SC18:G: RACE - Recursive Algebraic Coloring Engine

Christie Louis Alappat and Gerhard Wellein
Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg, Germany

1 Problem & Motivation

Sparse linear algebra is a key component in many scientific simulations ranging from quantum physics to fluid and structural mechanics. However, iterative numerical methods and important building blocks of sparse linear algebra frequently feature strong data dependencies, making them difficult to parallelize. Typically, loop-carried dependencies occur in iterative solvers (e.g., Kaczmarz, Gauss-Seidel) or preconditioners and write conflicts show up in the parallelization of building blocks such as symmetric sparse matrix-vector multiplication. Scalable, hardware-efficient parallelization of such methods and kernels is known to be a challenge. Multi-coloring is a widely used approach to enable parallelization of iterative solvers with distance-k dependency; e.g., the red-black Gauss-Seidel algorithm solves the distance-1 dependency problem. However, most of those standard solutions suffer from low performance on modern hardware, are highly problem specific, or require tailored sparse matrix storage formats.

RACE addresses these shortcomings by combining ideas from graph traversal and multi-coloring to ensure data locality, to generate appropriate levels of parallelism, and to enable hardware-efficient parallelization schemes. It is applicable to many problems (i.e., matrix structures) and general sparse data storage formats.

Outline

The paper is structured as follows. Section 2 describes the underlying dependency problems and conventional solutions. Section 3 demonstrates the major drawbacks of the existing approaches. We then introduce the RACE method in Section 4, its uniqueness and how its basic design addresses the existing problems. In Section 5 we compare RACE performance for thread-level parallelization of symmetric sparse matrix-vector multiplication (SymmSpMV) to available standard solutions including Intel MKL. Finally we use RACE to parallelize a sparse eigenvalue solver provided by Intel MKL and demonstrate RACE's superiority in terms of performance and attainable problem sizes.

2 Background & Related Work

Data dependencies often prevent a straightforward parallelization of sparse linear algebra kernels. As a representative and highly relevant example for a distance-2 dependency problem, we use symmetric sparse matrix-vector multiplication (SymmSpMV). Algorithm 1 shows the pseudo-code of the basic SymmSpMV kernel for upper triangular matrices stored in Compressed Row Storage (CRS) [7] format. The kernel exploits the symmetry of the matrix ($A_{ij} = A_{ji}$) to reduce storage size and overall memory traffic, which is known to be pivotal hardware bottleneck for this operation on all modern compute devices. However, SymmSpMV cannot be parallelized easily as different threads working on different rows in parallel could potentially write to the same element $b[col[idx]]$, causing write conflicts. In terms of graph theory this means a vertex (row in a matrix) and its distance-2 neighbors [9] cannot be operated on in parallel. Here we concentrate on such distance-2 dependency problems, although the underlying method and library is capable of handling the general case of distance-k dependencies as well.

A popular approach to solve the above problem is multi-coloring (MC). The earliest work on coloring is the red-black Gauss-Seidel scheme [6], which was applied to matrices with a known regular sparsity pattern. Later multi-coloring techniques were expanded using graph theory for general sparse matrices [10, 15]. Recent variants like algebraic block multi-coloring (ABMC) [14] tried to improve the performance of MC methods. In [8], MC was applied to the Kaczmarz iterative solver [16], which has the same distance-2 dependency as SymmSpMV. Specifically for SymmSpMV there has been no previous attempt to use multi-coloring techniques. General solutions for SymmSpMV are lock-based methods and thread-private target arrays [5, 11]. Depending on the matrix structure these solutions can lead to performance degradation due to serialization and massive increase in data traffic. Recent research in this direction uses specialized storage formats like CSB [2] or RSB [20], but this requires rewriting of existing code and substantial tuning efforts.

