DOI: 10.15439/2016F366
ACSIS, Vol. 8. ISSN 2300-5963

Efficient parallel evaluation of block properties of sparse matrices

Ivan Šimeček, Daniel Langr
Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems
Thákurova 9, 160 00, Praha, Czech Republic
Email: {xsimecek,langrd}@fit.cvut.cz

***Abstract*—Many storage formats for sparse matrices have been developed. Majority of these formats can be parametrized, so the algorithm for finding optimal parameters is crucial. For overall efficiency, it is important to reduce the execution time of this preprocessing. In this paper, we propose a new algorithm for the determination of the number of nonzero blocks of the given size in a sparse matrix. The proposed algorithm requires relatively a small amount of auxiliary memory. Our approach is based on the Morton reordering and bitwise manipulations. We also present a parallel (multithreaded) version and evaluate its performance and space complexity.**

I. Introduction

COMPUTATIONS with sparse matrices are widespread in scientific projects. Many storage formats for sparse matrices have been developed. The most straightforward approach is to accompany values of nonzero elements with their row and column indexes, which forms the *coordinate* format (COO, for details see Sec. I-C). If the local matrix nonzero elements are ordered lexicographically, then row indexes in COO can be substituted by the number of nonzero elements of each row. Such idea is represented by the *compressed sparse row* (CSR, for details see Sec. I-D) format. Due to matrix sparsity, the memory access patterns in common formats (like COO or CSR) are irregular and the utilization of cache suffers from low spatial and temporal locality, hence other (so called advanced) formats are used in practice. These advanced formats are usually parametrized,

A. General notation

We consider a matrix A of order $n \times n$, $A = (a_{i,j})$. The number of its nonzero elements is denoted by N. Matrix A is considered *sparse* if it is worth (for performance or any other reason) not to store this matrix in memory in a dense array. In the following text:

- We assume that indexes of all vectors and matrices start from zero.
- We assume that $1 \ll n \leq N \ll n^2$.
- The number of nonzero elements in submatrix B of matrix A is denoted by $\eta(B)$, so $\eta(A) = N$
- For any submatrix C, if $\eta(C) = 0$ then the submatrix C is called zero submatrix, otherwise it is called nonzero submatrix.
- If all elements in A have the same probability N/n^2 to be nonzero (independently on other elements) then A has a *uniform distribution* of nonzero elements.
- The parameter th denotes the number of threads used for the execution of an algorithm.

This work was supported by internal CTU grant No. SGS16/122/OHK3/1T/18 of the Czech Technical University in Prague. 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).

B. Banded matrices

Citing from Golub and Van Loan [1]:

Definition 1:

If all matrix elements are zero outside a diagonally bordered band whose range is determined by constants k_1 and k_2:

$$a_{i,j} = 0 \quad \text{if} \quad j < i - k_1 \quad \text{or} \quad j > i + k_2, \quad k_1, k_2 \geq 0.$$

Then the quantities k_1 and k_2 are called the left and right half-bandwidth, respectively. The bandwidth of the matrix (denoted by $\omega(A)$) is $k_1 + k_2 + 1$.

Definition 2: If $\omega(A) \ll n$, then A is *banded.*

C. The Coordinate (COO) format

The coordinate (COO) format is the simplest format for storing sparse matrices (see [2], [3]). The matrix A is represented by three linear arrays $values$, $xpos$, and $ypos$. The array $values[0, \ldots, N-1]$ stores the nonzero values of A, arrays $xpos[0, \ldots, N-1]$ and $ypos[0, \ldots, N-1]$ contain column and row indexes, respectively, of these nonzero values. The ordering of elements in this format is not prescribed.

D. The Compressed Sparse Row (CSR) format

The most common format for storing sparse matrices is the *compressed sparse row* (CSR) format (see [2]–[7]). The matrix A stored in the CSR format is represented by three linear arrays: $values$, $addr$, and ci. The array $values[0, \ldots, N-1]$ stores the nonzero elements of A, the array $addr[0, \ldots, n]$ contains indexes of initial nonzero elements of rows of A. The array $ci[0, \ldots, N-1]$ contains column indexes of nonzero elements of A.

E. Hierarchical formats

Many hierarchical formats are based on the idea of partition the matrix into square disjoint *blocks* of size $2^c \times 2^c$ rows/columns, where $c \in N^+$ is a formal parameter. In the further text, we will denote them as basic hierarchical format (BHF), for details see [6], [8], [9]. Coordinates of the upper left corners of these blocks are aligned to multiples of 2^c. Thus, indexes of nonzero elements are separated in two parts, indexes of blocks and indexes inside the blocks. Every such a region has *block row* and *block column* indexes. Let $B(c)$ denote the number of nonzero blocks for matrix A. The minimal number of nonzero blocks is equal to $B(c)_{min} = \lceil \frac{N}{2^{2c}} \rceil$, if all nonzero blocks contain only nonzero elements (i.e., are 100% dense). The maximal number of nonzero blocks is equal to $B(c)_{max} = \min\left(N, \lceil \frac{n}{2^c} \rceil^2\right)$, if each nonzero block contains exactly one nonzero element or if the whole matrix A is covered by nonzero blocks. This idea is for example behind formats: COOCOO format [10], ABHSF [8], [9], multilevel format [11], and so on. For all these formats, the optimal value of bits for each level should be computed. For this decision, the information about number of blocks $B(c)$ for given c is required.

