A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication

CHRISTIE ALAPPAT, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg
GEORG HAGER, Erlangen Regional Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg
OLAF SCHENK, Institute of Computational Science, Università della Svizzera italiana
JONAS THIES, Simulation and Software Technology, German Aerospace Center
ACHIM BASERMANN, Simulation and Software Technology, German Aerospace Center
ALAN R. BISHOP, Theory, Simulation and Computation, Los Alamos National Laboratory
HOLGER FEHSKE, Institute of Physics, University of Greifswald
GERHARD WELLEIN, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg

The symmetric sparse matrix-vector multiplication (SymmSpMV) is an important building block for many numerical linear algebra kernel operations or graph traversal applications. Parallelizing SymmSpMV on today's multicore platforms with up to 100 cores is difficult due to the need to manage conflicting updates on the result vector. Coloring approaches can be used to solve this problem without data duplication, but existing coloring algorithms do not take load balancing and deep memory hierarchies into account, hampering scalability and full-chip performance. In this work, we propose the recursive algebraic coloring engine (RACE), a novel coloring algorithm and open-source library implementation, which eliminates the shortcomings of previous coloring methods in terms of hardware efficiency and parallelization overhead. We describe the level construction, distance-*k* coloring, and load balancing steps in RACE, use it to parallelize SymmSpMV, and compare its performance on 31 sparse matrices with other state-of-the-art coloring techniques and Intel MKL on two modern multicore processors. RACE outperforms all other approaches substantially and behaves in accordance with the roofline model. Outliers are discussed and analyzed in detail. While we focus on SymmSpMV in this paper, our algorithm and software is applicable to any sparse matrix operation with data dependencies that can be resolved by distance-k coloring.

CCS Concepts: •**Mathematics of computing** → **Graph algorithms;**

Additional Key Words and Phrases: sparse matrix, sparse symmetric matrix-vector multiplication, graph algorithms, graph coloring, scheduling, memory hierarchies

ACM Reference format:
CHRISTIE ALAPPAT, GEORG HAGER, OLAF SCHENK, JONAS THIES, ACHIM BASERMANN, ALAN R. BISHOP, HOLGER FEHSKE, and GERHARD WELLEIN. xxxx. A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication. 1, 1, Article 1 (January xxxx), 39 pages.
DOI:

1 INTRODUCTION AND RELATED WORK

The efficient solution of linear systems or eigenvalue problems involving large sparse matrices has been an active research field in parallel and high performance computing for many decades. Well-known, traditional application areas include quantum physics, quantum chemistry or engineering.

xxxx. XXXX-XXXX/xxxx/1-ART1 $15.00
DOI:

In recent years, new fields such as social graph analysis [41] or spectral clustering in the context of learning algorithms [35, 45] have further increased the need for hardware-efficient, parallel sparse solvers and/or efficient matrix-free solvers. Assuming sufficiently large problems, the solvers are typically based on iterative subspace methods and may include advanced preconditioning techniques. In many methods, two components, sparse matrix-vector multiplication (SpMV) and coloring techniques, are crucial for hardware efficiency and parallel scalability. Typically, these two components are considered to be orthogonal, i.e., hardware efficiency for SpMV is mainly related to data formats and local structures while coloring is used to address dependencies in the enclosing iteration scheme. Interestingly, the hardware-efficient parallelization of symmetric SpMV has not attracted a lot of attention over the years, though symmetry is widespread in the application fields.

The SpMV operation is an essential building block in a number of applications such as algebraic multigrid methods, sparse iterative solvers, shortest path algorithms, breadth first search algorithms, and Markov cluster algorithms, and therefore it is an integral part of numerous scientific algorithms. In the past decades, much research has been focusing on designing new data structures, efficient algorithms, and parallelization techniques for the SpMV operation. Its performance is typically limited by main memory bandwidth. On cache-based architectures, the main factors that influence performance are spatial access locality to the matrix data and temporal locality when reusing the elements of the vectors involved. To address this problem, over the last two decades a plethora of partitioning techniques and data structures to improve SpMV on cache-based architectures have been suggested, including cache-oblivious methods using hypergraph partitioning. One of the first studies on temporal locality optimizations was done by Toledo [43], who investigated Cuthill–McKee (CM) ordering techniques on three-dimensional finite-element test matrices when used in combination with blocking into small dense blocks. Various authors [19, 47] used advanced data storage formats and techniques such as register and cache blocking for SpMV by splitting the matrix into several smaller $p \times q$ sparse submatrices and presented an analytic cache-aware model to determine the optimal block size. These algorithms are, e.g., included in OSKI [46], which is a collection of low-level primitives of tuned sparse kernels for modern cache-based superscalar machines. Kreutzer et al. [26] and Xing et al. [32] used techniques to improve SIMD efficiency and performance on many-core and GPU architectures. Recent work can be found, e.g., in [29–31]. Previous work on SpMV has also focused on reducing communication volume for distributed-memory parallelization, often by using variants of graph or hypergraph partitioning techniques [6]. Yzelman and Bisseling [49, 50] extended hypergraph partitioning techniques in a cache-oblivious method, permuting rows and columns of the input matrix using a recursive hypergraph-based sparse matrix partitioning scheme so that the resulting matrix exhibits cache-friendly behavior during the SpMV.

Despite SpMV being a bandwidth-limited operation, not much work has been done to exploit the symmetry property of symmetric matrices to reduce storage requirements and data transfers by using only the upper/lower triangular part of the matrix. The major challenge here is to resolve the potential write conflicts of explicit symmetric sparse matrix-vector multiplication (SymmSpMV) kernels in parallel processing. There are general solutions for such problems like lock based methods and thread private target arrays [10, 17, 27, 36]. However they have in common that their overhead may increase with the degree of parallelism. Another recent research direction is the use of specialized storage formats like CSB [4], RSB [34], CSX [10] combined with the use of bitmasked register blocking techniques as in [5]. As pointed out by [31] these approaches have drawbacks like missing backward compatibility and matrix conversion costs. Due to these problems there are only a very few standard libraries, like Intel MKL [20], that support primitives for efficient SymmSpMV operation. Another potential way of tackling this inherent data dependency problem is using a distance-2 coloring of the underlying undirected graph, which has not been investigated so far to the best of our knowledge.

Multicoloring (MC) reordering to tackle data dependencies is a very well established strategy in parallelization of iterative solvers. As it is applied to the underlying graph it is not bound to a specific data format and may use existing highly optimized (serial) kernels, i.e., it is orthogonal to general code optimization strategies. Prominent examples for MC in iterative solvers are Gauss-Seidel, incomplete Cholesky factorization or Kaczmarz method [11, 13, 22], where typically a distance-1 or distance-2 coloring is applied subject to the underlying dependencies of the iterative scheme. However, coloring changes the evaluation order of the original solver and may lead to worse convergence rates. This is different when using MC methods for parallelization of SymmSpMV where we only need to ensure that entries of the target vector is not written in parallel. Here we do not require strict serial ordering to get to the same result as in serial processing. In terms of hardware utilization long-standing MC methods often generate colorings which lack efficiency on modern cache-based processors. Studies have been made to increase their performance and improve inherent heuristics; an overview of the methods can be found in [14–16, 33]. However, for irregular and/or large sparse matrices MC may lead to load imbalance, frequent global synchronization, and loss of data locality, severely reducing (single-node) performance. These problems typically become more stringent for higher order distance colorings and larger matrices. The algebraic block multicoloring (ABMC) [21] proposed by Iwashita et al. in 2012 addresses some of these issues as it tries to increase data locality by applying graph partitioning (blocking) before coloring. Beyond the quality of the actual coloring, the time to generate it is also critical, especially for very large problems. Here, widely used and publicly available coloring packages such as COLPACK[16], Kokkos[24] and ZOLTAN[2, 3] speed-up the coloring process itself by parallelization and other heuristics.

Design and implementation of hardware efficient computational kernels can be supported by a structured performance engineering process based on white-box models. On the processor/node level, the most prominent model is the roofline model [48]. Its basic applicability as a reasonable light-speed estimate for SpMV was demonstrated already in [18], including an extension to sparse matrix multiple vector multiplication. The SpMV performance model was refined in [26] with a focus on modeling the performance impact of irregular accesses to the right-hand side (RHS) vector. It has been successfully used to model performance on CPUs and GPGPUs for SpMV kernels [26] and for augmented sparse matrix multiple vector kernels for Chebyshev filter diagonalization [25]. However, there is no extension towards explicit SymmSpMV, which shows increased computational intensity and irregular accesses to both involved vectors. Typically the expectation is that SymmSpMV should be approximately twice as fast as SpMV as only half of the matrix information needs to be stored and accessed.

Finally, there is a clear hardware trend towards processors with advanced vector-style processing, higher core counts and more complex cache hierarchies. Also, attainable bandwidth may increase even for "standard" CPU based systems through the use of high bandwidth memory solutions at the cost of very restricted memory sizes. A first step into this direction was the Intel Xeon Knights Landing processor. The specification of the ARM-based Fujitsu A64FX processor (to be used in the Post-K computer) may provide another blueprint for future processor configurations [12]: A 48-core processor supporting 512-bit SIMD execution units on top of 32 GiB HBM2 main memory, which provides a bandwidth of 1 TB/s. It is obvious that such hardware trends call for revisiting existing, time-critical components in simulation codes both in terms of scalability and hardware efficiency. Moreover, the potential of SymmSpMV to substantially reduce the memory footprint of sparse solvers needs to be exploited to meet the constraint of very limited memory space.

Contribution and Outline

This paper addresses the general problem of generating hardware efficient distance-k coloring of undirected graphs for modern multicore processors. As an application we choose parallelization of the SymmSpMV operation. We cover thread-level parallelization and focus on a single multicore processor. The main contributions can be summarized as follows:

- A new recursive algebraic coloring scheme (RACE) is proposed, which generates hardware efficient distance-k colorings of undirected graphs. Special emphasis in the design of RACE is put on achieving data locality, generating levels of parallelism matching the core count of the underlying multicore processor and load balancing for shared memory parallelization.
- We propose shared memory parallelization of SymmSpMV using a distance-2 coloring of the underlying undirected graph to avoid write conflicts and apply RACE for generating the colorings.
- A comprehensive performance study of shared memory parallel SymmSpMV using RACE demonstrates the benefit of our approach. Performance modeling is deployed to substantiate our performance measurements, and a comparison to existing coloring methods as well a vendor optimized library (Intel MKL) are presented. The broad applicability and the sustainability is validated by using a wide set of 31 test matrices and two very different generations of Intel Xeon processors.
- We extend the existing proven SpMV performance modeling approach to the SymmSpMV kernel. In the course of the performance analysis we further demonstrate why in some cases the ideal speedup may not be achievable.

We have implemented our graph coloring algorithms in the open source library recursive algebraic coloring engine (RACE).[1] Information required to reproduce the performance numbers provided in this paper is also available.[2]

This paper is organized as follows: Our software and hardware environment as well as the benchmark matrices are introduced in Section 2. In Section 3 we describe the properties of the SpMV and SymmSpMV kernels, including roofline performance limits, and motivate the need for an advanced coloring scheme. In Section 4 we detail the steps of the RACE algorithm via an artificial stencil matrix and show how recursive level group construction and coloring can be leveraged to exploit a desired level of parallelism for distance-k dependencies. The interaction between the parameters of the method and their impact on the parallel efficiency is studied in Section 5. Section 6 presents performance data for SymmSpMV for a wide range of matrices on two different multicore systems, comparing RACE with ABMC and MC as well as Intel MKL, and also shows the efficiency of RACE as defined by the roofline model yardstick. Section 7 concludes the paper and gives an outlook to future work.

2 HARDWARE AND SOFTWARE ENVIRONMENT

2.1 Hardware test bed

We conducted all benchmarks on a single CPU socket from Intel's Ivy Bridge EP and Skylake SP families, respectively, since these represent the oldest and the latest Intel architectures in active use within the scientific community at the time of writing:

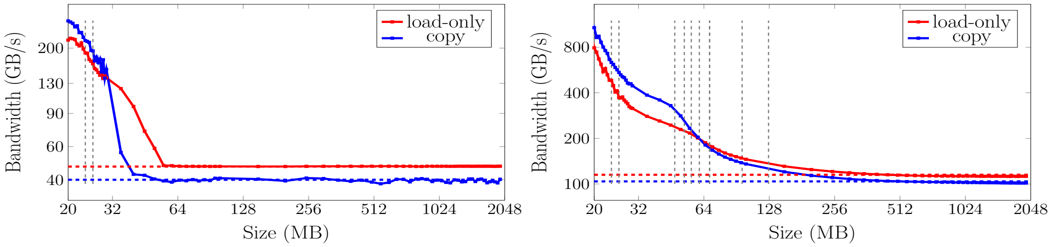
- The Intel Ivy Bridge EP architecture belongs to the class of "classic" designs with three inclusive cache levels. While the L1 and L2 caches are private to each core, the L3 cache is

[1] http://tiny.cc/RACElib

[2] http://tiny.cc/RACElib-AD

Table 1. Technical details (per socket) of the Intel CPUs used for the benchmarks.

Model name	Xeon® E5-2660	Xeon® Gold 6148
Microarchitecture	Ivy Bridge EP	Skylake SP
Base clock frequency	2.2 GHz	2.4 GHz
Uncore clock frequency	2.2 GHz	2.4 GHz
Physical cores per socket	10	20
L1D cache	10 × 32 KiB	20 × 32 KiB
L2 cache	10 × 256 KiB	20 × 1 MiB
L3 cache	25 MiB	27.5 MiB
L3 type	inclusive	noninclusive, victim
Main memory	32 GiB	48 GiB
Bandwidth per socket, load-only	47 GB/s	115 GB/s
Bandwidth per socket, copy	40 GB/s	104 GB/s



(a) *Ivy Bridge EP*

(b) *Skylake SP*

Fig. 1. Attained bandwidth versus total data size for a range from 20 MB to 2 GB. The dotted lines show the asymptotic bandwidth given in Table 1 for the load-only and copy benchmark. The benchmarks were performed on the full socket using the `likwid-bench` tool. The gray vertical lines correspond to the positions of matrices that might show caching effects; see Section 2.3. Note the logarithmic scales.

shared but scalable in terms of bandwidth. The processor supports the AVX instruction set extension, which is capable of 256-bit wide SIMD execution.

- Contrary to its predecessors, the Intel Skylake SP architecture has a shared but noninclusive victim L3 cache and much larger private L2 caches. The model we use in this work supports AVX-512, which features 512-bit wide SIMD execution.

Architectural details along with the attainable memory bandwidths are given in Table 1. All the measurements were made with CPU clock speeds fixed at the indicated base frequencies. Note that for the Skylake SP architecture the clock frequency is scaled down internally to 2.2 GHz when using multicore support and the AVX-512 instruction set; however, this is of minor importance for the algorithms discussed here.

As the attainable main memory bandwidth is the input parameter to the roofline model used later, we have carefully measured this value depending on the data set size for two access patterns (copy and load-only). The data presented in Figure 1 basically show the characteristic performance drop if the data set size is too large to fit into the last level cache (LLC), which is an L3 cache on both architectures (cf. Table 1 for the actual sizes). Interestingly there is no sharp drop at the exact size of the LLC but a rather steady performance decrease with enhanced data access rates also for data set sizes up to twice the LLC size on Ivy Bridge EP. For Skylake SP this effect is even more pronounced as the noninclusive victim L3 cache architecture only stores data which are not in the L2 cache; thus the available cache size for an application may be the aggregate sizes of the L2 and L3 caches on this architecture. The final bandwidth for the roofline model is chosen as the asymptotic value depicted in Figure 1. Of course caching effects are extremely sensitive to the data access pattern and thus the values presented here only provide simple upper bounds for the SymmSpMV kernel with its potentially strong irregular data access.

