New Storage Formats and Algorithms for Sparse Matrices

by

Ivan Šimeček

A thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague.

October 2015
Copyright © 2015 by Ivan Šimeček

This thesis is originated at the Department of Computer Systems at the Faculty of Information Technology of the Czech Technical University in Prague during years 2012-2015. At this place I would like to thank the leaders of my Faculty and Department and all other colleagues who have supported me directly or indirectly during this period. My special thanks go to my colleague D. Langr for his cooperation on the research of space-efficient formats for sparse matrices.
Abstract

The presented work deals with new storage formats and algorithms for sparse matrices. Our search for new algorithms and formats was motivated by absence of an available solution (at the beginning of the research). Firstly, there was no satisfactory solution for storing/loading large sparse matrices to/from a distributed file system. Secondly, although there were many sparse matrix storage formats, they were execution-efficient either only for a limited set of matrices or only for a limited set of operations. This work presents some solutions for these application domains. New formats and algorithms (usually with possibilities for their parallel execution) are presented. The theoretical part of this work contains a formal description of sparse matrix formats and a survey of state-of-the-art solutions. In the area of experiments and implementations, many modifications of the formats or related algorithms are tested. Results from these experiments have enabled us to gain the knowledge necessary for designing either space-efficient or execution-efficient formats that can be used in many areas of high-performance computing.

Keywords:
sparse matrix format, space complexity, sparse matrix-vector multiplication, parallel I/O, distributed memory, multithreaded execution, high performance computing, MPI, OpenMP, cache memory, quadtree format.
Preface

The submitted work presents a substantial part of long-term research performed in the field of new storage formats and related algorithms for sparse matrices. An important aspect of sparse computations is a choice of a used sparse format. Various formats and corresponding algorithms are presented. These formats and algorithms can be used in many applications. For example, efficient implementation of sparse matrix-vector multiplication is crucial for the performance of iterative linear solvers.

This thesis is divided into the following main chapters:

- Chapter 1, “Theoretical background and survey of the state-of-the-art”, presents a brief overall summary of commonly used sparse matrix formats and software technologies used for parallelization of algorithms.
- Chapter 2, “Contributions of the thesis”, deals with the description of new formats and algorithms. Their low space complexities (for formats) or high performance (for algorithms) were proved by measurements and by comparison with common formats or algorithms.
- Chapter 3, “Author’s relevant Papers”, highlights essential portions of published papers that represent the main new ideas or approaches. In this chapter, eight papers are presented in their unmodified form as published in conference proceedings or in journals. Each included paper is preceded by a short introduction detailing its most important contributions. A short discussion and summary regarding the papers is also given at the beginning and end of this chapter.
- Chapter 4, “Conclusions”, summarizes the achieved results presented in the third chapter and concludes the thesis.

Place and Date

Ivan Šimeček
Dedication

To my wife Lenka and our sons – Lukáš and David.
## Contents

**List of Figures** ix

1 **Theoretical background** 1
   1.1 Terminology and notation ........................................ 1
      1.1.1 General assumption and notation .......................... 1
      1.1.2 Matrix properties ......................................... 1
      1.1.3 Notation for pseudocodes and for parallel execution ....... 2
      1.1.4 Representing numbers and indexes in binary codes .......... 3
   1.2 Cache memory hierarchy ........................................ 3
      1.2.1 Definitions and terminology ................................ 3
   1.3 Used software technologies .................................... 5
      1.3.1 OpenMP .................................................... 5
         1.3.1.1 Execution model ...................................... 5
         1.3.1.2 Memory model ......................................... 5
      1.3.2 MPI ....................................................... 6
   1.4 Motivation of the thesis ....................................... 6
      1.4.1 Space-efficient formats .................................... 6
         1.4.1.1 Our assumptions for space-efficient formats .......... 7
         1.4.1.2 Our requirements for new space-efficient formats ... 8
      1.4.2 Execution-efficient formats ................................ 9
   1.5 State-of-art .................................................. 9
      1.5.1 Common sparse storage formats .............................. 10
         1.5.1.1 The Coordinate (COO) Format .......................... 10
         1.5.1.2 The Compressed Sparse Row (CSR) format ............. 10
         1.5.1.3 The Compressed Sparse Column (CSC) format .......... 11
         1.5.1.4 Register blocking formats ............................ 12
      1.5.2 State-of-the-art survey for space-efficient formats ....... 12
      1.5.3 State-of-art survey for execution-efficient formats ....... 13