F. Our requirements for an algorithm

Our assumptions and the requirements for an algorithm for computation of the number of nonzero blocks are as follows:

- The algorithm should be execution efficient (even in the multithreaded environment).
- Since, we are processing large sparse matrices, we assume that the space complexity (memory footprint) of the sparse matrix A is significant. We define an *in-place* algorithm as an algorithm that needs auxiliary data structures with space complexity strictly lower than the number of matrix nonzero elements. By in-place computation we mean an algorithm with $o(N)$ space complexity, in contrast to $O(N)$. It is the typical situation in HPC (high performance computing) that the matrix typically represents the largest object stored in the main memory. Hence, the algorithm should be also space-efficient, i.e., space complexity of all its temporary data should be much lower than space complexity of the matrix. When the computation of the number of nonzero blocks of a matrix cannot be performed in-place, then there might not be enough free memory to make this computation at all.
- In real situation, we don't need to compute the number of nonzero blocks for all c from $[1, \ldots, \lceil \log n \rceil]$ because some block sizes from this interval are simply too small or too large for the given purpose. Thus, we need to compute the number of nonzero blocks only for c from the given interval $[c\_min, \ldots, c\_max]$.

G. Overview of state-of-the-art

As far as we know, there are only two algorithms for the computation of the number of nonzero blocks (e.g., in [10]), both of them are based on sets. There are two main reasons for such low number:

- Authors rarely present algorithms for preprocessing of matrices, which are necessary for assessment of the suitability of their formats for a given application [6]. We have not found a case where format authors provided efficient parallel implementations of algorithms for the computation of the number of nonzero blocks in non-experimental forms.
- Register blocking formats (like SPARSITY [12] or [13] or CARB [5], [14]) store a matrix as a set of small dense blocks. Blocks can be linear (horizontal, vertical, or diagonal) or rectangular, a usual range of size is from 2 to 20. Since the block-size is not limited to the power of 2 and optimization criteria are more complex, a special transformation algorithm is used.
- Some formats (e.g., [11], [15]–[17]) skip this computation and use "typically good" values of the block-size.

The idea behind the two found solutions is to evaluate the number of blocks $B(c)$ for all values of parameter c from $[c\_min, \ldots, c\_max]$ using a set. We will consider four implementations of such an algorithm using different data-structure for implementation of the set.

1) First algorithms based on sets: All nonzero elements are mapped to block coordinates. These block coordinates are mapped to index $i \in \langle 0, \ldots, \lceil n/2^c \rceil^2 \rangle$ and put into set U. Finally, the cardinality of U is determined. It is equal to the number of blocks $B(c)$.

Algorithm 1 Determination of the number of blocks $B(c)$

```
1: procedure NUMBEROFBLOCKS1($I$,$c$)
Input: $In$ = a matrix in the CSR format
Input: $c$ = the parameter (logarithm of block size)
Output: $B$ = the number of blocks
2:     $B \leftarrow 0$;  $d \leftarrow 2^c$;  $p \leftarrow \lceil n/d \rceil$;
3:     construct the set $U$;
4:     $U = \emptyset$;
5:     for $y \leftarrow 1, In.n$ do
6:         for $j \leftarrow In.addr[y], In.addr[y+1] - 1$ do
7:             $x \leftarrow In.ci[j]$;
8:             $i \leftarrow \lfloor y/d \rfloor \cdot p + \lfloor x/d \rfloor$;
9:             put the element $i$ into $U$;
10:    $B \leftarrow |U|$;
11:    return $B$;
```

The time complexity of Algorithm 1, $T_1(n, N)$, consists of

- t_1 = time complexity of creating an empty set U at codeline (4),
- $t_2 = N \cdot t_{\text{ins}}$ = time complexity of inserting N elements at codeline (9) into the set U,
- t_3 = time complexity of computing the cardinality of U at codeline (10).

The time complexity depends on the data structure used for implementing the set U. Let $d = 2^c$ and $p = \lceil n/d \rceil$. Let us consider four basic implementations:

1) a bit array (of size of $p^2 = \lceil n/d \rceil^2 = \lceil n/2^c \rceil^2$ bits): Then

- t_1 is proportional to the range of indices i: $t_1 = O(p^2)$,
- $t_2 = N \cdot O(1) = O(N)$,
- t_3 is also proportional to the range of indices i: $t_3 = O(p^2)$.