2.2 External Tools and Software

The LIKWID [44] tool suite in version 4.3.2 was used, specifically `likwid-bench` for bandwidth benchmarks (see Table 1), and `likwid-perfctr` for counting hardware events and measuring derived metrics. LIKWID validates the quality of it's performance metrics and validation data is publicly available.[3] Overall the LIKWID data traffic measurements can be considered as highly accurate. Only the L3 data traffic measurement on Skylake SP fails the quantitative validation but it still provides good qualitative results.[4]

For coloring we used the COLPACK [16] library and METIS [23] version 5.1.0 for graph partitioning with the ABMC method. The Intel SpMP [42] library was employed for RCM bandwidth reduction, and the Intel MKL version 19.0.2 for some reference computations and comparisons.

All code was compiled with the Intel compiler in version 19.0.2 and the following compiler flags: `-fno-alias -xHost -O3` for Ivy Bridge EP and `-fno-alias -xCORE-AVX512 -O3` for Skylake SP.

2.3 Benchmark Matrices

Most test matrices were taken from the SuiteSparse Matrix Collection (formerly University of Florida Sparse Matrix Collection) [8] combining sets from two related papers [34, 38], which allows the reader to make a straightforward comparison of results. We also added some matrices from the Scalable Matrix Collection (ScaMaC) library[1], which allows for scalable generation of large matrices related to quantum physics applications. A brief description of the background of these matrices can be found in ScaMaC documentation.[5] All the matrices considered are real, although our underlying software would also support complex matrices. As mentioned before, we restrict ourselves to matrices representing fully connected undirected graphs. Table 2 gives an overview of the most important matrix properties like number of rows (N_r), total number of nonzeros (N_nz), average number of nonzeros per row (N_nzr), along with the bandwidth of the matrix without (bw) and with (bw_{RCM}) RCM preprocessing.

Due to the extended cache size as seen in Figure 1 it might happen that some of the matrices attain higher effective bandwidths due to partial/full caching, especially on Skylake SP. The ten potential candidates for the Skylake SP chip in terms of symmetric and full storage ($<$ 128 MB) are marked with an asterisk in Table 2, while only two among these (`offshore` and `parabolic_fem`) satisfy the criteria for Ivy Bridge EP ($<$ 40 MB). The corresponding data set size for storing the upper triangular part of these matrices have been labeled in Figure 1.

3 KERNELS

We evaluate our methods by parallelization of the SymmSpMV kernel using distance-2 coloring, which avoids concurrent updates of the same vector entries by different threads.

[3] https://github.com/RRZE-HPC/likwid/wiki/TestAccuracy

[4] https://github.com/RRZE-HPC/likwid/wiki/L2-L3-MEM-traffic-on-Intel-Skylake-SP-CascadeLake-SP

[5] https://alvbit.bitbucket.io/scamac_docs/_matrices_page.html

Table 2. Details of the benchmark matrices. N_r is the number of matrix rows and N_{nz} is the number of nonzeros. $N_{nzr} = N_{nz}/N_r$ is the average number of nonzeros per row. bw and bw_{RCM} refer to the matrix bandwidth without and with RCM preprocessing. The letter "C" in the parentheses of the matrix name indicates a corner case matrix that will be discussed in detail, while the letter "Q" marks a matrix from quantum physics that is not part of the SuiteSparse Matrix Collection. With an asterisk (*) we have labeled all the matrices which are less than 128 MB, which could potentially lead to some caching effects especially on the Skylake SP architecture.

Index	Matrix name	N_r	N_{nz}	N_{nzr}	bw	bw_{RCM}
1	crankseg_1* (C)	52,804	10,614,210	201.01	50,388	5126
2	ship_003*	121,728	8,086,034	66.43	3659	3833
3	pwtk*	217,918	11,634,424	53.39	189,331	2029
4	offshore*	259,789	4,242,673	16.33	237,738	19,534
5	F1	343,791	26,837,113	78.06	343,754	10,052
6	inline_1 (C)	503,712	36,816,342	73.09	502,403	6002
7	parabolic_fem* (C)	525,825	3,674,625	6.99	525,820	514
8	gsm_106857*	589,446	21,758,924	36.91	588,744	17,865
9	Fault_639	638,802	28,614,564	44.79	19,988	19,487
10	Hubbard-12* (Q)	853,776	11,098,164	13.00	232,848	38,780
11	Emilia_923	923,136	41,005,206	44.42	17,279	14,672
12	audikw_1	943,695	77,651,847	82.29	925,946	35,084
13	bone010	986,703	71,666,325	72.63	13,016	14,540
14	dielFilterV3real	1,102,824	89,306,020	80.98	1,036,475	25,637
15	thermal2*	1,228,045	8,580,313	6.99	1,226,000	797
16	Serena	1,391,349	64,531,701	46.38	81,578	84,947
17	Geo_1438	1,437,960	63,156,690	43.92	26,018	30,623
18	Hook_1498	1,498,023	60,917,445	40.67	29,036	28,994
19	Flan_1565	1,564,794	117,406,044	75.03	20,702	20,849
20	G3_circuit*	1,585,478	7,660,826	4.83	947,128	5068
21	Anderson-16.5* (Q)	2,097,152	14,680,064	7.00	1,198,372	24,620
22	FreeBosonChain-18 (Q)	3,124,550	38,936,700	12.46	2,042,975	131,749
23	nlpkkt120	3,542,400	96,845,792	27.34	1,814,521	86,876
24	channel-500x100x100-b050	4,802,000	90,164,744	18.78	600,299	23,766
25	HPCG-192	7,077,888	189,119,224	26.72	37,057	110,017
26	FreeFermionChain-26 (Q)	10,400,600	140,616,112	13.52	5,490,811	434,345
27	Spin-26 (Q)	10,400,600	145,608,400	14.00	709,995	211,828
28	Hubbard-14 (Q)	11,778,624	176,675,928	15.00	3,171,168	425,415
29	nlpkkt200	16,240,000	448,225,632	27.60	8,240,201	240,796
30	delaunay_n24	16,777,216	100,663,202	6.00	16,769,102	32,837
31	Graphene-4096 (C,Q)	16,777,216	218,013,704	13.00	4098	6145

Since the kernel is closely related to the sparse matrix-vector multiplication (SpMV) kernel by structure and computational intensity, we start with a discussion of SpMV and extend it towards SymmSpMV later. In all cases the aim is to derive realistic upper performance bounds, which can be estimated once the computational intensity and main memory bandwidth (b_s; see Table 1) are

Algorithm 1 SpMV using the CRS format: $b = Ax$

```
$double :: A[nnz], b[nrows], x[nrows]$
$integer :: col[nnz], rowPtr[nrows + 1], tmp$
for $row = 1 : nrows$ do
    $tmp = 0$
    for $idx = rowPtr[row] : (rowPtr[row + 1] - 1)$ do
        $tmp += A[idx] * x[col[idx]]$
    end for
    $b[row] = tmp$
end for
```

known [48], i.e.,

$$P_{\text{kernel}} = I_{\text{kernel}} \times b_S. \tag{1}$$

Since b_S depends on the ratio of load to store streams we present the model for both upper (load-only) and lower bound (copy) bandwidth cases. In the following we choose the compressed row storage (CRS) format for the implementation of SpMV as well as SymmSpMV and assume symmetric matrices.

3.1 SpMV

A baseline SpMV kernel is presented in Algorithm 1. It has no loop-carried dependencies, so parallelization of the outer loop using, e.g., OpenMP, is straightforward. Following the discussion in [26], its computational intensity is

$$I_{\text{SpMV}}(\alpha) = \frac{2}{8 + 4 + 8\alpha + 20/N_{\text{nzr}}} \frac{\text{flops}}{\text{bytes}}. \tag{2}$$

Here we assume that the matrix data ($A[], col[]$), the left-hand side (LHS) vector ($b[]$), and the row pointer information ($rowPtr[]$) are loaded only once from main memory, since these data structures are consecutively accessed. The intensity is calculated from the average cost of performing all computations required for one nonzero element of the matrix. Thus, contributions which are independent of the inner (short) loop are rescaled by N_{nzr}, which is the average number of nonzeros per row (i.e., the average length of the inner loop).

The 8α term quantifies the data traffic caused by accessing the RHS vector ($x[]$). The value of α depends on the matrix structure as well as on the RHS vector data set size and the available cache size. The minimum value of $\alpha = {N_{\text{nzr}}}^{-1}$ is attained if the RHS vector is only loaded once from main memory to the cache and all subsequent accesses in the same SpMV are cache hits. This limit is typically observed for matrices with low bandwidth (high access locality) or if the cache is large enough to hold the full RHS data during one SpMV. The actual value of α can be determined experimentally by measuring the data traffic when executing the SpMV; see [26] for more details.[6] The optimal value of $\alpha = {N_{\text{nzr}}}^{-1}$ together with the corresponding computational intensities for all matrices is shown in Table 3. The measured α_{SpMV} is used as a sensible lower bound for $\alpha_{SymmSpMV}$ values (see Section 3.2) in cases where advanced cache replacement strategies do not apply; therefore the table also presents the corresponding measured α_{SpMV} (= assumed $\alpha_{SymmSpMV}$) values for different matrices. Choosing the matrices 10, 22, and 31, which have approximately the same optimal α_{SpMV}, one can study the delicate influence of matrix structure (i.e., matrix bandwidth and

[6] In [26] the traffic for the row pointer was not accounted for, i.e., the denominator in (2) is larger by $\frac{4}{N_{\text{nzr}}}$ bytes. This error is only significant when N_{nzr} is small.

Table 3. The optimal value of α_{SpMV} is shown in column three. Following Equation (1) the maximum SpMV performance can be calculated for each architecture using the best intensity values ($I_{SpMV}(\alpha_{SpMV})$ in $\frac{\text{flops}}{\text{bytes}}$) shown in the fourth column. The assumed $\alpha_{SymmSpMV}$ on Skylake SP and Ivy Bridge EP architectures are presented in columns five and six, respectively. The assumed $\alpha_{SymmSpMV}$ is equal to the measured α_{SpMV} for all matrices except the ones marked with asterisk, where $\alpha_{SymmSpMV}$ is set to optimal $\alpha_{SymmSpMV}$ (= $1/N_{\text{nzr}}^{\text{symm}}$).

Index	Matrix name	α_{SpMV}	$I_{SpMV}(\alpha_{SpMV})$	Assumed $\alpha_{SymmSpMV}$	
		Optimal	Optimal	SKX	IVB
1	crankseg_1	0.0050	0.1648	0.0099*	0.0179
2	ship_003	0.0151	0.1610	0.0297*	0.0390
3	pwtk	0.0187	0.1597	0.0368*	0.0383
4	offshore	0.0612	0.1458	0.1154*	0.1058
5	F1	0.0128	0.1618	0.0253*	0.0436
6	inline_1	0.0137	0.1615	0.0137	0.0340
7	parabolic_fem	0.1431	0.1249	0.2504*	0.2250
8	gsm_106857	0.0271	0.1568	0.0528*	0.0946
9	Fault_639	0.0223	0.1584	0.0453	0.0861
10	Hubbard-12	0.0769	0.1413	0.1429*	0.2318
11	Emilia_923	0.0225	0.1583	0.0827	0.0855
12	audikw_1	0.0122	0.1621	0.0624	0.0638
13	bone010	0.0138	0.1615	0.0492	0.0523
14	dielFilterV3real	0.0123	0.1620	0.0728	0.0675
15	thermal2	0.1431	0.1249	0.2504*	0.2277
16	Serena	0.0216	0.1587	0.1006	0.1156
17	Geo_1438	0.0228	0.1583	0.0896	0.0917
18	Hook_1498	0.0246	0.1576	0.1031	0.0948
19	Flan_1565	0.0133	0.1616	0.0541	0.0525
20	G3_circuit	0.2070	0.1124	0.3429*	0.3360
21	Anderson-16.5	0.1429	0.1250	0.3634	0.3187
22	FreeBosonChain-18	0.0802	0.1404	0.2708	0.2628
23	nlpkkt120	0.0366	0.1536	0.1600	0.1656
24	channel-500x100x100-b050	0.0533	0.1482	0.1735	0.1339
25	HPCG-192	0.0374	0.1533	0.1358	0.1391
26	FreeFermionChain-26	0.0740	0.1421	0.3879	0.3973
27	Spin-26	0.0714	0.1429	0.3670	0.3518
28	Hubbard-14	0.0667	0.1442	0.3575	0.3598
29	nlpkkt200	0.0362	0.1537	0.1669	0.1720
30	delaunay_n24	0.1667	0.1200	0.4065	0.3192
31	Graphene-4096	0.0770	0.1413	0.1604	0.1278

number of rows; see Table 2) and the cache size on the actual data traffic, i.e., the measured values of α_{SpMV}.

For most of the ten candidate matrices on the Skylake SP architecture that could potentially show a caching effect (see Table 2) we observe the measured α_{SpMV} to be lower than optimal. In this case

Algorithm 2 SymmSpMV $b = Ax$, where A is an upper triangular matrix

```
for $row = 1 : nrows$ do
    $diag\_idx = rowPtr[row]$
    $b[row] += A[diag\_idx] * x[row]$
    $tmp = 0$
    for $idx = (rowPtr[row] + 1) : (rowPtr[row + 1] - 1)$ do
        $tmp += A[idx] * x[col[idx]]$
        $b[col[idx]] += A[idx] * x[row]$
    end for
    $b[row] += tmp$
end for
```

we set their $\alpha_{SymmSpMV}$ values to the optimal alpha value of SymmSpMV ($\alpha_{SymmSpMV}$) which will be defined in the following Section 3.2. These cases are marked with an asterisk in Table 3.

3.2 SymmSpMV

SymmSpMV exploits the symmetry of the matrix ($A_{ij} = A_{ji}$) to reduce storage size for matrix data and reduce the overall memory traffic by operating on the upper (or lower) half of the matrix only. Thus for every nonzero matrix entry we need to update two entries in the LHS vector ($b[]$) as shown in Algorithm 2.

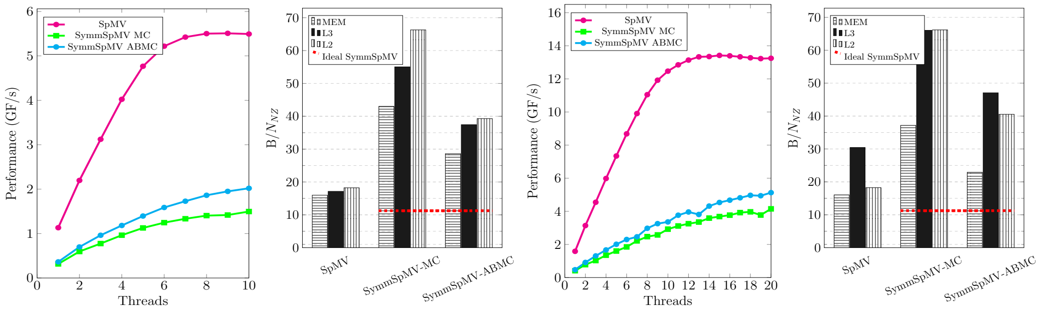
In line with the discussion above, the computational intensity of SymmSpMV is

$$I_{\mathrm{SymmSpMV}}(\alpha) = \frac{4}{8 + 4 + 24\alpha + 4/N_{\mathrm{nzr}}^{\mathrm{symm}}} \frac{\mathrm{flops}}{\mathrm{bytes}} \,, \tag{3}$$

$$\text{where } N_{\mathrm{nzr}}^{\mathrm{symm}} = (N_{\mathrm{nzr}} - 1)/2 + 1 \,. \tag{4}$$

For a given nonzero matrix element 4 flops are performed, which is twice the amount of work than in SpMV. In addition we have indirect access to the LHS vector (read and write) which triples the traffic contribution quantified by α. The only term scaled with $N_{\mathrm{nzr}}^{\mathrm{symm}}$ (number of nonzeros per row in the upper triangular part of the matrix) is the row pointer. The most optimistic value of α ($\alpha_{SymmSpMV}$) in this case is $1/N_{\mathrm{nzr}}^{\mathrm{symm}}$, which corresponds to a one time transfer of the LHS and RHS vectors. Note that the α for SpMV and SymmSpMV may be different even for the same matrix and the same compute device, as in the latter case the two vectors are accessed irregularly and compete for cache. Thus we can assume that the α value measured for SpMV (α_{SpMV}) is a lower bound for SymmSpMV. Table 3 show the assumed $\alpha_{SymmSpMV}$ values taken for performance modeling. Since an upper bound for the performance is the product of computational intensity and main memory bandwidth (see Equation (1)), this approach provides an upper performance bound for SymmSpMV. However, note that the performance models derived for matrices having caching effects (see Table 3) need not be strictly upper bound, as they heavily depends on the caching strategy of the underlying architecture.

