Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs

James D. Trotter
james@simula.no
Simula Research Laboratory
Oslo, Norway

Sinan Ekmekçibaşı
sinanekmekcibasi@gmail.com
Istanbul University-Cerrahpaşa
Istanbul, Turkey

Johannes Langguth
langguth@simula.no
Simula Research Laboratory
Oslo, Norway
University of Bergen
Bergen, Norway

Tugba Torun
ttorun@ku.edu.tr
Koç University
Istanbul, Turkey

Emre Düzakın
eduzakin18@ku.edu.tr
Koç University
Istanbul, Turkey

Aleksandar Ilic
aleksandar.ilic@inesc-id.pt
INESC-ID, IST, Universidade de Lisboa
Lisbon, Portugal

Didem Unat
dunat@ku.edu.tr
Koç University
Istanbul, Turkey

ABSTRACT

Many real-world computations involve sparse data structures in the form of sparse matrices. A common strategy for optimizing sparse matrix operations is to reorder a matrix to improve data locality. However, it's not always clear whether reordering will provide benefits over the unordered matrix, as its effectiveness depends on several factors, such as structural features of the matrix, the reordering algorithm and the hardware that is used. This paper aims to establish the relationship between matrix reordering algorithms and the performance of sparse matrix operations. We thoroughly evaluate six different matrix reordering algorithms on 490 matrices across eight multicore architectures, focusing on the commonly used sparse matrix-vector multiplication (SpMV) kernel. We find that reordering based on graph partitioning provides better SpMV performance than the alternatives for a large majority of matrices, and that the resulting performance is explained through a combination of data locality and load balancing concerns.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Computing methodologies** → **Shared memory algorithms**; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

matrix reordering, sparse matrix-vector multiply, multicore, graph partitioning

ACM Reference Format:
James D. Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3581784.3607046

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607046

1 INTRODUCTION

Sparse matrices arise from a wide variety of problems in scientific computing, graph theory, finance, and deep learning. Sparse matrix reordering is an optimization technique used to improve the efficiency of operations on sparse matrices by rearranging their rows and columns. Matrix reordering serves many purposes. It can be used to achieve lower work and storage requirements, improve data locality and cache reuse, expose additional parallelism or improve the effectiveness of other optimization techniques. Sparse direct solvers rely heavily on appropriate ordering to reduce fill-in during factorization, whereas iterative solvers can benefit from reordering through improved data locality.

Various reordering algorithms [1, 4, 8, 9, 14, 19, 28] have been proposed over the years. In the case of sparse direct solvers, it is well known that the right ordering can drastically reduce the number of operations required to perform factorisation [2]. For sparse matrix-vector multiplication (SpMV), one of the most frequently encountered sparse matrix operations, there are some examples of reordering being used to significantly improve performance (e.g., by a factor of 3.6× [28]). But most experimental evaluations are carried out with only a small number of matrices and reveal a mere

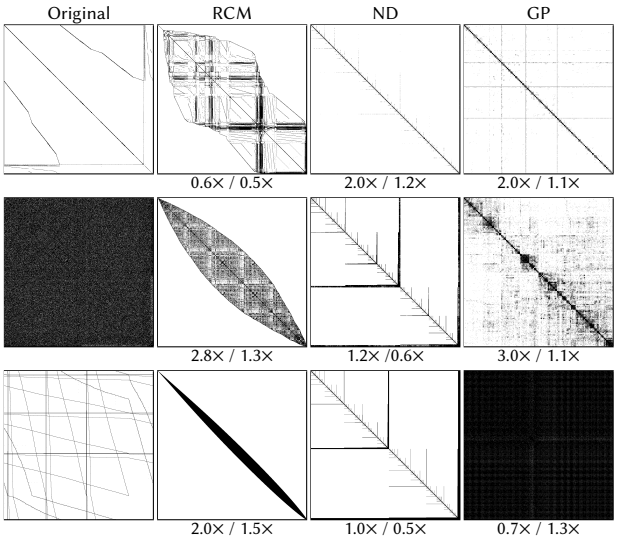

Figure 1: Matrices reordered with Reverse Cuthill-McKee (RCM), Nested Dissection (ND) and graph partitioning (GP). The numbers below represent speedup (or slowdown) of SpMV on 64-core AMD Epyc Milan and 36-core Intel Ice Lake CPUs, respectively.

10 % improvement in SpMV performance for commonly used ordering strategies [15, 23, 24]. Reordering faces several challenges, including the difficulty of finding an optimal ordering, matrices already having an efficient ordering, or the new ordering causing performance degradation by introducing load imbalance in parallel computations. Additionally, orderings that benefit one architecture may not be useful or even harmful for others.

To demonstrate this with a concrete example, Figure 1 displays a few matrices with their original sparsity patterns, along with their patterns after applying three frequently used reorderings. Additionally, the figure indicates the speedup (or slowdown) of SpMV over the unordered matrix in two different platforms. The figure highlights three main observations: (i) different reorderings lead to a diverse distribution of matrix nonzeros, which consequently results in significant performance improvement or degradation depending on the algorithm-matrix pair; (ii) although a reordering algorithm can enhance the performance of one matrix, it may reduce the performance of another; (iii) the efficacy of a reordering algorithm depends on the architecture.

In this paper, our primary objective is to offer a comprehensive analysis of the confusing and often contradictory results pertaining to matrix reorderings. To our knowledge, we present the most extensive performance study to date, involving 490 large matrices, six reordering algorithms, and eight modern multicore architectures commonly used in high-performance computing. Our evaluation employs two SpMV implementations: a standard 1D SpMV algorithm that performs an even row split, and a 2D algorithm that balances the number of nonzeros, thereby highlighting the effect of reordering on a widely used sparse computational kernel.

To explain the performance behaviors, we devise a set of metrics and order-dependent matrix features and attempt to correlate them with reordering performance. These features include objectives that matrix reordering algorithms attempt to minimise, such as matrix bandwidth, profile and others, as well as the amount of load imbalance in parallel SpMV.

Our key findings and contributions are:

(1) While reordering for SpMV can yield extreme speedups or slowdown ranging from 0.05 to 40×, the most typical result lies in the range 0.5 to 1.5×.
(2) Reordering based on graph and hypergraph partitioning provide far better SpMV performance than the alternatives for both 1D and 2D algorithms.
(3) Overall, reordering algorithms perform similarly across different hardware architectures, in spite of individual matrices whose performance is hardware-sensitive. This highlights the choice of reordering algorithm as a means to attain cross-architecture performance gains.
(4) The impact in load balance and data locality as a result of reordering can be used to categorise individual matrices and thus explain performance improvement or degradation.
(5) Among the matrix features considered, the most important one for SpMV performance coincides with the edge-cut objective used for graph partitioning-based ordering.
(6) Reordering time may vary by several orders of magnitude, but in general Gray ordering is fastest, and Reverse Cuthill-McKee (RCM) is the second fastest.

We anticipate that these findings will be valuable for programmers and algorithm designers who wish to make informed decisions about employing reordering algorithms for their workloads.

In the rest of the paper, Section 2 provides some background on various matrix reordering algorithms. Section 3 describes our methodology, as well as matrix features and criteria that are used to evaluate the reorderings. Next, Section 4 presents experimental results from matrix reordering and our findings. Section 5 discusses related works before presenting our conclusions in Section 6.

2 BACKGROUND

Before giving background on reordering algorithms, let us define some key terminology. Many reordering strategies arise from linear solvers where certain features of sparse matrices are desirable. The *bandwidth* of a sparse matrix is the width of the diagonal band that contains its nonzero elements, whereas the *profile* is a sum of distances from the leftmost element to the diagonal of each row (see Section 3.2). In sparse matrix factorization, *fill-in* refers to the appearance of additional nonzero elements in the factorization compared to the original matrix.