The total time complexity is $T_1(n,N) = O(N + p^2) = O(N + \lceil n/2^c \rceil^2)$ and the space complexity is $O(p^2) = O(\lceil n/2^c \rceil^2)$. These complexities are high (especially for small values of c) which makes this approach inefficient.

2) A linked list:

- $t_1 = N \cdot O(1) = O(N)$,
- $t_2 = N \cdot O(B(c)_{max})$,
- $t_3 = O(B(c)_{max})$.

The total time complexity is $T_1(n,N) = O(N \cdot B(c)_{max})$ and the space complexity is $O(B(c)_{max})$. This complexity is high which makes this approach inefficient.

3) A balanced binary search tree:

- $t_1 = O(1)$,
- t_2 is proportional to N and to the logarithm of the size of binary representation of index i: $t_{\text{ins}} = O(2\log p) = O(\log(n/2^c)) = O(\log n - c)$,
- $t_3 = O(B(c)_{max})$.

The total time complexity is $T_1(n,N) = O(B(c)_{max} + N(\log n - c))$ and the space complexity is $O(B(c)_{max})$, hence the approach is quite execution efficient, but space inefficient (especially for small values of c).

4) A hash table of size l (we assume a closed hashing scheme):

- $t_1 = O(l)$,
- $t_2 = N \cdot O(1)$ in ideal case,
- $t_3 = O(l)$.

We must carefully choose the value of parameter l to satisfy the no-collision assumption during the insertion of elements, this parameter should be greater than $B(c)_{max}$. Then, the total time complexity is $T_4(n,N) = O(N + l) = O(N + B(c)_{max})$ and the space complexity is $O(B(c)_{max})$, hence the approach is execution efficient, but space inefficient (especially for small values of c).

2) An improved algorithm: We can improve Algorithm 1 as follows: The matrix A is divided into disjoint horizontal strips whose height is equal to 2^c. Then the cardinality of set U (i.e., the number of blocks) is determined in each strip separately. The value of $B(c)_{max}$ is decreased to $\lceil \frac{n}{2^c} \rceil$ that occurs if the whole strip is covered by nonzero regions.

Algorithm 2 Determination of the number of blocks $B(c)$ (improved)

```
1: procedure NumberOfBlocks2($In$,$c$)
Input: $In$ = a matrix in the CSR format
Input: $c$ = the parameter (logarithm of block size)
Output: $B$ = the number of blocks
2:     $B \leftarrow 0$; $old \leftarrow -1$;
3:     $d \leftarrow 2^c$;
4:     create the set $U$;
5:     $U = \emptyset$;
6:     for $y \leftarrow 1, In.n$ do
7:         if $\lfloor y/d \rfloor > old$ then
8:             $B \leftarrow B + |U|$;
9:             $U = \emptyset$;
10:            $old \leftarrow \lfloor y/d \rfloor$;
11:        for $j \leftarrow In.addr[y], In.addr[y+1] - 1$ do
12:            $x \leftarrow In.ci[j]$;
13:            $i \leftarrow \lfloor x/d \rfloor$;
14:            put the element $i$ into $U$;
15:    $B \leftarrow B + |U|$;
16:    return $B$;
```

The time complexity of Algorithm 2, $T_2(n,N)$, consists of:

- $t_1 = p \cdot t_{\text{init}}$ = time complexity of creating empty sets at codelines (5) and (9),
- $t_2 = N \cdot t_{\text{ins}}$ = time complexity of inserting N elements at codeline (14),
- $t_3 = p \cdot t_{\text{enum}}$ = time complexity of determining the cardinality of the set at codelines (8) and (15).

Four basic implementations of the set data type provide the following time complexities:

1) a bit array (of size of $\lceil n/d \rceil = \lceil n/2^c \rceil$ bits):
   - $t_{\text{init}} = O(p)$,
   - $t_{\text{ins}} = O(1)$,
   - $t_{\text{enum}} = O(p)$,

   $T_2(n,N) = O(N + p^2)$ and the space complexity is $O(p)$. So, the space complexity decreased compared to same implementation of the set data type in Algorithm 1, but the time complexity remains high, especially for small values of c.

2) a linked list:
   - t_{init} remain the same as in the previous case,
   - $t_{\text{enum}} = t_{\text{ins}} = O(B(c)_{max})$ drops due to $B(c)_{max}$ decrease.

   The total time and the space complexities remain the same compared to the same implementation of the set data type in Algorithm 1.

3) a balanced binary search tree:
   - t_{init} remains the same as in the previous case,
   - $t_{\text{ins}} = O(\log p) = O(\log(n/2^c)) = O(\log n - c)$.
   - $t_{\text{enum}} = O(B(c)_{max})$ drops due to $B(c)_{max}$ decrease.