Comparing Equation (3) and Equation (2) it is obvious that the perfect speedup of 2× when using SymmSpMV instead of SpMV is only attainable in the limit of small α. Considering the large prefactor of the α contribution, any implementation of SymmSpMV must aim at ensuring high data locality. The indirect update of the LHS also has a large impact on parallelization strategy as two rows which have a nonzero in the same column cannot be computed in parallel. In a graph-based approach to this problem, this is equivalent to the constraint that only vertices which have at least distance two can be computed in parallel.



(a) SymmSpMV (b) Data traffic (c) SymmSpMV (d) Data traffic

Fig. 2. Scaling performance of SymmSpMV with MC and ABMC compared to SpMV on one socket of Ivy Bridge EP and Skylake SP is shown in Figures 2(a) and 2(c) respectively. Figures 2(b) and 2(d) show average data traffic per nonzero entry (N_{nz}) of the full matrix as measured with LIKWID for all cache levels and main memory on full socket of Ivy Bridge EP and Skylake SP respectively. The Spin-26 matrix was prepermuted with RCM.

3.3 Analysis of the SymmSpMV kernel using parallel coloring schemes for the Spin-26 matrix

As pointed out previously, the parallelization of the SymmSpMV kernel can be done via distance-2 coloring of the corresponding graph. The computational intensity, and hence the performance, depends on the data access patterns to the LHS and RHS vectors. As coloring schemes change those patterns, they may change the computational intensity, and we have to investigate this effect in more detail. We apply the basic MC scheme generated by COLPACK [16] to parallelize SymmSpMV and compare it with SpMV, which serves as our performance yardstick. Note that any required preprocessing is excluded from the timings. In Figure 2 we present performance and data transfer volumes for the *Spin-26* matrix on a single socket of the Ivy Bridge EP and Skylake SP systems. For SpMV we recover the well-known memory bandwidth saturation pattern as we fill the chip (Figures 2(a) and 2(c)). Measuring the actual data volume from main memory using LIKWID we find 16.24 and 16.36 bytes per nonzero matrix entry (Figures 2(b) and 2(d)) on Ivy Bridge EP and Skylake SP architectures. This corresponds to the denominator of I_{SpMV} in Equation (2), so we can determine $\alpha_{\mathrm{SpMV}} = 0.351$ for Ivy Bridge EP and 0.367 for Skylake SP, thus we can calculate an optimistic bound for the intensity of SymmSpMV according to Equation (3). Using the copy and the load-only bandwidth of Ivy Bridge EP (see Table 1) in Equation (1) we find a maximum attainable SymmSpMV performance range for this matrix of $P_{\mathrm{SymmSpMV}} = 7.63, \ldots, 8.96$ GF/s, while for Skylake SP we expect $P_{\mathrm{SymmSpMV}} = 19.49, \ldots, 21.55$ GF/s . This indicates a possible speedup of approximately 1.4× − 1.6× compared to the SpMV baseline (5.5 GF/s and 13.41 GF/s on Ivy Bridge EP and Skylake SP), the SymmSpMV implementation using MC falls short of this expectation and is more than three times slower than SpMV.

The reason for this decrease is the nature of the MC permutation. For distance-2 coloring, sets of structurally orthogonal rows have to be determined [15], i.e., rows that do not overlap in any column entry. These sets are referred to as colors. Figure 3 shows the corresponding permutation and the obtained sets of colors applied to a toy problem with high data locality. Different rows of the same color can be executed in parallel, but colors are operated one after the other. After MC a color may contain rows from very distant parts of the matrix, potentially destroying data locality. Assuming that the LLC can hold a maximum of six elements, we find that the RHS vector must

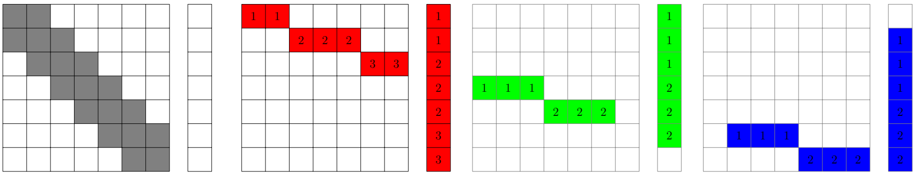


Fig. 3. Illustration of the increase of α by MC. Numbers represent thread ids. Note that this figure shows only rows of the matrix permuted according to MC, but in practice one would permute both rows and columns.

be loaded every time for each new color. This is the reason why we observe 3× more bytes per nonzero for SymmSpMV with MC compared to SpMV, as seen in Figures 2(b) and 2(d). However, our performance model indicates that SymmSpMV should exhibit only 0.7× the data traffic of SpMV (see red dotted line in Figures 2(b) and 2(d)). Of course this effect strongly depends on the matrix structure, the matrix size, and the cache size.

ABMC[21] tries to preserve data locality by first partitioning the entire matrix into blocks of specified size and then applying MC to these blocks. Threads then work in parallel between blocks of the same color. Along the lines of [39] we use METIS [23] to partition the matrix into blocks, and COLPACK for MC. The size of blocks for ABMC is determined by a parameter scan (range 4 . . . 128; see [21]). As stated above, the timing for the performance measurements excludes preprocessing and the parameter search. This method reduces the data traffic (see Figures 2(b) and 2(d)) as there is better data locality within a block. Consequently, the performance improves over plain MC (see Figures 2(a) and 2(c)). However, we are far off the performance model prediction. In addition to data locality, other factors like global synchronizations and false sharing also contribute to this failure. These effects strongly depend on the number of colors and in general increase with chromatic number. In the case of the Spin-26 matrix the overhead of synchronization is roughly 10% for the MC method. For most of the matrices considered in this work one can also observe a strong positive correlation between false sharing and the number of threads for SymmSpMV kernels due to the indirect writes in SymmSpMV.

4 RECURSIVE ALGEBRAIC COLORING ENGINE (RACE)

Our advanced coloring algorithm is based on three steps:

(1) level construction,
(2) distance-k coloring,
(3) load balancing.

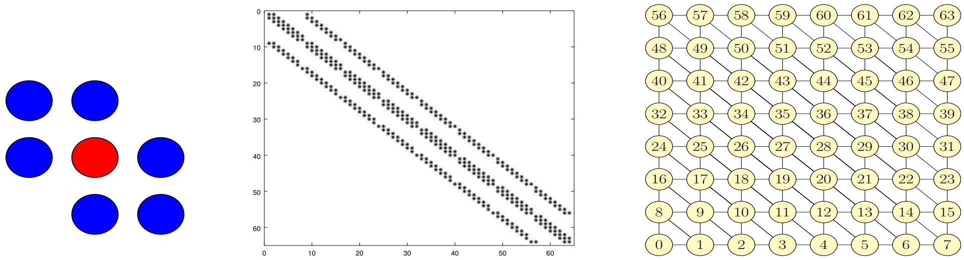
In the first step we apply a bandwidth reduction algorithm including level construction and matrix reordering. We then use the information from the level construction step to form subsets of levels which allow for hardware efficient distance-k coloring of the graph. Finally we present a concept to ensure load balancing between threads. These steps are applied recursively if required.

To illustrate the method we choose a simple matrix which is associated with an artificially constructed two-dimensional stencil as shown in Figure 4(a). The corresponding sparsity pattern and the graph of the matrix are shown in Figures 4(b) and 4(c) respectively.

Definitions

We need the following definitions from graph theory:

- **Graph:** $G = (V, E)$ represents a graph, with $V(G)$ denoting its set of vertices and $E(G)$ denoting its edges. Note that we restrict ourselves to irreducible undirected graphs.



(a) Stencil example (b) Sparsity pattern (c) Graph

Fig. 4. (a) Structure of an artificially designed stencil, (b) corresponding sparsity pattern of its matrix representation on an 8 × 8 lattice with Dirichlet boundary conditions, and (c) the graph representation of the matrix. The stencil structure was chosen for illustration purposes and does not represent any specific application scenario.

- **Neighborhood:** $N(u)$ is the neighborhood of a vertex u and is defined as

$$N(u) = \{\, v \in V(G) : (u, v) \in E(G) \,\} \ .$$

- ***k*th Neighborhood:** $N^k(u)$ of a vertex u is defined as

$$\begin{aligned} N^2(u) &= N(N(u)) \\ N^3(u) &= N^2(N(u)) \\ &\vdots \\ N^k(u) &= N^{k-1}(N(u)) \ . \end{aligned}$$

- **Subgraph:** In this paper a subgraph H of G specifically refers to the subgraph induced by vertices $V' \subseteq V(G)$ and is defined as

$$H = (V', \{\, (u, v) : (u, v) \in E(G) \text{ and } u, v \in V' \,\}) \ .$$

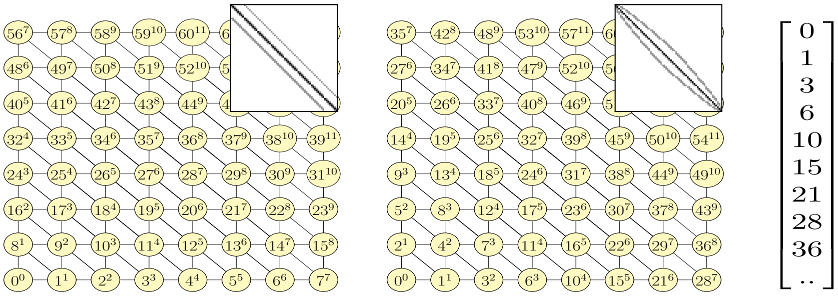
4.1 Level Construction

The first step of RACE is to determine different *levels* in the graph and permute the graph data structure. This we achieve using well-known bandwidth reduction algorithms such as reverse Cuthill McKee (RCM) [7] or breadth-first search (BFS) [28]. Although the RCM method is also implemented in RACE, we use the BFS reordering in the following for simpler illustration.

First we choose a *root* vertex and assign it to the first level, $L(0)$. For $i > 0$, level $L(i)$ is defined to contain vertices that are in the neighborhood of vertices in $L(i-1)$ but not in the neighborhood of vertices in $L(i-2)$ [9], i.e.,

$$L(i) = \begin{cases} root & \text{if } i = 0, \\ u : u \in N(L(i-1)) & \text{if } i = 1, \\ u : u \in N(L(i-1)) \cap \overline{N(L(i-2))} & \text{otherwise.} \end{cases} \tag{5}$$

From Equation (5) one finds that the ith level consists of all vertices that have a minimum distance i from the root node. Algorithm 3 shows how to determine this distance and thus set up the levels $L(i)$. We refer to the total number of levels obtained for a particular graph as N_ℓ. Figure 5(a) shows



(a) Level construction (b) Permuted graph (G') (c)

Fig. 5. (a) Levels of the original graph and (b) the permuted graph for the stencil example. Insets show the corresponding sparsity patterns. (c) Shows the entries of the `level_ptr` array associated with G'.

the N_ℓ=14 levels of our artificial stencil operator, where the index of each vertex (v) is the vertex number and the superscript represents the level number, i.e.,

$$v^i \implies v \in L(i)\ . \tag{6}$$

Note that the $L(i)$ are substantially different from the levels used in the "level-scheduling" [40] approach, which applies "depth first search."

After the levels have been determined, the matrix is permuted in the order of its levels, such that the vertices in $L(i)$ are stored consecutively and appear before those of $L(i+1)$. Figure 5 shows the graph ($G' = P(G)$) of the stencil example after applying this permutation (P) and demonstrates the enhanced spatial locality of the vertices within and between levels (see Figure 5(b)) as compared to the original (lexicographic) numbering (see Figure 5(a)). Until now the procedure is the same as BFS (or RCM).

As RACE uses information about the levels for resolving dependencies in the coloring step, we store the index of the entry point to each level in the permuted data structure (of G') in an array `level_ptr`$[0 : N_\ell]$, so that levels on G' can be identified as

$$L(i) = \{\, u : u \in [\texttt{level\_ptr}[i] : (\texttt{level\_ptr}[i+1] - 1)] \text{ and } u \in V(G')\,\}\ .$$

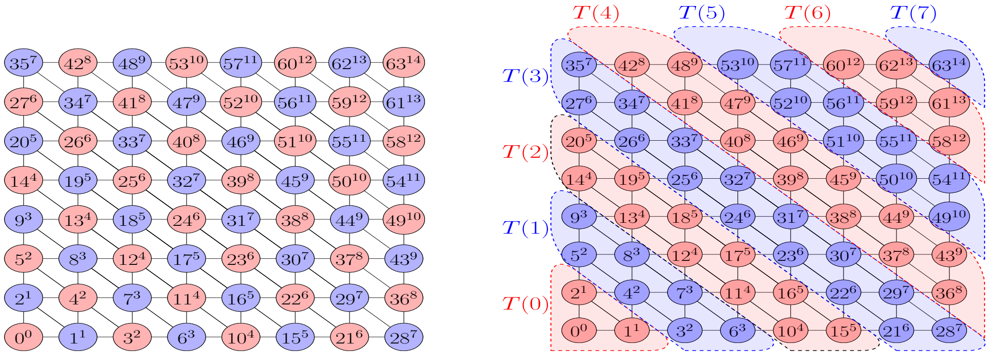
The entries of `level_ptr` for the stencil example are shown in Figure 5(c).

4.2 Distance-k coloring

The data structure generated above serves as the basis for our distance-k coloring procedure as it contains information about the neighborhood relation between the vertices of any two levels. Following the definition in [15], two vertices are called distance-k neighbors if the shortest path connecting them consists of at most k edges. This implies that u is a distance-k neighbor of v (referred to as $u \xrightarrow{k} v$) if

$$u \xrightarrow{k} v \iff v \in \{\, u \cup N(u) \cup N^2(u) \cup \cdots N^k(u)\,\}\ . \tag{7}$$

For the undirected graphs as used here, $u \xrightarrow{k} v$ also implies $v \xrightarrow{k} u$. Based on this definition we consider two vertices to be distance-k independent if they are not distance-k neighbors, and two levels are said to be distance-k independent if their vertices are mutually distance-k independent. Thus, levels $L(i)$ and $L(i \pm (k+j))$ of the permuted graph G' are distance-k independent for all $j \geq 1$,



(a) Distance-1 independent level groups

(b) Distance-2 independent level groups

Fig. 6. Forming distance-1 and distance-2 independent level groups for the stencil example.

denoted as

$$L(i) \overset{k}{\not\to} L(i \pm (k+j)) \forall j \geq 1 \, . \tag{8}$$

Equation (8) implies that if there is a gap of at least one level between any two levels (e.g., $L(i)$ and $L(i+2)$) all pairs of vertices between these two levels are distance-1 independent. Similarly if the gap consists of at least two levels between any two levels (e.g., $L(i)$ and $L(i+3)$) we have distance-2 independent levels, and so on.

The definition used in Equation (8) offers many choices for forming distance-k independent sets of vertices, which can then be executed in parallel. In Figure 6 we present one example each for distance-1 (Figure 6(a)) and distance-2 (Figure 6(b)) colorings of our stencil example. The distance-1 coloring uses a straightforward approach by assigning two colors to alternating levels, i.e., levels of a color can be calculated concurrently. In case of distance-2 independence we do not use three colors but rather aggregate two adjacent levels to form a *level group* (denoted by $T(i)$) and perform a distance-1 coloring on top of those groups. This guarantees that vertices of two level groups of the same color are distance-2 independent and can be executed in parallel. Here, the vertices in $T(0)$, $T(2)$, $T(4)$, and $T(6)$ can be operated on by four threads in parallel, i.e., one thread per level group. After synchronization the remaining four blue level groups can also be executed in parallel. This idea can be generalized such that for distance-k coloring, each level group contains k adjacent levels. Thus formed level groups are then distance-1 colored. Then, all level groups within a color can be executed in parallel. This simple approach allows one to generate workload for a maximum of $N_\ell/2k$ threads if distance-k coloring is requested.[7] Note that in all cases, vertices within a single level group are computed in their original order, which allows for good spatial access locality.

