Proceedings of the Federated Conference on Computer Science and Information Systems pp. 695–704
DOI: 10.15439/2016F35
ACSIS, Vol. 8. ISSN 2300-5963

Block Iterators for Sparse Matrices

Daniel Langr*†, Ivan Šimeček* and Tomáš Dytrych‡
* Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems
Thákurova 9, 160 00, Praha, Czech Republic
Email: {langrd,xsimecek}@fit.cvut.cz
† Výzkumný a zkušební letecký ústav, a.s.
Beranových 130, 199 05, Praha, Czech Republic
‡ Czech Academy of Sciences
Nuclear Physics Institute
Řež 130, 250 68, Řež, Czech Republic
Email: tdytrych@ujf.cas.cz

***Abstract*—Finding an optimal block size for a given sparse matrix forms an important problem for storage formats that partition matrices into uniformly-sized blocks. Finding a solution to this problem can take a significant amount of time, which, effectively, may negate the benefits that such a format brings into sparse-matrix computations. A key for an efficient solution is the ability to quickly iterate, for a particular block size, over matrix nonzero blocks. This work proposes an efficient parallel algorithm for this task and evaluate it experimentally on modern multi-core and many-core high performance computing (HPC) architectures.**

I. Introduction

STORAGE formats prescribe a way how sparse matrices are stored in a computer memory. Many designed formats are based on partitioning of matrices into blocks where

1) blocks have a uniform size,
2) this size is not fixed for a given format and may be chosen for each matrix individually.

We call such formats *uniformly-blocking formats* or, shortly, *UB formats*.

Considering a particular sparse matrix A and a particular UB format, we thus face a problem of finding an optimal block size (whatever this means). Typically, we want to find a block size that will provide highest performance of sparse matrix-vector multiplication (SpMV) performed with A. Generally, this task cannot be accomplished until the matrix is fully assembled or at least until its structure of nonzero elements is fully known, which implies that matrices cannot be assembled in UB formats directly (there are usually other reasons as well). Instead, one needs to

1) assemble A in some suitable (not-parametrized) simple storage format,
2) find an optimal block size for A,
3) transform A in memory from its original storage format to a given UB format.

The second and third steps form an important problem related to a given UB format. If the solution of this problem takes too long, it might effectively negate the benefits that a UB format brings into subsequent computations with A.

To find an optimal block size, there is usually no option other than

1) to form some set of possibly-optimal block sizes,
2) to evaluate an optimization criterion for all of them.

Note that this approach generally gives a pseudooptimal block size instead of an optimal one. For sake of simplicity, we do not distinguish between these two cases and call them both optimal throughout this text.

Evaluation of an optimization criterion for a given UB format, given matrix A, and a particular tested block size typically involves gathering some information about all nonzero blocks of A. We therefore need to examine all these nonzero blocks. Such a procedure can be described briefly as follows: *for all nonzero blocks of A, perform some calculations that contribute to the evaluation of an optimization criterion*. Thus, in fact, we need to *iterate over nonzero blocks of A*.

When an optimal block size is found, this iterative process has to be run once again within the third step mentioned above, i.e., during the final transformation of A to a given UB format.

This paper addresses the problem of fast iteration over nonzero blocks of a sparse matrix. We propose an efficient scalable parallel algorithm for a solution to this problem and evaluate it experimentally on modern multi-core and many-core HPC architectures, where matrices frequently emerge in multi-threaded programs. (In distributed-memory environments, objects of our concern are "local" matrices formed by nonzero elements mapped to particular application processes.)

II. Case Study

As an illustrative case study, we work throughout this text with the *adaptive-blocking hierarchical storage format*

This work was supported by the Czech Science Foundation under Grant No. 16-16772S. This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

Algorithm 1: Transformation of A to the ABHSF

Input: A: sparse matrix
Input: $\mathcal{S} = \{s_1, s_2, \ldots\}$: set of possibly-optimal block sizes
Data: M_{opt}, M, s_{opt}, i: auxiliary variables

```
M_opt ← 0
for i ← 1 to |S| do
    M ← 0
    for each nonzero block B of size s_i of A do
        find a space-optimal way W to store B in memory
        considering ⌈log2 s_i⌉ bits for in-block indexes;
        calculate the contribution c(B, W) of B stored in W
        to the memory footprint of A;
        M ← M + c(B, W)
    end
    if i = 1 or M < M_opt then
        s_opt ← s_i
        M_opt ← M
    end
end
for each nonzero block B of size s_opt of A do
    store B in memory in the ABHSF format
end
```

(ABHSF) [1], [2]. This format partitions A into uniform square blocks of size s and stores each block in memory in a space-optimal way. The optimization criterion of ABHSF is represented by the total memory footprint of A which is being minimized. This is a very common optimization criterion for storage formats in general (not only UB formats), since the performance of SpMV is limited by bandwidths of memory subsystems on modern HPC architectures [3].

Many UB formats work with in-block row and column indexes. Optimal (space-optimal) block sizes are then typically those that employ most or all of the available indexing bits. In case of the ABHSF and byte-padded in-block indexes, setting $s = 256$ is almost generally optimal or at least close to being optimal [1]. (Such a choice eliminates the discussed optimization problem, however, it does not eliminate the need to iterate over nonzero blocks of A; this process is still required for transformation of A into the ABHSF.)

On the other hand, if we really want to minimize the memory footprint of A stored in the ABHSF, we need to use the minimum possible number of bits for in-block indexes, i.e., $\lceil \log_2 s \rceil$. In such cases, any block size s can be generally optimal. Let $\mathcal{S} = \{s_1, s_2, \ldots\}$ denotes some set of possibly-optimal block sizes. The transformation of A into the ABHSF can then be written as Algorithm 1.

Algorithm 1 iterates over nonzero blocks of A exactly $|\mathcal{S}|+1$ times. Therefore, we want $|\mathcal{S}|$ to be

1) large enough to find the best possible block size,
2) small enough to prevent long algorithm running times.