The total time complexity remains the same compared to the same implementation of the set data type in Algorithm 1.

4) a hash table with the open hashing scheme (the size of the hash table is equal to l). It is much easier to satisfy the no collision assumption by insertion of elements (with lesser value of l).
   - t_{ins} is the same,
   - t_{init} and t_{enum} drops due to $B(c)_{max}$ and l decrease.

   The total time complexity remains the same.

3) The summary of state-of-the-art algorithms: None of the algorithm and implementation satisfies the requirements describe in Sec. I-F. The improved algorithm reduces memory requirements which allows implementations to be space efficient. On the other hand, existing algorithms are execution inefficient (especially for small values of c). The situation is getting worse for multithreaded execution. If each strip is computed in parallel then every thread has independent instance of the set and memory requirements will be th-times higher. If single strip is computed by multiple threads then the only one (shared) set is used, but every access is a critical section, hence the speedup would be minimal.

II. Our new approach

A. Main idea

In our approach, we utilize reordering of nonzero elements according to so-called Morton order (for details see [18]). Morton ordering is a mapping from an i-dimensional space onto a linear list of numbers. If we want to convert a certain set of integer coordinates to a Morton code, we have to interleave the binary representations of each coordinate. Here is an example of transformation from 3D coordinates into Morton code.

$$(x, y, z) = (5, 9, 1)_{10} = (0101, 1001, 0001)_2$$

Interleaving the bits results in: $(010\ 001\ 000\ 111)_2 = (1095)_{10}$-th cell along the co called Z-curve. For the determination of the number of blocks, we use the following lemma.

Lemma 1: Morton codes for all elements inside the block of size 2^c that is aligned to multiple of 2^c are the same except $2 \cdot c$ least significant bits.

The proof of Lemma 1 is obvious (based on the construction of Morton code). Hence, we can design an algorithm based on this lemma: in a sorted sequence of nonzero elements, we count differences in Morton codes of two adjacent items (more exactly: the positions of highest bit set in the result of logical XOR of two adjacent items). We call this algorithm Morton-based (see Algorithm 3).

Algorithm 3 Determination of the number of blocks $B(c)$ (new)

```
 1: procedure NumberOfBlocks3($In$,$c$)
Input: $A$ = a matrix in the COO format
Input: $c$ = the parameter (logarithm of block size)
Output: $B$ = the number of blocks
 2:     for $j \leftarrow 1, c\_max$ do
 3:         $number[j] \leftarrow 1$;
 4:     add to every nonzero element its Morton code;
 5:     sort nonzero elements on its Morton code;
 6:     $old \leftarrow A[0].Morton$;
 7:     for $i \leftarrow 1, N$ do
 8:         $new \leftarrow A[i].Morton$;
 9:         $diff \leftarrow XOR(new, old)$;
10:         $old \leftarrow new$;
11:         $k \leftarrow round\_up(Highest1(diff)/2)$;
12:             ▷ Highest1 = the index of the position of the
    highest bit set
13:         for $j \leftarrow 1, k$ do
14:             $number[j] \leftarrow number[j] + 1$;
15:     return $number[]$;
```

The time complexity of Algorithm 3, $T_3(n, N)$, consists of:

- $t_1 = N$ = time complexity of generating Morton codes at codeline (4),
- $t_2 = N \log N$ = time complexity of sorting. We assume the sorting algorithm with complexity $O(i \log i)$ for a array of length i.
- $t_3 = N \cdot c\_max$ = time complexity of for-cycle at codeline (7) mainly inner loop at codeline (13).

Overall time complexity is $T_3(n, N) = N(c\_max + \log N)$, so this algorithm is efficient, but requires additional space for Morton codes proportional to N. To avoid this, we propose two solutions:

1) Morton codes are not stored explicitly. They can overwrite COO storage format (arrays $xpos$ and $ypos$). After determination of the number of blocks, the coordinates (original values in these arrays) will be restored from Morton codes.
2) Computation of Morton code can be included in a compare function of sorting algorithm at codeline (5) in Algorithm 3.

B. Example of algorithm usage

Let us assume a very small example of a sparse matrix with $n = 8$ and $N = 12$. Instead of the values of the matrix elements, we deal only with binary flags indicating the existence of nonzero elements.

$$\mathbf{M^{(0)}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The steps of the new algorithm are as follows:

Step 1: (codeline 4) For each nonzero element, the Morton code is computed. Morton codes (for elements in lexicographic order) = { 000000, 010101, 000011, 010110, 001100, 001111, 011010, 110011, 101000, 111100, 101011, 111111 }.

Step 2: (codeline 5) The whole sequence is sorted according to Morton codes.

