) of CLX, ICL and ROME is about 116 Gbyte/s, 170 Gbyte/s, and 146 Gbyte/s, respectively.

2.2 Software

For compilation, Intel compilers (see Table 1 for version info) were used on Ubuntu 20.04.4 (CLX and ROME) and Red Hat Enterprise Linux 8.4 (ICL), respectively, with compiler flags `-O3 -xHOST`. All floating-point computations were done in double precision, while integers were 32 bits wide. The kernels were SIMD vectorized using pragmas to exploit the maximum SIMD width of the hardware, i.e., 256 bits on ROME and 512 bits on ICL and CLX. Threads were bound to cores in a closed (fill-type pinning) manner. To reduce fluctuations, each kernel was executed multiple times such that the overall runtime is greater than one second. The average performance of these runs was then reported. As the variation among multiple measurements was less than 5%, we do not show error bars.

For pinning, bandwidth benchmarks (see Fig. 1), and for counting hardware events we use the `likwid-pin`, `likwid-bench`, and `likwid-perfctr` tools from the `LIKWID` tool suite version 5.1.

2.3 Benchmark Matrices

Table 2 shows the sparse matrices used for the benchmarks and some of their properties: N_{r} is the total number of rows, N_{nz} is the total number of nonzero entries, and N_{nzr} is the average number of nonzero entries per row (i.e., $N_{\mathrm{nz}}/N_{\mathrm{r}}$). The matrices are ordered (top to bottom) according to increasing N_{nz}, and all are square since this is a requirement for the matrix power kernel (MPK). Most of the matrices were taken from the SuiteSparse Matrix Collection [31]. `HPCG-128-128-128` is the matrix found in the HPCG benchmark [32], with a problem size of 128^3. Further, large matrices from current research in the fields of quantum physics and cardiac electrophysiology have been included. The matrices from quantum physics were generated using the Scalable Matrix Collection (ScaMaC) library [33], while the ones from electrophysiology were taken from [34].

3 Matrix Power Kernel

The basic algorithmic workload addressed in this article is the computation of powers of a sparse matrix applied to a dense vector. The matrix power kernel (MPK) is defined as follows: For a given square, sparse matrix A and a dense vector x calculate all matrix powers $A^p x$ up to a maximum p_m $(p = 1, \ldots, p_m)$ and store all p_m resulting vectors $(y_p = A^p x)$ for subsequent calculations. We further define $y_0 := x$.

3.1 Baseline MPK Implementation

The standard approach to implement the MPK is to perform a sequence of p_m SpMV operations, i.e., $y_i = Ay_{i-1}$ with $i = 1, \ldots, p_m$, using standard SpMV implementations or library calls. We refer to this strategy as *baseline* MPK.

Fig. 2 shows a high-level representation of our baseline MPK together with an SpMV implementation that is known to provide good performance on CPUs for a wide variety of sparse matrix structures. The sparse matrix A is stored in the well-known CRS format, using the three arrays $rowPtr$, val, and col, which hold the row pointer information, values, and column indices of nonzero entries, respectively (see [35] for details). This information is passed (as global data) to the SpMV function along with the function parameters (line 9) representing the right-hand side (RHS) vector and the range of row indices for which the SpMV is to be computed.[1] The function then performs the SpMV operation (lines 12–20) and returns the resulting left-hand side (LHS) vector. Note that most SpMV implementations in libraries are unsuitable for the optimized MPK discussed later as they do not support SpMV on a subset of rows. Therefore we use our own

1. For the baseline implementation, the entire row range is specified.

TABLE 2
Details of the Benchmark Matrices

Index	Matrix name	N_{r}	N_{nz}	N_{nzr}
1	cfd2	123440	3087898	25.02
2	parabolic_fem	525825	3674625	6.99
3	xenon2	157464	3866688	24.56
4	cant	62451	4007383	64.17
5	offshore	259789	4242673	16.33
6	Hamrle3	1447360	5514242	3.81
7	bmw7st_1	141347	7339667	51.93
8	G3_circuit	1585478	7660826	4.83
9	shipsec1	140874	7813404	55.46
10	ship_003	121728	8086034	66.43
11	thermal2	1228045	8580313	6.99
12	gearbox	153746	9080404	59.06
13	crankseg_1	52804	10614210	201.01
14	pwtk	217918	11634424	53.39
15	rajat31	4690002	20316253	4.33
16	gsm_106857	589446	21758924	36.91
17	F1	343791	26837113	78.06
18	cage14	1505785	27130349	18.02
19	Fault_639	638802	28614564	44.79
20	inline_1	503712	36816342	73.09
21	RM07R	381689	37464962	98.16
22	Emilia_923	923136	41005206	44.42
23	ldoor	952203	46522475	48.86
24	af_shell10	1508065	52672325	34.93
25	HPCG-128-128-128	2097152	55742968	26.58
26	Hook_1498	1498023	60917445	40.67
27	Geo_1438	1437960	63156690	43.92
28	Serena	1391349	64531701	46.38
29	bone010	986703	71666325	72.63
30	audikw_1	943695	77651847	82.28
31	channel-500x100x100-b050	4802000	85362744	17.78
32	dielFilterV3real	1102824	89306020	80.98
33	nlpkkt120	3542400	96845792	27.34
34	ML_Geer	1504002	110879972	73.72
35	Flan_1565	1564794	117406044	75.03
36	stokes	11449533	349321980	30.51
37	nlpkkt240	27993600	774472352	27.67
38	Topi-real-256 (Q)	67108864	802160640	11.95
39	Graphene-8192 (Q)	67108864	872235016	13.00
40	Lynx649 (E)	64950632	978866282	15.07
41	Anderson-600 (Q)	216000000	1293840000	5.99
42	Lynx1151 (E)	115187228	1934489424	16.79

N_{r} is the number of rows, N_{nz} is the number of nonzeros, and N_{nzr} is the average number of nonzeros per row. The letters "Q" and "E" in parentheses mark the matrices from quantum physics and cardiac electrophysiology, respectively.

```
1:  $double :: val[N_{\mathrm{nz}}]$ //store values of nonzeros in $A$
2:  $int :: col[N_{\mathrm{nz}}], rowPtr[N_{\mathrm{r}}+1]$ //column index and row pointer of $A$
3:  $double :: y[N_{\mathrm{r}}, 0{:}p_m]$ //to store input and output vectors
4:  //Perform $p_m$ SpMVs
5:  for $p = 1 : p_m$ do
6:      $y[:, p]$=SpMV($y[:, p-1]$, 0 , $N_{\mathrm{r}}$-1)
7:  end for
8:  //Perform SpMV between $A$ and $in$ vector and store result in $out$.
    // $row\_s$ and $row\_e$ arguments are used to specify the start and end row
    //to which SpMV is applied.
9:  function SpMV($double :: in\_rhs[N_{\mathrm{r}}]$, $int :: row\_s$, $int :: row\_e$)
10:     $double :: out\_lhs[N_{\mathrm{r}}]$
11:     //Loop over rows
12:     #pragma omp parallel for schedule(static)
13:     for $row = row\_s : row\_e$ do
14:         $double :: tmp = 0$
15:         //Loop over nonzeros in row
16:         for $idx = rowPtr[row] : (rowPtr[row+1]-1)$ do
17:             $tmp$ += $val[idx] * in\_rhs[col[idx]]$
18:         end for
19:         $out\_lhs[row] = tmp$
20:     end for
21:     return $out\_lhs$
22: end function
```

Fig. 2. CRS-based MPK computing $A^{p_m}x$. The arrays val, col, and $rowPtr$ hold the CRS data structure of A. The input and output vectors are stored in the y matrix.

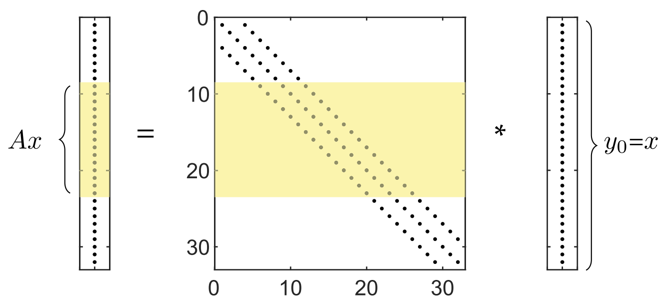
version of SpMV, which serves as the main kernel for both the baseline and the optimized version. We have ensured that our SpMV performs at least as good as Intel MKL with the standard CRS format.

The baseline MPK stores the $p_m + 1$ vectors $\{y_p\}$ in the matrix $y[:, 0 : p_m]$ (column-major order) and performs p_m back-to-back calls to the SpMV function (see lines 5–7 of Fig. 2). If the caches are too small to hold the entire matrix, it must be read p_m times from main memory. Consequently, the optimum (minimum) main memory balance for the CRS-based baseline MPK is $B_C = 6\,\mathrm{byte/flop}$ [10], [36], which is equivalent to $12\,\mathrm{bytes}$ of memory traffic per nonzero matrix entry. The baseline MPK thus reflects the strongly memory-bound performance characteristic of the underlying SpMV operation.

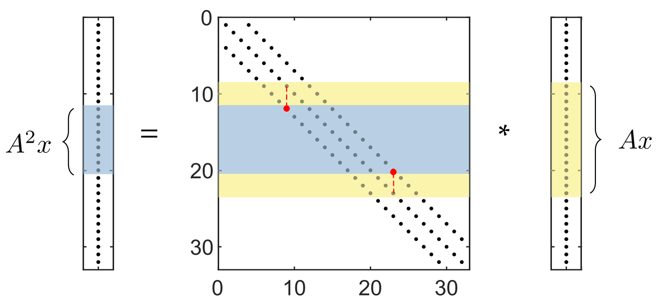
In order to evaluate the quality of optimized MPK implementations, we will measure the actual code balance $B_{C,m}$ and compare it with the theoretical baseline minimum ($6\,\mathrm{byte/flop}$) discussed above. The $B_{C,m}$ is obtained by measuring the actual data traffic (using `likwid-perfctr`) and dividing it by the minimum amount of floating-point operations to be performed, i.e., $2 \times N_{nz} \times p_{max}$. Where appropriate, measured code balance from within the cache hierarchy will also be reported.

3.2 Blocking Strategy for the MPK Implementation

As the same sparse matrix is repeatedly applied, there is substantial performance optimization potential via data transfer reduction by reusing matrix entries from the cache for the successive computation of multiple powers. The basic idea is to compute the SpMV partially for a block of A that fits into cache and reuse these matrix entries for the next SpMV, i.e., calculate another power on a smaller subset of the data. This approach is equivalent to temporal blocking for iterative stencil update schemes, where multiple updates on the same stencil data are computed in cache. Here the spatial stencil structure determines the dependencies between successive updates and geometric schemes for handling the spatial-temporal dependencies such as trapezoidal [37] or diamond blocking [38] are well established. To demonstrate the equivalent challenge in MPK, we show in Fig. 3a simple banded sparse matrix, which arises from a discretization of a toy stencil in one spatial dimension. In the first step (Fig. 3a), an SpMV operation is performed applying a block of the matrix (yellow rows), which fits into cache, to the input (RHS) vector x to calculate a part of Ax (yellow elements of LHS vector). In the next step (Fig. 3b), the updated vector elements serve as input and are used to calculate A^2x (blue elements of the LHS vector) by applying SpMV with a subset of the matrix block (blue rows). To fulfill the dependencies between these successive SpMV steps, the column indices of the subset of the matrix block (blue rows in Fig. 3b) must be in the range (indicated with red



(a) Ax update



(b) A^2x update

Fig. 3. Blocking successive matrix applications for a simple banded sparse matrix: (a) The RHS vector is the input vector x. Yellow elements of the LHS vector are updated to Ax. (b) The next update is performed on the blue block of A to compute A^2x on the blue elements of the LHS vector. Yellow matrix elements can be reused when computing A^2x on blue blocks.

line) of the row indices of the original matrix block (yellow rows). It is obvious that the overhead of this approach, which is quantified by the ratio of yellow to blue rows in Fig. 3b, increases with the bandwidth of the matrix (i.e., with longer-range stencils).