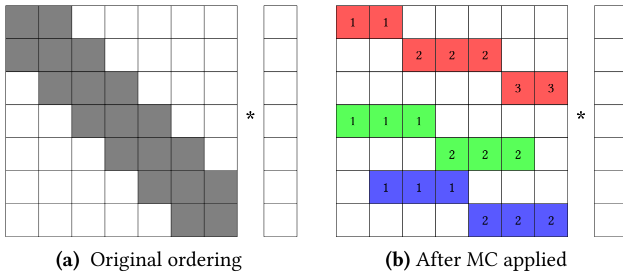
3 Uniqueness of the Approach

Multi-coloring methods can extract parallelism for kernels with data dependencies like SymmSpMV. For distance-2 coloring of a matrix, MC groups rows that do not overlap in

Algorithm 1 SymmSpMV kernel, $b = Ax$, in CRS format.

```
   //Loop over all matrix rows
1: for row = 1 : nrows do
2:     diag_idx = rowPtr[row]
3:     b[row]+ = A[diag_idx] * x[row]
       //Loop over all non-zero entries in a row
4:     for idx = rowPtr[row] + 1 : rowPtr[row + 1] do
5:         b[row]+ = A[idx] * x[col[idx]]
6:         b[col[idx]]+ = A[idx] * x[row]
```

ACM-SRC Grand Finals, 2019, USA
2019.



(a) Original ordering **(b)** After MC applied

Figure 1. Illustration of data locality degradation due to MC. Numbers represent thread id. Note that this figure shows only rows of matrix permuted according to MC, but in practice one would permute both rows and columns.

any column entries [9] (structurally orthogonal rows). These groups of rows are referred to as colors and parallelization can be done across the rows of a color (see Figure 1 for a simple example). However this process comes at the cost of destroying data locality in the matrix by the required permutations. In the SymmSpMV example (see algorithm 1) threads within a color operate on different rows having entirely different $col[idx]$ avoiding write conflicts in b vector. Note, that within a color (for e.g., red) none of the rows share same column index. As the matrix is traversed row by row (see algorithm 1) the original matrix has good data locality and most of the indirect vector accesses ($x[col[idx]]$ and $b[col[idx]]$) correspond to nearby elements that were loaded in the computation of previous rows. This ensures these vectors needs to be loaded only once from main memory, and the rest of the accesses are served by fast caches. However coloring the matrix destroys this data locality. For example in Figure 1b computing all the red colored rows leads to loading the entire vector completely. If the cache holds only six elements, computation on green and blue rows require loading almost the entire vector again from the slow main memory.

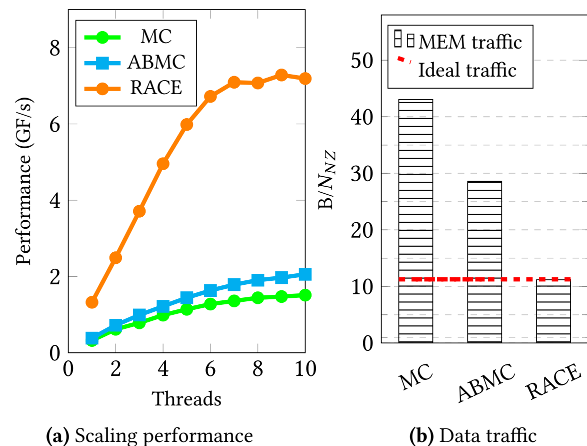
Destroying data locality along with secondary effects like synchronization costs and false sharing may, thus lead to severe performance degradation for MC methods. We demonstrate the impacts on performance and data transfer volumes for the SymmSpMV computations in Figure 2 for a single 10-core Intel Ivy Bridge EP (E5-2660 v2) CPU clocked at 2.2 GHz. The experiment was done on a large (number of rows = 10400600) `Spin-26` [24] matrix taken from quantum physics application. We find that performance of MC methods scale decently within a socket but are far off the RACE performance which saturates main memory bandwidth at 6-7 cores (Figure 2a). The reason for the large performance difference is given in Figure 2b which shows the average main memory data traffic per non-zero of the general matrix during SymmSpMV execution. It can be clearly seen that the memory traffic is almost 4× higher for the MC method compared to ideal traffic (red line) predicted by an appropriate performance model[1]. The extra data traffic is mainly due to the low data locality and thereby incurred extra accesses of the indirectly accessed vectors. Algebraic block multi-coloring (ABMC) tries to reduce the memory traffic by first partitioning the matrix into blocks and then applying coloring. This improves (reduces) the data traffic compared to MC but is still far from optimal in this case.