2.1 Reordering Algorithms

To provide an overview, we categorise reordering algorithms into 1) bandwidth-reducing orderings, 2) fill-reducing orderings 3) graph and hypergraph partitioning-based orderings, and 4) other orderings.

2.1.1 Bandwidth-Reducing Orderings. Well-known examples of such orderings include the Cuthill-McKee (CM) algorithm [4] and the method described by Gibbs et al. [12]. The CM ordering attempts to reduce the matrix bandwidth through a breadth-first search of

the undirected graph corresponding to a symmetric sparse matrix. The vertices of the graph, which correspond to rows and columns of the matrix, are ordered by choosing a starting vertex (e.g., by finding a pseudo-peripheral vertex [10]) and then traversing the graph in breadth-first search order, where the vertices at each level are sorted in ascending order by degree. In the end, after traversing the entire graph, the ordering may be reversed to obtain the more commonly used Reverse Cuthill-McKee (RCM) [19] ordering.

2.1.2 Fill-Reducing Orderings. Minimum degree orderings [1, 9, 11] arise in the context of reducing fill-in during sparse Cholesky factorisation. The elimination graph of a sparse symmetric matrix consists of a vertex for every row, as well as edges between any pair of vertices a and b for which row a has a nonzero above the diagonal in column b. At each step of the factorisation, one row is eliminated by removing a vertex and its edges in the elimination graph, and replacing it with a clique consisting of the former neighbours of the vertex. The new edges that are created by forming such a clique lead to fill-in of the Cholesky factor. The minimum degree algorithm is a graph-based heuristic to find node orderings with low amounts of fill by always selecting a vertex of least degree.

Another commonly used fill-reducing ordering is Nested dissection (ND) [8, 14], which is based on computing a vertex separator for the undirected graph of a symmetric sparse matrix. The two subgraphs that arise from removing the separator are ordered first, while rows and columns corresponding to the separator are moved to the end of the matrix. This process is applied recursively for the two subgraphs. The underlying motivation for the ND ordering is that it incurs low fill-in for sparse Cholesky factorization if the separators are small. Since the method relies on graph partitioning, it can be grouped under graph partitioning-based orderings as well.

2.1.3 (Hyper)graph Partitioning-based Orderings. Graph partitioning can be used to define an ordering by directly partitioning a matrix into a given number of parts, then grouping rows and columns by their assigned parts. This approach is frequently used in a distributed-memory setting to perform work division of sparse matrix operations, and the same idea can be applied to the shared-memory case. METIS [18] is a well-known graph partitioning tool that can be used to partition large irregular graphs. It is based on the multilevel paradigm which consists of the graph coarsening, initial partitioning, and uncoarsening phases. The aim of the partitioning is to minimize a partitioning objective, while obeying a load balancing criteria.

Hypergraph partitioning may similarly be used for reordering. PaToH [3] is a commonly-used hypergraph partitioning tool which is known to reflect the actual communication volume requirement of parallel SpMV. Hypergraphs are the generalization of graphs, in which the hyperedges (nets) can be incident to any number of vertices instead of exactly two vertices in simple graphs. The hypergraph partitioning problem is the task of dividing a hypergraph into roughly balanced parts such that the cut size is minimized. Other reorderings based on hypergraph partitioning include the separated block diagonal form proposed by Yzelman and Bisseling [27].

2.1.4 Other Orderings. A number of alternative matrix orderings have been proposed with the goal of improving data locality in SpMV, including approaches based on the travelling salesperson problem [17, 24] and space-filling curves [21]. One particular method, which we call Gray ordering [28], is motivated by microarchitectural concerns to reduce branch mispredictions and improve data locality for SpMV. First, to improve branch prediction, rows with similar nonzero density are grouped together (density reordering). Second, to improve locality, a bitmap-based reordering is applied, where each row is segmented into multiple sections of nonzeros (to construct the row bitmaps), which are then labeled and ordered based on the Gray code [16]. In general, the matrix is first split into dense and sparse submatrices according to the number of nonzeros in each row, while the density and bitmap reorderings are applied depending on the characteristics of each submatrix.

3 METHODOLOGY

This section outlines two SpMV algorithms used to evaluate various reordering algorithms, then introduces matrix features that are sensitive to ordering and may therefore relate to SpMV performance with different orderings. Finally, we describe the matrix reordering algorithms used in this study.

3.1 SpMV Kernels

We aim to study how matrix reordering affects the performance of sparse matrix-vector multiplication (SpMV) with shared-memory parallel kernels based on the popular compressed sparse row (CSR) format.

A sparse matrix A with M rows and N columns is defined by its K nonzeros, which we denote a_{i_k,j_k} for $k = 1, 2, \ldots, K$, where i_k and j_k are the row and column offsets of the kth nonzero, respectively. Any sparse matrix storage format must somehow store the row and column offsets as well as the nonzero values. The well-known CSR format groups nonzeros by rows in ascending order, and then compresses the row offsets to form row pointers, $r_1, r_2, \ldots, r_{M+1}$, such that r_i and $r_{i+1} - 1$ indicate the location of the first and last nonzeros of row i, respectively. Thus, multiplying A by a vector x to obtain another vector y amounts to computing the sum $y_i \leftarrow y_i + \sum_{k=r_i}^{r_{i+1}-1} a_{i_k,j_k} x_{j_k}$, for every row $i = 1, 2, \ldots, M$.

The standard method for performing the above SpMV computation in parallel is by partitioning the rows into equal-sized, contiguous blocks and assigning one block to each thread. (This is easily achieved in OpenMP with a single `#pragma omp for` directive.) We refer to this as the *1D algorithm*. Although this scheme is simple and works well in some cases, it suffers from load imbalance for many realistic sparse matrices due to the nonzeros being unevenly divided among threads.

As a result, we also consider a second SpMV kernel which is still based on the CSR storage format, but offers a more balanced workload among threads. Rather than partitioning the rows, we instead perform an equal partitioning of the matrix nonzeros. This produces a balanced workload among threads, at least in terms of nonzeros. Loop scheduling must now be performed manually, since the built-in loop scheduling directives of OpenMP are no longer sufficient. In addition, this approach requires each thread to handle its first and final row specially to avoid race conditions when updating the output vector y. We call this the *2D algorithm*.

Table 1: Sparse matrix reordering algorithms used in this study

Short Name	Reordering Algorithm	Description
RCM [19]	Reverse Cuthill–McKee	bandwidth reduction via breadth-first graph traversal
AMD [1]	Approximate minimum degree	local greedy strategy to reduce fill by selecting sparsest pivot row
ND [18]	Nested dissection	recursive divide-and-conquer using vertex separators to reduce fill
GP [18]	Graph partitioning	multi-level recursive graph partitioning with METIS using edge-cut objective
HP [3]	Hypergraph partitioning	column-net hypergraph partitioning with PaToH using cut-net metric
Gray [28]	Gray code ordering	splitting of sparse and dense rows and Gray code ordering

Our 2D algorithm is closely related to and may be considered a simplified version of the merge-based SpMV kernel of Merrill and Garland [20]. Both of these kernels entail a small preprocessing cost to find the balanced partitioning. Even for the more elaborate merge-based kernel, this cost is small enough to keep 2D algorithms competitive. Furthermore, for a given matrix and architecture, this represents a one time cost and can thus easily be amortized over multiple SpMV iterations. Therefore, we ignore this cost in our measurements.

We use our own implementation of the 1D and 2D algorithms. Parallelization is achieved via OpenMP using `static` scheduling. Since our experimental platforms are large NUMA machines, we use the first-touch policy to ensure that the data is placed close to the core using it.

3.2 Matrix Features for Reordering

This section defines four matrix features that depend heavily on matrix ordering. These metrics are *bandwidth, profile, off-diagonal nonzero count*, and *load imbalance factor*, and they are later used to explain the effect of reordering particularly with respect to SpMV performance.