The outlined MPK blocking approach can be generalized for sparse matrices with irregular structures. We define $\mathcal{I}$ to be a set of row indices of the matrix A. The corresponding set $\mathcal{C}(\mathcal{I})$ contains the column indices of all nonzero entries in the rows of $\mathcal{I}$, i.e., if $i \in \mathcal{I}$ then $j \in \mathcal{C}(i) \Leftrightarrow A_{i,j} \neq 0$. Based on this notation, the SpMV operation ($y = Ax$) for a given row index $i \in \mathcal{I}$ can be written as:

$$y_i = \sum_{j \in \mathcal{C}(i)} A_{i,j} x_j \tag{1}$$

If we apply the SpMV for all rows in $\mathcal{I}$ to a RHS y_{p-1}, then all corresponding row entries of the LHS vector are updated to power p. We can then apply to this vector another SpMV on a set of rows $\mathcal{K}$ for which $\mathcal{C}(\mathcal{K}) \subseteq \mathcal{I}$.

The choice of the set of row indices $\mathcal{I}$ for a given sparse matrix A is decisive to the performance of such a method: (i) The matrix elements associated with $\mathcal{I}$ and $\mathcal{C}(\mathcal{I})$ have to fit into cache and (ii) should be stored to enable high spatial and temporal locality without indirect access. Furthermore, (iii) the bandwidth of the matrix involved in the MPK should be as small as possible, i.e., the indices of $\mathcal{C}(\mathcal{I})$ have to be close to the set $\mathcal{I}$. A potential approach to address these challenges is to consider the SpMV operation as a graph traversal problem as done in the RACE coloring scheme [10]. Here, breadth-first search (BFS) [39] is applied to the graph underlying A; the BFS levels of A are stored consecutively. These levels allow us to identify appropriate parts of the matrix ($\mathcal{I}$ and $\mathcal{K}$) for blocking and how to traverse the full matrix (graph) systematically to update vector elements corresponding to all matrix powers while maintaining locality in accessing matrix and vector data. As an added benefit, the BFS reordering of the matrix reduces its bandwidth.

4 Level-Blocked MPK

The RACE coloring scheme has been developed to generate hardware-efficient distance-k colorings of graphs [10] using their BFS levels. It has been successfully applied to the shared-memory parallelization of symmetric SpMV providing unprecedented performance levels. Further it has been shown that the level-based approach allows to control dependencies in a parallel, symmetric SpMV operation and at the same time provides flexibility to ensure data locality and to adjust to the degree of parallelism required by modern multicore processors.

In the context of the MPK, the BFS levels are used to split the sparse matrix into smaller blocks which may fit into cache. Furthermore, the locality features of the levels are used to reduce the cache reuse distance for matrix entries and track the dependencies between levels of successive SpMVs. As the relevant features of the BFS levels are important for the MPK blocking, we first recapitulate the basic terminology and then the level-based approach of RACE in Section 4.1. From Section 4.2 onward we demonstrate how it is basically applied to the MPK problem and then show how data locality and efficient shared-memory parallelization can be achieved.

In this section, we restrict ourselves to symmetric matrices, i.e., undirected graphs. However, the proposed MPK blocking method is also applicable to non-symmetric square matrices. The following definitions from graph theory are used throughout the paper:

Graph: $G = (\mathcal{V}, \mathcal{E})$ represents a graph, with $\mathcal{V}(G)$ denoting a set of vertices and $\mathcal{E}(G)$ denoting its edges. For sparse matrices, $\mathcal{V}(G)$ consists of all row indices of the matrix and $\mathcal{E}(G)$ consists of edges between two vertices corresponding to the row (u) and the column indices (v) of the nonzero entries, i.e., $\{u, v\} \in \mathcal{E}(G) \Leftrightarrow A_{u,v} \neq 0$.

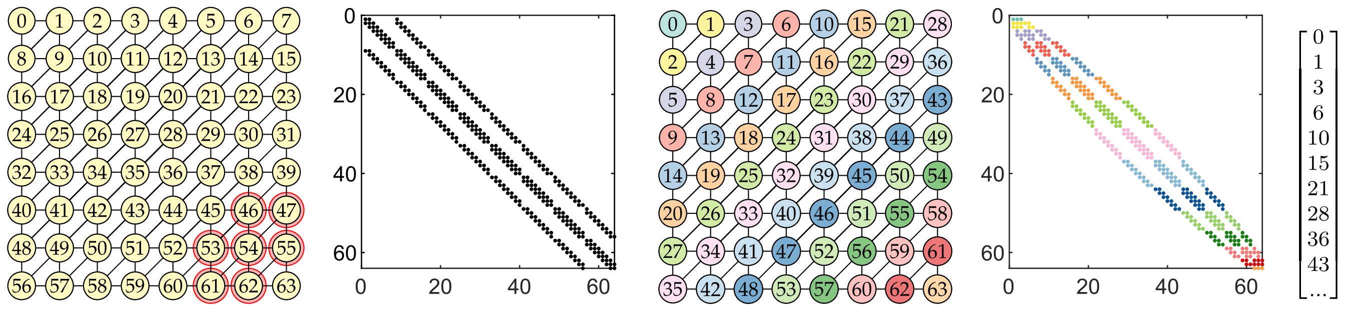
Neighborhood. The neighborhood of a vertex u is the set of vertices $\mathcal{N}(u) = \{v \in \mathcal{V}(G) : \{u, v\} \in \mathcal{E}(G)\}$.

Subgraph. A subgraph H of G specifically refers to the subgraph induced by vertices $\mathcal{V}' \subseteq \mathcal{V}(G)$ and is defined as the graph $H = (\mathcal{V}', \{\{u, v\} \in \mathcal{E}(G) \wedge u, v \in \mathcal{V}'\})$.

In the graph terminology, an SpMV operation ($y = Ax$) can be formulated as follows: If $G = (\mathcal{V}, \mathcal{E})$ is the graph representation of the sparse matrix A then for every vertex $u \in \mathcal{V}(G)$ calculate

$$y_u = \sum_{v \in \mathcal{N}(u)} A_{u,v} x_v \; . \tag{2}$$

Comparing (2) with (1), we can observe the equivalence between index-based (row index i and its related column indices $\mathcal{C}(i)$) and graph-based (vertex u and its neighborhood $\mathcal{N}(u)$) notations.



(a) Graph. (b) Sparsity pattern of matrix. (c) Permuted graph. (d) Permuted matrix. (e)

Fig. 4. Graph (a) and sparsity pattern (b) of the matrix associated with a 2d-7pt stencil on an 8×8 grid. In (a), the associated stencil is highlighted in red for an arbitrary vertex (54). (c) shows the permuted graph and (d) the sparsity pattern of the matrix after applying BFS reordering. The vertices (rows) of the graph (matrix) that belong to a level are represented with the same color. The `level_ptr` associated with the permuted graph/matrix is shown in (e).

To illustrate our method, a simple graph generated by applying a two-dimensional seven-point (2d-7pt) stencil to a square grid of size 8×8 will serve as an example. Fig. 4a shows the graph with each vertex numbered in lexicographic ordering. The associated stencil at a single grid point (vertex 54 and its neighborhood) is highlighted. The sparsity pattern of the corresponding matrix is shown in Fig. 4b.

4.1 Levels

The level formation in RACE is based on a BFS which assigns each vertex (row) of the graph (matrix) to a level. First, a *root vertex* v_{root} is chosen and assigned to the first level, $L(0)$.[2] The rest of the levels, $L(i)\ \forall\ i > 0$, are defined to contain vertices that are in the combined neighborhood of the vertices in the previous level $L(i-1)$ but have no level numbers assigned yet, i.e.,

$$L(i) = \begin{cases} v_{\text{root}} & \text{if } i = 0, \\ \{u : u \in \mathcal{N}(L(i-1)) \wedge \\ u : \{u \notin \{L(0), \ldots, L(i-1)\}\} & \text{else.} \end{cases} \tag{3}$$

Fig. 4c shows the 15 levels (indicated by different colors) generated by this procedure for the stencil graph if $v_{\text{root}} = 0$ is chosen. After level formation, the vertices are renumbered (compare vertex indices in Figs. 4a and 4c) such that those in the same level are numbered consecutively and the vertices in level $L(i-1)$ appear before those in $L(i)$. This permutation[3] increases data locality between neighboring vertices and results in a lens-shaped matrix with typically reduced bandwidth (see Fig. 4d). Since this improves the data locality of sparse matrix computations, such permutations are widely employed as preprocessing steps for SpMV-based algorithms [40].

As a consequence of the definition of levels, the neighborhood of all vertices in a given level $L(i)$ is clearly confined to the vertices within the previous, current, and next levels, i.e.:

$$\mathcal{N}(L(i)) \in \{L(i-1) \cup L(i) \cup L(i+1)\}, \text{ for } i > 0\ . \tag{4}$$

2. In this article, we always choose the first vertex (default setting in RACE) as the root node.

3. Note that a symmetric permutation is employed on the matrix, i.e., both rows and columns are permuted.

This property is crucial for the design of our level-based MPK blocking scheme as it defines the dependency between the computation of SpMVs for different levels at different matrix powers: To advance all vertices of $L(i)$ to $A^p x$, the calculation of $A^{p-1}x$ has to be completed on the levels $L(i-1)$, $L(i)$, and $L(i+1)$.

4.2 Level-Based Blocking of MPK

In Section 3, we discussed the baseline MPK and the potential matrix data reuse by blocking across the SpMV operations involved in the MPK. Further it has been shown that the graph formulation of the SpMV(2) together with the neighborhood relation (4) of the levels (3) provide a natural framework for the structured computation of the MPK. This includes the dependency between a level and its neighborhood; e.g., in Fig. 4c one can calculate the next matrix power for level $L(6)$ (with vertices $21, \ldots, 27$) only after the computation of the previous matrix power is complete on levels $L(5)$, $L(6)$, and $L(7)$ (containing vertices $15, \ldots, 35$).

We next introduce the *Lp diagram* to visualize the dependencies between levels in MPK calculations. In the Lp diagram, the indices of the levels $L(i)$ are on the x-axis and the matrix power stages ($1 \leq p \leq p_{max}$) are on the y-axis. Hence, each node (i, p) in the diagram represents an SpMV on the vertices in level i to compute part of the power p. Fig. 5 shows the Lp diagram for 15 levels and $p_{max} = 5$. To satisfy the dependencies in the level-based MPK blocking scheme, the nodes $(i-1, p-1)$, $(i, p-1)$, and $(i+1, p-1)$ need to be computed before SpMV can be applied to compute the node (i, p). The red arrows in Fig. 5 denote the dependency for the computation of $L(6)$ at $p = 4$, i.e., for the node (6, 4). The order of traversal in the Lp diagram is as follows:

- Each *diagonal*, defined by $i + p = \text{const}$, is traversed from bottom to top (starting at $p = 1$).
- Diagonals are traversed from left to right, i.e., starting with $p = 1$ for $L(0)$.