Choosing the same number of levels for each level group may, however, cause severe load imbalance depending on the matrix structure. In particular, the use of bandwidth reduction schemes such as BFS or RCM will further worsen this problem due to the lenslike shape of the reordered matrix (see inset of Figure 5(b)), leading to low workload for level groups containing the top and bottom rows of the matrix. Compare, e.g., $T(0)$ and $T(7)$ with $T(3)$ and $T(4)$ in Figure 6(b). However, Equation (8) does not require *exactly* k levels to be in a level group but only *at least* k. In the following we exploit this to alleviate the imbalance problem.

[7] This implies that as the number of levels increases, so does the parallelism. E.g., if the matrix contains at least one dense row, there is parallelism as $N_\ell = 2$ in this case.

4.3 Load balancing

The RACE load balancing scheme tries to balance the workload across level groups within each color for a given number of threads while maintaining data locality and the distance-k constraint between the two colors. To achieve this we use an idea similar to incremental graph partitioning [37]. The level groups containing low workload "grab" adjacent levels from neighboring level groups; overloaded level groups shift levels to adjacent level groups. One can either balance the number of rows (i.e., vertices) $N_{\mathrm{r}}(T(i))$ or the number of nonzeros (i.e., edges) $N_{\mathrm{nz}}(T(i))$. Both variants are supported by our implementation, and we choose balancing by number of rows in the following to demonstrate the method (see Algorithm 4).

For a given set of level groups we calculate the mean and variance of $N_{\mathrm{r}}(T(i))$ within each color (red and blue). The overall variance, which is the target of our minimization procedure, is then found by summing up the variances across colors. In order to reduce this value we first select the two level groups with largest negative/positive deviation from the mean (which is $T(5)$ and $T(4)$ in step 1 of Figure 7) and try to add/remove levels to/from them (see top row of Figure 7). When removing levels from a level group, the distance-k coloring is strictly maintained by keeping at least k levels in it. The shift of levels is done with the help of an array $T\_ptr[]$, which holds pointers to the beginning of each level group (see Figure 7), avoiding any copy operation. If shifting levels between the two level groups with the largest deviation does not lead to a lower overall variance, no levels are exchanged and we choose the next pair of level groups according to a ranking which is based on the absolute deviation from the mean (see Algorithm 4 for implementation details) and continue. Following this process in an iterative way we finally end up in a state of lowest overall variance at which no further moves are possible, either because they would violate the distance-k dependency or lead to an increase in overall variance. Figure 7 shows the load balancing procedure under a distance-2 constraint for some initial mapping of 17 levels to six level groups. Applying the procedure to our stencil example of size 16×16, requesting distance-2 coloring and ten level groups leads to the mapping shown in Figure 8(a). Note that level groups at the extreme ends have more levels due to fewer vertices (N_{r}) in each level, while level groups in the middle, having more vertices, maintain two levels to preserve the distance-2 constraint.

4.4 Recursion

As discussed in Section 4.2, the maximum degree of parallelism is limited by the total number of levels (N_{ℓ}) and may be further reduced by level aggregation. In case of our 16×16 stencil example the maximum possible parallelism is eight threads, which may cause load imbalance as seen in Figure 8(b). Hence, further parallelism must be found within the level groups. Compared to methods like MC we do not require all vertices in a level group to be distance-1 (or distance-k in general) independent. This is a consequence of our level-based approach, which requires distance-k independence between vertices of different levels but not within a level (see Equation (8)). There may be more parallelism hidden within the level groups, which can be interpreted as subgraphs. Thus we apply the three steps of our method recursively on selected subgraphs to exploit the parallelism within them.

In the following section we first demonstrate the basic idea in the context of distance-1 dependencies, which can be resolved within the given level group by design. However, for $k > 1$, vertices in a level group may have distance-k dependencies via vertices in adjacent level groups. We generalize our procedure to distance-k dependencies as a second step in Section 4.4.2. Finally, in Section 4.4.3 we apply the recursive scheme to our stencil example and introduce proper subgraph selection as well as global load balancing strategies.

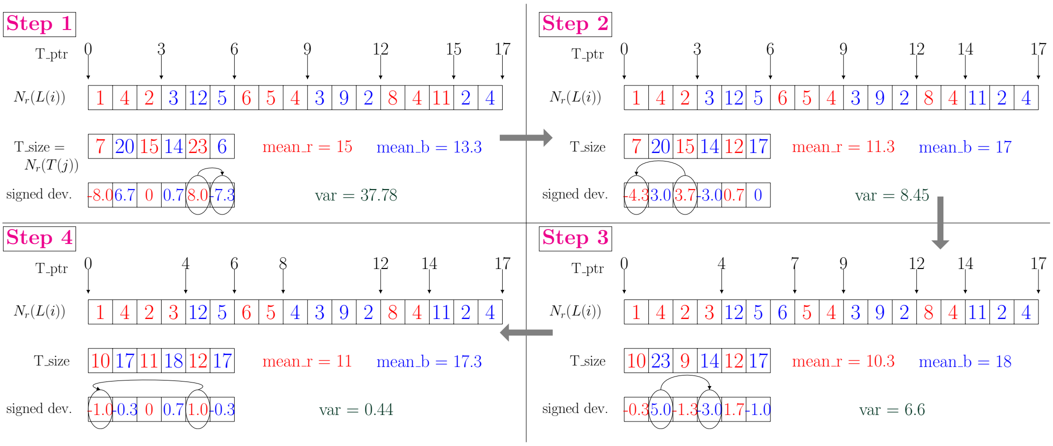
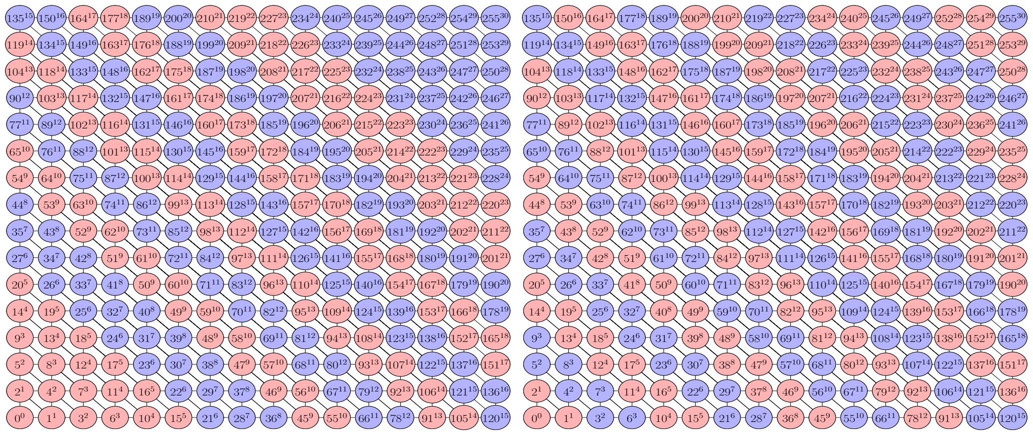


Fig. 7. All steps of the load balancing scheme, applied to an arbitrarily chosen initial distribution of 17 levels into six level groups for distance-2 coloring. Rebalancing steps are performed clockwise starting from top left. *mean_r* and *mean_b* denote the current average number of rows per level group and color. *var* is the overall variance.



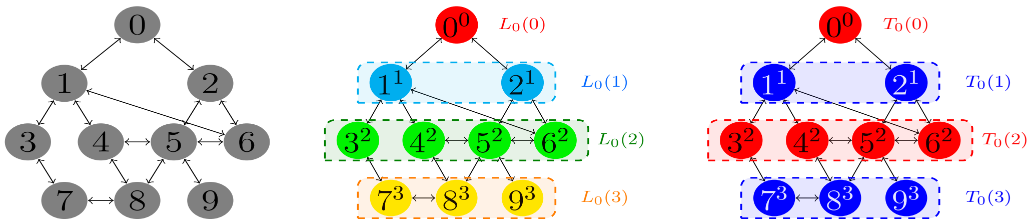
(a) Five threads

(b) Eight threads

Fig. 8. Domain size 16 × 16 for the stencil example and distance-2 dependency, (a) after load balancing for five threads, (b) after load balancing for eight threads.

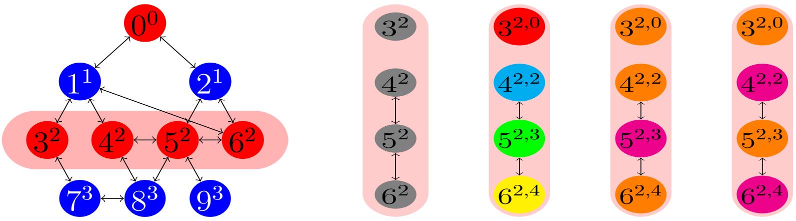
In order to visualize the basic concepts easily and discuss important corner cases of the recursive approach we start with the simple graph shown in Figure 9(a), which is not related to our stencil example. To distinguish between level groups at different stages s of the recursive procedure we add a subscript to the levels ($L_s(i)$) and level groups ($T_s(i)$) indicating the stage of recursion at which they are generated, with $s = 0$ being the original distribution before recursion is applied to any subgraph.

4.4.1 *Distance-*1 *dependency.* For the distance-1 coloring of the graph in Figure 9 we find that three of the four level groups of the initial stage still contain distance-1-independent vertices; e.g.,



(a) Example graph (b) Stage 0, levels in graph (c) distance-1 coloring

Fig. 9. Exposing potential for more parallelism in a graph with distance-1 coloring. $T_0(1), T_0(2)$, and $T_0(3)$ have internal unexposed parallelism. Note that the graph shown here is not related to the previous stencil example.



(a) (b) (c) (d) (e)

Fig. 10. Applying recursion to the subgraph induced by $T_0(2)$. Figure 10(b) shows the isolated subgraph, while Figure 10(c) presents the level construction step on the subgraph. Two potential distance-1 colorings of this subgraph are shown in Figures 10(d) and 10(e).

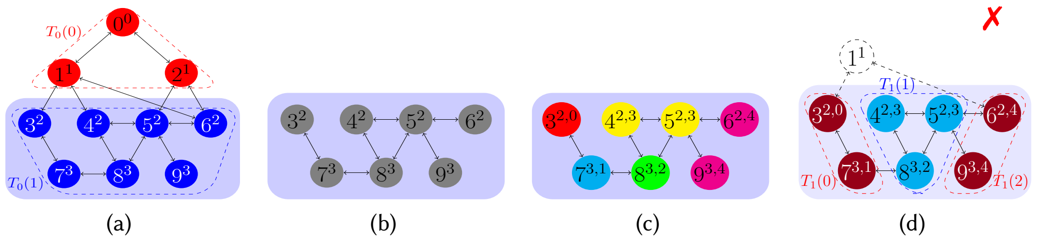
in $T_0(2)$ we have vertices $3 \overset{1}{\not\rightarrow} 4$ (3 distance-1 independent to 4), $3 \overset{1}{\not\rightarrow} 5$, $3 \overset{1}{\not\rightarrow} 6$, and $4 \overset{1}{\not\rightarrow} 6$, implying each of these pairs can be computed in parallel without any distance-1 conflicts. This parallelism is not exposed at the first stage ($s = 0$) as vertices in $L_0(i)$ are chosen such that they are distance-1 neighbors of $L_0(i-1)$, ignoring any vertex relations *within* $L_0(i)$.

Recursion starts with the selection of a subgraph of the matrix, which is discussed in more detail later (see Section 4.4.3). Here we choose the subgraph induced by $T_0(2)$. It can be isolated from the rest of the graph since the distance-1 coloring step in stage 0 has already produced independent level groups. Now we just need to repeat the three steps explained previously (Section 4.1–Section 4.3) on this subgraph.

Figure 10 shows an illustration of applying the first recursive step ($s = 1$) on $T_0(2)$, where we extend the definition of the vertex numbering in Equation (6) to the following:

$$v^{i,j,k\cdots} \implies v \in \{\, L_0(i) \cap L_1(j) \cap L_2(k) \cap \cdots \} \,. \tag{9}$$

At the end of the recursion (cf. Figures 10(d) and 10(e)) on $T_0(2)$, we obtain parallelism for two more threads in this case. Note that the subgraphs might have "islands" (groups of vertices that are not connected to the rest of the graph); e.g., vertex 3 and vertices 4,5,6 form two islands in Figure 10(b). Since an island is disconnected from the rest of the (sub)graph it can be executed independently and in parallel to it. To take advantage of this, the starting node in the next island is assigned a level number with an increment of two, as seen in Figure 10(c). This allows for two different colorings of the island, increasing the number of valid distance-1 configurations (cf. Figures 10(d)



(a) (b) (c) (d)

Fig. 11. Two level groups generated by a distance-2 coloring (Figure 11(a)). Figure 11(b) shows the subgraph induced by level group $T_0(1)$. Level construction on the selected subgraph is shown in Figure 11(c). Forming distance-2 independent level groups on these levels does not guarantee a distance-2 independence between the newly generated level groups of the same sweep (color) as seen in Figure 11(d).

and 10(e)). The selection of the optimal one will be done in the final load balancing step as described in Section 4.3.

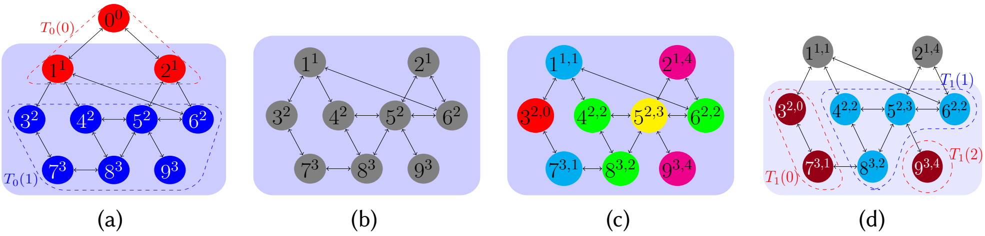
As this recursive process finds independent level groups (T_{s+1}) within a level group of the previous stage (T_s), the thread assigned to T_s has to spawn threads to parallelize within T_{s+1}.

4.4.2 Distance-k dependencies with $k > 1$. In general, it is insufficient to consider only the subgraphs induced by level groups in the recursion step, as can be seen in Figure 11(a) for distance-2 coloring. Applying the three steps (see Figures 11(b) to 11(d)) to the subgraph induced by $T_0(1)$ does not guarantee distance-2 independence between the new level groups $T_1(0)$ and $T_1(2)$. It is obvious that for general distance-k colorings two vertices a, b within a level group might be connected by a shared vertex c outside the level group. Thus, our three step procedure must be applied to a subgraph which contains the actual level group ($T_s(j)$) as well as its all distance-p neighbors, where $p = 1, 2, \ldots, (k-1)$.

This ensures that there is no vertex outside the subgraph which can mediate a distance-k dependency between vertices in the embedded level group ($T_s(j)$). We can now construct the new levels on this subgraph considering the neighborhood, but we only store the vertices in the new levels $L_{s+1}(:)$ that are in the embedded level group ($T_s(j)$). Next we apply distance-k coloring by aggregation of the new levels, leading to a set of level groups $T_{s+1}(:)$ within $T_s(j)$. Figure 12 demonstrates this approach to resolve the conflict shown in Figure 11(d). Figure 12(b) presents the subgraph containing the selected level group $T_0(1)$ and its distance-1 neighborhood.