2 **Contributions of the thesis** 16
2.1 Our new space-efficient formats ........................................... 16
  2.1.1 The entropy-based (EB) and arithmetical-coding-based (ACB) format 16
    2.1.1.1 The main idea ............................................. 16
    2.1.1.2 Transformation algorithms .............................. 17
    2.1.1.3 Results and applicability .............................. 17
  2.1.2 Minimal quadtree (MQT) format .................................. 17
    2.1.2.1 The main idea ............................................. 17
    2.1.2.2 Transformation algorithms .............................. 18
    2.1.2.3 Results and applicability .............................. 18
  2.1.3 The minimal binary tree (MBT) format ............................ 18
    2.1.3.1 The main idea ............................................. 18
    2.1.3.2 Transformation algorithms .............................. 19
    2.1.3.3 Results and applicability .............................. 20
  2.1.4 Minimal compressed formats ...................................... 20
    2.1.4.1 The main idea ............................................. 20
    2.1.4.2 Transformation algorithms .............................. 21
    2.1.4.3 Results and applicability .............................. 21
  2.1.5 COOCOO256 and COOCSR256 formats ............................... 21
    2.1.5.1 The main idea ............................................. 21
  2.1.6 Basic hierarchical (BH) formats .................................. 22
    2.1.6.1 The main idea ............................................. 22
    2.1.6.2 Transformation algorithms .............................. 22
    2.1.6.3 Results and applicability .............................. 23
  2.1.7 The advanced hierarchical (AH) format ............................ 23
    2.1.7.1 The main idea ............................................. 23
    2.1.7.2 Transformation algorithms .............................. 24
    2.1.7.3 Results and applicability .............................. 24
  2.2 Execution-efficient formats .......................................... 24
    2.2.1 Multilevel hierarchical formats ................................ 24
      2.2.1.1 The main idea ............................................. 24
      2.2.1.2 Transformation algorithms .............................. 25
List of Figures

1.1  a) an example of the sparse matrix, b) representation of this matrix in the COO format. ................................................................. 10
1.2  a) an example of a sparse matrix, b) representation of this matrix in the CSR format. ................................................................. 11
3.1  Interrelationship of the topics of individual relevant papers. .............. 114
1 Theoretical background and survey of the state-of-the-art

In this chapter, a brief overall summary (theoretical background, used SW technologies, survey of state-of-the-art, our motivation, etc.) is presented.

1.1 Terminology and notation

In the following text, we assume the following

1.1.1 General assumption and notation

- The set of real numbers is denoted by \( \mathbb{R} \). All vectors and matrix elements considered in this thesis are real.
- Indexes of all vectors and matrices are zero-based.
- We consider a large sparse 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 \textit{sparse} if it is worth (for performance or any other reason) not to store this matrix in memory in a dense array. Some alternative definitions of sparse matrix can be found in [31].
- The average number of nonzero elements per row is denoted by \textit{avg\_per\_row}. Obviously, \( \text{avg\_per\_row} = N/n \).
- We assume that \( 1 \ll n \ll N \ll M = n^2 \).
- The pattern of nonzero elements in \( A \) is unknown or random.
- The number of nonzero elements in submatrix \( B \) of matrix \( A \) is denoted by \( \eta(B) \), thus \( \eta(A) = N \). For any submatrix \( B \), if \( \eta(B) = 0 \) then the submatrix \( B \) is called zero submatrix , otherwise it is called nonzero submatrix.