Step 3: (codeline 7-14) We count difference in Morton codes of adjacent items. Morton codes (for elements the highest different bit set is marked) ={ 000000, $0000\bar{1}1$, $00\bar{1}100$, $0011\bar{1}1$, $0\bar{1}0101$, $0101\bar{1}0$, $01\bar{1}010$, $\bar{1}01000$, $1010\bar{1}1$, $1\bar{1}0011$, $11\bar{1}100$, $1111\bar{1}1$ }. The counters (in variable $numbers$) are increased according to the position of the highest bit set (e.g., $0000\bar{1}1 \Rightarrow$ increase $numbers[\mathit{1}]$ by one, $00\bar{1}100 \Rightarrow$ increase $numbers[\mathit{1}]$ and $numbers[\mathit{2}]$ by 1, etc.).

Step 4: (codeline 15) After traversing the example sequence, the counters are set to $B(c = 1) = numbers[\mathit{1}] = 7$, $B(2) = 4$, $B(3) = 1$.

C. Parallelization

1) SW technologies: The OpenMP API specification [19] is defined by a collection of compiler directives, library routines and environment variables extending the C, C++ and Fortran languages. These can be used to create portable parallel programs utilizing shared memory. The core of OpenMP is the so called *fork-join model* execution model. An application employing OpenMP usually begins as a single thread program and during execution uses multiple threads or even other devices to perform parallel tasks.

The OpenMP API provides a relaxed-consistency, shared memory model. All threads have access to the memory and each may have its own temporary view of the memory (which represents cache or other local storage used for caching). Each thread also have access to thread private memory, which cannot be accessed by any other thread. A single access to a variable is not guaranteed to be atomic with respect to other accesses of that variable, since it may be implemented with multiple load or store instructions. If multiple threads write without synchronization to the same memory unit, the data race occurs.

2) Multithreaded execution: The parallel (multithreaded) version of the algorithm is represented by Alg. 4. We simply make all steps (described in Sec. II-B) parallel:

Step 1 (Computation of Morton codes): The parallelization of this step is straightforward.

Step 2 (Sorting of Morton codes): The parallelization of this step is straightforward by using any parallel in-place sort (e.g., `sort` method from `std::algorithm` [20] or AQsort [21])

Step 3 (Counting the differences): The whole sequence of sorted Morton codes is partitioned into th disjoint chunks of consecutive elements. Each chunk contains approximately the same number of elements. Each thread counts the differences in the assigned chunk independently. After this computation, local (thread private) instances of $l\_number[]$ are summed up to the global values ($number[]$).

Algorithm 4 Determination of the number of blocks $B(c)$ (new, parallel version)

```
1: procedure NumberOfBlocks4($In$,$c$)
Input: $A$ = a matrix in the COO format
Input: $c$ = the parameter (logarithm of block size)
Output: $B$ = the number of blocks
2:     parallel add to every nonzero element its Morton code;
3:     parallel sort nonzero elements on its Morton code;
4:     $len = N/th$;
5:     start of parallel block
6:     $tid = get\_tid\_of\_current\_thread()$;
7:     if $tid = 0$ then
8:         $old \leftarrow A[0].Morton$;
9:         $start \leftarrow 1$;
10:        for $j \leftarrow 1, c\_max$ do
11:            $l\_number[j] \leftarrow 1$;
12:    else
13:        $old \leftarrow A[tid \cdot len - 1].Morton$;
14:        $start \leftarrow tid \cdot len$;
15:        for $j \leftarrow 1, c\_max$ do
16:            $l\_number[j] \leftarrow 0$;
17:    for $i \leftarrow start, (tid + 1) \cdot len - 1$ do
18:        $new \leftarrow A[i].Morton$;
19:        $diff \leftarrow XOR(new, old)$;
20:        $old \leftarrow new$;
21:        $k \leftarrow round\_up(Highest1(diff)/2)$;
22:        for $j \leftarrow 1, k$ do
23:            $l\_number[j] \leftarrow l\_number[j] + 1$;
24:    end of parallel block
25:    parallel reduction(+) of $l\_number$ into $number$;
26:    return $number[]$;
```

D. Modification of algorithm suitable for the CSR format

Our algorithms (see Alg. 3 and 4) require the input storage format that allows to reorder/sort its nonzero elements. Hence, we use the COO format. This format is used in HPC, but the CSR format is more frequent (see for example [22], [23]). For the CSR format, we propose the following modification.

Step 1 The whole matrix is partitioned into 2D disjoint regions (=chunks of consecutive rows). Starting row of each region are aligned to multiple of 2^{c_max}.

Step 2 For each region all nonzero elements in this region are extracted and their Morton codes are stored into a temporary array. These regions are proceeded by Alg. 3.

Step 3 The results for all regions are summed up to the global values.

The parallelization of this algorithm is easy: each region can be computed independently by a different thread. For good load balancing in OpenMP API even for matrices with non-uniform distribution (e.g., for banded matrices), so-called dynamic scheduling strategy is used.