This execution order, which is independent of the actual graph structure, ensures that the levels $L(i-1)$, $L(i)$, and $L(i+1)$ are updated to power stage $p-1$ before level $L(i)$ is advanced to power stage p. In Fig. 5, the order of all execution steps of this scheme is shown via the node numbers in the Lp diagram with $p_m = 5$.

Visualizations similar to Fig. 5 are often shown for one-dimensional (1D) radius-one stencils, where the x-axis

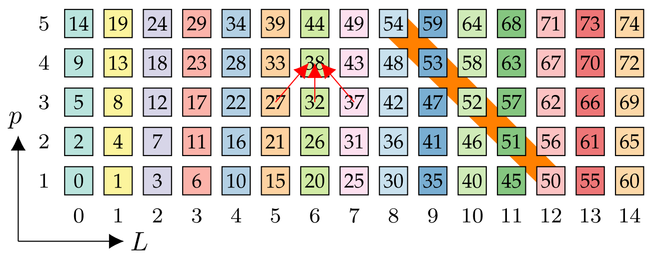


Fig. 5. Lp diagram with 15 levels ($L(0),\ldots,L(14)$) and a maximum power stage of $p_{max} = 5$. Level colors are the same as in Fig. 4c. Each node in the Lp diagram is numbered according to the execution order. For $p = 4$ and level $L(6)$, the explicit dependencies with levels at $p = 3$ are indicated with red arrows. The nodes highlighted in orange fulfill $i + p = 13$ ("diagonal").

```
1: //traverse diagonals of $Lp$ in ascending order $d = i + p$
2: for $d = 1 : L_m + p_m - 1$ do
3:     $p_{\text{start}} = \max(1, d - (L_m - 1))$
4:     $p_{\text{end}} = \min(d, p_m)$
5:     //traverse diagonal $d = i + p = const$ in ascending order of p
6:     for $p = p_{\text{start}} : p_{\text{end}}$ do
7:         $i = (d - p)$ //$i + p = d$ diagonal
8:         $y[p]$ = SpMV($y[p-1]$, $level\_ptr[i]$, $level\_ptr[i+1] - 1$)
9:     end for
10: end for
```

Fig. 6. Basic implementation of the level-blocked (LB) MPK algorithm. L_m is the total number of levels and p_m is the maximum matrix power. The SpMV function implementation from Fig. 2 is used.

represents the grid points and the y-axis shows iterations or time steps [16], [37], [38]. As we have shown above, our level-based MPK algorithm shows the same dependencies, with levels substituting grid points on the x-axis. This opens up a host of options, since we could draw from the large variety of temporal blocking optimizations developed for 1D stencils. Our approach is analogous to parallelogram-style temporal blocking; see [16] for a classification.

The reuse distance of a given level is a central quantity to characterize the cache locality of the level-blocked (LB) MPK. Within the Lp diagram, this quantity can be determined by the number of execution steps between two computations on the same level, i.e., one step in vertical direction. As the scheme traverses the Lp space in consecutive diagonals, a level computed at power p will be reused after $\tilde{d} + 1$ execution stages for the computation of the next power $p + 1$, where $\tilde{d}$ is the number of execution steps in the current diagonal. After the wind-up and before the wind-down phases at the left and right ends of the Lp diagram, we have $\tilde{d} = p_m$; hence, levels are reused after $p_m + 1$ execution steps. This can be observed from Fig. 5 if we concentrate on a single level, e.g., the vertices of $L(10)$ used in the 40th execution step to compute $p = 1$ are reused in the 46th step to compute $p = 2$. As the number of levels is typically much larger than the maximum power stage, we can assume a maximum reuse distance of $p_m + 1$ execution stages. This means if all the matrix entries associated with the $p_m + 1$ successive levels touched between two computations of a given $L(i)$ can be held in a cache, all accesses to this $L(i)$ can be served from the cache with the exception of the first one ($p = 1$), which requires main memory access. Assuming that cache accesses are much faster than memory accesses, the performance of the LB MPK implementation can improve by a factor of at most p_m as compared to the baseline MPK.

Implementation

Two basic implementation decisions for our LB MPK are guided by RACE. First, the complete algorithm operates on the permuted graph. Second, only two lean data structures are required to store the information on the permutation and the levels: The permutation vector (N_r entries) is required to recover the original ordering. The storage location of the first vertex (row) of each level are stored in the `level_ptr` array (one entry per level). Fig. 4e shows the `level_ptr` of our stencil example matrix (see Fig. 4d).

A straightforward implementation of our LB MPK is presented in Fig. 6. The algorithm first iterates over all diagonals of the Lp diagram in ascending order (line 2). Within a diagonal $d = i + p = const$, the computations are processed in increasing order of power p (line 6). Note that to account for the wind-up and the wind-down phase of the parallelogram, the starting and ending power stages are adjusted in lines 3 and 4 of the algorithm. Depending on the power p and the diagonal counter d, the actual level index i to use in the current iteration is calculated in line 7. Finally, in line 8 the vector ($y[p-1]$) containing the required information at power level $p - 1$ and the indices of the first and last row of $L(i)$ are passed to the SpMV function (shown in Fig. 2) to compute $A^p x$ on level $L(i)$. Note that OpenMP parallelization is done within the SpMV function using static scheduling (line 12 in Fig. 2). As there is an implicit barrier after the parallel workshare construct, all threads finish the computations on a given execution stage before proceeding to the next one. In order to reduce the start-up overhead at the parallel region encountered in each SpMV call, the parallel region is opened outside the SpMV routine in our implementation.

Note that the storage of each level is consecutive and the levels are stored in ascending index order. Therefore, the proposed method neither has irregular accesses to matrix entries nor does it have to store extra copies of matrix elements and perform redundant computations, which were required in previous work [15]. Moreover,the parallelization within the levels avoids load imbalance and redundant thread-local copies, which may add significant overhead for irregular matrices and high thread (or core) counts.

Performance analysis of naive version

The naive implementation of the LB MPK already results in a decent performance improvement for some of the matrices presented in Table 2. However, it often falls short of the predicted maximum p_m-fold speedup. For example, with $p_m = 4$ on one socket of CLX, 50% of the matrices in the table showed speedup of less than 10% and almost 10 matrices had a performance degradation compared to the baseline MPK. We choose two representative matrices, `pwtk` and `Flan_1565`, which are exemplary for the major performance shortcomings of the basic LB MPK and we will identify those in the following.

Fig. 7 shows the multithreaded performance and main memory code balance of the LB MPK (Fig. 6) with $p_m = 1$ and $p_m = 4$ along with the baseline MPK (Fig. 2) with $p_m = 4$ on one socket of CLX (20 cores) for both matrices. One

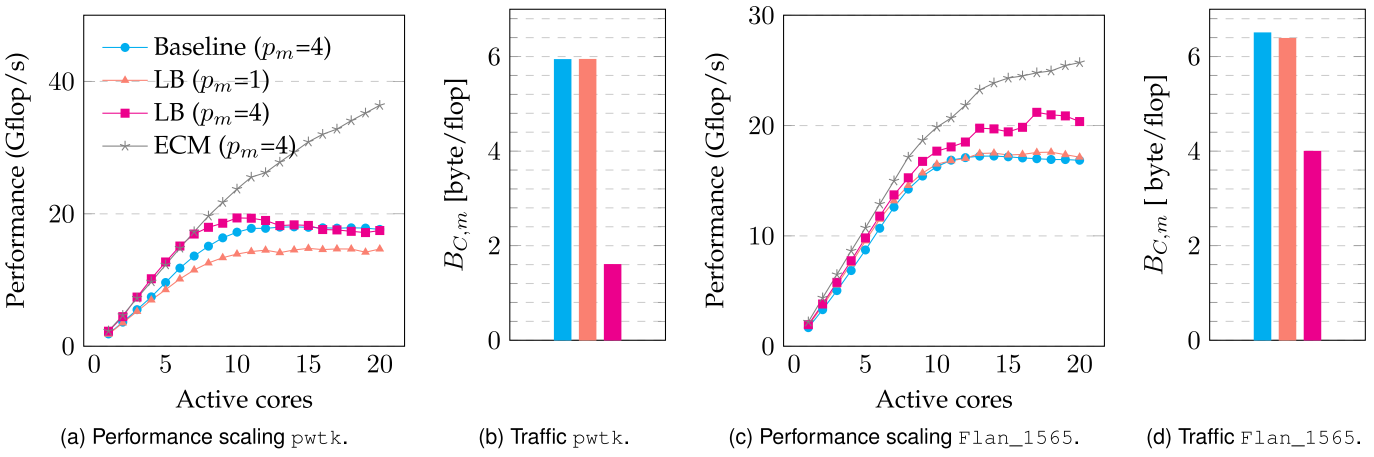


Fig. 7. Scaling performance and main memory traffic of our LB MPK implementation for Ax ($p_m = 1$) and A^4x ($p_m = 4$) in comparison to the baseline MPK on one socket of CLX for the `pwtk` and `Flan_1565` matrices. The stars show the phenomenological ECM performance model [41] (in gray) for the $p_m = 4$ case. The model assumes that the computation of first power of a level ($p = 1$) does not overlap for subsequent powers ($p > 1$).

may expect that LB MPK with $p_m = 1$ and the baseline MPK should deliver the same performance, independent of p_m. They both perform the memory-bound SpMV operations successively but with different execution order within each SpMV function, and their minimum code balance from main memory is $B_C = 6\,\mathrm{byte/flop}$ (see Section 3.1). For $p_m = 4$ case, a data traffic (i.e., B_C) reduction and performance speedup of at most 4× may be achieved when using LB MPK.

For `pwtk`, the typical memory bandwidth saturation pattern is observed for LB MPK ($p_m = 1$, triangles) and baseline MPK (circles) in Fig. 7a. The level-based implementation saturates at a lower level, although both variants attain the same minimum code balance of $B_C = 6\,\mathrm{byte/flop}$ (Fig. 7b). The characteristic behavior is the same for the LB MPK with $p_m = 4$ (squares): In line with the expectation, our method reduces the data traffic by a factor of approximately four ($B_{C,m} \approx 1.5\,\mathrm{byte/flop}$) but it fails to improve performance at the full socket level. It even falls behind the baseline MPK for larger core counts. Further analysis reveals a 1.6× increase in retired instructions[4] for LB MPK ($p_m = 4$) compared to the baseline approach. These instructions are executed in the spin-waiting loop of OpenMP barriers [42], indicating that the synchronization between threads (performed after each computation of a level) is a potential bottleneck. An analysis of the level structure of the `pwtk` matrix confirms the relevance of synchronization cost as the average level size is approximately 850 rows only. At an average of 53 nonzeros per row, the workload of a level is just too low to ignore the synchronization cost, which increases with thread count and may reach a few thousand cycles at a full socket.[5]

The `Flan_1565` matrix shows an opposite characteristic. The performance of LB MPK with $p_m = 1$ is in line with the baseline approach, and the level blocking with $p_m = 4$ achieves a performance improvement of 1.2× (see Fig. 7c). The moderate speedup of LB MPK is reflected in Fig. 7d by its rather high (measured) code balance of approximately $4\,\mathrm{byte/flop}$, indicating that level-blocking is not very cache efficient in this case. The matrix level structure plays a decisive role here as there is a rather small number of levels, some of them being large. Already one of these large levels, which may contain up to 20,000 rows (with about 75 nonzeros per row) has a size of roughly $18\,\mathrm{MB}$, which is more than half of the L3 cache size of the CPU. Moreover, the small number of levels in combination with imbalanced level sizes may cause the irregular performance scaling of LB MPK ($p_m = 4$) in Fig. 7c.