As main memory bandwidth is the main bottleneck on modern compute devices, this extra traffic reflects directly on the performance. This is seen in Figure 2a, where the performance is shown in giga floating point operations in seconds (GF/s). The ideal performance as predicted by performance model is ≈ 7.6 GF/s (not shown in figure) for this matrix, but MC and ABMC are well below this limit. However our RACE method closely approaches the ideal values both for the data traffic and performance and provides a speed-up of almost 4× compared to other methods.

4 RACE Method

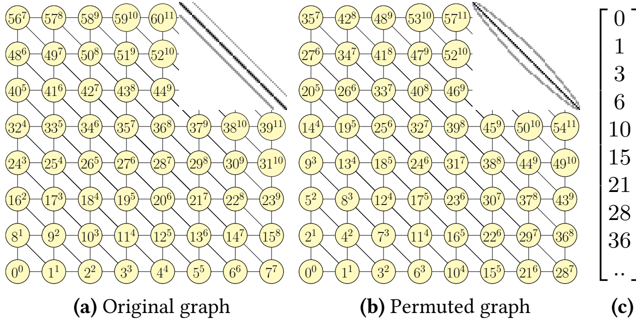
RACE was designed with the shortcomings of coloring approaches in mind. The idea is to have a general hardware-friendly approach applicable even for simple matrix storage formats like CRS. The RACE method consists of three steps: (1) level construction, (2) distance-k coloring, and (3) load balancing. Depending on the matrix and hardware the steps are applied recursively if required. To illustrate the method we choose a simple matrix which is associated with an artificially constructed two-dimensional-seven-point (2d-7pt) stencil. Figure 3a shows the corresponding graph and the

[1] See Section 5 for more details on modeling.



(a) Scaling performance **(b)** Data traffic

Figure 2. (a) Performance of SymmSpMV with MC and ABMC compared to RACE. (b) Average main memory data traffic in bytes (B) per nonzero entry (N_{nz}) of the full matrix as measured with LIKWID tool [26]. The ideal data traffic as predicted by performance model is shown for reference.



(a) Original graph **(b)** Permuted graph **(c)**

Figure 3. (a) Original graph of the 2d-7pt example, domain size 8×8. (b) Graph after permutation according to levels. The level numbers are denoted on the superscript of the vertices. Figures in inset show the corresponding sparsity pattern of the matrix. (c) `level_ptr`

sparsity pattern (see inset) of the matrix. In this paper we restrict ourselves to matrices representing strongly connected undirected graphs.

4.1 Level Construction

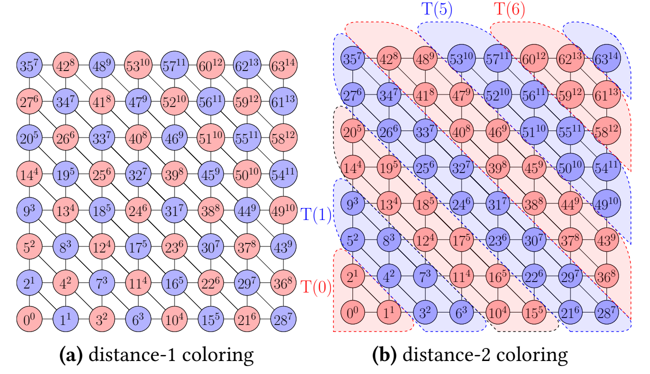
In the first step we determine the levels of a graph and permute the data structure accordingly. Here, we use well-known bandwidth reduction algorithms like Reverse Cuthill McKee (RCM)[3] or Breadth-first search (BFS)[19]. Although RCM is implemented in RACE, in the following we apply BFS reordering for better illustration. We start with choosing a root vertex and assign it to the first level ($L(0)$). The next levels $L(i)$ are defined to contain all vertices that are directly related to the previous level $L(i-1)$ but are not in $L(i-2)$. This implies that the i-th level consists of all vertices that have a minimum distance of i from the root node. In Figure 3 the level numbers (i) are denoted in the superscript of the vertices.