The *bandwidth* and *profile* give an indication of whether nonzeros are clustered near the main diagonal, which in turn may lead to better data locality for SpMV [25]. For an N-by-N sparse matrix A, the bandwidth is the largest distance of any nonzero to the main diagonal, $\max_{a_{i,j}\neq 0} |i-j|$, whereas the profile [12] is a sum over every row of the distance from the leftmost entry to the diagonal, $\sum_{i=1}^{N} i - \min\left\{j \mid a_{i,j} \neq 0\right\}$.

Assuming that the matrix is partitioned into N-by-N equal-sized blocks, we count the total number of nonzeros that do not fall into any diagonal blocks. We call this the *off-diagonal nonzero count*, and it is essentially the same as the edge-cut metric which is minimised by graph partitioning, if one assumes that rows are divided equally among the threads, as in the 1D SpMV algorithm.

Finally, to capture effects of load imbalance in shared-memory parallel SpMV, we also consider a *load imbalance factor*. This is defined as the ratio of the maximum number of nonzeros assigned to a single thread to the average number of nonzeros per thread. Thus, if every thread has the same number of nonzeros, the imbalance factor is 1. On the other hand, a thread having twice the number of nonzeros compared to the average yields an imbalance factor of 2.

3.3 Chosen Reordering Algorithms

For this study, we have chosen a diverse set of reordering algorithms with relatively different objectives for evaluation. Table 1 gives an overview of our chosen algorithms. We employ the Reverse Cuthill-McKee (RCM) algorithm [19], which is obtained by simply reversing the usual Cuthill-McKee ordering and is known to work better in practice when used for factorising symmetric, positive definite matrices.

From the fill-in reducing group, variations of the minimum degree reordering algorithm are often used in practice, such as multiple minimum degree (MMD) [9] and approximate minimum degree (AMD) [1]. We chose the latter that is based on merely approximating the degree of a vertex, which reduces the runtime complexity. For the AMD and ND orderings, we use reordering routines from SuiteSparse [1] and METIS [18] libraries, respectively.

Next, we include a graph partitioning-based reordering, which will be referred to as *GP*. For GP, we also use METIS, which offers two options for the partitioning objective. The first one is edge-cut, i.e., the number of edges connecting vertices in different partitions, and the second is total communication volume. With respect to partitioning a sparse matrix, the load balance criteria can be chosen to balance the number of rows or the number of nonzeros within the parts. The latter is achieved by weighting each vertex in the graph by the number of nonzeros in the corresponding row. For this study, we use the edge-cut objective and an unweighted graph which implies balancing the number of rows in each part. Moreover, the number of parts to be created by the partitioner is chosen to match the number of CPU cores for the hardware used in this study (see Table 2) by partitioning into 16, 32, 48, 64, 72 or 128 parts.

We also include a hypergraph partitioning reordering, which will be referred to as *HP*. In HP, we employ PaToH [3] with the column-net model in which the rows and columns of the coefficient matrix are represented with vertices and nets, respectively. PaToH includes two metrics that can be used as the partitioning objective, namely cut-net and connectivity metrics. In the column-net model, the former metric corresponds to minimizing the number of nonzero column segments, while the latter corresponds to minimizing the off-diagonal nonzero count. In HP, we adopt the 128-way partitioning of matrices using PaToH with cut-net metric and the same balancing criteria as in GP.

The final algorithm is the Gray code ordering using the parameters suggested by Zhao et al. [28], meaning that 16 bits are used for bitmap-ordering and rows with more than 20 nonzeros are considered to be dense.

The RCM, AMD, ND and GP reorderings assume a structurally symmetric matrix A. We therefore use the symmetrization, $A + A^T$, to obtain these reorderings whenever the sparsity pattern of A is unsymmetric. The HP and Gray reorderings, on the other hand, apply

Table 2: Hardware used in our experiments.

	Skylake	Ice Lake	Naples	Rome	Milan A	Milan B	TX2	Hi1620
CPUs	Intel Xeon Gold 6130	Intel Xeon Platinum 8360Y	AMD Epyc 7601	AMD Epyc 7302P	AMD Epyc 7413	AMD Epyc 7763	Cavium TX2 CN9980	HiSilicon Kunpeng 920-6426
Instr. set	x86-64	x86-64	x86-64	x86-64	x86-64	x86-64	ARMv8.1	ARMv8.2
Microarch.	Skylake	Ice Lake	Zen	Zen 2	Zen 3	Zen 3	Vulcan	TaiShan v110
Sockets	2	2	2	1	2	2	2	2
Cores	2×16	2×36	2×32	1×16	2×24	2×64	2×32	2×64
Freq. [GHz]	1.9–3.6	2.4–3.5	2.7–3.2	1.5–3.3	2.5–3.5	2.5–3.5	2.0–2.5	2.6
L1I/core [KiB]	32	32	64	32	32	32	32	64
L1D/core [KiB]	32	48	32	32	32	32	32	64
L2/core [KiB]	1024	1280	512	512	512	512	256	512
L3/socket [MiB]	22	54	64	16	128	256	32	64
Bandwidth [GB/s]	256	409.6	342	204.8	409.6	409.6	342	342

naturally to symmetric as well as unsymmetric matrices. With the exception of the Gray algorithm, the methods considered here produce symmetric reorderings, meaning that the same permutation is applied to the matrix rows and columns. For the Gray algorithm, only the matrix rows are reordered, and the reordering is therefore unsymmetric. Finally, while some matrices possess additional structure in their original ordering (e.g., being made up of small, dense blocks), these kinds of structures are not taken into account when reordering.

4 EVALUATION

4.1 Experimental Setup

The hardware used in our experiments is shown in Table 2. All codes are compiled with GCC 11.2.0 with the `-O3` and `-march=native` options on each node, and the test systems are running Ubuntu 18.04.6.

Our evaluation relies on the SuiteSparse Matrix Collection [5]. We apply six reorderings (see Table 1) to 490 matrices that are square, non-complex and have between 1 million and 1 billion nonzeros. On converting the matrices to CSR format, column offsets are stored as 32-bit integers and nonzero values as double precision floating point numbers. In the case of symmetric matrices, whenever an offdiagonal nonzero is encountered, two nonzeros are inserted into the CSR representation, one in the upper and another in the lower triangle of the matrix.

Each SpMV run is repeated 100 times, and we take the maximum performance among the runs. This represents the peak performance of a system with a warm cache and is less susceptible to noise than the average. Note that for smaller matrices used in our evaluation, some or all of the data may fit in last-level cache. For example, the AMD Epyc 7763 has the largest last-level cache at a total of 512 MiB. Only 77 matrices have more than 45 million nonzeros, which is the minimum size needed to exceed the capacity of the last-level cache if matrices are stored in CSR format.

4.2 Reordering for 1D SpMV

Prior to reordering, we observe that the performance of the 1D SpMV algorithm varies greatly from one matrix to another, as expected. For example, the typical range for the 128-core Milan B is about 50–120 Gflop/s with a median value of about 80 Gflop/s. As a reference, we measure SpMV performance for a tall-and-skinny, dense matrix of 96 000 rows and 4 000 columns stored in CSR format. This represents a case where the vectors fit in cache and are likely to benefit from prefetching due to the predictable access pattern. The matrix data, on the other hand, does not fit in cache and must therefore be read from main memory. The measured performance when using 128 cores on Milan B is about 53 Gflop/s or 317 GB/s, which is about 77 % of the theoretical peak memory bandwidth.

Our first experiment measures the SpMV speedup using the 1D algorithm for all reorderings compared to the original ordering, using all 490 matrices in the test set. Figure 2 illustrates the results for all architectures. Note that some outliers are not shown to save space.