In the following three sections we describe three optimizations of the LB MPK, which are motivated by the performance shortcomings identified above. The first two are targeted at reducing the synchronization cost by forming larger levels ("level groups") and substituting the expensive barrier by point-to-point synchronization. The third optimization improves performance on matrices with dominant, bulky levels by recursively splitting these up ("recursion") to improve cache efficiency.

4.3 Level Groups (LG)

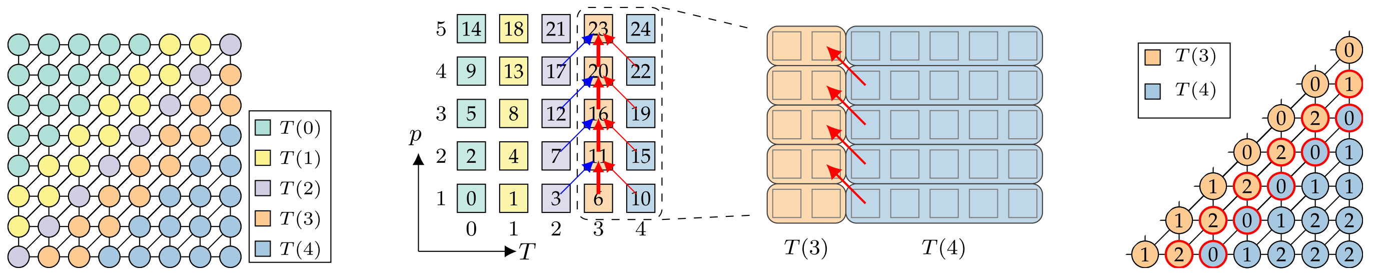
The formation of larger levels follows the idea presented in [10]: Successive levels are aggregated into so-called *level groups*. This allows our LB MPK to operate on these level groups instead of the original levels. Fig. 8a shows the fifteen levels of Fig. 4c being clustered into five level groups $T(0)$–$T(4)$ ($T(i)$ denotes ith level group). The Lp diagram can easily be adapted by replacing the levels by the level groups on the x-axis (see Fig. 8b).[6] Still, the same parallelogram-style blocking can be applied by traversing the level groups using the same rules as for the levels. Parallel execution is performed within a level group, and all threads synchronize after the computation of each group. This strategy satisfies the neighborhood dependencies between levels as required by the LB MPK.

The cache reuse requirements of the LB MPK impose strict limits on the size of the level groups. As discussed in

4. Using the event `INSTR_RETIRED_ANY` in `likwid-perfctr`.

5. For the full CLX socket (ignoring hyper-threading) and the software environment used, a minimum barrier cost of 2,900 cycles was measured by direct barrier benchmarking.

6. For the sake of uniformity we keep the name "Lp" for the diagram instead of "Tp," although here we plot level groups (T) instead of levels (L) on the x-axis.



(a) Graph of the matrix. (b) Lp diagram. (c) Lp diagram zoomed view. (d) Subgraph of T(3) and T(4).

Fig. 8. (a) Levels as in Fig. 4c being consolidated to five level groups ($T(0) - T(4)$). (b) Lp diagram corresponding to the level groups and the execution order of each level group at different power stages. The bold red arrow (vertical) corresponds to the dependency with all the levels of the same level group $T(i)$ at the previous power stage $p-1$, and the slanted red arrow corresponds to the dependency with the lowest-indexed level of next level group $T(i+1)$ at the previous power stage. The blue arrow corresponds to a dependency that is automatically fulfilled by the execution order. (c) Zoomed-in view of the $T(3)$ and $T(4)$ level groups in the Lp diagram. The levels within the level group are seen as square nodes and the dependency between levels in $T(i)$ and $T(i+1)$ are clearly visible. The subgraph corresponding to the zoomed region is shown in (d). The vertices drawn with red circles correspond to the two boundary levels between which synchronization in southeast direction has to be established. The numbers on the vertices represent the id of the thread (tid) working on that vertex.

Section 4.2, $p_m + 1$ neighboring level groups have to be kept in cache. Therefore, if we assume neighboring level groups to be of similar size, the following criterion has to be satisfied by the ith level group $T(i)$:

$$(p_m + 1) \times N_{\mathrm{nz}}\ (T(i)) \times 12\,\text{bytes} < fC, \tag{5}$$

where $N_{\mathrm{nz}}\ (T(i))$ is the number of nonzeros in $T(i)$, C is a parameter representing the available cache size (in bytes), and f is a safety factor. The cache size parameter is typically chosen to be less than or equal to the physical size of the cache(s) targeted for level blocking. The safety factor ($f = 0.5$ in this work) accounts for extra traffic from other data structures and inefficiencies of the cache replacement policies. The left part of inequality (5) is the total memory traffic generated by accessing p_m+1 level groups (assuming 12 bytes per nonzero entry of the matrix, see Section 3.1), and the right part is the effective cache size. If (5) is satisfied then level group $T(i)$ can be reused from cache for $p_m > 1$; otherwise, at least parts of it must be loaded from main memory.

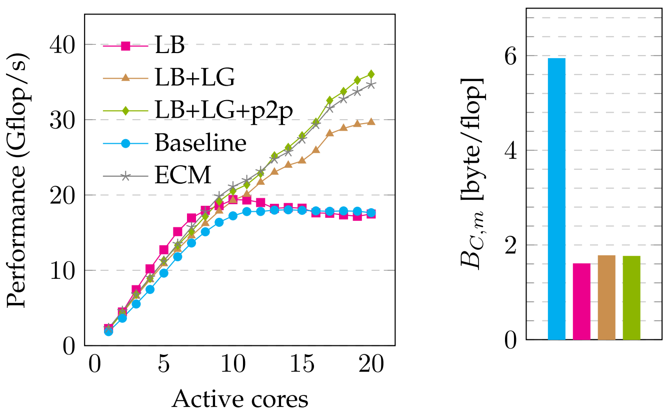
Inequality (5) is crucial to the construction process of the level groups. We form the first level group $T(0)$ by accumulating levels $L(0)$ ... $L(j)$ up to the largest j for which $N_{\mathrm{nz}}\ (L(0)) + \ldots + N_{\mathrm{nz}}\ (L(j)) = N_{\mathrm{nz}}\ (T(0))$ satisfies (5). The same procedure is repeated starting from level $L(j+1)$ to find $T(1)$, and successively forming the other level groups. It can be seen from Fig. 8a that this procedure creates level groups with almost equal numbers of nonzero elements. In regions where levels contain fewer nonzeros per level, more levels are aggregated (see $T(0)$ in Fig. 8a) while in regions with bulkier levels, even a single level can form a level group (see $T(2)$ in Fig. 8a). As a result, the number of level groups is typically much smaller than the number of levels.

In the level-group-based scheme, synchronization only happens after the computation of each level group, which greatly diminishes the impact of barriers in case of LB MPK for the pwtk matrix: The performance of the LB MPK for $p_m = 4$ (LB+LG; triangles in Fig. 9a) improves on a full socket by $1.6\times$ compared to the baseline MPK approach. At the same time, we only encounter a minor increase in the measured code balance (see Fig. 9b) since the condition (5) limits the size of the level groups. Also the overhead from extra instructions reduces from 60% for the naive LB MPK version to only 7%. The cache size parameter $C = 35\,\text{MB}$ has been set to target the aggregate physical size of L3 and L2 caches of CLX.

4.4 Point-to-Point (P2P) Synchronization

The concept of level groups allows us to relax the lockstep-like synchronization by eliminating the OpenMP barrier (implicit barrier in line 12 of Fig. 2) after computation of a level group. The parallel LB MPK must ensure that the computations on the following levels and level groups are completed before the computation of power p for a given level group $T(i)$: (A) the same level group $T(i)$ with previous power $p-1$ (bottom neighbor in Lp diagram), (B) the highest-indexed (rightmost) level of $T(i-1)$ with power $p-1$ (southwest neighbor in Lp diagram), and (C) the lowest-indexed (leftmost) level of $T(i+1)$ with power $p-1$ (southeast neighbor in Lp diagram).

Note that the most stringent condition (A) can be enforced without a global barrier synchronization since it is only relevant when a level group $T(i)$ is visited again for computing the next power on it, which happens after a full diagonal traversal. We thus implemented a customized locking mechanism (see below for details), which allows



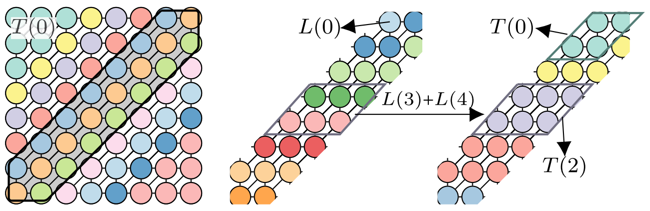
(a) Performance scaling. (b) Memory traffic.

Fig. 9. (a) Performance improvement of LB MPK using level group (LG) optimizations and point-to-point synchronization (p2p) for the pwtk matrix with $p_m = 4$ on CLX. (b) Memory traffic of the four variants shown in (a).

threads to spread out over a full diagonal of the Lp diagram. Due to the diagonal traversal scheme of the Lp diagram, condition (A) implies condition (B) as the southwest neighbor of a level group is always visited before its bottom neighbor (see numbering of execution order in 8b). Finally, a similar locking mechanism is required to ensure condition (C). Here, only the completion of the relevant boundary level of the southeast neighbor has to be ensured (see Fig. 8c).

The locking mechanism is implemented using `volatile int` arrays, `omp atomic` directives, and spin-waiting loops. For ease of tracking threads, we do not use OpenMP worksharing schemes; instead we manually assign the vertices in each level group to the N_t threads in a static manner. The numbers in Fig. 8d illustrate such a thread assignment in level groups $T(3)$ and $T(4)$ for a total of three threads, i.e., $N_t = 3$. To satisfy condition (A), a `volatile int` array `U` of size equal to total number of level groups (L_m) is defined and set to zero in the initialization phase. Each thread after finishing work on xth level group, $T(x)$, atomically increments `U[`x`]` by one. Condition (A) implies that, in order to start working on a level group $T(y)$ at power p, each thread has to ensure all the threads have finished computing the previous power $p-1$ of $T(y)$. This is ensured by checking if `U[`y`]` $= (p-1) \times N_t$; if it is not, the thread waits in a spin-waiting loop till the other threads finish their computations. Similarly, condition (C) is ensured by a two-dimensional `volatile int` array `V` having the same dimension of the Lp graph, i.e., $L_m \times p_m$. Here only threads working at the boundary levels of the two nearby level groups need to interact. For example, in Fig. 8d to satisfy condition (C), thread 0 working on the first level of $T(4)$ has to finish the power $p-1$ computation before thread 1 and 2 can start the power p computation on the last level of $T(3)$. To achieve this, the thread(s) working on the first level of the level group $T(x)$ atomically increment `V[`x`][`p`]` by one after performing the power p computation. The thread(s) that compute power p on last level of a level group $T(y)$ then checks if the first level of the southeast neighbor has completed computation, i.e., if `V[`$y+1$`][`$p-1$`]` $=$ `H[`$y+1$`]`, where `H` is a precomputed array which stores the number of threads that work on the first level of each level group. If the equality is not satisfied, the thread waits in a spin-waiting loop until it is.