After the levels are determined we permute (reorder) the matrix (and graph) according to the levels such that the vertices in $L(i)$ appear before $L(i+1)$. Figure 3b shows the graph and matrix after applying the permutation. Note that the vertex numbering in the permuted graph has changed compared to the original lexicographically ordered matrix. It is well known that such a permutation improves data locality, and it was previously applied to sparse matrix computations without dependencies [21].

In order to resolve dependencies, RACE additionally keeps information about the levels by storing the index of the first vertex corresponding to each level in a data structure called `level_ptr` (see Figure 3c).

4.2 Distance-k Coloring

The distance-k coloring step uses the information of the `level_ptr` to resolve dependencies. Two vertices are called



(a) distance-1 coloring **(b)** distance-2 coloring

Figure 4. Example of distance-1 and distance-2 coloring of the matrix shown in Figure 3

distance-k neighbors if the shortest path connecting them consists of at most k edges [9]. This implies two vertices are distance-k independent if they are not distance-k neighbors. Based on this definition it can be proven that vertices between levels $L(i)$ and $L(i \pm (k+j))$ are distance-k independent $\forall j \geq 1$. The levels that satisfy this criterion are called distance-k independent levels.

The above approach allows for many choices to form distance-k independent levels. Figure 4 shows one such possibility for distance-1 and distance-2 coloring each. As $L(i)$ and $L(i \pm 2)$ are distance-1 independent, the distance-1 coloring assigns two colors to alternating levels. In case of distance-2 we group two adjacent levels and apply distance-1 coloring to the groups. These groups of levels are called level-groups and the $i-th$ level-group is denoted as $T(i)$ (see Figure 4b). For distance-1 coloring shown in Figure 4a the levels and level-groups coincide ($L(i) = T(i)$). In both cases all the vertices between level-groups of same color are distance-1/distance-2 independent and can be executed in parallel. For example, in case of distance-2, level-groups $T(0), T(2), T(4)$ and $T(6)$ can be executed by four threads in parallel. After synchronization the remaining four blue level-groups can be executed in parallel. Note that within a level-group/level the vertices are computed serially without destroying any data locality.

Choosing the same number of levels per level-group may cause severe load imbalance depending on the matrix. For example, in Figure 4b level-groups at extreme ends $T(0), T(7)$ have a relatively low number of vertices (proportional to computational work) compared to the level-groups in the middle ($T(3), T(4)$).

4.3 Load Balancing

RACE applies a load-balancing scheme among the threads within each color. It generates just the right number of level-groups as required by the hardware (i.e., the number of available threads) and then applies a load-balancing algorithm that minimizes the variance among the number

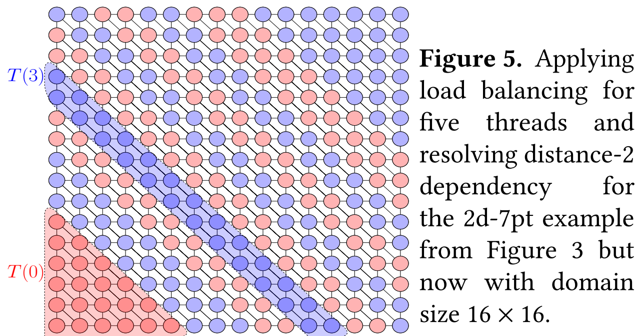


Figure 5. Applying load balancing for five threads and resolving distance-2 dependency for the 2d-7pt example from Figure 3 but now with domain size 16×16.

of vertices within level-groups of the same color. To this end, level-groups containing few vertices grab adjacent levels from neighboring level-groups; overloaded level-groups shift levels to adjacent level-groups. To maintain distance-k independence between level-groups of the same color the algorithm enforces at least k levels per level-group. This shifting process is applied iteratively until it reaches the minimum possible variance or no further moves are possible due to distance-k coloring.