Matrix	abbr	n	N	avg_per_row
`circuitM5`	m1	$5.56 \cdot 10^6$	$5.95 \cdot 10^7$	10.7
`nlpkkt120`	m2	$3.54 \cdot 10^6$	$5.02 \cdot 10^7$	14.1
`ldoor`	m3	$9.52 \cdot 10^5$	$2.37 \cdot 10^7$	24.9
`TSOPF_RS_b2383`	m4	$3.81 \cdot 10^4$	$1.62 \cdot 10^7$	42.5
`mouse_gene`	m5	$4.51 \cdot 10^4$	$1.45 \cdot 10^7$	32.1
`t2em`	m6	$9.25 \cdot 10^5$	$4.59 \cdot 10^6$	5.0
`bmw7st_1`	m7	$1.41 \cdot 10^5$	$3.74 \cdot 10^6$	26.5
`amazon0312`	m8	$4.01 \cdot 10^5$	$3.20 \cdot 10^6$	8.0
`thread`	m9	$2.97 \cdot 10^4$	$2.25 \cdot 10^6$	75.8
`gupta2`	m10	$6.21 \cdot 10^4$	$2.16 \cdot 10^6$	34.8

TABLE I: Characteristics of the testing matrices and their abbrevation in the further text.

III. Evaluation of the results

A. Testing matrices

We have used 10 testing matrices from various application domains from the University of Florida Sparse Matrix Collection [24]. Table I shows the characteristics of the testing matrices.

B. Used HW and SW

The execution times were measured on a server with following HW and SW parameters:

- 2 × CPU Intel Xeon Processor E5-2620 v2 (15MB L3 cache per CPU),
- CPU cores: 6 per CPU, 12 in total,
- Memory size: 32 GB RAM, total max. memory bandwidth: 51.2 GB/s,
- Peak single precision floating point performance 0.48 Tflops (using base clocks),
- OS Linux, C++ compiler (g++) version 4.8.3 with switches `-O3 -march=native -mavx -fopenmp`.

We measure elapsed wall clock times using OpenMP function `omp_get_wtime()`.

C. Evaluation of results

1) Comparison of sequential algorithms: Tables II and III show the comparison of measured times for different algorithms for the determination of the number of blocks. From this table, we can conclude that our Morton-based algorithm is always faster for larger matrices. The reason is the following: the time complexity of classical algorithm is $O(n^2+N)$, hence for smaller matrices (with small order) both components are approximately the same and algorithm is execution-efficient. On the other hand, for larger matrices in the complexity of the classical algorithm the component n^2 become dominant and algorithm is execution-inefficient. For small matrices the initial overhead of the Morton-based algorithm is significant. For larger matrices, this overhead become negligible and this algorithm is execution-efficient.

Matrix	CL($th = 1$)	CL($th = 12$)	NEW($th = 1$)	NEW($th = 12$)
m1	2834	265.9	3.56	0.348
m2	1149	107.8	3.19	0.267
m3	82.8	7.88	1.28	0.107
m4	0.313	0.053	0.959	0.243
m5	0.523	0.044	1.19	0.103
m6	77.3	7.22	0.209	0.018
m7	1.49	0.124	0.204	0.017
m8	14.2	1.26	0.284	0.024
m9	0.091	0.008	0.132	0.012
m10	0.284	0.024	0.142	0.020

TABLE II: Measured times in seconds for the determination of the number of blocks ($c\_max = 8$): CL denotes the classical improved algorithm, NEW denotes the Morton-based algorithm.

Matrix	CL($th = 1$)	CL($th = 12$)	NEW($th = 1$)	NEW($th = 12$)
m1	2839	414	4.38	1.52
m2	1151	168	4.19	0.476
m3	83.4	11.7	1.67	0.212
m4	0.562	0.566	0.563	0.565
m5	0.742	0.746	0.743	0.749
m6	77.4	11.5	0.273	0.041
m7	1.41	0.670	0.259	0.135
m8	14.4	2.39	0.388	0.130
m9	0.128	0.128	0.157	0.157
m10	0.309	0.325	0.154	0.154

TABLE III: Measured times in seconds for the determination of the number of blocks ($c\_max = 16$): CL denotes the classical (improved) algorithm, NEW denotes the Morton-based algorithm.