One way to get close to both these outcomes is to consider only block sizes

$$\mathcal{S} = \{2^k : 1 \leq k \leq k_{\max}\}, \tag{1}$$

which implies maximum utilization of all k bits for in-block indexes. The $k_{\max}$ parameter, which corresponds to $|\mathcal{S}|$, determines the upper bound for tested block sizes. In practice, setting $k_{\max} = 10$ is typically sufficient, which implies 11 iterations over nonzero blocks of A while testing block sizes $s = 2, 4, 8, \ldots, 1024$ within Algorithm 1.

III. Notation

Let A be an $m \times n$ sparse matrix, where $a_{i,j}$ denotes the value of an element of A located in its ith row and jth column. As a mathematical object, A can be written as

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}. \tag{2}$$

However, within computer programs, we typically work only with nonzero elements of sparse matrices (or, even only with nonzero elements from a single triangular part if a matrix exhibits some kind of symmetry). An element of A is determined by its value, row index, and column index; let us write it as a triplet $(i, j, a_{i,j})$. As a data structure, we can consider A as a *set of matrix nonzero elements*:

$$A = \big\{(i, j, a_{i,j}) : 1 \leq i \leq m,\ 1 \leq j \leq n,\ a_{i,j} \neq 0\big\}. \tag{3}$$

Moreover, nonzero elements stored in memory are accessible in some order, which is typically prescribed by a given storage format. If this order matters, we can consider A as a *sequence of matrix nonzero elements*:

$$A = \big((i_l, j_l, a_{i_l,j_l})\big)_{l=1}^{nnz}, \quad a_{i_l,j_l} \neq 0, \tag{4}$$

where nnz denotes the number of nonzero elements of A.

In the text below, we use forms (2), (3), and (4) interchangeably, while preferring the particular one in dependence on the actual context. For the sake of simplicity, we also consider partitioning into square blocks only; the generalization for rectangular blocks is straightforward.

Partitioning A into square blocks of size s yields an $M \times N$ block matrix, where $M = \lceil m/s \rceil$ and $N = \lceil n/s \rceil$. For indexing block rows and columns, we use capital letters I and J, respectively. A block is called nonzero if it contains at least one nonzero matrix element.

A matrix element $(i, j, a_{i,j})$ belongs to a block with indexes

$$I = \lfloor (i-1)/s \rfloor + 1 \quad \text{and} \quad J = \lfloor (j-1)/s \rfloor + 1. \tag{5}$$

By using $\backslash$ for integer division, we can rewrite (5) as

$$I = (i-1) \backslash s + 1 \quad \text{and} \quad J = (j-1) \backslash s + 1. \tag{6}$$

Element's in-block indexes can be found correspondingly as $\big[(i-1) \mod s\big] + 1$ and $\big[(j-1) \mod s\big] + 1$.

Note that the calculations of block indexes and local in-block indexes for nonzero matrix elements involve integer division and modulo, which are relatively expensive arithmetic operations [4]. When possibly-optimal block sizes are chosen according to (1), both integer division and modulo can be substituted by much faster *logical shift* and *bitwise AND* operations.

Algorithm 2: Iteration over nonzero blocks of A: variant 1

```
Input: A: sparse matrix
Input: s: block size
Data: B: nonzero elements of a single block
Data: I, J: indexes
1  for I ← 1 to ⌈m/s⌉ do
2      for J ← 1 to ⌈n/s⌉ do
3          B ← {}
4          for all (i, j, a_{i,j}) ∈ A do
5              if (i − 1)\s + 1 = I and (j − 1)\s + 1 = J then
6                  B ← B ∪ {(i, j, a_{i,j})}
7              end
8          end
9          if B ≠ {} then
10             process block B with indexes I and J
11         end
12     end
13 end
```

Algorithm 3: Iteration over nonzero blocks of A: variant 2

```
Input: A: sparse matrix
Input: s: block size
Data: B_{I,J}: nonzero elements of a block in Ith block row and
      Jth block column
Data: I, J: indexes
1  for I ← 1 to ⌈m/s⌉ do
2      for J ← 1 to ⌈n/s⌉ do B_{I,J} ← {}
3  end
4  for all (i, j, a_{i,j}) ∈ A do
5      I ← (i − 1)\s + 1
6      J ← (j − 1)\s + 1
7      B_{I,J} ← B_{I,J} ∪ {(i, j, a_{i,j})}
8  end
9  for I ← 1 to ⌈m/s⌉ do
10     for J ← 1 to ⌈n/s⌉ do
11         if B_{I,J} ≠ {} then
12             process block B_{I,J} with indexes I and J
13         end
14     end
15 end
```

IV. Algorithms

Let us now analyse the problem of iteration over the nonzero blocks of A. In the most generic case, we have, at the outset, no knowledge which blocks of A are nonzero and which nonzero elements of A belong to these blocks. There are basically two ways to find this out:

1) to iterate over all blocks of A and for each block find its nonzero elements;
2) to iterate over all nonzero elements of A, find for each of them the corresponding block (6), and save the information that the element belongs to this block.

Pseudocodes for these two options are provided as Algorithms 2 and 3, respectively. Processing of blocks is application-dependent; it might, e.g., represent the calculation of blocks contributions to the optimization criterion (line 5–7 of Algorithm 1) or the storage of blocks in memory (line 15).

Algorithm 2 have low memory requirements; its auxiliary space is

$$S_2(A, s) = O(s^2),$$

since, at a given time, only nonzero elements for a single block need to be kept in memory. The drawback of this algorithm is its high time complexity

$$T_2(A, s) = \Theta(m \cdot n \cdot nnz/s^2).$$

As for Algorithm 3, its time complexity is considerably lower, namely

$$T_3(A, s) = \Theta(m \cdot n/s^2 + nnz).$$

However, the auxiliary space of Algorithm 3 is

$$S_3(A, s) = O(m \cdot n/s^2 + nnz),$$

since one needs to save the information about all nonzero elements for each nonzero block. Moreover, an implementation of this algorithm would likely require some complex dynamic data structure, which might introduce problems with memory fragmentation and expensive insertion/look-up operations.

Whenever working with sparse matrices, we generally want to avoid algorithms with $\Omega(nnz)$ auxiliary space as much as possible. Within many running instances of HPC programs, matrices are the largest objects in a computer memory and their sizes determine an extent of underlying computational problems. Any $\Omega(nnz)$ auxiliary space algorithm (such as Algorithm 3) thus, in effect, considerably limits the size of a problem being solved.