Figure 5 shows the graph of the 2d-7pt example at size 16×16 after load balancing. Here distance-2 coloring and five threads (i.e., ten level-groups) were the input to the load balancer. Note that level-groups at extreme ends (e.g., $T(0)$) have more levels since here each level has fewer vertices, whereas bigger level-groups (e.g., $T(3)$) in the middle maintain two levels to preserve distance-k coloring.

4.4 Recursion

However, using the steps above the generated parallelism is limited by the total number of levels, and the load balancing can be a problem as it gets closer to this limit. To match the high levels of parallelism of modern compute devices we use recursion. The formulation of RACE allows to simply select a level-group (sub-graph) and apply the three steps recursively on this sub-graph to exploit the parallelism within this level-group. The thread that was originally assigned to the level-group spawns other parallel threads in a nested manner. The selection of the level-group to be used for recursion and final load balancing is done by a global load balancing algorithm.

5 Results & Contribution

We evaluate the performance of RACE by parallelizing the SymmSpMV kernel shown in Algorithm 1. This allows a clear picture of the performance advantage of RACE. Finally, we use RACE to parallelize an eigenvalue solver, and compare against standard approaches.

5.1 Analysis of SymmSpMV Performance

Matrix-vector multiplication is frequently used in numerical algorithms. In many cases, however, the lack of an efficient and generic SymmSpMV implementation leads to the full (general) matrix being stored and used even if it is symmetric, which wastes not only CPU cycles but also memory. Modern HBM (High Bandwidth Memory) technology with its rather limited memory sizes makes this problem even more severe.

In this section we carry out experiments using a SymmSpMV kernel. We choose most of the test matrices from the public SuiteSparse Matrix Collection [4], that are frequently used in related publications [20, 22], as well as some from the quantum physics context in which RACE was developed [1]. The experiments are run on one Intel Skylake SP Gold 6148 CPU (20 threads) at a fixed clock speed of 2.4 GHz. The reported performance is purely for the SymmSpMV computation as in practical applications these kernels are called multiple times, making other costs (like setup time) negligible.

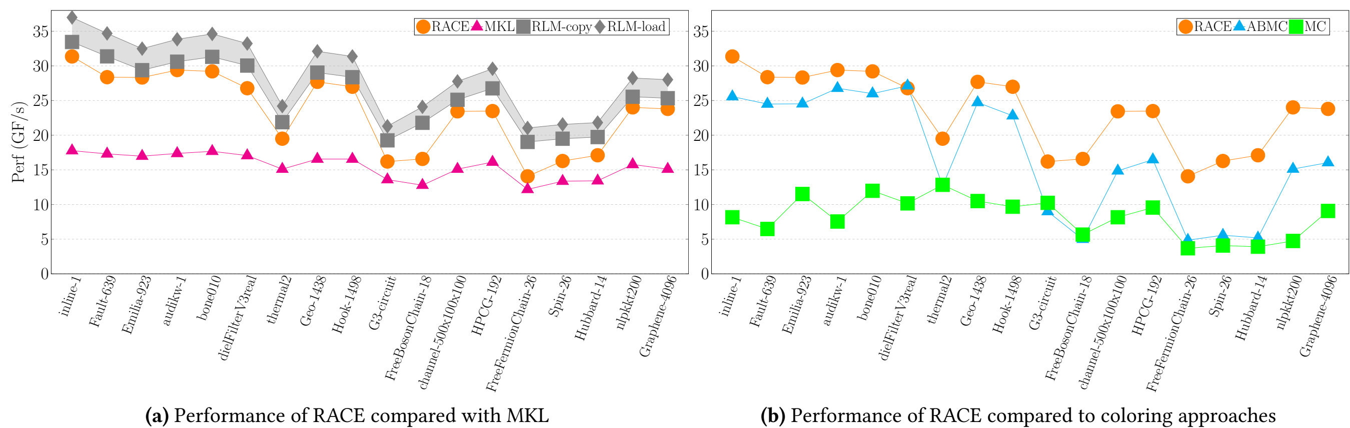
To establish a sensible performance baseline we use the Roofline model (RLM) [27] along the lines of [18] but adjusted for the SymmSpMV kernel. Figure 6a shows the performance of RACE on different matrices along with the range of upper performance bounds based on two saturated memory bandwidth measurements (RLM-load and RLM-copy). In almost all cases, RACE attains more than 85% of the possible maximum. This can be attributed to the good data locality and minimal data traffic, which we have already demonstrated in Figure 2b for the `Spin-26` matrix.