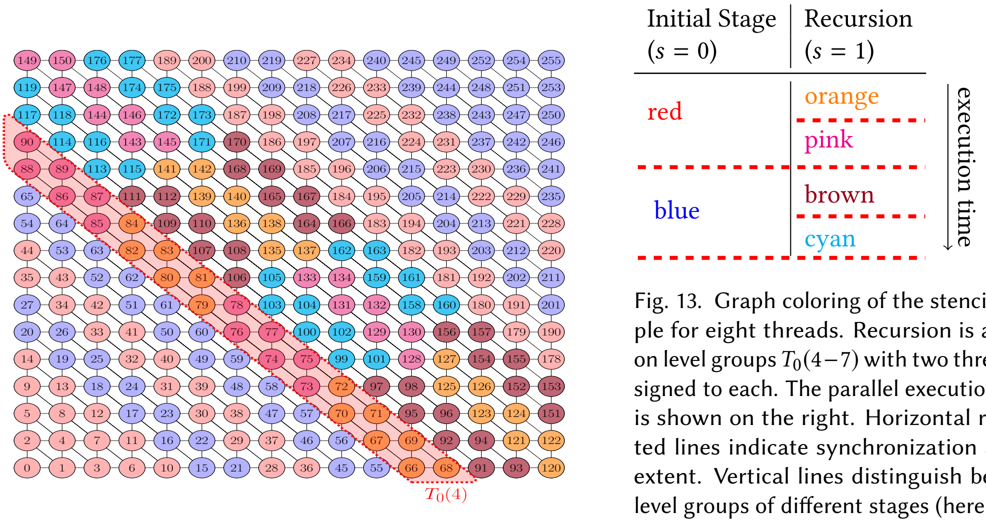
Level construction is performed on the subgraph (Figure 12(c)), but the new levels only contain vertices of $T_0(1)$, i.e., $L_1(1) = \{7^{3,1}\}$. Finally, distance-2 coloring by aggregation of two adjacent levels is performed, leading to three level groups of the second stage $s = 1$ (Figure 12(d)), i.e., $T_1(0) = \{L_1(0) \cup L_1(1)\}$. Now vertices 3 and 6 are mapped to level groups of different colors. Note that the permutation step on the newly generated levels is not shown but is performed as well to maintain data locality.

4.4.3 Level group construction and global load balancing. The recursive refinement of level groups allows us to tackle load imbalance problems and limited degree of parallelism as we are no longer restricted by the one thread per level group constraint. Instead, we have the opportunity to form level groups and assign appropriate thread counts to them such that the load per thread approaches the optimal value, i.e., the total workload divided by the number of threads available. Pairs of adjacent level groups having different colors within a stage, i.e., $T_s(i)$ and $T_s(i+1)$ with $i = 0, 2, 4, \ldots$, are typically handled by the same threads, so we assign an equal number of threads to them. We



(a) (b) (c) (d)

Fig. 12. Correct procedure for distance-2 coloring of level group $T_0(1)$. The subgraph as shown in Figure 12(b) contains level group $T_0(1)$ and its distance-1 neighborhood. A level construction step is applied to this subgraph in Figure 12(c). Distance-2 coloring by level aggregation leading to level groups of stage 1 is shown in Figure 12(d); we get three level groups at the end of the recursion on $T_0(1)$.



Initial Stage ($s = 0$)	Recursion ($s = 1$)
red	orange
	pink
blue	brown
	cyan

Fig. 13. Graph coloring of the stencil example for eight threads. Recursion is applied on level groups $T_0(4-7)$ with two threads assigned to each. The parallel execution order is shown on the right. Horizontal red dotted lines indicate synchronization and its extent. Vertical lines distinguish between level groups of different stages (here T_0 and T_1) which can run in parallel.

then apply recursion to the level groups with more than one thread assigned. Starting with the original graph as the base level group ($T_{-1}(0)$) to which all available threads $N_t(T_{-1}(0)) = N_t$ and all vertices $N_r(T_{-1}(0)) = N_r{}^{\text{total}}$ are assigned, we perform the following steps to form level groups $T_s(:)$ at stage $s \geq 0$ to which we assign $N_t(T_s(:))$ threads. To illustrate the procedure we use the 16×16 stencil example and construct a coloring scheme for eight threads (see Figure 13).

(1) Assign weights to all levels at stage (s) of the recursion. Assuming that $L_s(i) \subset T_{s-1}(j)$, its weight is defined by

$$w(L_s(i)) = \frac{N_r(L_s(i))}{\frac{N_r(T_{s-1}(j))}{N_t(T_{s-1}(j))}} = \frac{N_r(L_s(i))}{N_r(T_{s-1}(j))} N_t(T_{s-1}(j)).$$

For a given level group ($T_{s-1}(j)$) that has to be split up ($N_t(T_{s-1}(j)) > 1$), the weight describes the fraction of the optimal load per thread, $\frac{N_r(T_{s-1}(j))}{N_t(T_{s-1}(j))}$, in the specific level ($L_s(i)$).

Requesting $N_{\mathrm{t}}(T_{-1}(0)) = 8$ threads for the $N_{\mathrm{r}}(T_{-1}(0)) = 256$ vertices of the stencil example in Figure 13 produces the following weights for the initial ($s = 0$) levels:

$$\{w(L_0(0)), w(L_0(1)), w(L_0(2)), ...\} = \left\{\frac{1}{256} \times 8, \frac{2}{256} \times 8, \frac{3}{256} \times 8, ...\right\}.$$

(2) The above definition implies that if the weight is close to a natural number b, the corresponding workload is near optimal for operation with b threads. Thus, starting with $L_s(0)$ we aggregate successive levels until their combined weight forms a number a close to a natural number b. Distance-k coloring is ensured by enforcing it to aggregate at least $2 \times k$ levels, i.e., for distance-2 coloring at least four levels (two for red and two for blue). Closeness to the natural number is quantified by a parameter ϵ defined as

$$\epsilon = 1 - \mathrm{abs}(a - b), \text{ where } b = \max(1, [a])$$
$$\text{and } [a] \text{ is the nearest integer to } a,$$

and controlled by the criterion

$$\epsilon > \epsilon_s, \text{where the } \epsilon_s \in [0.5, 1) \text{ are user defined parameters.}$$

The choice of this parameter may be different for every stage of recursion. Once we find a collection of successive levels satisfying this criterion, the natural number b is fixed. We try to further increase the number of levels to test if there exists a number $a' > a$ which is closer to b leading to an ϵ value closer to one. We finally choose the set of levels with the best ϵ value and define them to form $T_s(0)$ and $T_s(1)$ which are to be executed by $N_{\mathrm{t}}(T_s(0)) = N_{\mathrm{t}}(T_s(1)) = b$ threads. In Figure 13 we choose $\epsilon_s = 0.6$, which selects the first seven levels to form $T_0(0)$ and $T_0(1)$. As their combined weight is $\frac{28}{32} = 0.875$, one thread will execute these two level groups.

(3) We continue with subsequent pairs of level groups ($T_s(i), T_s(i+1); i = 2, 4...$) by applying this procedure starting with the very next level. Finally, once all the levels have been touched, a total of $N_{\mathrm{t}}(T_{s-1}(j))$ threads have been assigned to the levels $L_s(i) \subset T_{s-1}(j)$. For example, for $T_0(4)$ and $T_0(5)$ in Figure 13 two threads satisfy the criterion as the total weight of the four levels included is $\frac{54}{32} = 1.69$.

(4) The distribution between adjacent red and blue level groups which are assigned to the same thread(s) as well as the final global load balancing is performed using a slight modification of the scheme presented in Section 4.3 (shown at the beginning of Algorithm 4 in appendix A): Now the calculation of mean and variance must consider the number of threads ($N_{\mathrm{t}}(T_s(j))$) assigned to each level group. The worker array now has to be replaced by the number of threads assigned to each level group ($N_{\mathrm{t}}(T_s(j))$). The algorithm then tries to minimize the variance of the number of vertices per thread in level groups. Ideally, after this step the load per thread in each level group should approach the optimal value given above.

Once the level group of stage s has been formed, the recursion and the above procedure are separately applied to all new level groups with more than one thread assigned. This continues until every level group is assigned to one thread. The depth of the recursion is determined by the parameter ϵ_s and depends on the matrix structure as well as degree of parallelism requested.

For our stencil example in Figure 13 the inner four level groups of stage $s = 0$ required one stage of recursion. This led to 16 level groups at stage $s = 1$, as we require four new level groups per recursion to schedule two threads. In terms of parallel computation, first the red vertices will be computed in parallel with the orange ones using four threads for both colors. Once the orange vertices are done, each pair of threads assigned to $T_0(4)$ and $T_0(6)$ synchronize locally (i.e., within $T_0(4)$ and $T_0(6)$ separately). Then the pink vertices are computed followed by a global

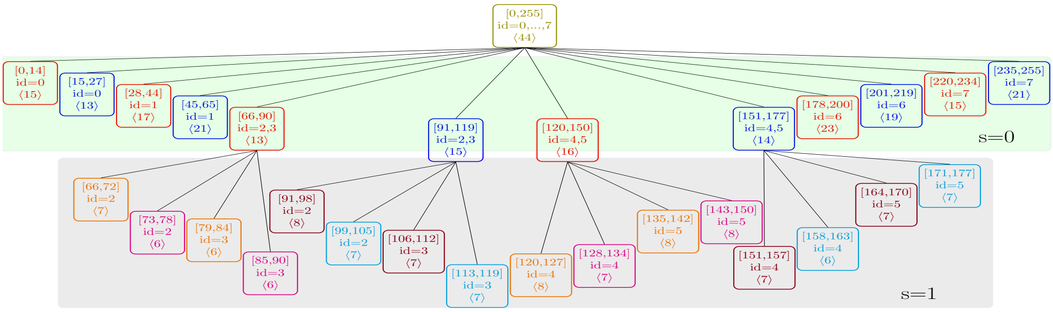


Fig. 14. The internal tree structure of RACE representing the stencil example for domain size 16 × 16 and eight threads. The range [. . .] specified in each leaf represents the vertices belonging to each level group and the id refers to the thread id assigned to each level group assuming compact pinning . The last entry $\langle N_{\mathrm{r}}^{\mathrm{eff}} \rangle$ gives the effective row count introduced in Section 5.

synchronization of all threads. The scheme continues with the blue vertices and the brown/cyan ones, which represent the two blue level groups to which recursion has been applied (see table in Figure 13).

The recursive nature of our scheme can be best described by a tree data structure, where every node represents one level group and the maximum depth is equivalent to the maximum level (stage) of recursion. The data structure for the colored graph in Figure 13 and its thread assignments are shown in Figure 14. The root node represents our baseline level group $T_{-1}(0)$ comprising all 256 vertices and all eight threads (having unique $id = 0, \ldots, 7$). The first level of child nodes gives the initial ($s = 0$) distribution, with each node storing the information of a level group including its color. Threads are mapped consecutively to the level groups. The red $T_0(4)$ level group, which consists of vertices $66, \ldots, 90$ (omitting the superscript for level numbers), is executed by threads with $id = 2, 3$. Applying recursion to $T_0(4)$, this node spawns four new child nodes at stage $s = 1$, i.e., level groups $T_1(0, \ldots, 3) \subset T_0(4)$, to be executed by the two threads. Synchronization only happens between threads having the same parent node after executing the same color. Note that actual computations are only performed on the leaf nodes of the final tree.

5 PARAMETER STUDY

The RACE method has a set of input parameters $\{\epsilon_s; s = 0, 1, \ldots\}$ which control the assignment of threads to adjacent level groups. To determine useful settings, we analyze the interaction between these input parameters, the number of threads used, and the parallel efficiency of the generated workload distribution.

As the internal tree structure contains all information about the final workload distribution, we can use it to identify the critical path in terms of workload and thus the parallel efficiency. To this end we introduce the *effective row count* for every node (or level group) $N_{\mathrm{r}}^{\mathrm{eff}}(T_s(i))$, which is a measure for the absolute runtime to calculate the corresponding level group. For level groups that are not further refined (leaf nodes) this value is their actual workload, i.e., the number of rows assigned to them ($N_{\mathrm{r}}^{\mathrm{eff}}(T_0(0)) = 15$ in Figure 14). For an inner node, the *effective row count* is the sum of the maximum workload (i.e., the maximum *effective row count* value) across each of the two colors of its child nodes:

$$N_{\mathrm{r}}^{\mathrm{eff}}(T_s(i)) = \max\left(N_{\mathrm{r}}^{\mathrm{eff}}(T_{s+1}(j) \subseteq T_s(i))\right) + \max\left(N_{\mathrm{r}}^{\mathrm{eff}}(T_{s+1}(j+1) \subseteq T_s(i))\right)$$
$$\text{for } j = 0, 2, \ldots$$

Such a definition is based on the idea that nodes at a given stage s have to synchronize with each other and have to wait for their siblings with the largest workload in each sweep (color). Propagating this information upwards on the tree until we reach the root node constructs the critical path in terms of longest runtime taking into account single thread workloads, dependencies, and synchronizations. Thus, the final value in the root node $N_\mathrm{r}^\mathrm{eff}(T_{-1}(0))$ can be considered as the effective maximum workload of a single thread. Dividing the globally optimal workload per thread, ${N_\mathrm{r}}^\mathrm{total}/N_\mathrm{t}$, by this number gives the parallel efficiency (η) of our workload distribution:

$$\eta = \frac{{N_\mathrm{r}}^\mathrm{total}}{N_\mathrm{r}^\mathrm{eff}(T_{-1}(0)) \times N_\mathrm{t}}.$$

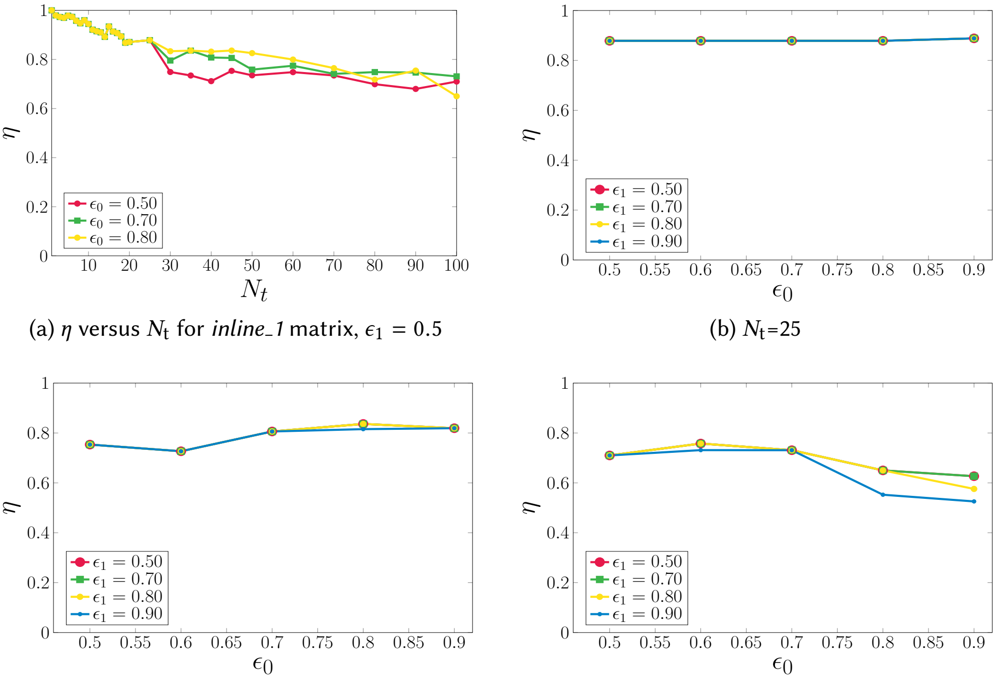
For the tree presented in Figure 14, the parallel efficiency is limited to $\eta = \frac{256}{44\times 8} = 0.73$ on eight threads, i.e., the maximum parallel speedup is 5.8.

5.1 Parameter analysis and selection

The parallel efficiency η as defined above can be calculated for any given matrix, number of threads N_t, and choice of $\{\epsilon_s; s = 0, 1, \ldots\}$; it reflects the quality of parallelism generated by RACE for the problem at hand. This way we can understand the interaction between these parameters and identify useful choices for the ϵ_s. Of course, running a specific kernel such as SymmSpMV on actual hardware will add further hardware and software constraints such as attainable memory bandwidth or cost of synchronization.