To avoid the high time complexity of Algorithm 2 as well as the high auxiliary space of Algorithm 3, we propose another solution for iteration over nonzero block of A that works as follows:

1) The nonzero elements of A are reordered such that the nonzero elements of each block are laid out consecutively (grouped together) in memory. In other words, *the nonzero elements are sorted with respect to blocks*.
2) A single iteration over nonzero elements is performed while elements of each nonzero block are identified and processed.

The pseudocode of such a solution is provided as Algorithm 4. Its time complexity and auxiliary space is dominated by the sorting step (line 1). Let us assume that we use an in-place randomized quicksort with time complexity $O\big(nnz \cdot log_2(nnz)\big)$ and auxiliary space $O\big(\log_2(nnz)\big)$. The overall time complexity of Algorithm 4 then will be

$$T_4(A, s) = O\big(nnz \cdot \log_2(nnz)\big)$$

and its auxiliary space

$$S_4(A, s) = O\big(\log_2(nnz)\big)$$

as well.

Algorithm 4 reduces both the time complexity of Algorithm 2 and the auxiliary space of Algorithm 3, however, at the following price: it requires A to be provided in such a format that facilitates reordering/sorting its nonzero elements. There is

Algorithm 4: Iteration over nonzero blocks of A: variant 3

Input: A: sparse matrix
Input: s: block size
Data: I, I', J, J', l, l_1: indexes

1 $\big((i_l, j_l, a_{i_l,j_l})\big)_{l=1}^{nnz} \leftarrow$ sort A with respect to blocks
2 $l_1 \leftarrow 1$
3 $I \leftarrow (i_1 - 1) \backslash s + 1$
4 $J \leftarrow (j_1 - 1) \backslash s + 1$
5 **for** $l \leftarrow 2$ to nnz **do**
6 $\quad I' \leftarrow (i_l - 1) \backslash s + 1$
7 $\quad J' \leftarrow (j_l - 1) \backslash s + 1$
8 $\quad$ **if** $I' \neq I$ **or** $J' \neq J$ **then**
9 $\quad\quad$ process block with indexes I and J that contains nonzero elements $\big((i_q, j_q, a_{i_q,j_q})\big)_{q=l_1}^{l-1}$
10 $\quad\quad l_1 \leftarrow l$
11 $\quad\quad I \leftarrow I'$
12 $\quad\quad J \leftarrow J'$
13 $\quad$ **end**
14 **end**
15 process block with indexes I and J that contains nonzero elements $\big((i_q, j_q, a_{i_q,j_q})\big)_{q=l_1}^{nnz}$

practically only one candidate—the *coordinate* storage format (COO) [5], [6]; it consists of three arrays containing row indexes, column indexes, and values of nonzero elements. At the same time, it does not prescribe any particular ordering for these arrays.

To require A to be initially in COO is not as restrictive in practice as it might seem, since:

1) any sparse matrix can be easily and quickly transformed into COO regardless of its original storage format,
2) COO is the most convenient format for assembling sparse matrices (newly generated nonzero elements are simply appended to the corresponding COO arrays).

A scenario where matrices are first assembled in COO and then transformed to another, computationally more suitable, storage format (such as some UB format) is thus perfectly viable for HPC programs.

To sort the nonzero elements with respect to blocks, we can define sorting keys by using the pairs of I and J block indexes calculated by (5). For example, if we want blocks to be sorted lexicographically, we can calculate sorting keys as $I \cdot N + J$. Again, note that choosing (1) for possibly-optimal block sizes implies faster calculation of sorting keys and therefore, in effect, likely faster sorting step within Algorithm 4.

A. Parallelization

Parallelization of (expensive) Algorithms 2 and 3 is straightforward. In Algorithm 2, we can parallelize the inner-most loop (line 4) while synchronizing concurrent updates of B at line 6. In Algorithm 3, we can parallelize the loops over blocks (lines 1–2 and 9–10) as well as the loop over nonzero matrix elements (line 4) while using thread-local I and J indexes and synchronizing concurrent updates to $B_{I,J}$ at line 7.

Parallelization of Algorithm 4 is a bit more complex; we propose its multi-threaded variant as Algorithm 5. Note that

Algorithm 5: Parallel iteration over nonzero blocks of A

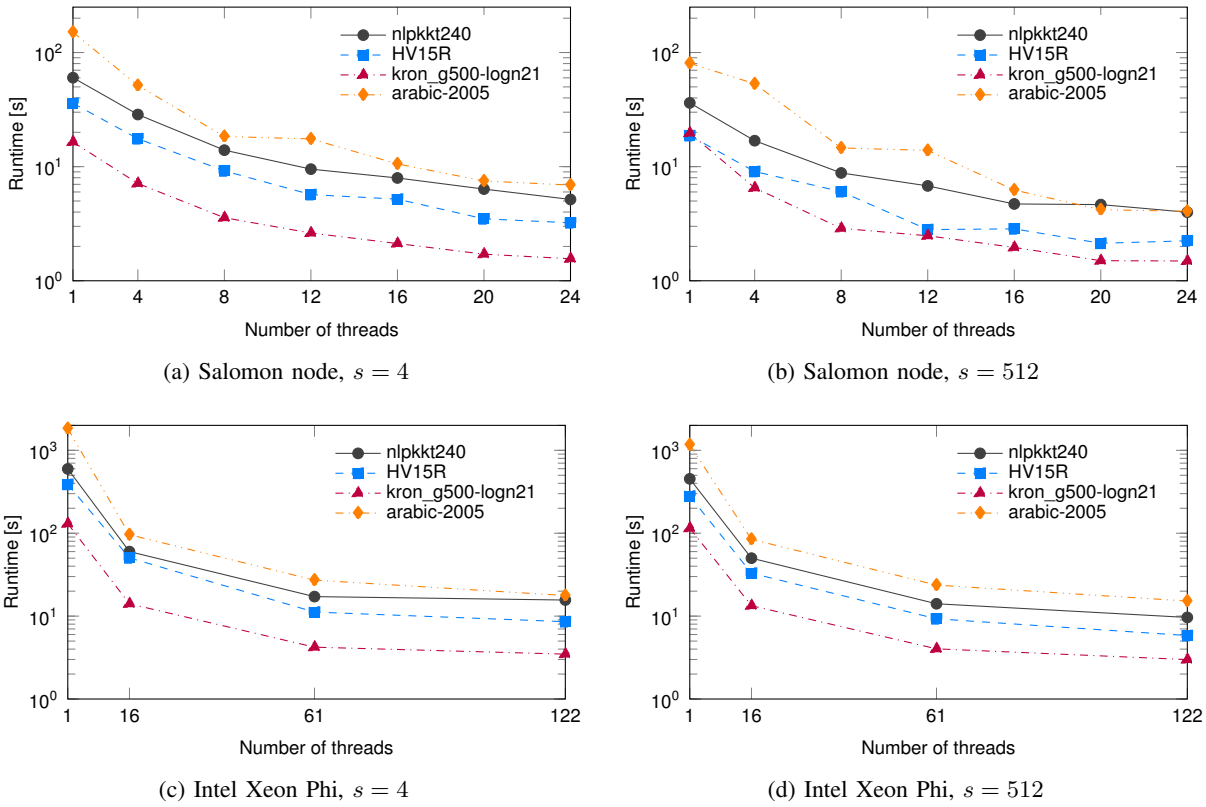
Input: A: sparse matrix
Input: s: block size
Input: T: number of threads
Data: I, I', J, J', l, l_1, t: thread-private indexes
Data: $tb[]$: thread-shared integer array of size $T + 1$