Fig. 9a shows the performance scaling of this implementation (LB+LG+p2p; diamonds) in comparison to the other variants; it yields a performance boost of $1.2\times$ over the version with level groups and barrier synchronization (LB +LG). A part of this speedup comes from the reduced synchronization cost. The rest is due to the relaxation of lock step synchronization that allows for overlap between memory and cache transfers, i.e., some threads can work on the memory-bound phase ($p=1$) while the rest work on a cache-bound phase ($p > 1$). The optimization thus brings us close to our phenomenological ECM model (stars in Fig. 9a) and results in a $2\times$ speedup over the baseline approach. Note that as the sizes of level groups change, traffic within inner cache levels will also change. Since the ECM model uses this data traffic as input, it results in slightly different models when sizes of level groups change. This can be observed for example by comparing Figs. 7a and 9a.



(a) Graph. (b) Levels. (c) Level groups.

Fig. 10. (a) Level groups in the graph. The shaded subgraph shows the level groups with more than six rows, where recursive treatment is applied. (b) BFS levels within the subgraph. (c) Level groups formed from the levels within the subgraph.

4.5 Recursion

The negative impact of bulky levels (which do not satisfy (5)) on main memory traffic for the LB MPK approach (see Fig. 7d) has been identified and discussed for the `Flan_1565` matrix in Section 4.2. In the RACE coloring scheme [10], a recursive approach has been presented to generate higher levels of parallelism within bulky levels. The same method can be used in our context to successively generate new levels or level groups of reduced size until they fit into cache. The idea is to apply the LB MPK presented so far to the subgraph defined by a single level or a set of consecutive levels. As a result, a new set of smaller levels is generated for this subgraph. If some of the new levels still violate (5), the procedure is applied again to the new subgraph defined by these levels. This procedure can be continued until all levels fit into a cache.

We start by locating (consecutive) levels that do not fit in a cache and isolate the subgraph formed by these levels. BFS is applied first to this subgraph, and then a set of level groups is formed from these BFS levels. The resulting level groups are typically smaller than the previous ones as neighboring vertices outside the subgraph do not need to be considered. Fig. 10 illustrates this procedure for our stencil example and a hypothetical cache which satisfies (5) for level groups $T(i)$ containing no more than six vertices. We find that the three bulkier level groups (containing one level each) $T(4) - T(6)$ do not satisfy the condition. The subgraph induced by these three levels is formed (shaded with gray background in Fig. 10a), and we identify the eight BFS levels of this subgraph (Fig. 10b). Following the discussion in Section 4.3, the level groups of the subgraph are constructed (Fig. 10c). They are now small enough to satisfy (5) and the process stops.

In general, the procedure can be applied recursively until the level groups satisfy (5) or a user-specified maximum recursion stage s_m is reached, where $s_m = 0$ is the case without any recursion. In the following, s ($\leq s_m$) denotes the current recursion stage. The maximum recursion stage should, however, be limited as applying the recursion step leads to loss of data locality at the boundaries of the subgraph. This happens because the subgraphs are permuted (BFS) without taking into account the neighbors outside the subgraph. Fig. 11 demonstrates this effect by comparing the matrix structure of our stencil example without recursion ($s_m = 0$) and with one recursion step ($s_m = 1$) applied to the inner levels. The matrix bandwidth increases for the boundary elements of the subgraph because of the mismatch of the vertex numberings outside and inside the subgraph. While

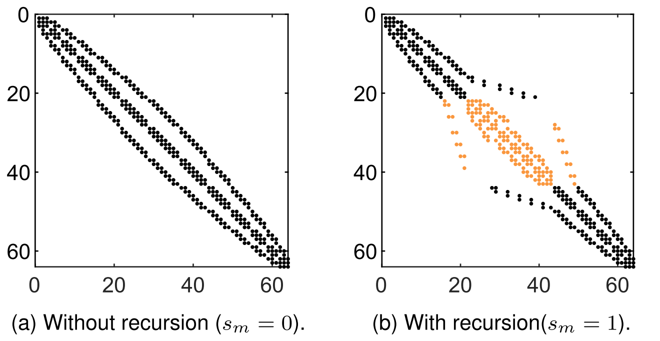


Fig. 11. Sparsity pattern of the stencil example matrix without (a) and with (b) recursion. The entries of submatrix where recursion is applied is shown with orange color in (b).

access to the matrix elements remains linear, the more irregular accesses to the right-hand side vector may impact the overall MPK performance. Note that the graphical representation in Fig. 11b exaggerates this effect, since in our toy problem the subgraph represents a substantial fraction of the full problem. The performance influence of the maximum recursion stage s_m is discussed later in Section 5.3.

As each subgraph (formed from consecutive levels) of a recursion stage creates its own level groups, we construct Lp diagrams for each subgraph, i.e., Lp^s represents the Lp diagrams of recursion stage s. Fig. 12 shows the two Lp diagrams of the stencil example for $p_m = 2$: Lp^0 representing $s = 0$ on the full graph (Fig. 10a), and Lp^1 after the first recursion stage of the subgraph corresponding to level groups in Fig. 10b. Note that the numbering of the execution order is local to each Lp^s diagram. All level groups of a subgraph of Lp^s to which recursion is applied have the same execution order in Lp^s (e.g., the subgraph related to $T(4)$–$T(6)$ in Lp^0 is executed in step 8 of Lp^0 in Fig. 12). The actual execution order of the vertices in this subgraph is determined by Lp^{s+1} (see Lp^1 in Fig. 12). In general, the actual execution of a given vertex is determined by the Lp diagram associated with the highest recursion stage of the vertex. Of course the execution order in the Lp^s diagrams still needs to maintain the data dependencies of the LB MPK. With $p_m = 2$ as used in Fig. 12 we can still maintain our diagonal-type execution order within the diagrams: $T(7)$ of Lp^0 is updated to $p = 1$ at step 7. Lp^1 is calculated as step 8 of Lp^0. In step 9 of Lp^0, $T(3)$ is updated to $p = 2$.

For $p_m > 2$, the dependency relations between execution order of Lp^s and Lp^{s+1} are more complicated. This is depicted in Fig. 13, where Lp^0 with $p_m = 3, 5$ is shown for 15 level groups and $T(6)$–$T(8)$ form the subgraph on which Lp^1 is built. Actually, all nodes in the parallelogram formed by the diagonals in Lp^s (Lp^0 in our example) that cross the

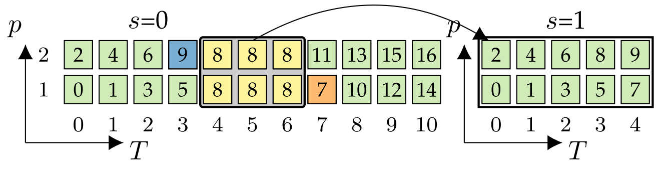


Fig. 12. The Lp diagram for $p_m = 2$. *Left*: Lp diagram of the $s = 0$ recursion stage (Lp^0), which contains level groups of the entire graph seen in Fig. 10a. The level groups selected for recursion are highlighted. *Right*: Lp diagram at $s = 1$ (Lp^1), which consists of the level groups shown in Fig. 10b. The execution order of the Lp graph is shown with numbers.

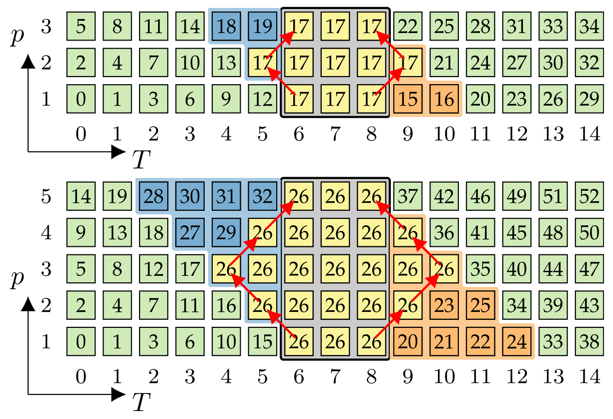
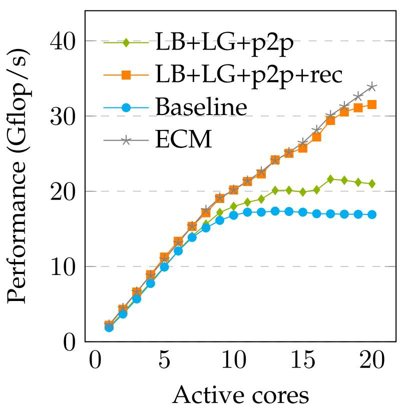


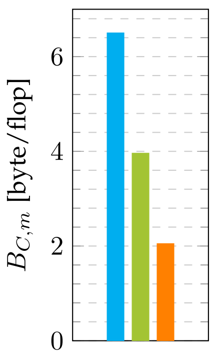
Fig. 13. Lp^0 diagrams with $p_m = 3$ (above) and $p_m = 5$ (below) corresponding to an arbitrary graph where recursion has to be applied to level groups $T(6)$–$T(8)$ forming Lp^1 (not shown). The red arrows show the longest input (output) dependency from (to) the boundary points of the recursive region.

subgraph to be refined have dependency relations to the vertices in this subgraph. Within the parallelogram, there are three different types of dependencies: (i) Nodes which provide input only to Lp^{s+1} and which need to be calculated before Lp^{s+1} (orange color in Fig. 13), (ii) nodes which have only an output dependency on Lp^{s+1} and need to be calculated after Lp^{s+1} (blue color in Fig. 13), (iii) nodes within the "diamond" embedded in the parallelogram, which have input and output dependencies related to the computations in Lp^{s+1}. The nodes within (i) and (ii) can be processed using the execution order as given by the Lp^s diagram. However, the nodes within the "diamond" (iii) have to proceed in coordination, therefore they all follow the execution order of the Lp^{s+1} diagram. This means that the recursive treatment is applied not just to the subgraph (here $T(6)$–$T(8)$) but also to the boundary levels within the diamond. In case of $p_m = 5$ (illustrated in the lower panel of Fig. 13), the boundary level groups at left ($T(4)$ and $T(5)$) as well as right ($T(9)$ and $T(10)$) will also be considered for the recursion. The vertices of these boundary level groups are then permuted within each level group according to the dependencies that arise from the subgraph. These refined level groups within the boundary levels are thus used when executing the recursive part using the Lp^{s+1} diagram. For example, in Fig. 13 (below), the calculation of $p = 2$, $p = 3$, and $p = 4$ at $T(5)$ is carried out using the refined levels. Since the permutations are conducted only within each boundary level group, the execution order of the parent Lp diagrams (here Lp^0) remain unchanged. This means that the computation of $p = 1$ and $p = 5$ at $T(5)$ can proceed without any change using the execution order of Lp^0. This diamond-type execution structure is well known from diamond tiling [38] applied to stencils. Note that this recursive refinement approach is not limited to a single subgraph of a given Lp^s. However, if multiple subgraphs need to be refined, the parallelograms formed by these subgraphs must not overlap.

The parallelization within each of the recursive stages follows the same procedure as explained in Section 4.4, with two modifications: First, each recursive stage needs to define and work with a separate set of arrays U, V, and H (see Section 4.4) to lock their corresponding part of the Lp^s diagram. Second, the vertices having dependencies (i) and



(a) Performance scaling.