As a first step we can limit the parameter space by simple corner case analysis. Setting all parameters close to one requests high-quality load balancing but may prevent our balancing scheme from terminating. In the extreme case of $\{\epsilon_s = 1; s = 0, 1, \ldots\}$ the scheme may generate only two level groups (one of each color) in each recursion, assign all threads to them, and may further attempt to refine them in the same way. The lowest possible value of ϵ_s is the maximum deviation of a real number from its nearest integer, which is 0.5. A range of [0.5,0.9] for the ϵ_s is therefore used in the following. For a basic analysis we have selected the *inline_1* matrix (see Table 2) as it has a rather small amount of parallelism and allows us to cover basic scenarios. In Figure 15 we demonstrate the impact of different choices for ϵ_0 and ϵ_1 on the parallel efficiency for thread counts up to 100, which is a useful limit for modern CPU-based compute nodes. For $s > 1$ we always set the minimum value of $\epsilon_s = 0.5$. The limited parallelism can be clearly observed in Figure 15(a), with efficiency steadily decreasing with increasing thread count. At $\epsilon_1 = 0.5$ there is only a minor impact of the parameter ϵ_0. In Figures 15(b) to 15(d) the interplay between these two parameters is analyzed at different thread counts in more detail. We find that up to intermediate parallelism ($N_\mathrm{t} = 50$) the exact choice has only a minor impact on the parallel efficiency (see y-axis scaling). For larger parallelism the interplay becomes more intricate, where too large values of $\epsilon_{0,1}$ may lead to stronger imbalance. Based on this evaluation, we choose $\epsilon_{0,1} = 0.8$ and $\epsilon_s = 0.5$ for $s > 1$ for all subsequent performance measurements. The quality of this choice in terms of parallel efficiency for all matrices is presented in Figure 16. Here we plot the η value for all the matrices over a large thread count. We find that our parameter setting achieves parallel efficiencies of 80% or higher for a substantial fraction of the matrices up to intermediate thread counts. Representing the upper (lower) values in Figure 16 is the best (worst) case matrix *Graphene-4096* (*crankseg_1*), exhibiting almost perfect (very low) parallel efficiency at intermediate to high thread counts.

Finally, we evaluate the scalability of RACE using these two corner cases and the *inline_1* matrix as well as the *parabolic_fem* matrix, which is small enough to fit into the cache. In Figure 17 we mimic scaling tests on one Skylake processor with up to 20 cores (i.e., threads) and plot the parallel efficiency η as well as the maximum number of threads which can be “perfectly” used $N_\mathrm{t}^\mathrm{eff}$ (i.e.,



(a) η versus N_t for *inline_1* matrix, $\epsilon_1 = 0.5$

(b) N_t=25

(c) N_t=45

(d) N_t=100

Fig. 15. Parameter study on the *inline_1* matrix. In Figures 15(b) to 15(d) each of the lines in the plot are iso-ϵ_1 and impact of η with respect to ϵ_0 is shown. ϵ_s for $s > 1$ is fixed to 0.5.

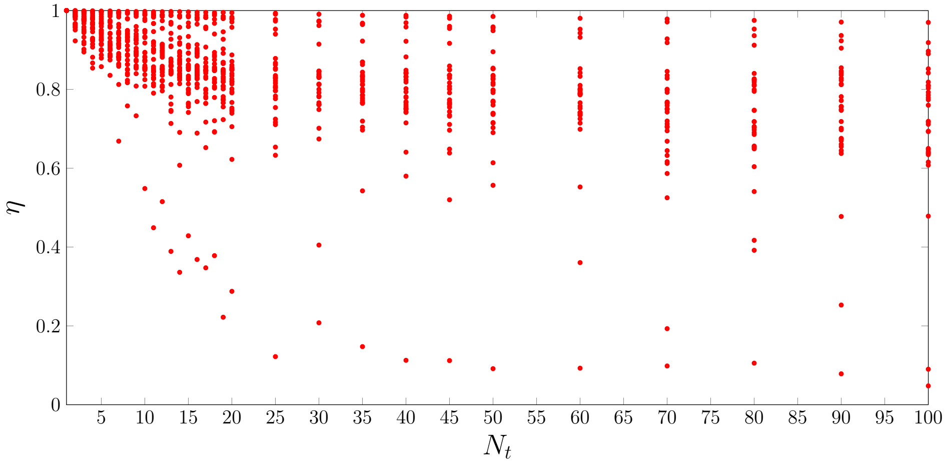
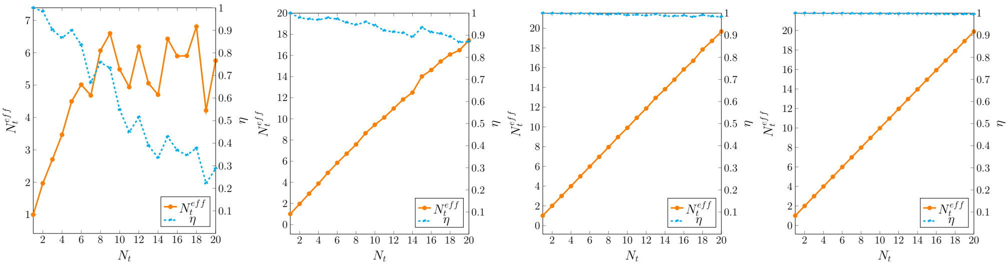


Fig. 16. Parallel efficiency η versus N_t for all test matrices with $\epsilon_{0,1} = 0.8$ and $\epsilon_{s>1} = 0.5$.

$N_t^{\text{eff}} = \eta \times N_t$). The unfavorable structure of the *crankseg_1* matrix puts strict limits on parallelism even for low thread counts. The combination of small matrix size with a rather dense population (see



(a) *crankseg_1* (b) *inline_1* (c) *parabolic_fem* (d) *Graphene-4096*

Fig. 17. N_t^{eff} and η versus N_t for the four corner case matrices, with the same settings used in experiment runs. N_t^{eff} is defined as $\eta \times N_t$.

Table 2) leads to large inner levels when constructing the graph, triggering strong load imbalance if using more than six threads. A search for better ϵ_s slightly changes the characteristic scaling but not the maximum parallelism that can be extracted. For the *inline_1* matrix we find a weak but steady decrease of the parallel efficiency, which is in good agreement with the discussion of Figure 15. The other two matrices scale very well in the range of thread counts considered.

The corresponding performance measurements for the SymmSpMV kernel (see Section 3.2) on a single Skylake SP processor chip with 20 cores are shown in Figure 18.[8] For the *crankseg_1* matrix (see Figure 18(a)) we recover the limited scaling due to load imbalance as theoretically predicted. A performance maximum is at nine cores, where the maximum SpMV performance can be slightly exceeded. However, based on the roofline performance model given by Equations (1) and (3) together with the matrix parameters from Table 2, a theoretical speedup of approximately two as compared to SpMV can be expected for the full processor chip under best conditions. Indeed, in case of the *inline_1* and *Graphene-4096* matrices, performance scales almost linearly until the main memory bandwidth bottleneck is hit. The saturated performance is in good agreement with the roofline limits. Note that even though the *inline_1* matrix does not exhibit perfect theoretical efficiency ($\eta \approx 0.85$ at $N_t = 20$), it still generates sufficient parallelism to achieve main memory saturation: The memory bottleneck can mitigate a limited load imbalance.

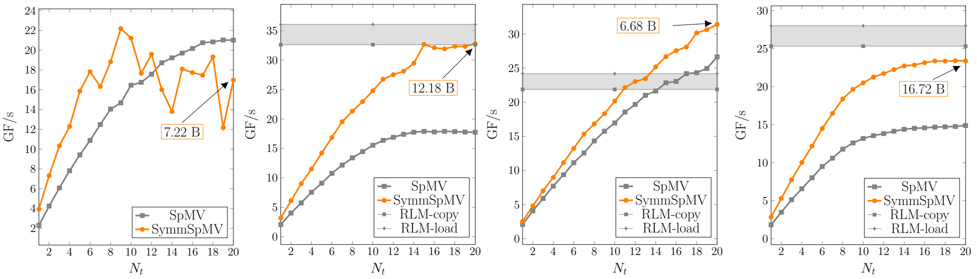
The peculiar performance behavior of *parabolic_fem* (see Figures 17(c) and 18(c)) is due to its smallness ($\approx$ 23 MB), which lets it fit into the caches of the Skylake processor (LLC size = 28 MB). Thus, performance is not limited by the main memory bandwidth constraint and the roofline model limits do not apply.

We have demonstrated that a simple choice for the only set of RACE input parameters $\{\epsilon_s; s = 0, 1, \ldots\}$ can extract sufficient parallelism for most matrices considered in this study. Moreover, the parallel efficiency as calculated by RACE in combination with the roofline performance model is a good indication for scalability and maximum performance of the actual computations.

6 PERFORMANCE EVALUATION OF SYMMSPMV USING RACE

We evaluate the performance of the SymmSpMV based on parallelization and reordering performed by RACE and compare it with the two MC approaches introduced above and the Intel MKL. As a yardstick for baseline performance we choose the general SpMV kernel and use the performance model introduced in Section 3 to quantify the quality of our absolute performance numbers. As the

[8] For the benchmarking setup see Section 6.



(a) *crankseg_1* (b) *inline_1* (c) *parabolic_fem* (d) *Graphene-4096*

Fig. 18. Parallel performance measurements of SymmSpMV with RACE on one Skylake SP socket for the four corner case matrices. The performance of the basic SpMV kernel is presented for reference. For the matrices Figures 18(b) to 18(d) the maximum roofline performance limits Equation (1) are given using the computational intensity Equation (3) for the two extreme cases of load-only memory bandwidth (RLM-load) and copy memory bandwidth (RLM-copy). The measured full socket main memory data traffic per nonzero entry of the symmetric matrix (in bytes) for the SymmSpMV operation is also shown, where values below 12 bytes indicate caching of the matrix entries.

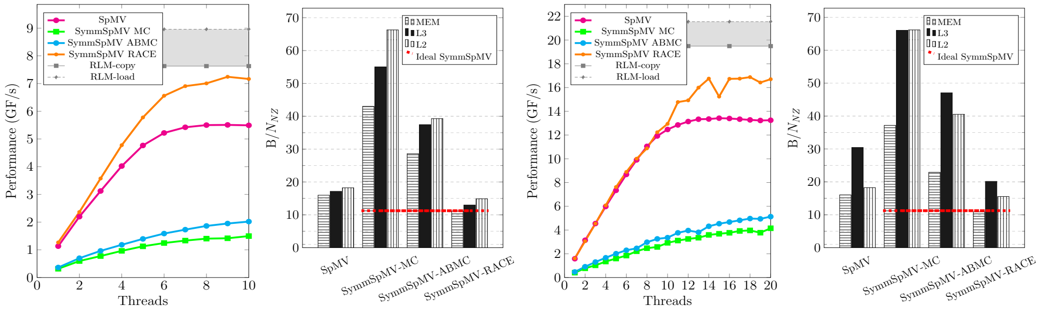
deviations between different measurement runs are less than 5%, we do not show the error bar in our performance measurements.

6.1 Experimental Setup

All matrix data are encoded in the CRS format. For the SymmSpMV only the nonzeros of the upper triangular matrix are stored. In the case of RACE and the coloring approaches every thread executes the SymmSpMV kernel Algorithm 2 with appropriate outer loop boundary settings depending on the color (MC, ABMC) or level groups (RACE) to be computed. In order to ensure vectorization of the inner loop in Algorithm 2 we use the SIMD pragma `#pragma simd reduction(+:tmp) vectorlength(VECWIDTH)`. Here `VECWIDTH` is the maximum vector width supported by the architecture, i.e., `VECWIDTH = 4 (8)` for Ivy Bridge EP (Skylake SP).

The Intel MKL offers two choices for the two sparse matrix kernels under consideration: First, CRS based data structures are provided and are used in the subroutines (`mkl_cspblas_dcsrgemv` for SpMV and `mkl_cspblas_dcsrsymv` for SymmSpMV) without any modification (MKL). This mode of operation is deprecated from Intel MKL.v.18. Instead, the inspector-executor mode (MKL-IE) is recommended to be used. Here, the user initially provides the matrix along with hints (e.g., symmetry) and operations to be carried out to the inspector routine (`mkl_sparse_set_mv_hint`). Then an optimization routine (`mkl_sparse_optimize`) is called where the matrix is preprocessed based on the inspector information to achieve best performance and highest parallelism for the problem at hand. The subroutine `mkl_sparse_d_mv` is then used to do the SpMV or SymmSpMV operations on this optimized matrix structure. This approach does not provide any insight into which kernel or data structure is actually used “under the hood.”

In the performance measurements the kernels are executed multiple times in order to ensure reasonable measurement times and average out potential performance fluctuations. Doing successive invocations of the kernel on the same two vectors, however, may lead to unrealistic caching of these vectors if the number of rows is small enough. Thus, we use two ring buffers (at least of 50 MB each) holding separate vectors of size N_r. After each kernel invocation we switch to the next vector in the two buffers. This way we mimic the typical structure of iterative sparse solvers where



(a) SymmSpMV (b) Data traffic (c) SymmSpMV (d) Data traffic

Fig. 19. Performance (Figure 19(a)) and data traffic (Figure 19(b)) analysis for SymmSpMV kernel with Spin-26 matrix using MC, ABMC, and RACE on a single socket of Ivy Bridge EP. The corresponding measurements for a single socket of Skylake SP is shown in Figures 19(c) and 19(d). The roofline performance model (using copy and load-only bandwidth) and the performance of the SpMV kernel is plotted for reference in scaling plots Figures 19(a) and 19(c). The average data traffic per nonzero entry (N_{nzr}) of the full matrix as measured with LIKWID for all cache levels and main memory is shown together with the minimal value for main memory access (horizontal dashed line) in Figures 19(b) and 19(d).

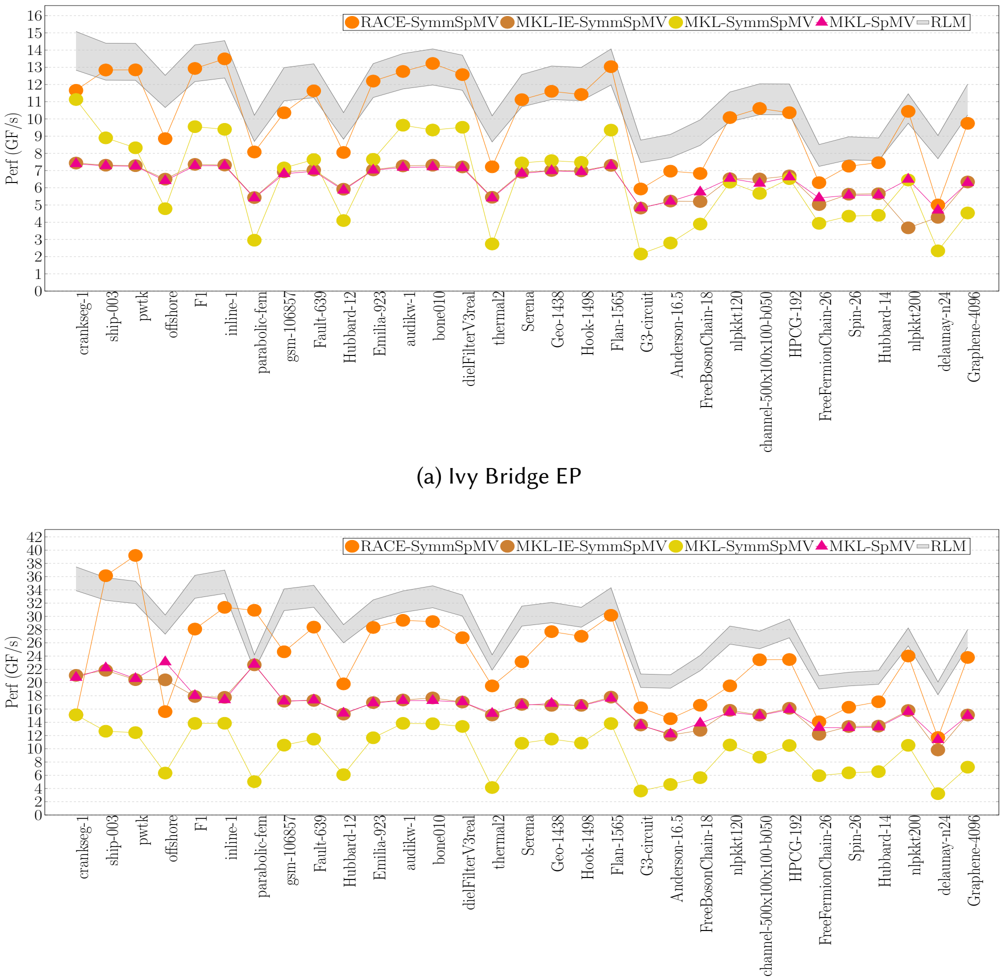
between successive matrix-vector operations other data intensive kernels are executed, e.g., several Level 1 BLAS routines or preconditioning steps. We run over these two buffers 100 iterations (times) and report the mean performance.