1 $\big((i_l, j_l, a_{i_l,j_l})\big)_{l=1}^{nnz} \leftarrow$ sort A in parallel with respect to blocks
2 $tb[1] \leftarrow 1$
3 $tb[T + 1] \leftarrow nnz + 1$
4 **for all** threads **do in parallel**
5 $\quad t \leftarrow$ current thread number (between 1 and T)
6 $\quad$ **if** $t > 1$ **then**
7 $\quad\quad l \leftarrow \big\lceil nnz \cdot (t - 1) \big\rceil \backslash T + 1$
8 $\quad\quad I \leftarrow (i_1 - 1) \backslash s + 1$
9 $\quad\quad J \leftarrow (j_1 - 1) \backslash s + 1$
10 $\quad\quad l \leftarrow l + 1$
11 $\quad\quad$ **while** $l \leq nnz$ **do**
12 $\quad\quad\quad I' \leftarrow (i_1 - 1) \backslash s + 1$
13 $\quad\quad\quad J' \leftarrow (j_1 - 1) \backslash s + 1$
14 $\quad\quad\quad$ **if** $I' \neq I$ **or** $J' \neq J$ **then break**
15 $\quad\quad\quad l \leftarrow l + 1$
16 $\quad\quad$ **end**
17 $\quad\quad tb[t] \leftarrow l$
18 $\quad$ **end**
19 $\quad$ perform barrier to synchronize threads
20 $\quad l_1 \leftarrow tb[t]$
21 $\quad I \leftarrow (i_{l_1} - 1) \backslash s + 1$
22 $\quad J \leftarrow (j_{l_1} - 1) \backslash s + 1$
23 $\quad$ **for** $l \leftarrow tb[t] + 1$ to $tb[t + 1] - 1$ **do**
24 $\quad\quad I' \leftarrow (i_l - 1) \backslash s + 1$
25 $\quad\quad J' \leftarrow (j_l - 1) \backslash s + 1$
26 $\quad\quad$ **if** $I' \neq I$ **or** $J' \neq J$ **then**
27 $\quad\quad\quad$ process block with indexes I and J that contains nonzero elements $\big((i_q, j_q, a_{i_q,j_q})\big)_{q=l_1}^{l-1}$
28 $\quad\quad\quad l_1 \leftarrow l$
29 $\quad\quad\quad I \leftarrow I'$
30 $\quad\quad\quad J \leftarrow J'$
31 $\quad\quad$ **end**
32 $\quad$ **end**
33 $\quad$ process block with indexes I and J that contains nonzero elements $\big((i_q, j_q, a_{i_q,j_q})\big)_{q=l_1}^{tb[t+1]-1}$
34 **end**

we cannot simply parallelize the main loop of Algorithm 4 (line 5), since its uniform splitting would generally cause threads to start with nonzero elements that are not, in sequence (4), first within corresponding blocks. Algorithm 5 therefore splits the load among threads such that:

1) an amortized number of nonzero elements processed by each thread is nnz/T, where T denotes the number of threads;
2) all nonzero elements of each particular block are processed by a single thread only.

Such splitting is calculated at lines 2–18 of Algorithm 5 and stored into an auxiliary array $tb[]$. Each thread can then process its exclusive portion of nonzero elements independently of other threads (lines 20–33). Threads are required to be indexed from 1 to T; if not, some mapping from thread IDs to such indexing must be provided.



(a) Salomon node, $s = 4$

(b) Salomon node, $s = 512$

(c) Intel Xeon Phi, $s = 4$

(d) Intel Xeon Phi, $s = 512$

Fig. 1: Strong scalability of Algorithm 5 with the *do-nothing processor* measured for different architectures, different matrices, and different block sizes.

V. Experiments

We have conducted an extensive experimental study to evaluate Algorithm 5. Within this study, we worked with matrices from the University of Florida Sparse Matrix Collection (UFSMC) [7]. Matrices that we used are listed in Appendix; their characteristics can be found at the UFSMC web pages[1]. We tried to choose matrices emerging in a wide range of scientific and engineering disciplines and thus having different properties, such as:

- different types of elements—real, complex, integer, binary;
- different sizes and shapes—square, rectangular;
- different kinds of symmetries—unsymmetric, symmetric, Hermitian;
- different numbers of nonzero elements—from $1.1 \cdot 10^7$ of the kim2 matrix to $6.4 \cdot 10^8$ of the arabic-2005 matrix;
- different densities, i.e., relative counts $nnz/(m \cdot n)$, of nonzero elements—from $5.12 \cdot 10^{-7}$ of the nlpkkt240 matrix to $1.11 \cdot 10^{-2}$ of the TSOPF_RS_b2328 matrix;
- different patterns of nonzero elements.

The matrices were read on the input from files downloaded from the UFSMC. All these files stored nonzero elements of matrices in the *reverse lexicograhical order* (RLO) and in the same order, we stored the elements in memory in the COO format as the first step of our benchmark program.