(b) Memory traffic.

Fig. 14. (a) Performance improvement of LB MPK using recursion (squares) compared to the one without recursion (diamonds) for the `Flan_1565` matrix with $p_m = 4$ on one socket of CLX. Both versions use level groups and p2p optimizations. The performance of the baseline approach as well as the ECM model is also shown for reference. (b) Measured memory traffic of the three variants on the left.

(ii) in Lp^s have to be computed before and after the computation of Lp^{s+1}, respectively. This can be ensured by checking the corresponding values of elements in array `U` as shown in Section 4.4.

The impact of the presented recursion scheme on the performance of the LB MPK method for the `Flan_1565` matrix with $p_m = 4$ is shown in Fig. 14a. We used a cache size parameter $C = 45\,\text{MB}$ for LB MPK methods and set $s_m = 4$ for the case with recursion (squares). In this setting, the Lp^0 diagram has three subgraphs to which recursive treatment is applied. Via improved cache reuse, the recursion improves the full-socket performance by a factor of almost $1.4\times$ compared to the version without recursion. This comes with a corresponding reduction of almost $2\times$ in main memory data traffic (Fig. 14b). Compared to the baseline MPK approach, we achieve an overall reduction of main memory traffic by $3.2\times$ and an increase in performance by $1.8\times$ on a full socket of CLX. These numbers and the (close to) linear scaling of our method indicate that main memory access is no longer the performance bottleneck.

4.6 RACE

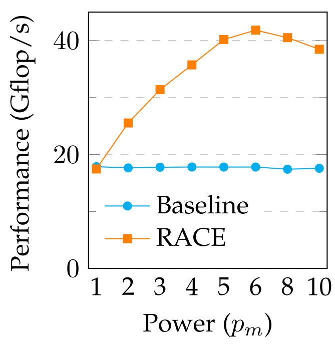
The LB MPK algorithm including all optimizations discussed above has been implemented in the RACE library (code available at [43]). In the following we therefore refer to our LB MPK implementation as "RACE MPK." The library supports both preprocessing and execution phases of the LB MPK. For preprocessing, RACE requires the matrix, highest power p_m, cache size C, and maximum recursion stage s_m as input and returns the permutation vector as output. The user then has to pass the permuted matrix and a call-back function to RACE for execution. RACE will execute the call-back function in parallel (using OpenMP threading) according to the internally created `level_ptr` and Lp diagrams.

5 Parameter Study

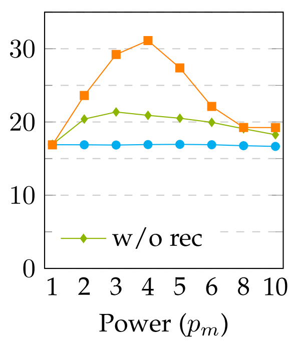
Our RACE MPK as introduced in the previous section has three input parameters: the maximum power p_m, the cache size C, and the maximum recursion stage s_m. In this section we discuss the qualitative impact of these parameters on the performance of RACE MPK.

5.1 Influence of p_m

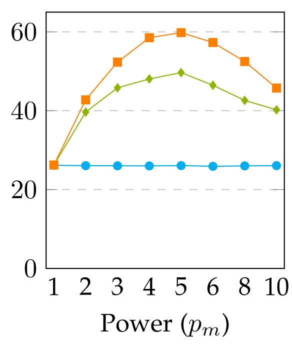
Ideally, RACE MPK requires to access main memory for each level group exactly once at $p = 1$. The remaining $p_m - 1$ accesses can potentially be served from the cache(s) (see Figs. 9b and 14b). As a consequence, cache utilization and performance should increase with p_m. However, as p_m gets larger, the number of level groups grows and their size must reduce as condition (5) has to be fulfilled, which results in higher synchronization cost. These opposing effects result in a typical performance pattern as shown in Fig. 15a for the `pwtk` matrix on CLX. Initially the performance increases almost linearly with p_m but starts to drop gradually at larger p_m ($\approx$ 6–8 in our example). For matrices that require recursion, the performance drop is more prominent and occurs at a lower p_m as shown in Fig. 15b for the `Flan_1565` matrix on CLX. The additional overhead at the boundaries of the recursively refined level groups (see discussion in Section 4.5) add another performance penalty. Of course, the p_m value at which performance starts to decrease depends on the matrix and the cache size. This can be observed by comparing the performance of `Flan_1565` on the three architectures (Figs. 15b, 15c, and 15d). On ROME (Fig. 15d) with its large last-level cache, the matrix does not require recursion at all and the performance increases up to $p_m = 10$, where the RACE MPK achieves a speedup of $4\times$



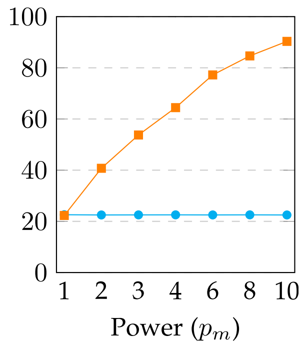
(a) `pwtk`, CLX.



(b) `Flan_1565`, CLX.

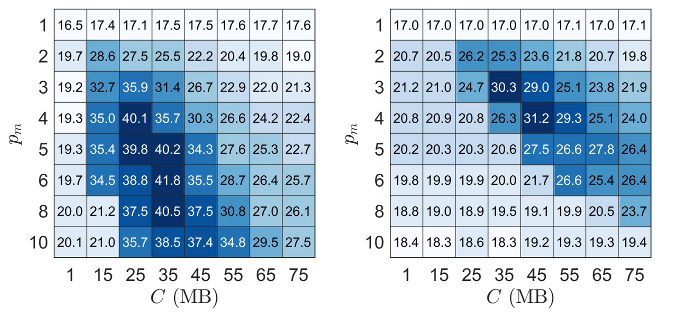


(c) `Flan_1565`, ICL.



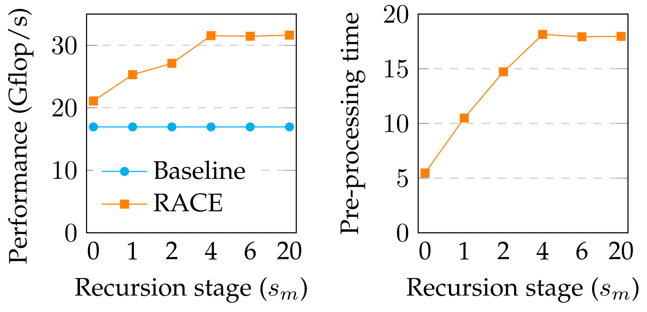
(d) `Flan_1565`, ROME.

Fig. 15. Performance as a function of maximum power p_m for RACE and the baseline implementation of MPK. For cases where recursion yields a speedup, we also plot the performance of RACE without recursion (in green) for comparison.



(a) `pwtk`. (b) `Flan_1565`.

Fig. 16. Influence of cache size C and power p_m on performance (in Gflop/s) of the RACE MPK using all cores of a socket on CLX.



(a) Performance. (b) Preprocessing cost.

Fig. 17. (a) Performance influence of maximum recursive stage s_m on the performance of the `Flan_1565` matrix with $p_m = 4$ and $C = 45$ MB on one socket of CLX. (b) Corresponding preprocessing cost of RACE in equivalent number of SpMVs.

compared to the baseline MPK. The ICL (Fig. 15c) and CLX (Fig. 15b) CPUs need recursion to achieve best performance. The maximum performance is attained at p_m values of 5 and 4, resulting in speedups of 2.3× and 1.8× with respect to the MPK baseline on these two architectures. Note that performance improvements decrease with decreasing cache sizes.

For applications computing $A^k x$ using RACE MPK, the best strategy is to identify the optimal p_m value p_m^{opt} and perform the $A^{p_m^{\text{opt}}} x$ computations multiple times (if $k > p_m^{\text{opt}}$) until the power k is reached. If k is not a multiple of p_m^{opt}, the remainder computations can be done using MPK kernels with $p_m < p_m^{\text{opt}}$.

5.2 Influence of C

The interaction of cache size C and highest power p_m is shown as a heatmap in Fig. 16 for the `pwtk` and `Flan_1565` matrices on CLX. The optimal C value is between 25 and 45 MB irrespective of p_m and the matrix. This is in good qualitative agreement with the aggregate size of the L3 (27.5 MiB, victim) and L2 cache (20 MiB) of CLX. Of course, the RACE MPK method works best when blocking for the biggest available cache. Smaller C values lead to smaller level groups (see (5)) and therefore higher synchronization and recursion overheads. On the other hand, C values bigger than the total cache size will obviously provoke cache misses.

5.3 Influence of s_m

For matrices that require recursion to fulfill (5), the maximum recursion depth s_m may stop the recursion procedure even if the condition is still violated for some level groups. Fig. 17a depicts the performance behavior of the `Flan_1565` matrix with $p_m = 4$ on CLX as a function of s_m. Initially, the performance increases with s_m as the level groups become smaller. When (5) is fulfilled at $s_m = 4$ for all level groups, performance saturates. Note that increasing s_m does not always have the positive performance effect as observed for `Flan_1565`. The overhead at the boundaries of the refined subgraphs may overcompensate the gains of increased cache efficiency. For example, in case of the `RM07R` matrix on ICL (not shown in plots) with $p_m = 3$ ($=p_m^{\text{opt}}$) it was found that $s_m = 0$ (no recursion) achieves 1.2× better performance than $s_m = 13$, where all the level groups fit in cache. Of course, the optimal value of s_m is determined by an intricate interplay of cache properties and matrix properties and thus cannot be found analytically. Typically, recursion should only be applied if condition (5) cannot be fulfilled with $s_m = 0$. In this scenario, the larger the matrix, the deeper the required recursion since bigger matrices tend to have bulky levels. For moderately large matrices ($N_{\text{nz}} \lesssim 2 \times 10^6$), recursion of up to $s_m = 15, \dots, 20$ should be scanned for best performance, while for large matrices it is advisable to test larger recursion depths ($s_m = 40, \dots, 80$).

The preprocessing cost increases with s_m as levels have to be found for recursive subgraphs. This can be seen in Fig. 17b for the `Flan_1565` matrix, where the preprocessing cost (shown in equivalent SpMVs) increases with s_m up to $s_m = 4$. The construction of levels (BFS) dominates the preprocessing time. The other parameters p_m and C do not have a considerable impact on preprocessing time as changing them does not require to generate new levels.

6 Performance Evaluation

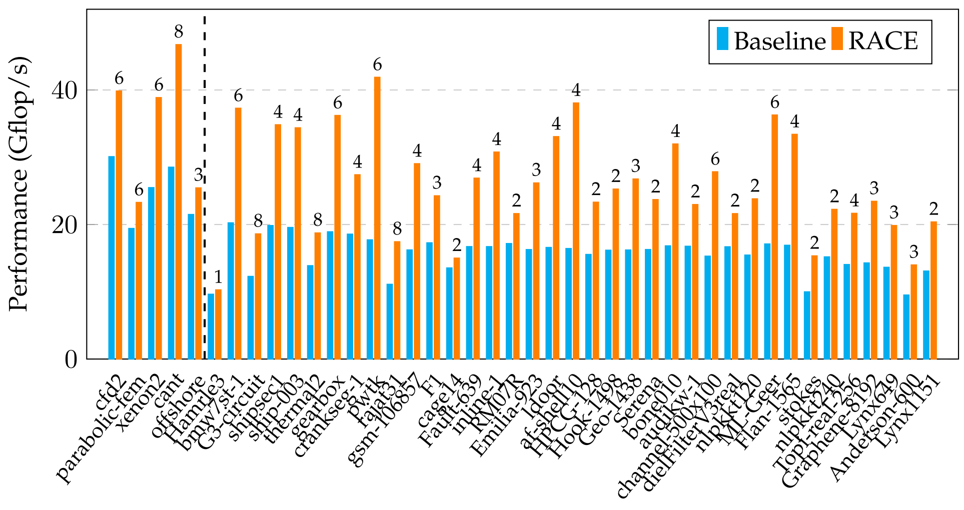
In this section we investigate the performance of RACE MPK and compare it against the baseline MPK for 42 different sparse matrices commonly seen in literature. The details of these matrices can be found in Table 2.