1.1.2 Matrix properties

- A matrix \( A \) is \textit{regular} if \( \det(A) \neq 0 \), and it is \textit{singular} otherwise.
- A matrix \( A \) is \textit{symmetric} if \( A = A^T \).
• A matrix $A$ is **structural symmetric** if $\forall i, j; \ a_{i,j} \neq 0 \iff a_{j,i} \neq 0$.

• A matrix $A$ is **diagonal** if $\forall i \neq j; \ a_{i,j} = 0$.

• Citing from Golub and Van Loan [10]:

**Definition 1.1.1** 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 $k_1 + k_2 + 1$.

If $\omega(A) \ll n$, i.e., nonzero elements occurring only around the main diagonal then $A$ is **banded**.

• If $A$ has the same number of nonzero elements in each row then nonzero elements are distributed in $A$ uniformly and we denote $A$ as a **uniform matrix**.

### 1.1.3 Notation for pseudocodes and for parallel execution

In the following pseudocodes and examples, is used the following syntax:

• $x++$ is a C-like shortcut for $x = x + 1$. Similarly for other unary arithmetic operators.

• $x+ = y$ is a C-like shortcut for $x = x + y$. Similarly for other binary arithmetic operators.

• In a shared-memory environment, the parameter $th$ denotes a number of threads used for the execution.

• In a distributed-memory environment:
  
  - Let $P$ be the number of processors. The matrix $A$ is partitioned among $P$ processors $p_1, \ldots, p_P$ of a given massive parallel computer system (MPCS).
  
  - The MPCS uses some variant of a parallel I/O that allows to read/write a separate file for each process independently.
1.1.4 Representing numbers and indexes in binary codes

Elements of vectors and matrices are represented in a computer memory by a floating-point datatype, where $S_D$ denotes the bitsize of this floating-point datatype. If not stated otherwise, all indexes are represented by an integral data type, where $S_I$ denotes the bitsize of this datatype.

Let us have an array of $\xi$ elements indexed from 0 to $\xi - 1$. The minimum number of bits of an unsigned indexing data type is

$$S_{\text{MIN}}(\xi) = \left\lceil \log_2 \xi \right\rceil.$$  

The value $S_{\text{MIN}}$ is the minimum number of bits, but it is usually padded to whole bytes ($S_{\text{BYTE}}$ bits)

$$S_{\text{BYTE}}(\xi) = 8 \cdot \left\lceil S_{\text{MIN}}(\xi)/8 \right\rceil,$$

or it is padded to the nearest power of 2 bytes ($S_{\text{POW}}$ bits)

$$S_{\text{POW}}(\xi) = 2^\eta, \quad \text{where } \eta = \left\lceil \log_2 S_{\text{MIN}}(\xi) \right\rceil.$$  

If not stated otherwise, all space complexities are in bits and we use simply $S(\xi)$ instead of $S_{\text{MIN}}(\xi)$.

1.2 Cache memory hierarchy

1.2.1 Definitions and terminology

- **Memory hierarchy**: All present computer systems have multiple levels of memory, where each level is of different size and speed (the smaller the memory, the faster). The levels closer to the CPU are faster, but smaller, because they are more expensive (relatively to the size).

- **Cache**: It is a small and fast memory between processor registers and the main memory on the system bus. It is often made from static RAM chips (since they do not need refreshment). It is used temporarily to store the most often used variables, which decreases the average latency and increases the memory bandwidth. In the
following text, $DC_S$ denotes the size of the data part of a cache in bytes, and $IC_S$ denotes the size of the instruction part of a cache in bytes.