The measurements were performed on the following two shared-memory HPC architectures:

1) nodes of the Salomon supercomputer operated by IT4Innovations National Supercomputing Center in Ostrava, Czech Republic, having two 12-core Intel Xeon E5-2680v3 CPUs and 128 GB RAM per node;
2) Intel Xeon Phi coprocessor type 7120P with 16 GB RAM.

Benchmark codes were written in C++ and we used the GNU g++ compiler version 5.1.0 on Salomon and Intel icpc compiler version 16.0.1 for Intel Xeon Phi builds.

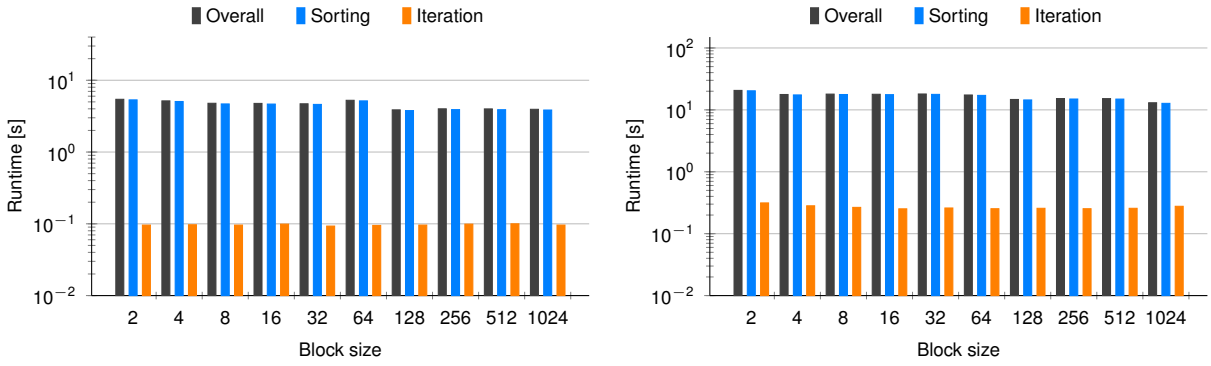
Parallelization was implemented with OpenMP. As for sorting (line 1 of Algorithm 5), we used AQsort[2]—an OpenMP-based multi-threaded variant of in-place quicksort that can work with multiple arrays, such as the arrays of the COO storage format in our case.

A. Processors

Lines 27 and 33 of Algorithm 5 contain processing of found nonzero blocks. Within this study, we invoked two different block *processors* at these points. The first one did nothing useful at all, which allowed us to evaluate the algorithm itself

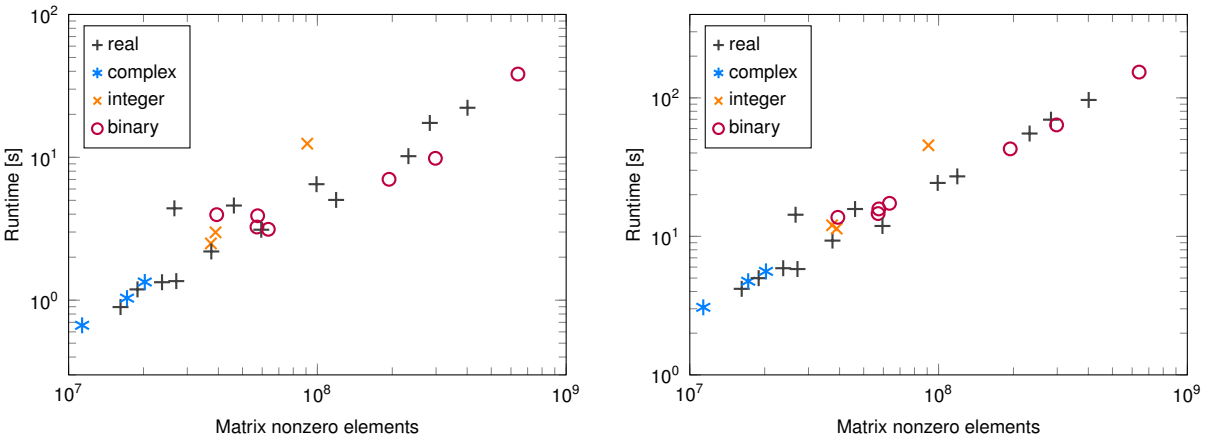
[1] http://www.cise.ufl.edu/research/sparse/matrices/

[2] https://github.com/DanielLangr/AQsort



(a) Salomon node, nlpkkt240, $T = 24$ (b) Intel Xeon Phi, arabic-2005, $T = 122$

Fig. 2: Runtime of Algorithm 5 (*Overall*) and its two phases (*Sorting* and *Iteration*) with the *do-nothing processor*, measured for different architectures, different matrices, different block sizes, and the optimal number of threads.



(a) Salomon node, $T = 24$ (b) Intel Xeon Phi, $T = 122$

Fig. 3: Aggregated runtime of Algorithm 5 with the *do-nothing processor* run 10 times in a row for block sizes $s = 2, 4, 8, \ldots, 1024$, measured for 26 matrices from the UFSMC, different architectures and the optimal number of threads.

without any application-dependent computations; we call this processor a *do-nothing processor*.[3]

The second processor was designed for the problem of finding an optimal block size when storing A in the ABHSF; we call it the *ABHSF-opt processor*. This processor calculated and summed the contributions of blocks to the overall memory footprint of A.