6.1 Experimental Setup

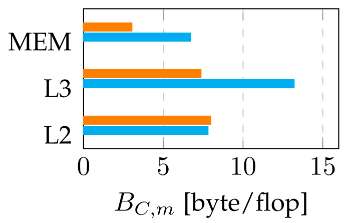
All matrices were stored in the CRS data storage format (see Section 3.1). We used all the cores on one CPU socket and one thread per core. To ensure vectorization of the kernels we used `#pragma simd vectorlength(VECLEN) reduction(+:tmp)` on the innermost loop of the SpMV (see Fig. 2). The vector length (`VECLEN`) was specified explicitly and was chosen to be the maximum SIMD width of the hardware.

For both baseline and RACE, the matrices were preprocessed with RCM reordering using the Intel SpMP [44] library if it improved the performance. The baseline method was parallelized using the `#pragma omp parallel for schedule(static)` workshare construct along the outermost loop (over matrix rows).[7] RACE is parallelized using OpenMP pragmas by manually assigning the vertices in each level group to the threads and implementing the point-to-point synchronization

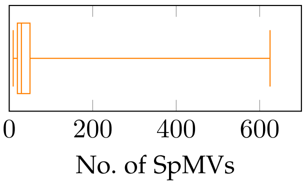
7. Note that static scheduling was chosen as the benchmark matrices (see Table 2) did not have highly imbalanced row lengths.



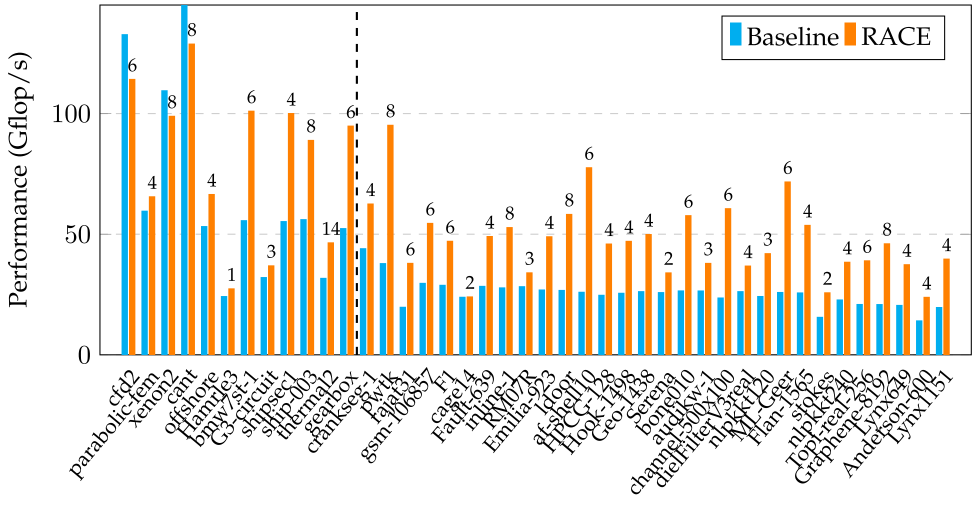
(a) Performance, CLX.



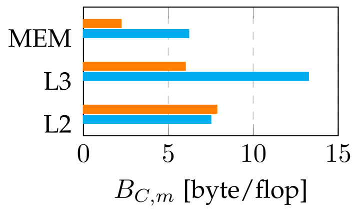
(b) Data traffic, CLX.



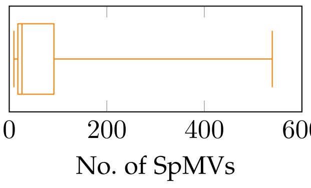
(c) Preprocessing cost, CLX.



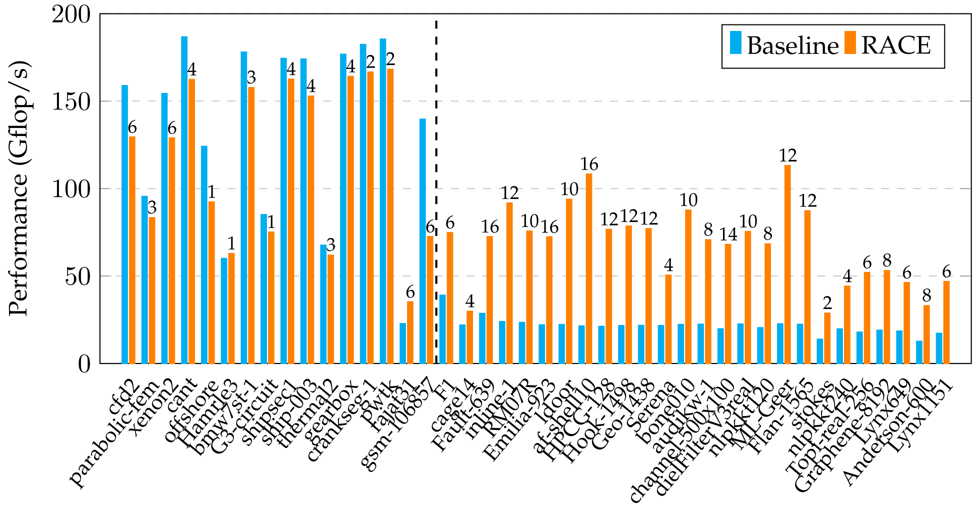
(d) Performance, ICL.



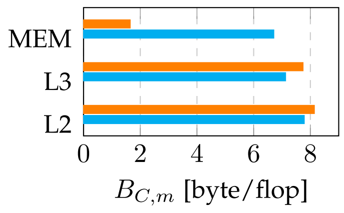
(e) Data traffic, ICL.



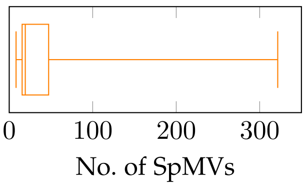
(f) Preprocessing cost, ICL.



(g) Performance, ROME.



(h) Data traffic, ROME.



(i) Preprocessing cost, ROME.

Fig. 18. (a), (d), (g): Performance comparison between baseline and RACE MPK on CLX, ICL, and ROME, respectively. The dashed line represents the total available cache size and the numbers show the tuned p_m values corresponding to the RACE performance. (b),(e), (h): L2, L3, and memory code balance of RACE MPK and baseline approach on the three architectures. The memory and cache data traffic shown is the average across all the in-memory matrices (i.e., to right of dashed line in the respective performance plot). (c), (f), (i): Statistics of the preprocessing cost of RACE MPK for all in-memory matrices. The cost is shown as the number of SpMVs that can be executed in the given time.

mechanisms discussed in Section 4.4. The parameter space of RACE (see Section 5) was tuned in the following range: $p_m \in \{[1{:}1{:}3] \cup [4{:}2{:}16]\}$,[8] C in the range of total cache (L3+L2) size of the hardware, and $s_m \in \{0, 1, 2, 4, 6, 20, 80\}$. More specifically, the parameter space of C (in MB) is [25:10:45] for CLX, [65:10:105] for ICL, and [100:50:250] for ROME.

6.2 Results

Figs. 18a, 18b, and 18c show the performance of baseline and RACE MPK on CLX, ICL, and ROME, respectively. The matrices are ordered (left to right) according to increasing data-set size (number of nonzeros). The vertical lines represent the total cache size of the respective hardware and thus categorize matrices into memory-resident (right of line) and cache-resident (left of line) scenarios.

For the smallest matrices, RACE does not usually show significant speedup over the baseline method as these matrices comfortably fit in cache. However, as the working set approaches the cache size, RACE starts to develop clear performance advantages. On CLX and ICL, this effect is pronounced already for larger "in-cache" matrices, while for ROME the benefit of RACE MPK starts exactly at the boundary between cache- and memory-resident matrices. There are two main reasons for this: (i) The transition between L3 and main memory bandwidth on Intel architectures is gradual compared to AMD ROME (see Fig. 1), and (ii) the L3 and L2 caches have almost similar sizes on both Intel architectures, and the blocking in RACE targets the combined L3 and L2 caches. Therefore, for smaller matrices that fit into the L3 cache, RACE can reduce the L2 traffic compared to the baseline method. On the other hand, for ROME the L3 cache is considerably bigger than the L2 and hence the blocking is performed only in the L3 cache, thereby bearing no benefit for matrices fitting in the L3 cache.

For all memory-resident matrices RACE has a clear performance advantage on all architectures, achieving typical speedups of 2× to 5× compared to the baseline MPK. This is correlated with the measurements of the average L2, L3, and main memory traffic shown in Figs. 18b, 18e, and 18h. Here the baseline MPK approach is close to the SpMV's minimum traffic limit of $6\,\mathrm{byte/flop}$[9], indicating the absence of caching of matrix elements. In most cases the baseline approach is also strongly memory bandwidth bound and thus performs close to the optimistic (memory-bound) roofline limit (i.e., b_{Mem}/B_C) of 19, 28, and 24 Gflop/s on CLX, ICL, and ROME, respectively. For RACE we find a memory traffic less than the minimum SpMV limit on all the three architectures due to caching of the matrix elements. On CLX and ICL, even the L3 traffic reduces substantially as the large (aggregate) L2 cache contributes substantially to the blocking. The reduced data traffic of RACE results in a performance higher than the SpMV in-memory roofline limit and the baseline approach. Correlated with the reduction of main memory traffic, RACE achieves the highest speedups on ROME where we observe an average (maximum) performance gain of 3.2× (5×). On ICL and CLX, we observe an average speedup of almost 1.9× and 1.6×, respectively, and a maximum speedup of 3× and 2.3×.

The significantly higher performance (as well as speedup) of RACE on ROME compared to the Intel systems can be attributed to its larger L3 cache and higher L3 bandwidth (see Fig. 1). A larger L3 allows to cache level groups for higher p_m values (see (5)). This can be observed in the tuned p_m values annotated with numbers on top of the RACE performance bars. We see that for the same matrices the p_m values on ROME are higher than that of ICL and CLX. This allows for matrix elements to be cached longer on ROME and results in an average memory traffic reduction of 4.1× (see Fig. 18h) compared to the baseline, while on ICL and CLX the reduction is 2.8× and 2.2×, respectively.