The distribution of speedups vary considerably between different reordering techniques. On the other hand, the overall picture is roughly the same regardless of the hardware used. Every ordering has outliers with extreme speedup or slowdown. The greatest slowdown is a factor of 0.05×, whereas the largest speedup is about 40×. If we disregard outliers and consider only the portion between the lower and upper quartiles (i.e., the coloured boxes, which comprises half of the matrices and thus the most typical case), then the speedup ranges from about 0.5 to 1.5×.

With respect to the various orderings, the median speedups of RCM, GP and HP are greater than 1, meaning that SpMV performance improves for more than 50 % of the matrices. Additionally, GP is best with speedup for about 75 % of matrices on every CPU and more matrices achieving higher speedups. This is closely followed by HP, which shows slightly smaller speedups in general and performs noticeably worse on Naples in particular. Next, the median speedups of ND and AMD are close to 1 or slightly less than 1, respectively. These methods are thus equally likely to yield a speedup as they are to result in a slowdown. Finally, the Gray ordering stands out by resulting in slowdowns in most cases. On Skylake in particular, 75 % of matrices reordered with Gray experience a slowdown of 0.9× or worse.

We also compute the geometric mean over the speedups in order to provide a better overview. Results are given in Table 3. They

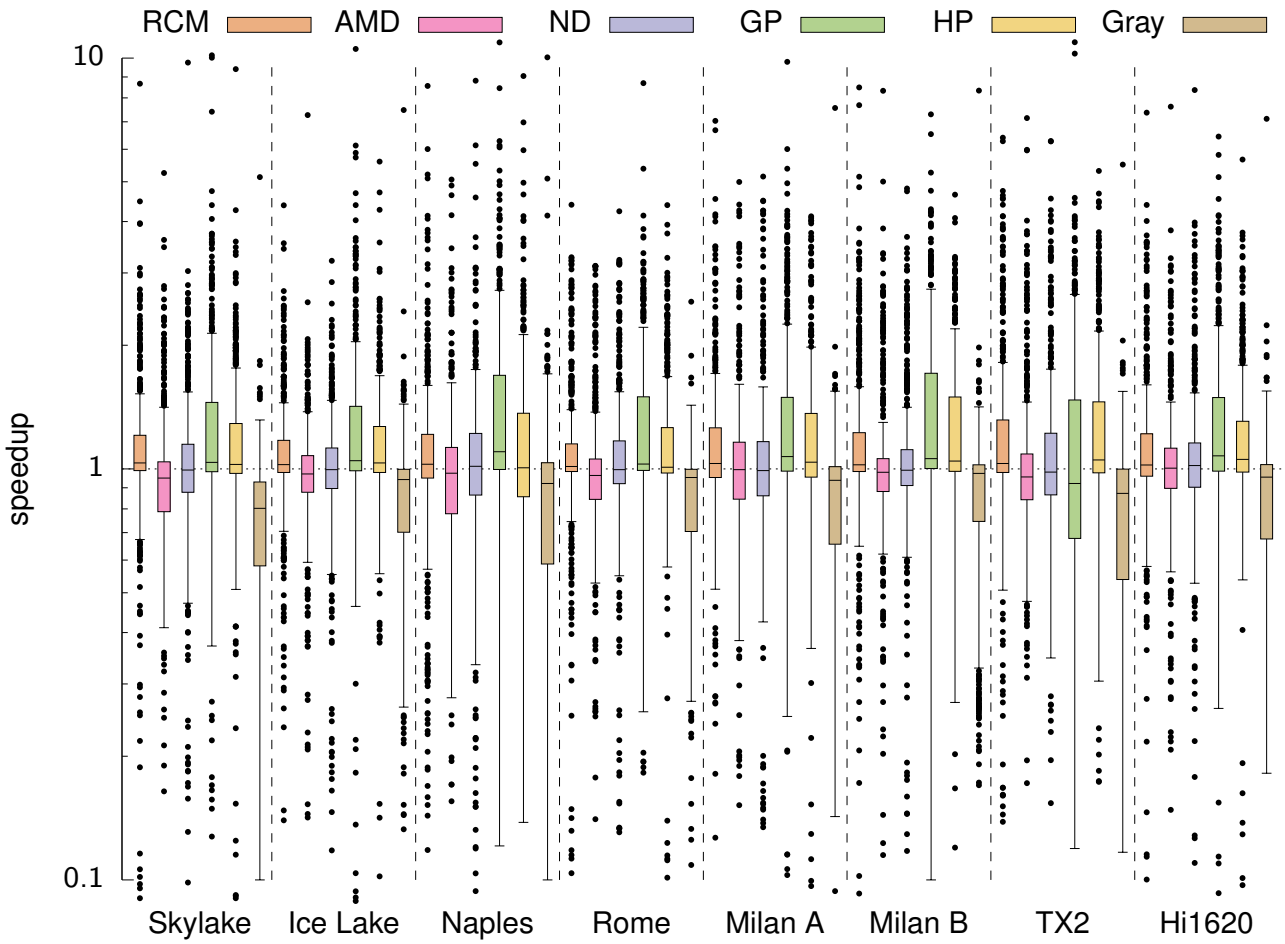

Figure 2: Speedup of sparse matrix-vector multiplication using 1D algorithm after reordering. For each box, the middle line represents the median and endpoints represent the lower and upper quartiles. Some outliers are not shown to save space.

Table 3: Geometric mean of the speedups of the different reorderings and architectures compared to the original order for all 490 matrices in the 1D algorithm.

1D	RCM	AMD	ND	GP	HP	Gray	Mean
Skylake	1.054	0.933	0.990	1.189	1.099	0.700	0.981
Ice Lake	1.039	0.939	0.981	1.183	1.100	0.744	0.987
Naples	1.025	0.939	0.978	1.206	1.083	0.757	0.988
Rome	1.032	0.946	0.989	1.197	1.089	0.767	0.994
Milan A	1.039	0.956	0.992	1.198	1.096	0.771	1.000
Milan B	1.048	0.963	0.999	1.212	1.110	0.778	1.009
TX2	1.060	0.969	1.007	1.224	1.123	0.768	1.015
Hi1620	1.061	0.973	1.007	1.228	1.128	0.772	1.017
Mean	1.045	0.952	0.993	1.205	1.103	0.757	0.999

clearly show our first key finding; Graph partitioning provides far better SpMV performance than the alternatives.

4.3 Reordering for 2D SpMV

As discussed in Section 3.1, the 1D algorithm does not ensure load balance. We thus repeat the above experiment for the 2D SpMV algorithm. Results are shown in Figure 3.

Compared to the 1D algorithm, there are fewer and less extreme outliers and the impact of reordering is less pronounced for most architectures. Moreover, the difference between reordering strategies is smaller. On the other hand, the ARM-based CPUs, TX2 and Hi1620, benefit immensely, especially from RCM, ND, and GP. Hi1620 also benefits from Gray ordering in this case. However, we note that the initial performance of both 1D and 2D algorithms on the ARM CPUs is quite low, with median values of 20–30 Gflop/s. We suspect that further tuning and improved compiler support would improve instruction-level parallelism and alleviate performance bottlenecks of the ARM CPUs.