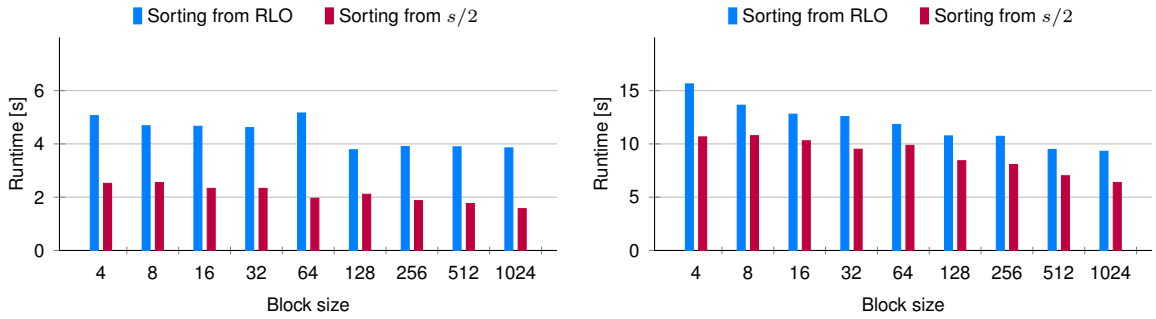
B. Scalability

First, we measured the strong scalability of Algorithm 5; the results for 4 different matrices and 2 different block sizes are shown in Fig. 1. In all cases, parallelization led to a considerable reduction of runtime required for the iteration over nonzero blocks of A. This runtime was dominated by the sorting phase of the algorithm (see Section V-C for details), thus, consequently, the overall scalability of Algorithm 5 was determined by the scalability of AQsort within our study. The maximum number of threads, i.e., 24 for Salomon nodes and 122 for Intel Xeon Phi, was chosen experimentally; beyond these points, runtime of AQsort started to grow significantly.

[3]A processor that would do anything at all might be optimized away by the compiler. We therefore designed the do-nothing processor such that it summed the number of nonzero elements of blocks, which also allowed us to verify that the algorithm correctly iterated over all nonzero blocks of A.

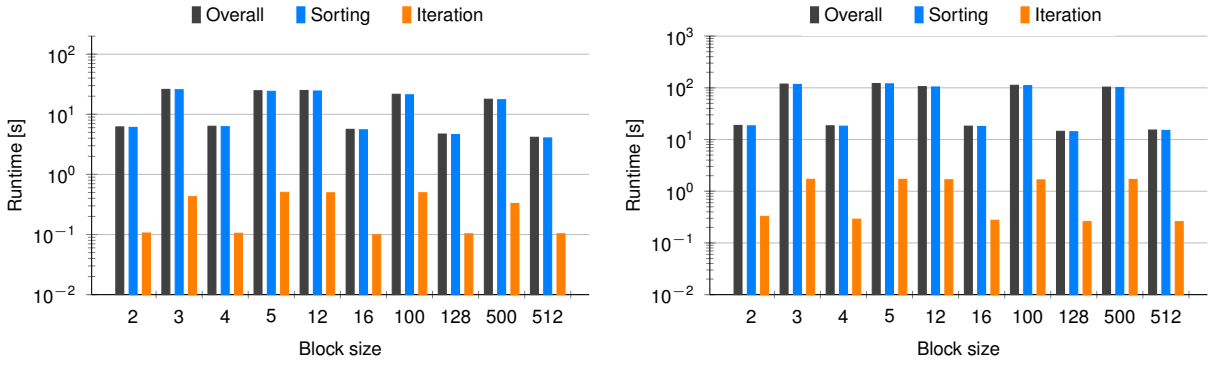
C. Algorithm Phases

The second experiment evaluated the contributions of the *sorting* and *iterations* phases of Algorithm 5 to its *overall* runtime; the results are presented by Fig. 2. The set of tested block sizes was selected according to (1) while setting $k_{\max} = 10$. The sorting phase of Algorithm 5 clearly dominates the overall algorithm runtime. The runtime of the iteration phase (with the *do-nothing* processor in this case) is practically negligible. We can also notice that larger block sizes yielded slightly faster sorting due to lower number of distinct sorting keys (less nonzero elements need to be swapped in memory).



(a) Salomon node, nlpkkt240, $T = 24$ (b) Intel Xeon Phi, nlpkkt240, $T = 122$

Fig. 4: Runtime of the sorting phase of Algorithm 5 for different block sizes when nonzero elements were sorted from the RLO (*Sorting from RLO*) and from the ordering given by half a block size (*Sorting from* $s/2$).



(a) Salomon node, nlpkkt240, $T = 24$ (b) Intel Xeon Phi, arabic-2005, $T = 122$

Fig. 5: Runtime of Algorithm 5 (*Overall*) and its two phases (*Sorting* and *Iteration*) with the *do-nothing processor*, measured for different architectures, different matrices, different block sizes, and the optimal number of threads.

D. Multiple Block Sizes

Within the problem of finding an optimal block size, we need to iterate over nonzero blocks of matrices multiple times while testing different block sizes. In the next experiment, we therefore run Algorithm 5 ten times in a row, while testing block sizes $s = 2, 4, 8, \ldots, 1024$ as proposed in Section II. We used 26 matrices from the UFSMC (listed in Appendix) and, for each one, measured the aggregated runtime of all 10 algorithm runs. The results are presented by Fig. 3, which shows runtimes as a function of the number of matrix nonzero elements.

We can observe that the relation of runtime and nnz was roughly linear. Though, for a constant T, the time complexity of Algorithm 5 is $O\big(n \cdot \log_2(n)\big)$, modern implementations of parallel quicksorts yield in practice such a linear growth of runtime on modern multi-core and many-core architectures (details are beyond the scope of this text).

E. Initial Ordering Effects

Fig. 2 shows the runtimes for sorting of nonzero elements of matrices from the RLO to the block-aware ordering. However, when we are looking for an optimal block size from $\mathcal{S}$ for a given matrix, we initially need to sort its nonzero elements from the input ordering (the RLO in our case) only once for the tested block size s_1. Then, for all other tested block sizes $s_k : k > 1$, the sorting algorithm takes as an input nonzero elements sorted with respect to the block size s_{k-1}.

Within our study, we considered block sizes $s_k = 2^k$, which implies $s_k = 2 \cdot s_{k-1}$ for $k > 1$. Moreover, we defined sorting keys according to the lexicographical ordering of blocks. Consequently, for $k > 1$, the nonzero elements were on the input of Algorithm 5 partially sorted, which should result in shorter sorting times. We performed an experiment to verify this assumption; the results are shown in Fig. 4. They clearly indicate that AQsort was able to take the advantage of such partially sorted data; the amount of spared time was significant, especially on Salomon CPU-based nodes.