In Figure 6a we compare against the SymmSpMV implementation of the latest version of Intel MKL [13], which uses the Inspector-Executor routines. The comparisons show that RACE outperforms MKL by a factor of 1.5× on an average. A simple analysis shows that the performance of the Intel MKL SymmSpMV kernel coincides with the matrix-vector multiplication using the full matrix (SpMV). We can only speculate (due to it being closed source) that MKL converts the symmetric matrix to a full matrix internally and then does a general SpMV operation.

We also compare RACE with two widely used coloring methods, MC and ABMC. For MC we apply the multi-coloring scheme generated by the COLPACK [10] library to parallelize the SymmSpMV kernel. In the ABMC method we first partition the matrix into blocks using METIS [17] and then apply coloring via COLPACK. The size of blocks was determined by a parameter scan (range $4 \ldots 128$, see [14]). Figure 6b shows the resulting performance data. Overall the MC method is not competitive, while ABMC delivers modest performance (85% of RACE) for small matrices. For large matrices, however, where data locality plays a vital role, ABMC falls substantially behind RACE as the indirect access to the vectors impairs temporal and spatial access locality. Overall, RACE shows an average speedup of 1.6× compared to ABMC, while in some cases the speedup is as high as 3×.

5.2 FEAST with RACE

FEAST[23] is a modern algorithm to compute inner eigenvalues. It uses contour integration to generate a subspace



(a) Performance of RACE compared with MKL

(b) Performance of RACE compared to coloring approaches

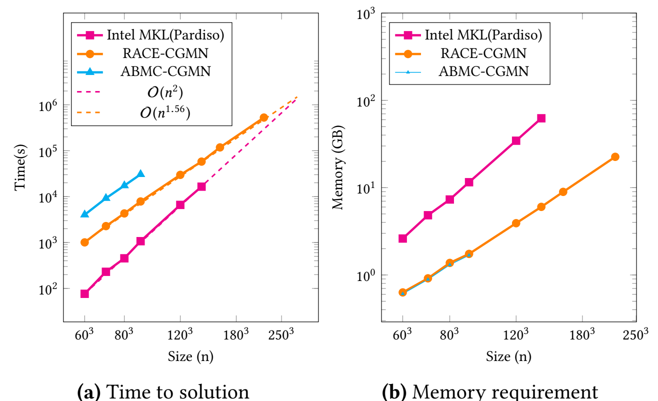
Figure 6. SymmSpMV performance of RACE compared to other methods. The Roofline model for SymmSpMV is shown in fig. 6a for reference. Note that the matrices are ordered according to increasing number of rows.

(reduced system) containing the eigenvalues, which are then solved using a classic Rayleigh-Ritz procedure. The solver is well-suited for very large sparse systems, and is of particular interest in the field of quantum mechanics. The hot spot of the algorithm (more than 95%) is a solver for shifted linear systems ($A - \sigma I = b$). These systems are, however, highly ill-conditioned, posing severe convergence problems for most linear iterative solvers. The standard approach is therefore to use direct solvers. However, in [8] it has been shown that the Kaczmarz iterative solver accelerated by a Conjugate Gradient (CG) method (the so-called CGMN solver [12]) is a robust alternative to direct solvers. Similar to SymmSpMV, the Kaczmarz method has a distance-2 dependency, making it difficult to parallelize. In [8], multi-coloring was used to parallelize the kernel. We have implemented a shared-memory parallel version of CGMN with RACE for use in FEAST.