While the 2D algorithm creates a perfect load balance with respect to the number of nonzeros per thread, execution time may not be fully balanced if cache locality differs in different parts of the matrix and thus in different threads. For that reason, the 2D algorithm can be slower than the 1D algorithm in rare cases. Nonetheless,

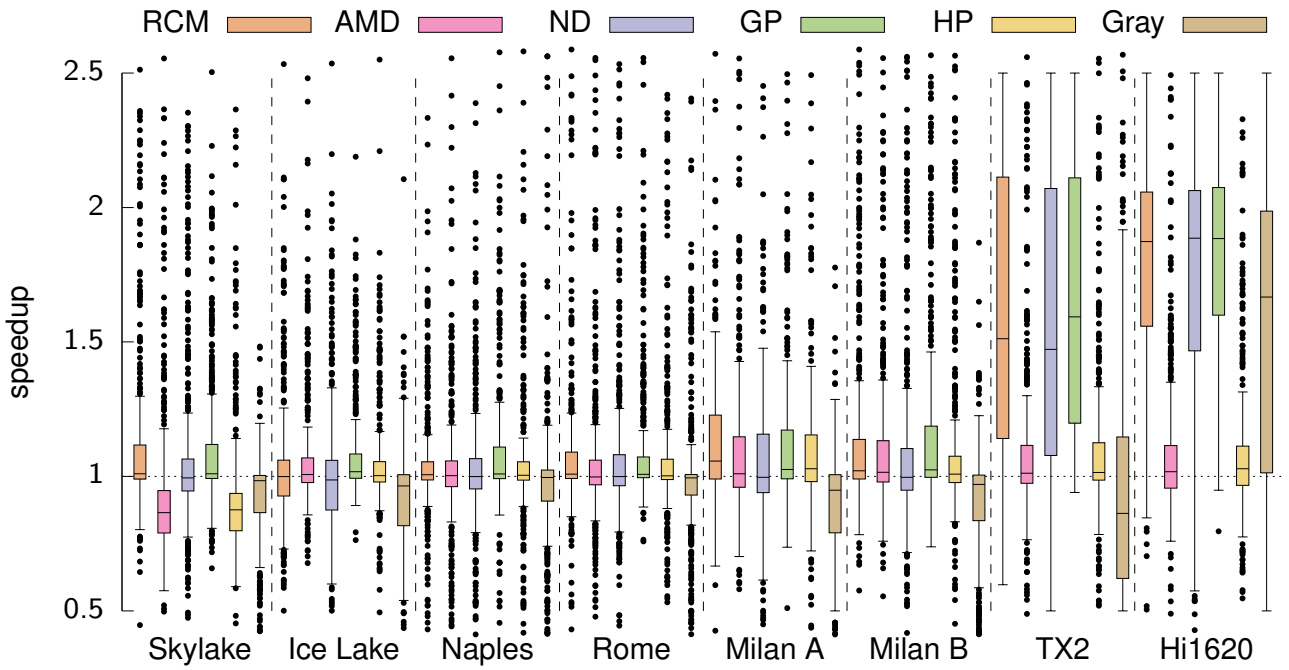

Figure 3: Speedup of the nonzero-balanced CSR SpMV kernel (2D algorithm) after reordering. Note that some outliers are not shown to save space.

Table 4: Geometric mean of the speedups of the different reorderings and architectures compared to the original order for all 490 matrices in the 2D algorithm.

2D	RCM	AMD	ND	GP	HP	Gray	Mean
Skylake	1.081	0.909	1.034	1.090	0.892	0.906	0.982
Ice Lake	1.043	0.979	1.007	1.088	0.959	0.899	0.994
Naples	1.026	1.005	1.018	1.112	1.003	0.909	1.010
Rome	1.038	1.017	1.028	1.106	1.015	0.920	1.019
Milan A	1.058	1.034	1.036	1.111	1.029	0.908	1.027
Milan B	1.074	1.050	1.045	1.123	1.040	0.899	1.036
TX2	1.134	1.059	1.100	1.186	1.049	0.893	1.066
Hi1620	1.197	1.062	1.160	1.250	1.051	0.944	1.106
Mean	1.080	1.013	1.052	1.132	1.003	0.910	1.029

the 2D algorithm typically observes a considerable speedup over the 1D algorithm for many matrices. For example, for 25 % of the matrices on the Rome processor, a speedup of more than 1.1× or more is observed when comparing the 2D and 1D algorithms with the same ordering, and the largest speedup for an individual matrix is about 10×. Since all other CPUs have more cores, most of their speedups are even higher.

The geometric mean of the speedups is shown in Table 4. The numbers show that most algorithms improve while the speedups of GP and HP are reduced compared to 1D. GP is still superior, but HP drops from second place to second to last place, with only Gray giving weaker results.

To understand this result, remember that RCM, ND, AMD and also Gray do not provide load balancing. Once this drawback is removed, it becomes clear that RCM and ND provide good cache reuse. The fact that these reorderings are much stronger when using a 2D algorithm constitutes our second key finding.

4.4 In-depth Performance Analysis

The previously presented results show that the obtainable speedups greatly vary across different matrices, algorithms and platforms. To better capture the dynamics of this complex interaction, we now present an in-depth analysis that aims to uncover what are the most common scenarios that result in performance improvement (or degradation), and what are the major causes for that behaviour. To make the analysis more focused, we concentrate on three out of the eight multicore CPUs in our study.

Based on how reordering impacts the load balance and whether SpMV performance improves, degrades or remains unchanged after reordering, we identified 6 classes of common scenarios. Figure 4 analyzes SpMV performance with respect to representative matrices from each class, across 3 platforms from different vendors (AMD, Intel and ARM), for both 1D and 2D SpMV algorithms and all 6 reordering schemes. *Classes 1*, *2* and *3* depict different scenarios where reordering improves performance, in *Class 4* no significant performance difference is observed, while reordering in *Classes 5* and *6* degrades performance when compared to the original matrix.

Class 1 covers cases where the original and reordered matrices are load balanced and significant speedups are observed for both 1D and 2D algorithms. This is seen in Figure 4, where the original and almost all reordered `333SP` matrices from *Class 1* have an imbalance