For all methods and libraries the input matrices have been preprocessed with RCM bandwidth reduction using the Intel SpMP library [42]. This provides the same or better performance on all matrices as compared to the original ordering. If not otherwise noted we use the full processor chip and assign one thread to each core. As we focus on a single chip and sub-NUMA clustering (SNC) is not enabled on Skylake SP, no NUMA data placement effects impact our results.

6.2 Results

Before we evaluate the performance across the full set of matrices presented in Table 2 we return to the analysis of the SymmSpMV performance and data traffic for the Spin-26 matrix which we have presented in Section 3.3 for the established coloring approaches.

6.2.1 Analysis of SymmSpMV kernel using RACE for the Spin-26 matrix. The shortcomings in terms of performance and excessive data transfer for parallelization of SymmSpMV using MC and ABMC have been demonstrated in Figure 2. We extend this evaluation by comparison with the RACE results in Figure 19. The figures clearly demonstrate the ability of RACE to ensure high data locality in the parallel SymmSpMV kernel. The actual main memory traffic achieved is inline with the minimum traffic for that matrix (see discussion in Section 3.3) and a factor of up to 4× lower than the coloring approaches. Correspondingly, RACE SymmSpMV performance is at least 3.3× higher than its best competitor and 25% better than the SpMV kernel on both architectures. It achieves more than 84% of the roofline performance limit based on the copy main memory performance. Note that the indirect update of the LHS vector will generate a store instruction for every inner loop iteration (see Algorithm 2), while the SpMV kernel only does a final store at the end of the inner loop iteration. In combination with the low number of nonzeros per row (N_{nzr}) of the Spin-26 matrix, the "copy" induced limit poses a realistic upper performance bound.



(a) Ivy Bridge EP

(b) Skylake SP

Fig. 20. Performance of SymmSpMV executed with RACE compared to the performance model and Intel MKL implementations. SpMV performance obtained using Intel MKL library is also shown for reference. The model prediction is derived for bandwidths in the range of load and copy bandwidth, and using the measured α_{SpMV} shown in Table 3.

6.2.2 Analyzing absolute performance of RACE. We now extend our RACE performance investigation to the full set of test matrices presented in Table 2. In Figures 20(a) and 20(b) the performance results for the full Ivy Bridge EP processor chip (10 cores) and the full Skylake SP processor chip (20 cores) are presented along with the upper roofline limits and the performance of the baseline SpMV kernel using Intel MKL. The matrices are arranged along the abscissa according to the ascending number of rows (N_r), i.e., increasing size of the two vectors involved in SymmSpMV. Overall RACE performance comes close to or matches our performance model for many test cases on both architectures. A comparison of the architectures shows that the corner case matrices crankseg_1 and parabolic_fem have a strikingly different behavior for RACE. For crankseg_1 this is caused by the limited amount of parallelism in its structures. Here we refer to the discussion of Figure 17(a) where best performance and highest parallelism ($N_\mathrm{t}^\mathrm{eff}$) were achieved at approximately 10 cores. Using only 9 cores on Skylake SP lifts the SymmSpMV performance of `crankseg_1` slightly above the SpMV level of the MKL. The `parabolic_fem` has been chosen to fit into the LLC of the Skylake SP architecture to provide a corner case where scalability is not intrinsically limited by main memory bandwidth (see Figure 17(c)) and thus our roofline performance limit does not apply for this matrix on Skylake SP. However, on Ivy Bridge EP the matrix data set just exceeds the LLC and the performance is inline with our model.

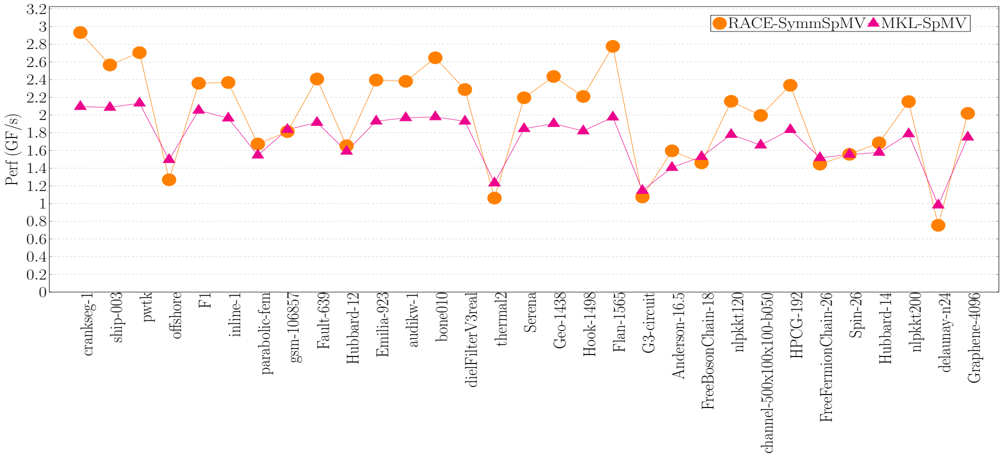
On both architectures a characteristic drop in performance levels is encountered around the `Flan_1565` and `G3_circuit` matrices, where the aggregate size of the two vectors (25 MB) approaches the available LLC sizes. For smaller matrices we have a higher chance that the vectors stay in the cache during the SymmSpMV, i.e., the vectors must only be transferred once between main memory and the processor for every kernel invocation. For larger matrices (i.e., larger N_r) the reuse of vector data during a single SymmSpMV kernel decreases and vector entries may be accessed several times from the main memory. This is reflected by the increase in measured α_{SpMV} (assumed $\alpha_{SymmSpMV}$) values for matrices with index 20 and higher in Table 3.

In short, RACE has an average speedup of 1.4× and 1.5× compared to SpMV on the Skylake SP and Ivy Bridge EP architectures, respectively. On Skylake SP RACE SymmSpMV attains on an average 87% and 80% of the roofline performance limits predicted using the copy and load bandwidth, respectively, while on Ivy Bridge EP we are 91% and 83% close to the respective performance models.

The MKL implementations of SymmSpMV deserves a special consideration in this context. Therefore, in Figure 20 we also compare our approach with the two Intel MKL options described above. For the MKL-IE variant we specify exploiting the symmetry of the matrix when calling the inspector routine. On the Ivy Bridge EP architecture, RACE always provides superior performance levels and the best performing Intel variant depends on the underlying matrix. On the Skylake SP, however, MKL-IE always outperforms the deprecated MKL routine and is superior to RACE for two matrices (`crankseg-1,offshore`). These are the same matrices where RACE is slower than the MKL SpMV kernel (see Figure 20(b)). It can be clearly seen that the MKL-IE data for SymmSpMV are identical with the MKL SpMV numbers presented in Figure 20, i.e., the inspector calls the baseline SpMV kernel and uses the full matrix, though it knows about the symmetry of the matrix. One reason for that strategy might be that the parallelization approach used in the deprecated MKL implementation for SymmSpMV is not scalable which would explain the fact that MKL is worse than MKL-IE for all cases on Skylake SP. As neither the algorithm used to parallelize the SymmSpMV nor its low level code implementation is known, we refrain from a deep analysis of the Intel performance behavior. In summary we find that RACE is on average 1.4× faster than the best Intel variant and can achieve speedups of up to 2×. Note that on Skylake SP the best MKL variant is always MKL-IE, which has almost twice the memory footprint compared to the SymmSpMV with RACE.

6.2.3 Single core performance.

Although single core performance is often considered not to be crucial for the full chip SymmSpMV performance, we demonstrate that it is vital to explain some of the performance behaviors. For example the drop in SymmSpMV performance for matrices like `Hubbard-12` and `delaunay_n24` strongly correlates with the lower performance of the baseline SpMV (see Figure 20). These matrices are characterized by a rather low N_nzr and a larger α_{SpMV} value. Note that α_{SpMV} measured for the SpMV kernel mainly accounts for the RHS vector traffic and the actual $\alpha_{SymmSpMV}$ may even be higher as SymmSpMV requires two vectors to stay in cache concurrently. Moreover, for these matrices the inner loop lengths are typically very short



(a) Skylake SP

Fig. 21. Single core performance of SymmSpMV executed with RACE compared to SpMV performance using Intel MKL.

(approximately $N_{\text{nzr}}/2$ on average) and consequently the SIMD vectorization performed by the compiler may become inefficient. This leads to lower single core performance as shown in Figure 21 for the Skylake SP architecture, where bad performance of SymmSpMV and SpMV can often be correlated with a small N_{nzr} value. For several matrices these combined effects overcompensate the reduced matrix data traffic of the SymmSpMV leading to worse single core performance than running SpMV with the full matrix. Using the `delaunay_n24` matrix as a representative for this class of matrices we demonstrate the basic challenge for SymmSpMV to exploit its basic performance advantage over SpMV in Figure 22. Starting with an approximately 25% lower single core performance (0.75 GF/s versus 0.98 GF/s) but having a 50% higher roofline performance limit (approximately 18 GF/s; see Figure 20(b)) than the SpMV, the SymmSpMV is not able to saturate the main memory bandwidth of the Skylake SP on its 20 cores. As speculated above, the single core performance is limited by inefficient SIMD vectorization of the extremely short inner loop and switching back to scalar code does improve performance by 15% (see Figure 22). As we are still substantially off the bandwidth limit we see this benefit over the full chip. Using chips with larger core counts would allow for further improving the SymmSpMV performance of this matrix. The same arguments hold for the `offshore` matrix but here the effect compared to SpMV performance is even more pronounced on Skylake SP. Here the full matrix can at least partially be held in the large aggregate cache between successive kernel invocations and its performance is not limited by the main memory bandwidth. In terms of caching effects we have also further identified at least partial caching of the matrix for `ship-003` and `pwtk` test cases by analyzing the overall data traffic in the kernel invocations. This is inline with their higher performance levels presented in Figure 20(b).

6.2.4 Comparing RACE with MC and ABMC. Having well understood the performance characteristics of SymmSpMV with RACE we finally compare this with the performance achieved by the two coloring methods in Figure 23. Here the underlying algorithm as well as implementation are known and are closely related to our approach. Overall the MC is not competitive and provides

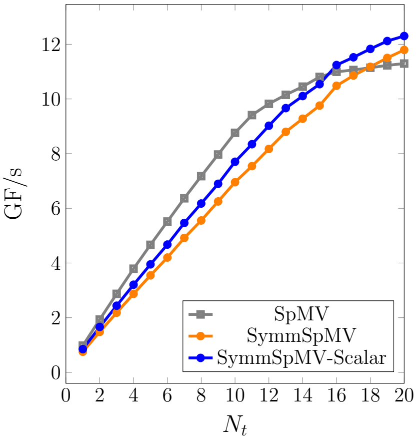


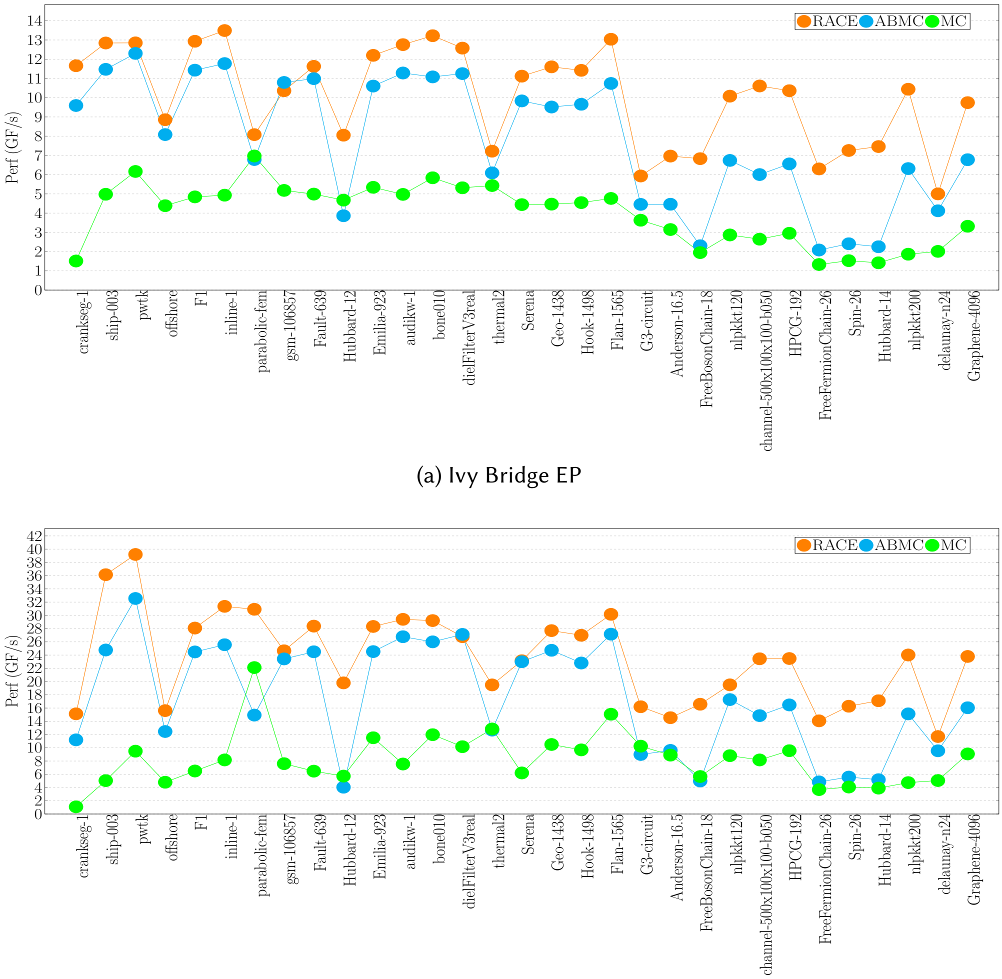
Fig. 22. Parallel performance of SymmSpMV (with RACE) and SpMV (with Intel MKL) for the delaunay_n24 matrix on one socket of Skylake SP. To disable vectorization (SymmSpMV-Scalar) we set VECWIDTH = 1 when compiling the SymmSpMV kernel.

low performance levels for almost all the matrices on both architectures. The ABMC shows similar performance characteristics as RACE until the two vectors involved in SymmSpMV approach the size of the caches (cf. discussion of Figure 20). For matrices with sufficiently small N_r (left in the diagram) the method can achieve between 70% and 90% of RACE performance on most cases. For matrices in the right part of the diagram with their higher N_r and α_{SpMV} values, the ABMC falls substantially behind RACE. Here, the strict orientation of the RACE design towards data locality in the vector accesses delivers its full power. See also the data transfer discussion in Section 6.2.1 for the `Spin-26` matrix. In total there are only three cases where ABMC performance is on a par with or slightly above the RACE measurement and the average speedup of RACE is 1.5× and 1.65× for Ivy Bridge EP and Skylake SP, respectively. Note that all three methods use the same baseline kernels and thus performance differences between the methods do not arise from different low level code but from the ability to generate appropriate degrees of parallelism and to maintain data locality.

7 CONCLUSION AND OUTLOOK

In this paper we have developed RACE, a coloring algorithm and open-source library implementation for exploiting parallelism in algorithms with inherent dependencies. RACE generates hardware-efficient distance-k colorings of undirected graphs and puts emphasis on data access locality, load balancing, and parallelism that is adapted to the number of cores of the underlying architecture. We demonstrated these benefits by applying RACE to symmetric sparse matrix-vector multiplication (SymmSpMV) on modern multicore architectures and compared its performance against standard multicoloring, algebraic block multicoloring, and Intel MKL implementations. Average and maximum speedups of 1.4 and 2, respectively, could be observed across a representative set of 31 matrices on two modern Intel processors. Our entire experimental and performance analysis process was backed by the Roofline performance model, corroborating the optimality of the RACE approach in terms of resource utilization and shedding some new light on the challenges of the SymmSpMV kernel on modern hardware. We demonstrated that RACE runs very close to the Roofline limit for most of the 31 test cases. Outliers were analyzed and discussed in detail.