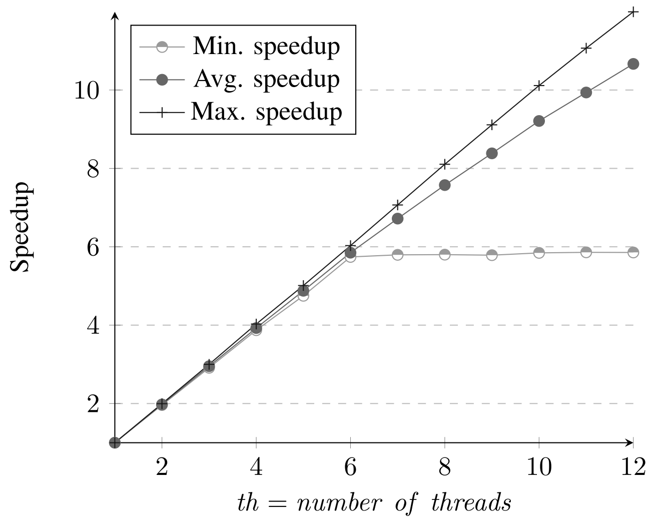

Fig. 1: Speedups for classical (improved) algorithm with $c\_max = 8$.

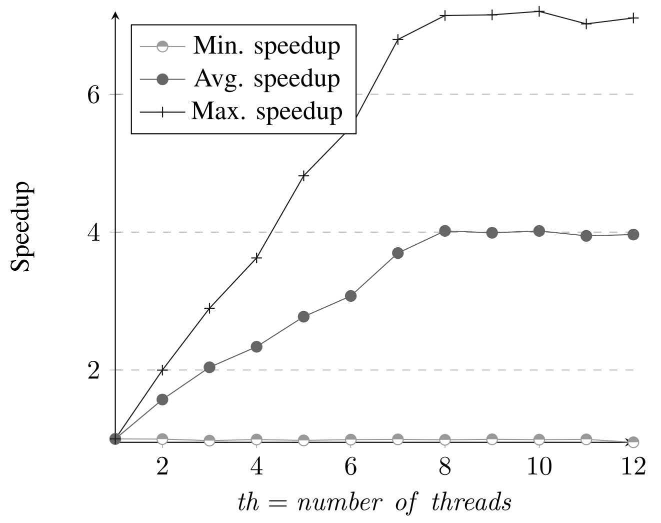

Fig. 2: Speedups for classical (improved) algorithm with $c\_max = 16$.

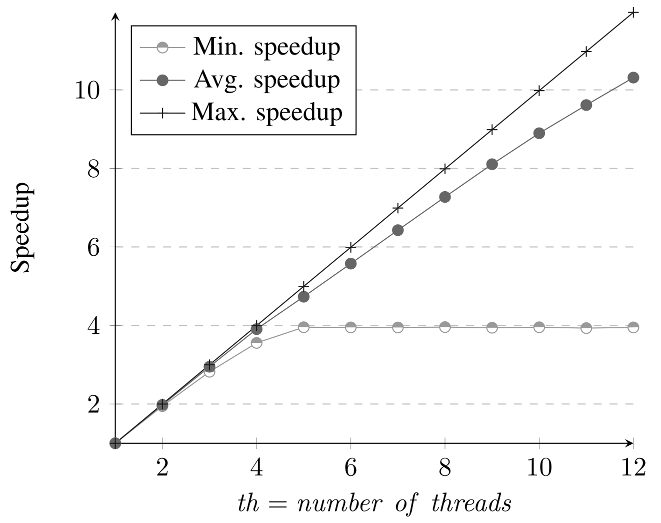

Fig. 3: Speedups for Morton-based algorithm with $c\_max = 8$.

2) Comparison of parallel algorithms: Tables II and III show the comparison of measured times for different algorithms for the determination of the number of blocks. From Fig. 1, 2, 3, and 4 we can conclude that both algorithms scale very well (speedup is equal to the number of threads) for $c\_max = 8$. For $c\_max = 16$, the scalability is getting worse since parallelization is based on regions (see Section I-G3 and II-D). For small matrices (with small order), the parameter 2^{c_max} is comparable with the order of the matrix. In this case, the load-balance is not good and majority of threads is idle and the speedup is almost independent on the number of threads.

IV. Conclusions

This paper presents the design of a new algorithm for the for the calculation of the number of blocks in sparse matrices. This

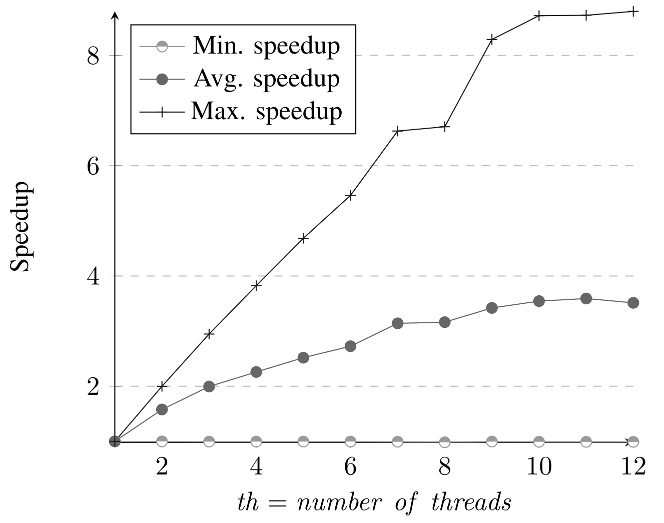

Fig. 4: Speedups for Morton-based algorithm with $c\_max = 16$.

algorithm is crucial for preprocessing of matrices into some advanced storage formats. We have also developed a parallel version of this algorithm. We performed experiments on the parallel system and their results showed that the proposed algorithm is both execution- and space-efficient.