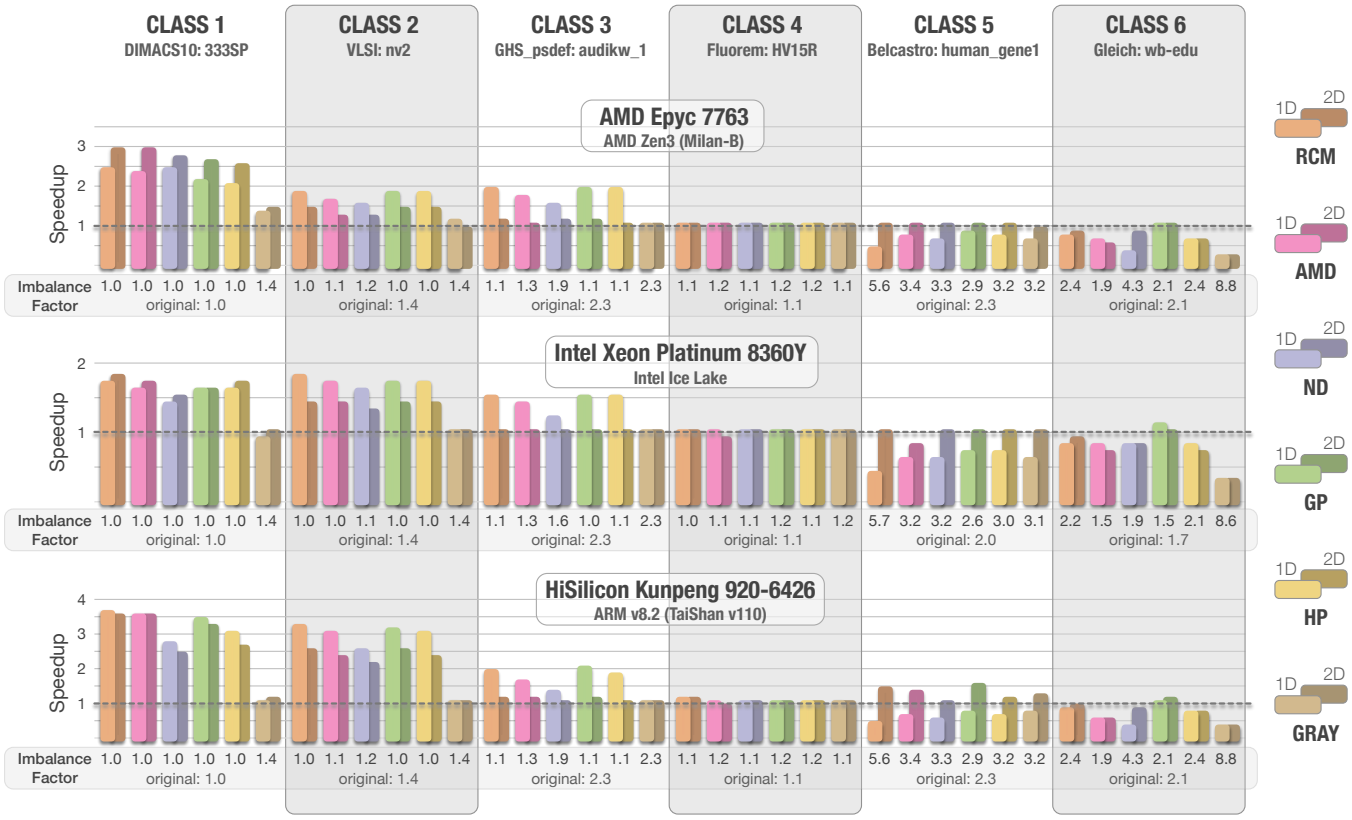

Figure 4: Performance analysis of matrix classes for different reordering schemes, SpMV algorithms and platforms. For each class, speedups and imbalance factors are shown for a representative matrix whose name is shown at the top.

factor (see Section 3.2) of 1.0 for the 1D SpMV[1], suggesting that *Class 1* reorderings do not have significant impact on load balancing. As such, the speedups obtained for both 1D and 2D SpMV mainly demonstrate the ability of the reordering schemes to provide better data locality and cache reuse. A notable exception in Figure 4 is Gray reordering, which induces some load imbalance (i.e., an imbalance factor of 1.4), resulting in marginal improvements on AMD and Intel, or even performance degradation on ARM.

A similar scenario can be observed for *Class 2* (see `nv2` in Figure 4), where reordering additionally provides better load balancing (i.e., the imbalance factor is lower after reordering). Since, in *Class 2*, the speedups are still observed for both 1D and 2D SpMV, it showcases the ability of reordering schemes to provide better data locality and load balancing. In contrast, reordering of *Class 3* matrices (see `audikw_1` in Figure 4) mainly improves the load balancing, since speedups are only observed for the 1D SpMV and there is no change in performance for the 2D SpMV.

In *Class 4*, original and reordered matrices deliver similar performance for both 1D and 2D SpMV. As shown for `HV15R` in Figure 4, reordering does not significantly impact load balancing, thus suggesting that both original and reordered matrices are equally capable of exploiting the data locality (e.g., data fits in cache either way). On the other hand, *Class 5* shows that the reordered matrices can provoke load imbalance in 1D execution, thus resulting in performance degradation, which does not occur in the inherently load-balanced 2D SpMV. Finally, *Class 6* depicts the case where different reordering schemes can diversely impact SpMV performance, indicating a need for new approaches to efficient matrix reordering.

As shown in Figure 4, it is worth noting that the matrices from different classes maintain very similar behaviour across all three platforms. However, the range of attainable speedups is highly affected by the platform specifics, e.g., the highest range is observed for the ARM system, followed by the AMD and Intel platforms.

4.5 Matrix Features and Metric Analysis

Figure 5 depicts performance profiles [7] to compare different reordering methods in terms of bandwidth, profile, off-diagonal nonzero count, and SpMV runtime relative to the best performing method for a given fraction of the matrices in the dataset. For the sake of brevity, we consider only the SpMV runtime on the 128-core Milan B system (see Table 2). A point (x, y) on a profile means that the respective method is within a factor x of the best result for a fraction y of the matrices. For example, in the upper left plot, the point $(1.0, 0.78)$ on the RCM curve means that for 78 % of the matrices, RCM results in the lowest bandwidth among all the orderings. Similarly, in the lower right plot, the point $(1.1, 0.8)$

[1]Due to its nature, load balancing with the 2D algorithm is guaranteed, i.e., its imbalance factor is always 1.0 and it does not depend on matrix features.

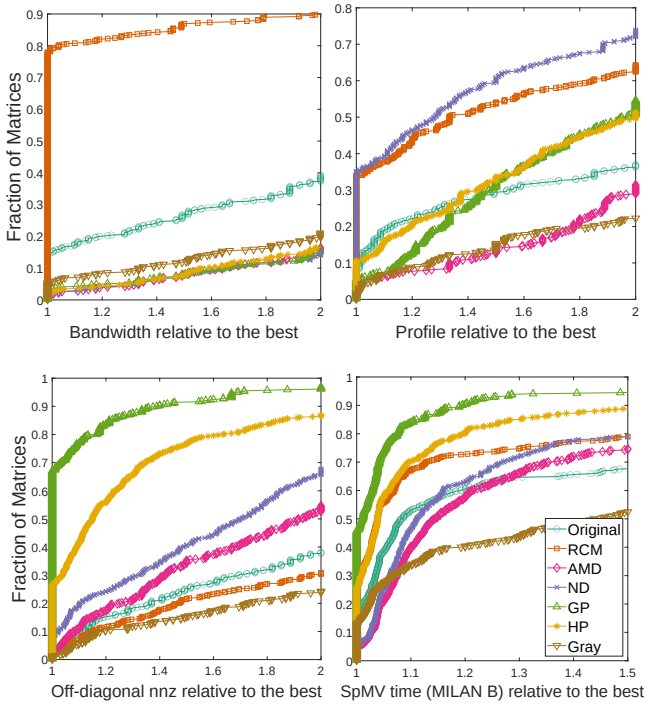

Figure 5: Performance profiles comparing bandwidth, profile, off-diagonal nonzero count and SpMV runtime.

on the GP curve means that for 80 % of the matrices, SpMV with the GP ordering is at most 10 % slower than the best SpMV time obtained by any method. Generally speaking, a curve closer to the top left corner is interpreted as better.

Figure 5 shows that, in terms of reducing the matrix bandwidth, RCM is the best method for almost 80 % of the matrices. The success of RCM in reducing bandwidth is expected since it is primarily exploited for that purpose. What might be more surprising is that all the other methods are worse than the original ordering in that regard. As for reducing the matrix profile, we see ND as the best method closely followed by RCM. Regarding the nonzero count in off-diagonal blocks, GP is the winner with the best performance for nearly 65 % of the instances. This is expected since GP with the edge-cut objective directly aims to minimize the off-diagonal nonzero count. The second method in this case is HP, which can also be expected since the cut-net objective aims to minimize off-diagonal nonzero segments, thus having an indirect yet strong relation with the off-diagonal nonzero count.

Finally, considering the performance profile for SpMV runtimes in Figure 5, it most closely resembles the performance profile for the off-diagonal nonzero count. This suggests that the off-diagonal nonzero count is a much more important feature than profile and bandwidth with respect to SpMV performance. Here, we again see GP and HP as the first and second most effective methods. Thereafter, RCM is the third best method in reducing SpMV time, even though it is not that successful in reducing the off-diagonal nonzero count. This might be explained by the success of RCM in reducing bandwidth, which might increase cache reuse and hence serve the SpMV effectiveness indirectly. Meanwhile, we expect

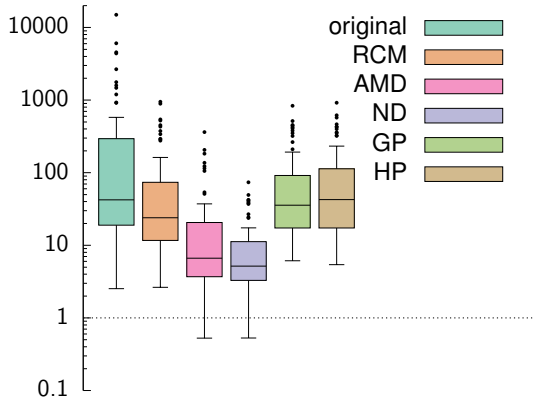

Figure 6: Nonzero ratio in Cholesky factor L to nonzeros in $A = LL^T$ for different orderings. The middle line in each box is the median, whereas the endpoints of each box correspond to the lower and upper quartiles.

the success of ND for reducing profile to be more effective in its performance for reducing fill-in, as we will observe in Section 4.6.