Similar to other coloring algorithms, the RACE method is not limited to the SymmSpMV kernel and can be used to efficiently parallelize solvers and kernels having general distance-k dependencies. Moreover, due to the level-based formulation of RACE, the framework has an added advantage that allows us to address other classes of problems. Future work with RACE will involve variants



(a) Ivy Bridge EP

(b) Skylake SP

Fig. 23. Comparison of SymmSpMV performance between RACE and coloring variants MC and ABMC. Matrices are arranged in increasing number of rows (N_r).

of linear solvers and kernel operations like in-place matrix powers and polynomials, which are of high interest in the scientific community.

ACKNOWLEDGMENTS

The project is funded by the German DFG priority programme 1648 “Software for Exascale Computing (SPPEXA)” and the Swiss National Science Foundation (SNF) under the projects “Dual-Phase Steels – From Micro to Macro Properties (EXASTEEL-2)” (DFG, SNF) and “Equipping Sparse Solvers for Exascale (ESSEX-II)” (DFG). The authors wish to thank Andreas Alvermann for providing access to his ScaMaC library, Thomas Gruber for supporting our LIKWID measurements and Moritz Kreutzer for helpful discussions.

ACRONYMS

$L(i)$ ith level.
$L_s(i)$ ith level at stage s.
N_ℓ total levels in the graph.
N_r number of rows.
N_t number of threads.
N_nzr average number of nonzeros per row.
$N_\mathrm{nzr}^\mathrm{symm}$ average number of nonzeros per row in symmetric matrix.
N_nz total number of nonzeros.
$N_\mathrm{r}^\mathrm{eff}$ effective number of rows.
$N_\mathrm{t}^\mathrm{eff}$ effective number of threads.
$T(i)$ ith level group.
$T_s(i)$ ith level group at stage s.
η theoretical parallel efficiency.
b_s single socket bandwidth.
s stage number of recursion.

ABMC algebraic block multicoloring.

BFS breadth-first search.

CRS compressed row storage.

Intel MKL Intel math kernel library.

LLC last level cache.

MC multicoloring.

RACE recursive algebraic coloring engine.
RCM reverse Cuthill McKee.

SNC sub-NUMA clustering.
SpMV sparse matrix-vector multiplication.
SymmSpMV symmetric sparse matrix-vector multiplication.

REFERENCES

[1] Andreas Alvermann. 2019. ScaMaC: The Scalable Matrix Collection. https://bitbucket.org/essex/matrixcollection/.

[2] D. Bozdağ, Ü. Çatalyürek, A. Gebremedhin, F. Manne, E. Boman, and F. Özgüner. 2010. Distributed-Memory Parallel Algorithms for Distance-2 Coloring and Related Problems in Derivative Computation. *SIAM Journal on Scientific Computing* 32, 4 (2010), 2418–2446. https://doi.org/10.1137/080732158 arXiv:https://doi.org/10.1137/080732158

[3] Doruk Bozdağ, Assefaw H. Gebremedhin, Fredrik Manne, Erik G. Boman, and Umit V. Catalyurek. 2008. A framework for scalable greedy coloring on distributed-memory parallel computers. *J. Parallel and Distrib. Comput.* 68, 4 (2008), 515 – 535. https://doi.org/10.1016/j.jpdc.2007.08.002

[4] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[5] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*. IEEE Computer Society, Washington, DC, USA, 721–733. https://doi.org/10.1109/IPDPS.2011.73

[6] U.V. Catalyurek and C. Aykanat. 1999). Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Systems* 10, 7 (1999)), 673–693. https://doi.org/10.1109/71.780863

[7] Elizabeth Cuthill. 1972. *Several Strategies for Reducing the Bandwidth of Matrices.* Springer US, Boston, MA, 157–166. https://doi.org/10.1007/978-1-4615-8675-3_14

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[9] Josep Díaz, Jordi Petit, and Maria Serna. 2002. A Survey of Graph Layout Problems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 313–356. https://doi.org/10.1145/568522.568523

[10] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2018. SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms. *ACM Trans. Math. Softw.* 44, 3, Article 26 (Jan. 2018), 32 pages. https://doi.org/10.1145/3134442

[11] D. J. Evans. 1984. Parallel S.O.R. Iterative Methods. *Parallel Comput.* 1, 1 (Aug. 1984), 3–18. https://doi.org/10.1016/S0167-8191(84)90380-6

[12] Fujitsu. 2019. Retrieved 2019/05/14 from https://www.fujitsu.com/global/Images/post-k_supercomputer_with_fujitsu%27s_original_cpu_a64fx_powered_by_arm_isa.pdf

[13] Martin Galgon, Lukas Krämer, Jonas Thies, Achim Basermann, and Bruno Lang. 2015. On the Parallel Iterative Solution of Linear Systems Arising in the FEAST Algorithm for Computing Inner Eigenvalues. *Parallel Comput.* 49, C (Nov. 2015), 153–163. https://doi.org/10.1016/j.parco.2015.06.005

[14] Assefaw Hadish Gebremedhin and Fredrik Manne. 2000. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience* 12, 12 (2000), 1131–1146.

[15] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. 2002. Parallel Distance-k Coloring Algorithms for Numerical Optimization. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing (Euro-Par '02)*. Springer-Verlag, London, UK, UK, 912–921. http://dl.acm.org/citation.cfm?id=646667.699892

[16] Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen. 2013. ColPack: Software for Graph Coloring and Related Problems in Scientific Computing. *ACM Trans. Math. Softw.* 40, 1, Article 1 (Oct. 2013), 31 pages. https://doi.org/10.1145/2513109.2513110 Software last accessed on 2019/02/25.

[17] T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas, and N. Koziris. 2013. Improving the Performance of the Symmetric Sparse Matrix-Vector Multiplication in Multicore. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 273–283. https://doi.org/10.1109/IPDPS.2013.43

[18] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. 2000. Towards Realistic Performance Bounds for Implicit CFD Codes. In *Parallel Computational Fluid Dynamics 1999*, D. Keyes, J. Periaux, A. Ecer, N. Satofuka, and P. Fox (Eds.). Elsevier, 241–248. https://doi.org/10.1016/B978-044482851-4.50030-X

[19] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158. https://doi.org/10.1177/1094342004041296 arXiv:https://doi.org/10.1177/1094342004041296

[20] Intel. 2019. Intel Math Kernel Library. https://software.intel.com/en-us/mkl

[21] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. 2012. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 474–483. https://doi.org/10.1109/IPDPS.2012.51

[22] Mark T. Jones and Paul E. Plassmann. 1994. Scalable Iterative Solution of Sparse Linear Systems. *Parallel Comput.* 20, 5 (May 1994), 753–773. https://doi.org/10.1016/0167-8191(94)90004-3

[23] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[24] Kokkos. 2018. Kokkos C++ Performance Portability Programming EcoSystem: The Programming Model - Parallel Execution and Memory Abstraction. http://trilinos.sandia.gov/packages/kokkos

[25] Moritz Kreutzer, Dominik Ernst, Alan R. Bishop, Holger Fehske, Georg Hager, Kengo Nakajima, and Gerhard Wellein. 2018. Chebyshev Filter Diagonalization on Modern Manycore Processors and GPGPUs. In *High Performance Computing*, Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis (Eds.). Springer International Publishing, Cham, 329–349.

[26] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423. https://doi.org/10.1137/130930352

[27] M. Krotkiewski and M. Dabrowski. 2010. Parallel Symmetric Sparse Matrix-vector Product on Scalar Multi-core CPUs. *Parallel Comput.* 36, 4 (April 2010), 181–198. https://doi.org/10.1016/j.parco.2010.02.003

[28] C. Y. Lee. 1961. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers* EC-10, 3 (Sept 1961), 346–365. https://doi.org/10.1109/TEC.1961.5219222

[29] Ang Li, Weifeng Liu, Mads R. B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and Analyzing the Real Impact of Modern On-package Memory on HPC Scientific Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17).* ACM, New York, NY, USA, Article 26, 14 pages. https://doi.org/10.1145/3126908.3126931

[30] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15).* ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

[31] Weifeng Liu and Brian Vinter. 2015. Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors. *Parallel Comput.* 49, C (Nov. 2015), 179–193. https://doi.org/10.1016/j.parco.2015.04.004

[32] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13).* ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/2464996.2465013

[33] H. Lu, M. Halappanavar, D. Chavarra-Miranda, A. H. Gebremedhin, A. Panyala, and A. Kalyanaraman. 2017. Algorithms for Balanced Graph Colorings with Applications in Parallel Computing. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (May 2017), 1240–1256. https://doi.org/10.1109/TPDS.2016.2620142

[34] Michele Martone. 2014. Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix-vector Multiplication with the Recursive Sparse Blocks Format. *Parallel Comput.* 40, 7 (July 2014), 251–270. https://doi.org/10.1016/j.parco.2014.03.008

[35] James McQueen, Marina Meilă, Jacob VanderPlas, and Zhongyue Zhang. 2016. Megaman: Scalable Manifold Learning in Python. *Journal of Machine Learning Research* 17, 148 (2016), 1–5. http://jmlr.org/papers/v17/16-109.html

[36] P. Mironowicz, A. Dziekonski, and M. Mrozowski. 2015. A Task-Scheduling Approach for Efficient Sparse Symmetric Matrix-Vector Multiplication on a GPU. *SIAM Journal on Scientific Computing* 37, 6 (2015), C643–C666. https://doi.org/10.1137/14097135X

[37] Chao-Wei Ou and Sanjay Ranka. 1997. Parallel Incremental Graph Partitioning. *IEEE Trans. Parallel Distrib. Syst.* 8, 8 (Aug. 1997), 884–896. https://doi.org/10.1109/71.605773

[38] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488 (ISC 2014).* Springer-Verlag New York, Inc., New York, NY, USA, 124–140. https://doi.org/10.1007/978-3-319-07518-1_8

[39] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey. 2014. Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and its Application to Unstructured Matrices. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis.* 945–955. https://doi.org/10.1109/SC.2014.82

[40] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718003 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003

[41] Toby Simpson, Dimosthenis Pasadakis, Drosos Kourounis, Kohei Fujita, Takuma Yamaguchi, Tsuyoshi Ichimura, and Olaf Schenk. 2018. Balanced Graph Partition Refinement Using the Graph p-Laplacian. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '18).* ACM, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3218176.3218232

[42] SpMP Development Team. [n.d.]. Sparse matrix pre-processing library. Retrieved 2019/02/25 from https://github.com/IntelLabs/SpMP

[43] S. Toledo. 1997. Improving the Memory-system Performance of Sparse-matrix Vector Multiplication. *IBM J. Res. Dev.* 41, 6 (Nov. 1997), 711–726. https://doi.org/10.1147/rd.416.0711

[44] J. Treibig, G. Hager, and G. Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. (2010).

[45] Ulrike von Luxburg. 2007. A tutorial on spectral clustering. *Statistics and Computing* 17, 4 (01 Dec 2007), 395–416. https://doi.org/10.1007/s11222-007-9033-z

[46] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 1 (2005), 521. http://stacks.iop.org/1742-6596/16/i=1/a=071

[47] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. *Parallel Comput.* 35, 3 (March 2009), 178–194. https://doi.org/10.1016/j.parco.2008.12.006

[48] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[49] A. Yzelman and R. Bisseling. 2009. Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM Journal on Scientific Computing* 31, 4 (2009), 3128–3154. https://doi.org/10.1137/080733243 arXiv:https://doi.org/10.1137/080733243

[50] Albert-Jan Nicholas Yzelman. 2011. *Fast sparse matrix-vector multiplication by partitioning and reordering*. Ph.D. Dissertation. Utrecht University, Utrecht.

A ALGORITHMS

Algorithm 3 Construction of levels

```
$integer :: root = n$      % Choose starting node
$bool :: marked\_all = false$      % Stopping criteria
$integer :: N = nrows(graph)$
$integer :: distFromRoot[N] = \{-1\}$
$integer :: curr\_children[] = \{\}$
$curr\_children.push\_back(root)$;
$integer :: currLvl = 0$
while $!marked\_all$ do
    $marked\_all$ = true
    $integer :: nxt\_children[] = \{\}$
    for $i = 1 : size(curr\_children)$ do
        if $distFromRoot[curr\_children[i]] == -1$ then
            $distFromRoot[curr\_children[i]] = currLvl$
            for $j$ in $graph[curr\_children[i]].children$ do
                if $distFromRoot[j] == -1$ then
                    $nxt\_children.push\_back(j)$
                end if
            end for
        end if
    end for
    $curr\_children = nxt\_children$
    $currLvl = currLvl + 1$
end while
```

Algorithm 4 Load Balancing for two sweep, distance-2, two colors

```
if section 4.3 then
    $integer :: nthreads = N_{\mathrm{t}}$
    $integer :: len = 2 * nthreads$    % number of level groups
    $integer :: worker[len] = 1$
    $integer :: T\_ptr[len+1] = linspace(0, N_\ell, len)$  % level group pointer
else
    $integer :: nthreads = n_t(T_{s-1}(i))$ % $i$ is the index of level group in stage $s-1$
                                            % where recursion is applied.
    $integer :: len = 2 * nthreads$    % number of level groups
    $integer :: worker[len] = [n_t(T_s(0)), ..., n_t(T_s(len-1))] = b$
    $integer :: T\_ptr[len+1] = linspace(0, N_\ell, len)$
end if
$bool :: exit = false$
$integer :: T\_size[len], absRankIdx[len], rankIdx[len], currRank$
$double :: mean\_r, mean\_b, diff[len], var, newVar$
while !($exit$) do
    $T\_size[:]$ = update($T\_ptr[:]$)  % $T\_size$ contains nrows in each level group
    $integer :: T\_size\_worker[:] = T\_size[:]/worker[:]$
    $mean\_r$ = sum($T\_size\_worker[0:2:len-1]$) / $nthreads$ %mean of red color
    $mean\_b$ = sum($T\_size\_worker[1:2:len-1]$) / $nthreads$ %mean of blue color
    $diff[0:2:len-1] = T\_size\_worker[0:2:len-1].-mean\_r$
    $diff[1:2:len-1] = T\_size\_worker[1:2:len-1].-mean\_b$
    $var$ = dot_product($diff, diff$)/len % overall variance
    $absRankIdx$ = argsort(-abs($diff$)) % ranking according to absolute deviation
    $rankIdx$ = argsort($diff$) % ranking according to signed deviation
    $currRank = 0, newVar = var$
    $integer :: old\_T\_ptr[len+1] = T\_ptr[:], acquireIdx, giveIdx$
    while $newVar \geq var$ do
        $T\_ptr = old\_T\_ptr$
        $bool :: fail$=true
        if $diff[absRankIdx[currRank]] < 0$ then
            for $el$ in $rankIdx[(len-1):-1:0]$ do
                if $(T\_Ptr[el+1] - T\_ptr[el]) > 2$ then
                    $acquireIdx$ = el
                    $fail$=false
                    $break$
                end if
            end for
            shift($T\_ptr, acquireIdx, currRank$) % shifts $T\_ptr$ by 1 from $acquireIdx$
                                % to $currRank$ if $currIdx < acquireIdx$ else shift by -1
        else if $(T\_ptr[currRank+1] - T\_ptr[currRank]) > 2$ then
            $giveIdx = rankIdx[0]$
            $fail$=false
            shift($T\_ptr, currRank, giveIdx$)
        end if
```

```
46:         if $!fail$ then
47:             $newVar$ = calculate_variance($T\_ptr$) % as seen in Line 17 to Line 23
48:         end if
49:         if ($currRank == (len - 1)$) && ($newVar \geq var$) then
50:             $T\_Ptr = old\_T\_ptr$
51:             $exit$ = true
52:             $break$
53:         end if
54:         $currRank += 1$
55:     end while
56: end while
```