F. Block Sizes Effects

Recall that in the previous text, we made an assumption that setting block sizes $s_k = 2^k$ should provide faster runs of Algorithm 5 due to the possibility of calculation of block indexes I and J by using cheap logical and bitwise operations. However, if we need to test block sizes other than the powers of 2, we cannot avoid integer division (6). To evaluate the

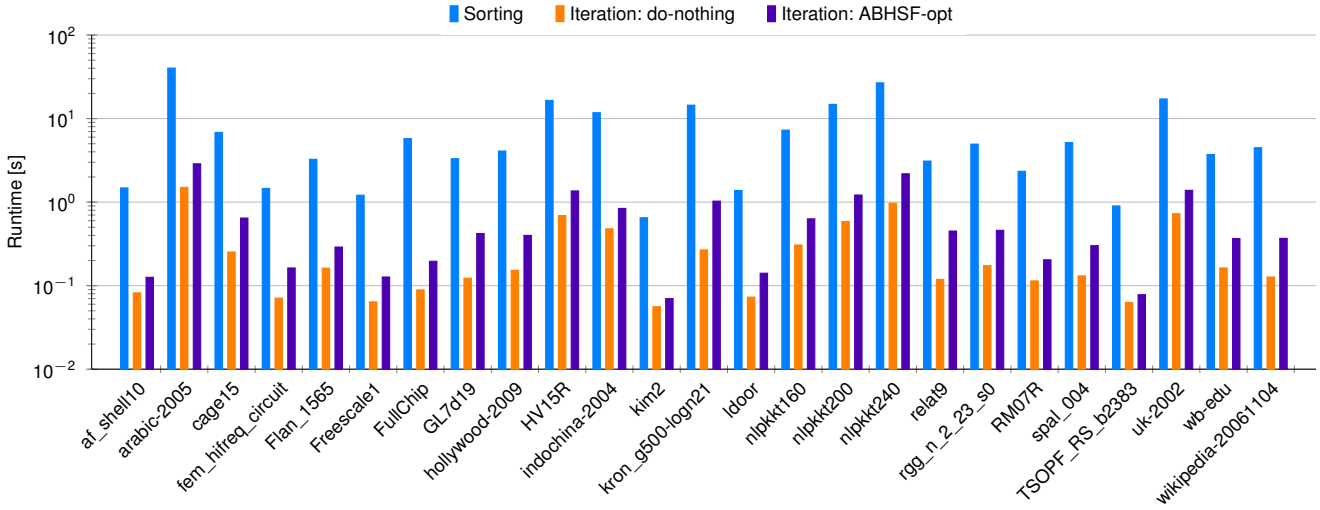


Fig. 6: Aggregated runtime of sorting and iteration phases of Algorithm 5 run 10 times in a row for block sizes $s = 2, 4, 8, \ldots, 1024$, measured for different matrices on a Salomon node using 24 threads. The iteration phases were measured for both the *do-nothing* and *ABHSF-opt* processors.

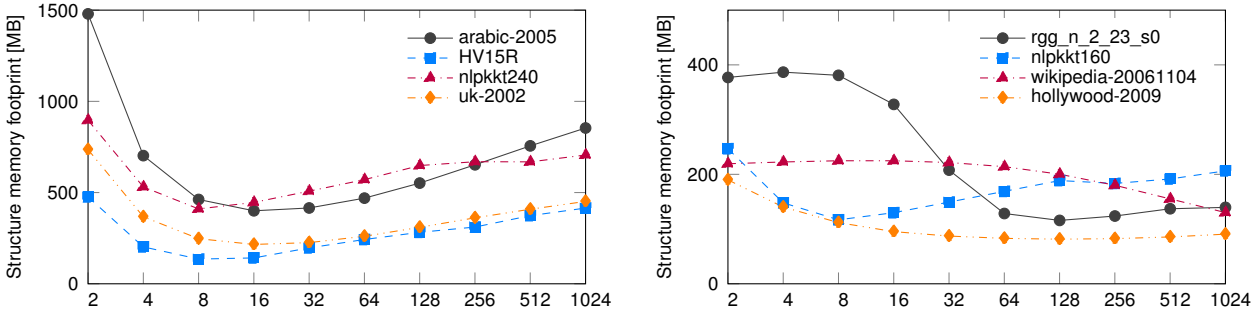


Fig. 7: Matrix structure memory footprints for different matrices stored in the ABHSF and different block sizes.

difference between both cases, we measured runtimes of Algorithm 5 and both its phases (sorting and iteration) for several block sizes of both classes $s = 2^k$ and $s \neq 2^k$; the results are presented in Fig. 5.

The conclusion is obvious—the way of deriving block indexes for the nonzero elements had a tremendous impact on the algorithm. Its runtime grew by almost a factor of 4 and 7 on Salomon nodes and Intel Xeon Phi, respectively, when using integer division instead of bitwise/logical operations.

G. ABHSF

Up to now, we presented measurements that used the *do-nothing processor* designed to evaluate Algorithm 5 itself. However, in practice, we iterate over nonzero blocks of sparse matrices to do something useful and the question is how the algorithm runtime will change in such cases. To answer this question, we substituted the *do-nothing processor* with the *ABHSF-opt* processor introduced in Section V-A and used Algorithm 5 for finding optimal block sizes for all tested matrices. The results of this experiment are presented in Fig. 6. They show aggregated runtimes of all 10 sorting phases as well as 10 iteration phases, and, for comparison, we show results for both types of block processors. We can observe that with the *ABHSF-opt processor*, the iteration phase took considerably longer times in comparison with the *do-nothing* processor. However, the overall runtime of the whole algorithm was still dominated by its sorting phase.

In regard to memory footprints of sparse matrices, we can usually focus only on *matrix structure memory footprints*, i.e., memory footprints of the information describing the structure of nonzero elements (compression of the values of nonzero elements pays off only for special kinds of matrices where same values emerge many times). For illustration, we show in Fig. 7 the relation between block sizes and the matrix structure memory footprints of selected matrices stored in the ABHSF. For most of the tested matrices, we have found only a single minimum, which typically corresponded to block sizes of 8 or 16 (left side of Fig. 7 and the nlpkkt160 matrix). However, we have also observed few "pathological" cases with different behavior (right side of Fig. 7). For example, the minimum

TABLE I: Matrix structure memory footprints in MB for selected matrices stored in the COO and CSR storage formats with 32-bit indexes and the ABHSF with optimal block sizes.