Overall, our key finding is that SpMV performance benefits most from the reordering methods that reduce the number of nonzeros in off-diagonal blocks most.

4.6 Fill-in for Sparse Cholesky Factorisation

We computed the fill-in created by sparse Cholesky factorisation using the row counting algorithm of Gilbert et al. [13] for symmetric, positive definite matrices in SuiteSparse with different orderings. The Gray code ordering is not included, since it does not preserve symmetry and therefore cannot be used for this factorisation. Figure 6 compares matrix orderings with respect to the ratio of nonzeros in the Cholesky factor L to nonzeros in $A = LL^T$ for 78 of the largest matrices.

As expected, the fill-reducing orderings, AMD and ND, produce the least fill-in. While RCM, GP and HP are considerably less effective, they still typically produce better results than the original ordering. With respect to the features discussed in Section 4.5, other matrix features may be needed to explain the reduction of fill-in due to reordering.

4.7 Reordering Overhead

To compare the cost of different reorderings, Table 5 shows the reordering time on Ice Lake (see Table 2) for ten of the largest matrices while covering different application domains, including road networks, semiconductor design, genome sequencing, solid and fluid mechanics. Roughly speaking, Gray ordering is always fastest and RCM is usually the second fastest, whereas HP and ND are typically the slowest. While the SpMV kernels described in Section 3.1 are parallelized with OpenMP, the reordering algorithms themselves are currently serial. Although we believe the implementations of the chosen reordering algorithms to be reasonably efficient, there may be room for further improve their performance.

Depending on the matrix and ordering algorithm, the time required for reordering ranges from a few hundred to several million

Table 5: Time (in seconds) to reorder a matrix on Ice Lake. For comparison, execution time of a single CSR SpMV iteration using 72 threads is also shown.

Matrix Name	RCM	AMD	ND	GP	HP	Gray	SpMV
delaunay_n24	5.3	18.6	210	10.9	125	2.9	0.010
europe_osm	15.4	18.9	437	31.5	151	7.5	0.013
Flan_1565	1.1	2.5	18.3	4.7	120	0.2	0.004
HV15R	6.5	8.3	118	31.5	429	0.3	0.011
indochina-2004	30.9	80.3	163	29.6	219	2.0	0.072
kmer_V1r	117	5 047	2 915	1 333	7 865	64.2	0.084
kron_g500-logn21	59.4	366	183	229	22 382	1.3	0.010
mycielskian19	24.4	80.1	131	739	177	3.1	0.132
nlpkkt240	14.3	71.7	869	53.8	1 040	4.9	0.035
vas_stokes_4M	4.2	17.0	146	22.6	243	1.2	0.010

SpMV operations with the unreordered matrix using 72 cores on the same machine. To achieve overall savings from reordering, the number of SpMV operations performed must exceed the ratio of the reordering time to the difference between the unreordered and reordered SpMV. For example, reordering *europe_osm* with RCM takes 15.4 seconds and improves SpMV performance by 22 % on Ice Lake. Since a single SpMV iteration before reordering takes 0.013 seconds, then approximately $15.4/(0.013 \times (1 - 1/1.22)) \approx 6\,569$ SpMV iterations are needed to save time overall. The reordering cost is high, but in the context of scientific computing it can often be amortised over thousands or millions of SpMV iterations with the same matrix in the course of a simulation.

5 RELATED WORK

The importance of fill-reducing orderings for sparse direct solvers has been known for a long time and a comprehensive overview is given by Davis et al. [6]. Early on, Temam and Jalby [25] devised a quantitative model of data locality for SpMV and discussed the merits of RCM and ND orderings. Toledo [26] found little or no difference in SpMV performance when comparing RCM and ND orderings of a small set of matrices to their original ordering. More recently, Gkountouvas et al. [15] found average improvements of 10–20 % in SpMV performance on 16- and 24-core CPUs after applying RCM reordering. Similarly, Pinar and Heath [24] found that RCM resulted in an average speedup of 10 % on a set of 10 matrices for a CSR SpMV kernel when combined with a register blocking optimization. Further, they proposed a column reordering strategy based on a formulation of the travelling salesperson problem (TSP) which yielded an average speedup of 25 %. Another TSP-based ordering to improve data locality was studied by Heras et al. [17] and Pichel et al. [22], who found that both RCM and their proposed ordering led to improved data locality and fewer cache misses for SpMV. In [23], it was shown that reordering using RCM, AMD and ND improves SpMV performance by 10–15 % on average for a set of 14 matrices, while the TSP-based reordering strategy reduced SpMV execution time ranging from a few percent up to 75 % depending on the matrix.

6 CONCLUSION

This paper provides a comprehensive analysis of the relationship between matrix reordering algorithms and their performance on SpMV. Through a large sparse matrix dataset using six broadly used reordering algorithms on eight state-of-the-art multicore architectures, we showed how the effectiveness of reordering relates to various factors, such as reordering algorithm, load balancing concerns, and off-diagonal nonzero counts. Some of our findings are as expected, such as AMD and ND producing the least fill-in for Cholesky factorisation, whereas other findings are counterintuitive, such the performance of reordering algorithms not varying much across different architectures. Overall, this paper sheds light on matrix reorderings and provides practical guidance for their effective use in sparse computations. Future work will explore new metrics to analyze reordering algorithms, expand the study to GPUs, and use machine learning to predict the most effective reordering algorithm.

ACKNOWLEDGMENTS

This work was supported by the SparCity project of the European High-Performance Computing Joint Undertaking under grant agreement No. 956213. Koç University is supported by the Turkish Science and Technology Research Centre Grant No. 120N003, and INESC-ID is supported by Fundação para a Ciência e a Tecnologia (FCT) through the UIDB/50021/2020 project. The research presented in this paper has also made use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

REFERENCES

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 2004. Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm. *ACM Trans. Math. Softw.* 30, 3 (Sept. 2004), 381—-388. https://doi.org/10.1145/1024074.1024081

[2] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'excellent, and Xiaoye S. Li. 2001. Analysis and Comparison of Two General Sparse Solvers for Distributed Memory Computers. *ACM Trans. Math. Softw.* 27, 4 (Dec. 2001), 388–421. https://doi.org/10.1145/504210.504212

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

[4] E. Cuthill and J. McKee. 1969. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference (ACM '69)*. Association for Computing Machinery, New York, NY, USA, 157–172. https://doi.org/10.1145/800195.805928

[5] 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

[6] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566. https://doi.org/10.1017/S0962492916000076

[7] Elizabeth D Dolan and Jorge J Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming* 91, 2 (2002), 201–213.

[8] Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363. https://doi.org/10.1137/0710032

[9] Alan George and Joseph WH Liu. 1989. The evolution of the minimum degree ordering algorithm. *SIAM Rev.* 31, 1 (1989), 1–19.

[10] Alan George and Joseph W. H. Liu. 1979. An Implementation of a Pseudoperipheral Node Finder. *ACM Trans. Math. Software* 5, 3 (1979), 284–295. https://doi.org/10.1145/355841.355845

[11] Alan George and David R. McIntyre. 1978. On the Application of the Minimum Degree Algorithm to Finite Element Systems. *SIAM J. Numer. Anal.* 15, 1 (1978), 90–112. http://www.jstor.org/stable/2156565

[12] Norman E. Gibbs, William G. Poole, and Paul K. Stockmeyer. 1976. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM J. Numer. Anal.* 13, 2 (1976), 236–250.