- **Locality**: The main idea of the cache hierarchy is based on 2 basic principles. The first one is called the **spatial locality** which states that code or data statically close together in the program will also be probably executed or accessed together. Hence, if the CPU reads some information, the adjacent data are copied into the same cache block. The second basic principle is called the **temporal locality** (sometimes called locality of reference) which states that programs tend to reuse data and instructions they have recently used. Hence, caches keep data close to the CPU to allow repeated fast accesses. The amount of data is at least one order of magnitude higher than can be stored in registers.

- **Cache hit and cache miss**: A cache hit denotes the situation when the CPU asks for some information and is already in the cache. On the other hand, a cache miss is a worse situation: the CPU asks for information that cache does not contain and that must be read from the next (i.e., slower) level in the hierarchy.

- **Cache hit ratio**: The cache hit ratio is the ratio between the number of cache hits and of all memory requests.

- **Cache miss ratio**: The cache miss ratio is the ratio between the number of cache misses and of all memory requests. Obviously, $\text{Cache miss ratio} = 1 - \text{Cache hit ratio}$.

- **Cache line**: It is the amount of data transferred between the main memory and the cache. The corresponding bus transactions are called cache-line fill for loading a block from the main memory to the cache or write-back operation for updating the main memory from the cache. One cache line consists of one cache block and information describing the cache line replacement strategy and cache states.

- **Cache block**: It is a data part of a cache line. The cache block size in bytes is denoted by $B_S$.

- **Cache set**: One set consists of $s$ independent lines. The number of sets in the cache is denoted by $h$.

We distinguish three types of cache misses:

**Compulsory** (sometimes called *intrinsic* or *cold*) misses that occur if the required memory block is not in the cache since it is accessed for the first time.
Thrashing misses (sometimes called cross-interference, conflict, or capacity misses) that occur if the required memory block is not in the cache even though it was previously loaded, since it has been replaced prematurely from the cache due to the capacity or other reasons.

Coherency misses that occur when a cache line that would otherwise be present in the thread’s cache has been invalidated by a write from another thread.

1.3 Used software technologies

1.3.1 OpenMP

The OpenMP API [25] specification 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 process of parallelization is however not automated, the programmer is responsible for correct usage of the API and avoidance of race conditions, deadlock and other data consistency issues related to shared memory environment.

1.3.1.1 Execution model

The core of OpenMP is the so called fork-join 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 parallel construct causes the thread to create a team of itself and several more additional threads and becoming master thread.

To coordinate tasks and data access in parallel regions, synchronization constructs and library routines can be used. These include constructs for critical sections, atomic operations or explicit barriers.

1.3.1.2 Memory model

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.

1.3.2 MPI

MPI (Message Passing Interface) [11] is a cross-platform language-independent communications protocol for parallel processing utilizing distributed memory. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in different computer programming languages such as Fortran, C, C++ and Java. Both point-to-point and collective communication are supported. MPI’s goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today. Actual distributed memory supercomputers such as computer clusters often run such programs.

1.4 Motivation of the thesis

In this section, the main motivation for this thesis is described: The maximal memory bandwidth and parallel I/O subsystem can be performance bottlenecks in sparse computations, e.g., loading or storing large sparse matrices from/to a distributed file system can take significant amounts of time. Sparse storage formats (SSFs) describe a way how sparse matrices are stored in computer memory. In this thesis, new formats are shown to minimize the indexing overhead and consequently reduce main memory traffic or the parallel I/O traffic.

1.4.1 Space-efficient formats

The space complexity (sometime called memory footprint) of representation of sparse matrices depends strongly on the used matrix storage format. Within our papers [B.6, B.7] or [RP3, RP4, RP6], weaknesses of previously developed solutions for space-efficient formats for storing of large sparse matrices were discussed. This thesis investigates memory-
efficient storage formats for large sparse matrices (LSMs). These matrices that due to their sizes must be stored and processed by massively parallel computer systems (MPCSs) with distributed memory architecture consisting of processor cores. The motivation of our work was in applications with LSMs that must be stored in a distributed file system using a parallel I/O subsystem. The parallel I/O subsystem can be performance bottleneck and loading or storing such matrices from/to a distributed file system are costly operations. We reduced this time by reducing the space complexity of the LSMs.