6.3 Preprocessing Cost

Now that the performance behavior of RACE is understood, we need to investigate its preprocessing overhead. The box plots in Figs. 18c, 18f, and 18i show statistics of RACE's preprocessing cost for memory-resident matrices. These cost is shown in equivalent number of SpMVs that can be executed during the time required for preprocessing. In general, the cost reduces as the cache size of the architecture increases, i.e., on ROME the preprocessing time is well under the time of 40 SpMVs for most matrices while on Intel systems the equivalent SpMV invocations is around 50 SpMVs. This is due to larger cache sizes requiring fewer recursion stages (s_m), since the preprocessing cost increases with s_m (see Fig. 17b). For same reason, larger matrices tend to incur higher preprocessing cost as more recursion stages are typically required to make parts of the matrix fit into the caches.

Most of the preprocessing time ($>95\%$) is spent on determining the levels using BFS. In RACE we use a parallel BFS implementation similar to the top-down approach from [46], where the parallelization is accomplished by distributing the vertices in a level (frontier) to different threads. However, this method lacks sufficient parallelism if the number of vertices in a level is too small. This is the case with the `RM07R` matrix, which is an outlier in the preprocessing cost on all three architectures. Here, a lot of levels contain only one vertex and preprocessing is largely sequential.

7 Conclusion and Outlook

In this article we have developed a level-based blocking algorithm (RACE MPK) to increase the performance of sparse matrix power kernels (MPK). The RACE algorithm uses levels, generated by breadth-first search, to increase temporal access locality for the matrix entries by reusing them for successive power computations. Various hardware-oriented algorithmic optimization strategies such as level grouping, point-to-point synchronization, and recursive application of the level-blocking scheme are introduced to further improve the performance of RACE MPK. A thorough performance analysis on a representative set of 42 matrices shows that RACE MPK outperforms a standard MPK implementation by an average factor of 2× and 3.5× on modern Intel and AMD CPUs.

8. In the format [start value : increment : end value].

9. The L3 traffic measurements using `likwid-perfctr` is double on CLX and ICL as the current version of `likwid-perfctr` cannot distinguish traffic between main memory and L2 cache with L3 and L2 caches; see [45] for details.

The MPK finds its use in a large variety of applications, especially in the field of communication-avoiding algorithms [12], s-step Krylov solvers [47], polynomial preconditioners [48], Chebyshev time-propagation [49] and exponential time integration [50]. The time-consuming part in most of these applications is the MPK computation, which can be accelerated with RACE's blocking scheme. Future work in this direction includes integrating RACE MPK into communication-avoiding s-step Krylov solvers and polynomial preconditioners from the Trilinos [51] framework. For multi-node MPK computations, our level-based cache blocking scheme can be integrated with existing ideas (e.g, [11], [24]) to enable a highly efficient distributed MPK scheme with low inter-node communication overhead. Another interesting research direction that we are currently pursuing is the development of a GPU implementation for the cache-blocked MPK method. Here, the two main challenges are the rather small cache size and high synchronization cost on GPUs.

Acknowledgments

The authors thank Johannes Langguth for providing the Lynx matrices from the cardiac electrophysiology field. The authors also thank Kengo Nakajima for helpful discussions within the JHPCN project "Innovative Multigrid Methods II." The authors would also 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.

References

[1] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, Jan. 2017.

[2] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia: Society for Industrial and Applied Mathematics, 1994.

[3] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, Dec. 2003.

[4] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. 21st Annu. Symp. Parallelism Algorithms Architectures*, 2009, pp. 233–244.

[5] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.

[6] L. Oliker, X. Li, P. Husbands, and R. Biswas, "Effects of ordering strategies and programming paradigms on sparse matrix computations," *SIAM Rev.*, vol. 44, no. 3, pp. 373–393, 2002.

[7] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. Yelick, "When cache blocking sparse matrix vector multiply works and why," *Applicable Algebra Eng. Commun. Comput.*, vol. 18, no. 3, pp. 297–311, 2007.

[8] P. Balaprakash et al., "Autotuning in high-performance computing applications," *Proc. IEEE*, vol. 106, no. 11, pp. 2068–2083, Nov. 2018.

[9] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, 2019, pp. 300–314.

[10] C. Alappat et al., "A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, Jun. 2020.

[11] J. Demmel, M. F. Hoemmen, M. Mohiyuddin, and K. A. Yelick, "Avoiding communication in computing Krylov subspaces," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007–123, Oct. 2007. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007–123.html

[12] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, USA, 2010, Art. no. aAI3413388.

[13] E. Carson, "Communication-avoiding Krylov subspace methods in theory and practice," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug. 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015–179.html

[14] J. Dongarra et al., "With extreme computing, the rules have changed," *Comput. Sci. Eng.*, vol. 19, no. 3, pp. 52–62, 2017.

[15] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–12.

[16] T. Muranushi and J. Makino, "Optimal temporal blocking for stencil computation," *Procedia Comput. Sci.*, vol. 51, pp. 1303–1312, 2015.

[17] J. Morlan, S. Kamil, and A. Fox, "Auto-tuning the matrix powers kernel with sejits," in *Proc. High Perform. Comput. Comput. Sci.*, 2013, pp. 391–403.

[18] M. M. Strout et al., "Generalizing run-time tiling with the loop chain abstraction," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 1136–1145.

[19] M. M. Strout, L. Carter, and J. Ferrante, "Rescheduling for locality in sparse matrix computations," in *Proc. Comput. Sci.*, 2001, pp. 137–146.

[20] D. Huber, M. Schreiber, and M. Schulz, "Graph-based multi-core higher-order time integration of linear autonomous partial differential equations," *J. Comput. Sci.*, vol. 53, 2021, Art. no. 101349.

[21] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, "Improving the performance of CA-GMRES on multicores with multiple GPUs," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 382–391.

[22] I. Yamazaki, S. Rajamanickam, E. G. Boman, M. Hoemmen, M. A. Heroux, and S. Tomov, "Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster," in *Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2014, pp. 933–944.

[23] Distributed GPU based matrix power kernel for geoscience applications, 2021. [Online]. Available: https://doi.org/10.2118/203947-MS

[24] E. Vatai, U. Singhal, and R. Suda, "Diamond matrix powers kernels," in *Proc. Int. Conf. High Perform. Comput.*, 2020, pp. 102–113.

[25] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, 2009.

[26] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 676–687.

[27] H. Wang and A. Chandramowlishwaran, "Pencil: A pipelined algorithm for distributed stencils," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–16.

[28] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, 2019, pp. 300–314.

[29] "Top 500: June 2021 list," [Online]. Available: https://www.top500.org/lists/top500/2021/06/

[30] C. L. Alappat, J. Hofmann, G. Hager, H. Fehske, A. R. Bishop, and G. Wellein, "Understanding HPC benchmark performance on Intel Broadwell and Cascade Lake processors," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., Berlin, Germany: Springer, 2020, pp. 412–433.

[31] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[32] M. A. Heroux and J. Dongarra, "Toward a new metric for ranking high performance computing systems," 2013. [Online]. Available: https://www.osti.gov/biblio/1089988

[33] Andreas Alvermann, "ScaMaC: The scalable matrix collection," 2019. [Online]. Available: https://bitbucket.org/essex/matrixcollection/

[34] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai, "Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes," *IEEE Micro*, vol. 35, no. 4, pp. 6–15, Apr. 2015.

[35] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computations," Research Institute for Advanced Computer Science, Tech. Rep., 1990.

[36] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Towards realistic performance bounds for implicit CFD codes," in *Proceedings of Parallel*. New York, NY, USA: Elsevier, 1999, pp. 233–240.

[37] M. Frigo and V. Strumpen, "The memory behavior of cache oblivious stencil computations," *J. Supercomputing*, vol. 39, no. 2, pp. 93–112, 2007.

[38] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM J. Sci. Comput.*, vol. 37, no. 4, pp. C439–C464, 2015.

[39] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. EC-10, no. 3, pp. 346–365, Sep. 1961.

[40] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, "Parallelization of reordering algorithms for bandwidth and wavefront reduction," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 921–932.

[41] T. M. Malas, G. Hager, H. Ltaief, and D. E. Keyes, "Multidimensional intratile parallelization for memory-starved stencil computations," *ACM Trans. Parallel Comput.*, vol. 4, no. 3, pp. 12:1–12:32, Dec. 2017.

[42] T. Röhl, J. Eitzinger, G. Hager, and G. Wellein, "Validation of hardware events for successful performance pattern identification in high performance computing," in *Tools for High Performance Computing*, A. Knüpfer, T. Hilbrich, C. Niethammer, J. Gracia, W. E. Nagel, and M. M. Resch Eds., Berlin, Germany: Springer, 2016, pp. 17–28.

[43] C. Alappat, "Recursive algebraic coloring engine library version 0.5.0," 2019. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/RRZE-HPC/RACE

[44] SpMP Development Team, "Sparse matrix pre-processing library." Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/IntelLabs/SpMP

[45] "L2 L3 MEM traffic on Intel Skylake SP CascadeLake SP." Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/RRZE-HPC/likwid/wiki/L2-L3-MEM-traffic-on-Intel-Skylake-SP-CascadeLake-SP

[46] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2012, pp. 1–10.

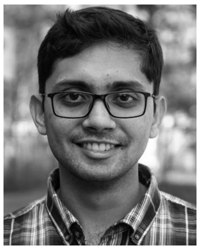
[47] A. Chronopoulos and C. Gear, "S-step iterative methods for symmetric linear systems," *J. Comput. Appl. Math.*, vol. 25, no. 2, pp. 153–168, 1989.

[48] J. A. Loe, H. K. Thornquist, and E. G. Boman, "Polynomial preconditioned GMRES in trilinos: Practical considerations for high-performance computing," 2020. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611976137.4

[49] H. Fehske, J. Schleede, G. Schubert, G. Wellein, V. S. Filinov, and A. R. Bishop, "Numerical approaches to time evolution of complex quantum systems," *Phys. Lett. A*, vol. 373, no. 25, pp. 2182–2188, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0375960109004927

[50] A. Y. Suhov, "An accurate polynomial approximation of exponential integrators," *J. Sci. Comput.*, vol. 60, no. 3, pp. 684–698, 2014.

[51] The Trilinos project team, "The Trilinos project website," 2021. Accessed: Aug. 06, 2021. [Online]. Available: https://trilinos.github.io



Christie Alappat received the master's degree in honors from the Bavarian Graduate School of Computational Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg. He is currently working toward the PhD degree under the guidance of professor Gerhard Wellein. His research interests include performance engineering, sparse matrix and graph algorithms, iterative linear solvers, and eigenvalue computations.

Georg Hager received the doctorate (PhD) and the Habilitation degrees in Computational Physics from the University of Greifswald, Germany. He leads the Training & Support Division with Erlangen National High Performance Computing Center (NHR@FAU) and is an associate lecturer at the Institute of Physics with 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 modeling of out-of-lockstep behavior in large-scale parallel codes.

Olaf Schenk (Senior Member, IEEE) received the Diploma (MSc) degree in mathematics from the University of Karlsruhe, Germany and the doctorate (PhD) degree in electrical engineering and information technology from the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland. He is a full professor with the Institute of Computing within the Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland, where he heads the Advanced Computing Laboratory. His research interests include extreme-scale simulations in computational algorithms, data science, application software, programming, and software tools.

Gerhard Wellein received the Diploma (MSc) degree and the doctorate (PhD) degree in physics from the University of Bayreuth, Germany. He is a professor with the Department of Computer Science with Friedrich-Alexander-Universität Erlangen-Nürnberg and heads the Erlangen National Center for High-Performance Computing (NHR@FAU). His research interests focus on performance modeling and performance engineering, architecture-specific code optimization, and hardware-efficient building blocks for sparse linear algebra and stencil solvers.