Acknowledgments

The authors would like to thank M. Václavík of the Czech Technical University in Prague for providing an access to the Star university cluster.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd ed.).* Baltimore: Johns Hopkins, 1996.

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

[3] 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.

[4] I. Šimeček and P. Tvrdík, "A new approach for accelerating the sparse matrix-vector multiplication," in *Proceedings of 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '06).* Los Alamitos: IEEE Computer Society, 2006, pp. 156–163. [Online]. Available: http://dl.acm.org/citation.cfm?id=1264261

[5] ——, "Sparse matrix-vector multiplication — final solution?" in *Parallel Processing and Applied Mathematics*, ser. PPAM'07, vol. 4967. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 156–165. [Online]. Available: http://www.springerlink.com/content/48x1345471067304/

[6] 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.

[7] B. Bylina, J. Bylina, P. Stpiczyński, and D. Szałkowski, "Performance analysis of multicore and multinodal implementation of spmv operation," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. P. M. Ganzha, L. Maciaszek, Ed., vol. 2. IEEE, 2014, pp. pages 569–576. [Online]. Available: http://dx.doi.org/10.15439/2014F313

[8] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, "Adaptive-blocking hierarchical storage format for sparse matrices," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE Xplore Digital Library, September 2012, pp. 545–551.

[9] 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, September 2013, pp. 479–486.

[10] I. Šimeček, D. Langr, and P. Tvrdík, "Space-efficient sparse matrix storage formats for massively parallel systems," in *High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, ser. HPCC'12, Liverpool, Great Britain, june 2012, pp. 54–60.

[11] I. Šimeček and D. Langr, "Space and execution efficient formats for modern processor architectures," in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Sept 2015, pp. 98–105.

[12] E. Im, *Optimizing the Performance of Sparse Matrix-Vector Multiplication - dissertation thesis*, University of California at Berkeley, 2001.

[13] M. Martone, M. Paprzycki, and S. Filippone, "An improved sparse matrix-vector multiply based on recursive sparse blocks layout," in *Large-Scale Scientific Computing*, ser. Lecture Notes in Computer Science, I. Lirkov, S. Margenov, and J. Waśniewski, Eds. Springer Berlin Heidelberg, 2012, vol. 7116, pp. 606–613. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29843-1_69

[14] P. Tvrdík and I. Šimeček, "A new diagonal blocking format and model of cache behavior for sparse matrices," in *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, ser. PPAM'05, vol. 12, no. 4. Poznan, Poland: Springer-Verlag, 2005, pp. 164–171. [Online]. Available: http://dl.acm.org/citation.cfm?id=2096870.2096894

[15] I. Šimeček and D. Langr, "Space-efficient sparse matrix storage formats with 8-bit indices," in *Seminar on Numerical Analysis*. Liberec: Technical University of Liberec, 2012, pp. 161–164. [Online]. Available: http://shimi.webzdarma.cz/vyzkum/sna12/article.pdf

[16] M. Martone *et al.*, "On the usage of 16 bit indices in recursively stored sparse matrices," *Symbolic and Numeric Algorithms for Scientific Computing*, vol. 0, pp. 57–64, 2010.

[17] ——, "Use of hybrid recursive CSR/COO data structures in sparse matrices-vector multiplication," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, Wisla, Poland, October 2010.

[18] G. M. Morton, *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. IBM Ltd., 1966.

[19] OpenMP Architecture Review Board, "Openmp application program interface," online, 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[20] J. Singler and B. Konsik, "The gnu libstdc++ parallel mode: Software engineering considerations," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, ser. IWMSE '08. New York, NY, USA: ACM, 2008, pp. 15–22. [Online]. Available: http://doi.acm.org/10.1145/1370082.1370089

[21] D. Langr, "Parallel multi-array in-place sort with openmp." [Online]. Available: https://github.com/DanielLangr/AQsort

[22] J. Cáceres, B. Barán, and C. Schaerer, "Implementation of a distributed parallel in time scheme using petsc for a parabolic optimal control problem," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. P. M. Ganzha, L. Maciaszek, Ed., vol. 2. IEEE, 2014, pp. pages 577–586. [Online]. Available: http://dx.doi.org/10.15439/2014F340

[23] S. Fialko, "Parallel finite element solver for multi-core computers," in *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*, Sept 2012, pp. 525–532.

[24] T. A. Davis, "The university of florida sparse matrix collection," *NA DIGEST*, vol. 92, 1994.