[13] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. 1994. An Efficient Algorithm to Compute Row and Column Counts for Sparse Cholesky Factorization. *SIAM J. Matrix Anal. Appl.* 15, 4 (1994), 1075–1091. https://doi.org/10.1137/S0895479892236921

[14] J. R. Gilbert and R. E. Tarjan. 1987. The Analysis of a Nested Dissection Algorithm. *Numer. Math.* 50, 4 (feb 1987), 377–404. https://doi.org/10.1007/BF01396660

[15] Theodoros Gkountouvas, Vasileios Karakasis, Kornilios Kourtis, Georgios Goumas, and Nectarios 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

[16] Sardar Anisul Haque and Shahadat Hossain. 2009. A Note on the Performance of Sparse Matrix-vector Multiplication with Column Reordering. In *2009 International Conference on Computing, Engineering and Information*. 23–26. https://doi.org/10.1109/ICC.2009.40

[17] D.B. Heras, V. Blanco, J.C. Cabaleiro, and F.F. Rivera. 2001. Modeling and improving locality for the sparse-matrix–vector product on cache memories. *Future Generation Computer Systems* 18, 1 (2001), 55–67. https://doi.org/10.1016/S0167-739X(00)00075-3

[18] 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

[19] Wai-Hung Liu and Andrew H Sherman. 1976. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.* 13, 2 (1976), 198–213.

[20] Duane Merrill and Michael Garland. 2016. Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190

[21] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak 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] J.C. Pichel, D.B. Heras, J.C. Cabaleiro, and F.F. Rivera. 2004. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings*. 66–71. https://doi.org/10.1109/EMPDP.2004.1271429

[23] Juan C Pichel, David E Singh, and Jesús Carretero. 2008. Reordering algorithms for increasing locality on multicore processors. In *2008 10th IEEE International Conference on High Performance Computing and Communications*. IEEE, 123–130. https://doi.org/10.1109/HPCC.2008.96

[24] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing* (Portland, Oregon, USA) *(SC '99)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/331532.331562

[25] O. Temam and W. Jalby. 1992. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. 578–587. https://doi.org/10.1109/SUPERC.1992.236646

[26] 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

[27] A. N. Yzelman and Rob H. 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

[28] Haoran Zhao, Tian Xia, Chenyang Li, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2020. Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 601–609. https://doi.org/10.1109/ICCD50377.2020.00105

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

The paper "Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs" compares various strategies for reordering sparse matrices. The purpose of reordering is to improve performance of sparse matrix operations, for example, by reducing fill-in resulting from sparse Cholesky factorisation or improving data locality in sparse matrix-vector multiplication (SpMV). Many reordering strategies have been proposed in the literature and the current paper provides a thorough comparison of several of the most popular methods.

The main computational artifacts needed for this comparison are provided by the following dataset: James D. Trotter. (2023). Performance measurements for "Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs" (1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.7821491.

The data set consists of performance measurements that were collected on the eX3 cluster, a Norwegian, experimental research infrastructure for exploration of exascale computing. These performance measurements allow one to independently audit and reproduce claims made in the paper, particularly related to the performance of SpMV kernels with respect to 490 sparse matrices, 6 matrix orderings and 8 multicore CPUs.

In addition, the following software is provided:

(1) Source code for Libmtx 0.4.0, which was used to perform matrix reordering with Reverse Cuthill-McKee, Nested Dissection and Graph Par titioning based on METIS.
(2) Source code for a utility that was used to perform matrix reordering with Hypergraph Partitioning based on PaToH.
(3) Source code for SparseBase 0.3.1, which was used to perform matrix reordering with Approximate Minimum Degree and Gray order.
(4) Source code for SpMV with two different kernels for matrices in compressed sparse row (CSR) format.
(5) Source code for sparse Cholesky factorization and for counting fill-in.
(6) Source code for computing matrix features.

The software can be found at: Trotter, James D. (2023). Software for "Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs" (1.0.0). Zenodo. https://doi.org/10.5281/zenodo.7837264.

Finally, we provide a dataset with permutations and reordered matrices in Matrix Market format for 6 orderings and 3 selected matrices (these matrices appear in Fig. 1 of the paper). The dataset can be found at: James D. Trotter. (2023). Matrix reorderings for "Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs" (1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.7837367

REPRODUCIBILITY OF EXPERIMENTS

The experimental results, which constitute the computational artifacts, are provided in a human-readable, tabular format using plain-text ASCII. This format may be readily consumed by gnuplot to create plots or imported into commonly used spreadsheet tools for further analysis.

Performance measurements are provided based on an SpMV kernel using the compressed sparse row (CSR) storage format with 7 matrix orderings. One file is provided for each of 8 multicore CPU systems considered in the paper:

(1) Skylake: `csr_all_xeongold16q_032_threads_ss490.txt`
(2) Ice Lake: `csr_all_habanaq_072_threads_ss490.txt`
(3) Naples: `csr_all_defq_064_threads_ss490.txt`
(4) Rome: `csr_all_rome16q_016_threads_ss490.txt`
(5) Milan A: `csr_all_fpgaq_048_threads_ss490.txt`
(6) Milan B: `csr_all_milanq_128_threads_ss490.txt`
(7) TX2: `csr_all_armq_064_threads_ss490.txt`
(8) Hi1620: `csr_all_huaq_128_threads_ss490.txt`

A corresponding set of files and performance measurements are provided for a second SpMV kernel that is also studied in the paper.

Each file consists of 490 rows and 54 columns. Each row corresponds to a different matrix from the SuiteSparse Matrix Collection (https://sparse.tamu.edu/). The first 5 columns specify some general information about the matrix, such as its group and name, as well as the number of rows, columns and nonzeros. Column 6 specifies the number of threads used for the experiment (which depends on the CPU). The remaining columns are grouped according to the 7 different matrix orderings that were studied, in the following order: original, Reverse Cuthill-McKee (RCM), Nested Dissection (ND), Approximate Minimum Degree (AMD), Graph Partitioning (GP), Hypergraph Partitioning (HP), and Gray ordering. For each ordering, the following 7 columns are given:

(1) Minimum number of nonzeros processed by any thread by the SpMV kernel
(2) Maximum number of nonzeros processed by any thread by the SpMV kernel
(3) Mean number of nonzeros processed per thread by the SpMV kernel
(4) Imbalance factor, which is the ratio of the maximum to the mean number of nonzeros processed per thread by the SpMV kernel
(5) Time (in seconds) to perform a single SpMV iteration; this was measured by taking the minimum out of 100 SpMV iterations performed
(6) Maximum performance (in Gflop/s) for a single SpMV iteration; this was measured by taking twice the number of matrix nonzeros and dividing by the minimum time out of 100 SpMV iterations performed.
(7) Mean performance (in Gflop/s) for a single SpMV iteration; this was measured by taking twice the number of matrix nonzeros and dividing by the mean time of the 97 last SpMV iterations performed (i.e., the first 3 SpMV iterations are ignored).

The results in Fig. 1 of the paper show speedup (or slowdown) resulting from reordering with respect to 3 reorderings and 3 selected matrices. These results can be reproduced by inspecting the

performance results that were collected on the Milan B and Ice Lake systems for the three matrices Freescale/Freescale2, SNAP/com-Amazon and GenBank/kmer_V1r. Specifically, the numbers displayed in the figure are obtained by dividing the maximum performance measured for the respective orderings (i.e., RCM, ND and GP) by the maximum performance measured for the original ordering.

The results presented in Figs. 2 and 3 show the speedup of SpMV as a result of reordering for the two SpMV kernels considered in the paper. In this case, gnuplot scripts are provided to reproduce the figures from the data files described above.