Matrix	COO	CSR	ABHSF
arabic-2005	4882.8	2528.2	400.1
hollywood-2009	438.8	223.8	81.5
HV15R	2159.7	1087.5	135.4
nlpkkt160	907.4	485.5	116.9
nlpkkt240	3061.2	1637.4	410.3
rgg_n_2_23_s0	484.5	274.2	115.8
uk-2002	2274.4	1207.9	215.7
wikipedia-20061104	300.5	162.2	130.1

for the wikipedia-20061104 matrix was not even found within the whole tested range $s = 2, 4, 8, \ldots, 1024$; such a result might indicate that the ABHSF is not a suitable format for this matrix.

We also present in Table I the comparison of the matrix structure memory footprints for selected matrices and 3 storage formats—COO, the *compressed sparse row* (CSR) format, and the ABHSF. CSR is likely the most commonly used format for sparse matrices, together with its *compressed sparse column* (CSC) counterpart (they are also often abbreviated as CRS and CCS). The measurements revealed that storing sparse matrices in the ABHSF can result in substantial memory savings.

VI. Related Work

We have proposed an algorithm for the purpose of storing matrices in a file system in the ABHSF [2, Algorithm 1]. This algorithm served as a starting point for the development of Algorithm 4 that was generalized for any UB format.

For examples of designed UB formats, see, e.g., [1], [5], [8]–[21]

VII. Conclusions

The contribution of this paper is an efficient scalable parallel algorithm for fast iteration over nonzero blocks of sparse matrices. This algorithm is a building block of a process of transformation of sparse matrices into UB storage formats. We have presented an extensive experimental study with the proposed algorithm using matrices from the UFSMC that came from different scientific and engineering disciplines and thus featured different characteristics. Measurements conducted on modern multi-core and many-core HPC architectures revealed that if the set of tested block sizes is chosen properly, the process of finding an optimal block size takes up to tens of seconds even for very large matrices.

The remaining question is whether or not it pays off to transform matrices into UB formats. The answer to this question is highly application-dependent. For instance, if a matrix is used within an iterative linear solver or an eigensolver, we would first need to know how many SpMV operations are applied to a given matrix and how much time this operation takes. In our future work, we want to focus on the ABHSF and undertake a research that should tell how many SpMV-based iterations need to be done with a given matrix to reduce the overall application runtime when considering matrix storage in this format.

Appendix

The list of sparse matrices from the UFSMC used in the experiments: 3Dspectralwave, af_shell10, arabic-2005, cage15, fem_hifreq_circuit, Flan_1565, Freescale1, FullChip, GL7d19, hollywood-2009, HV15R, indochina-2004, kim2, kron_g500-logn21, ldoor, nlpkkt160, nlpkkt200, nlpkkt260, relat9, rgg_n_2_23_s0, RM07R, spal_004, TSOPF_RS_b2383, uk-2002, wb-edu, wikipedia-20061104.

Acknowledgments

The authors acknowledge support from P. Tvrdík from the Czech Technical University in Prague, P. Vrchota and J. Fiala Výzkumný a zkušební letecký ústav, a.s., and M. Pajr from CQK Holding and IHPCI. The authors would like to thank M. Václavík of the Czech Technical University in Prague for providing an access to an Intel Xeon Phi accelerator installed at the Star university cluster.

References

[1] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, "Adaptive-blocking hierarchical storage format for sparse matrices," in *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2012)*. IEEE Xplore Digital Library, 2012, pp. 545–551.

[2] D. Langr, I. Šimeček, and P. Tvrdík, "Storing sparse matrices in the adaptive-blocking hierarchical storage format," in *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2013)*. IEEE Xplore Digital Library, 2013, pp. 479–486.

[3] D. Langr and P. Tvrdík, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 428–440, 2016. doi: 10.1109/TPDS.2015.2401575

[4] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," 2016, accessed April 8, 2016 at http://www.agner.org/optimize/instruction_tables.pdf.

[5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. Philadelphia, PA: SIAM, 1994.

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. ISBN 0898715342

[7] T. A. Davis and Y. F. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011. doi: 10.1145/2049662.2049663

[8] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient SMVM kernels," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009. doi: 10.1145/1542275.1542294. ISBN 978-1-60558-498-0 pp. 100–109.

[9] ——, "A library for pattern-based sparse matrix vector multiply," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 62–87, 2011. doi: 10.1007/s10766-010-0145-2

[10] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009. doi: 10.1145/1583991.1584053. ISBN 978-1-60558-606-9 pp. 233–244.

[11] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. IEEE Computer Society, 2011. doi: 10.1109/IPDPS.2011.73 pp. 721–733.

[12] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010. doi: 10.1145/1693453.1693471 pp. 115–126.

[13] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," in *Proceedings of the International Conference on Computational Science (ICCS 2001), Part I*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, vol. 2073, pp. 127–136.

[14] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004. doi: 10.1177/1094342004041296

[15] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, Aug 2009. doi: 10.1109/CSE.2009.223 pp. 247–256.

[16] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance modeling and analysis of cache blocking in sparse matrix vector multiply," University of California, Tech. Rep. UCB/CSD-04-1335, 2004.

[17] ——, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007. doi: 10.1007/s00200-007-0038-9

[18] I. Šimeček, D. Langr, and P. Tvrdík, "Space-efficient sparse matrix storage formats for massively parallel systems," in *Proceedings of the 14th IEEE International Conference of High Performance Computing and Communications (HPCC 2012)*. IEEE Computer Society, 2012. doi: 10.1109/HPCC.2012.18 pp. 54–60.

[19] I. Šimeček and D. Langr, "Space and execution efficient formats for modern processor architectures," in *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2015)*. IEEE Computer Society, 2015. doi: 10.1109/SYNASC.2015.24 pp. 98–105.

[20] F. S. Smailbegovic, G. N. Gaydadjiev, and S. Vassiliadis, "Sparse Matrix Storage Format," in *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2005*, 2005, pp. 445–448.

[21] P. Stathis, S. Vassiliadis, and S. Cotofana, "A hierarchical sparse matrix storage format for vector processors," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, p. 61.