We use the FEAST implementation of Intel MKL, which by default employs the PARDISO direct solver [25], but its Reverse Communication Interface (RCI) allows us to plug our CGMN implementation instead. In the following experiment we find ten inner eigenvalues of a simple discrete Laplacian matrix to an accuracy of 10^{-8}. Figure 7 shows the measured time and memory footprint of the default MKL version (using PARDISO) and the CGMN versions parallelized using both RACE and ABMC for different matrix sizes. In line with the observations in Section 5.1, ABMC is a factor of 4× slower than RACE. The time required by the default MKL with PARDISO is smaller than with CGMN using RACE for small sizes; however, the gap gets smaller as the size grows due to the direct solvers having a higher time complexity (here $\approx O(n^2)$, see Figure 7) compared to iterative methods ($\approx O(n^{1.5})$). Moreover, the direct solver requires more memory, and the memory requirement grows much faster (see Figure 7(b)) than with CGMN. In our experiment the direct solver ran out of the memory at problem sizes beyond 140^3, while CGMN using RACE used less than 10% of space at this point. Thus, CGMN with RACE can solve much larger problems compared to direct solvers, which is a major advantage in fields like quantum physics.

Conclusion

We have shown the parallelization problems that commonly occur in sparse computations and discussed on the drawbacks of existing approaches. We then introduced and described the RACE method, its novelties and how it mitigates the shortcomings of existing methods. Finally we have seen the application of the method in numerical linear algebra for achieving high performance and solving large problems on modern compute devices.



(a) Time to solution

(b) Memory requirement

Figure 7. Comparison of FEAST with default Intel MKL direct solver and iterative solver CGMN, parallelized using RACE and run on one Skylake SP Platinum 8160 CPU (24 threads).

Acknowledgments

We would like to thank Georg Hager, Olaf Schenk, Jonas Thies and Thomas Gruber for providing valuable insights, comments and technical support throughout the work. We gratefully acknowledge the compute resources and support provided by the Erlangen Regional Computing Center (RRZE) and RWTH Aachen.

References

[1] 2016. Equipping Sparse Solvers for Exascale - ESSEX-II. https://blogs.fau.de/essex/activities.

[2] 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

[3] 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

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

[5] 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

[6] 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

[7] William F. Tinney and John W. Walker. 1967. Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization. *Proc. IEEE* 55 (12 1967), 1801 – 1809. https://doi.org/10.1109/PROC.1967.6011

[8] 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

[9] 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

[10] 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

[11] 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

[12] Dan Gordon and Rachel Gordon. 2008. CGMN revisited: robust and efficient solution of stiff linear systems derived from elliptic partial differential equations. *ACM Trans. on Mathematical Software* (2008).

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

[14] 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

[15] 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

[16] S. Kaczmarz. 1937. Angenäherte Auflösung von Systemen linearer Gleichungen. *Bulletin International de l'Académie Polonaise des Sciences et des Lettres* 35 (1937), 355–357.

[17] 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

[18] 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

[19] 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

[20] 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

[21] L. Oliker, X. Li, P. Husbands, and R. Biswas. 2002. Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations. *SIAM Rev.* 44, 3 (2002), 373–393. https://doi.org/10.1137/S00361445003820

[22] 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

[23] Eric Polizzi. 2009. A Density Matrix-based Algorithm for Solving Eigenvalue Problems. *CoRR* abs/0901.2665 (2009). arXiv:0901.2665

[24] Röhrig-Zöllner, Melven and Thies, Jonas and Kreutzer, Moritz and Alvermann, Andreas and Pieper, Andreas and Basermann, Achim and Hager, Georg and Wellein, Gerhard and Fehske, H. 2015. Increasing the Performance of the Jacobi–Davidson Method by Blocking. *SIAM Journal on Scientific Computing* 37 (12 2015), C697–C722. https://doi.org/10.1137/140976017

[25] O. Schenk, K. Gärtner, and W. Fichtner. 2000. Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors. *BIT Numerical Mathematics* 40, 1 (01 Mar 2000), 158–176. https://doi.org/10.1023/A:1022326604210

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

[27] 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