1.4.1.1 Our assumptions for space-efficient formats

Our research addresses computations with LSMs satisfying at least one of the following conditions:

1. The LSM is used repeatedly and the computation of its elements is slow and it takes more time than its later reading from a file system.

2. Construction of a LSM is memory-intensive. It needs significant amount of memory for auxiliary data structures, typically of the same order of magnitude as the amount of memory required for storing the LSM itself.

3. A solver requires the LSM in another format than is produced by a matrix generator and the conversion between these formats cannot be performed effectively on-the-fly.

4. Computational tasks with LSMs need check-pointing and recovery from failures of the MPCSs. We assume that a distributed-memory parallel computation with a LSM needs longer time. To avoid recomputations in case of a system failure, we need to save a state of these long-run processes to allow fast recovery. This is especially important nowadays (and will be more in the future) when MPCSs consist of tens or hundreds of thousands of processor cores.

If at least one of these conditions is met, we might need to store LSMs into a file system. And since the file system access is usually of orders of magnitude slower compared to the memory access, we want to store matrices in a way that minimizes their memory requirements.
1.4.1.2 Our requirements for new space-efficient formats

The requirements for a new storage format are as follows:

1. One of MPCS’s bottleneck lies in parallel I/O bandwidth. Therefore we require that the new format should be space-efficient, in order to keep resulting file sizes as low as possible.

2. We want to access LSMs files linearly. Since nowadays I/O file operations are processed by hard discs, linear access with minimal amount of seek operations leads to a maximal efficiency of reading.

3. For the designed format, there must also exist a space-efficient algorithm with small algorithmic complexity for remapping from/to the common storage formats. Due to this assumption, the time complexity of remapping is negligible in comparison to I/O file operations complexity.

Unfortunately, it is hard to satisfy all requirements at the same time, because they are generally in contradiction. This work is inspired by some real applications, for example ab initio calculations of medium-mass atomic nuclei (for future details see [8,18]).

There are several other storage formats specialized for given areas (e.g., compression of text, picture or video). They can be used for compression of sparse matrices, but none of them satisfies all these four requirements:

1. non-lossy compression,

2. possibility of massively parallel execution,

3. space efficiency (high compression rate),

4. high speed compression/decompression.

In this thesis, the compression of the information describing the structure of LSMs (i.e., the locations of nonzero elements) is discussed. The values of the nonzero elements are unchanged, because their compression depends strongly on the application. For some application areas, the values of nonzero elements are implicit and only the information about the structure of a LSM is stored (for example, incident matrices of unweighed graphs). Alternatively, we can interleave computations with reading of nonzero elements. For example, we can divide the process of a sparse matrix factorization into these steps:
1. read the matrix structure,

2. do in parallel:
   (a) perform the symbolic factorization and
   (b) read the values of nonzero elements of the matrix,

3. perform the numeric factorization.

1.4.2 Execution-efficient formats

For modern processor architectures, the main bottleneck of sparse routines is in the limited memory bandwidth. Experimental results [17] led us to modify the storage scheme in order to reduce the indexing overhead and the overall memory traffic. Our assumptions and the requirements for execution-efficient formats are as follows:

- The new format should be space-efficient, because we try to keep the resulting memory footprint as low as possible.

- In this thesis, we aimed at processing of large sparse matrices, therefore we assume that the space complexity (memory footprint) of the sparse matrix $A$ is higher than the cache size. Consequently, the matrix $A$ must be reloaded from the main memory for the every execution of sparse operation e.g., sparse matrix-vector multiplication (SpMV) or sparse transposed matrix-vector multiplication (SpMTV).

- For the designed format, there must also be fast and space-efficient algorithm for transformation of matrices from the common storage formats.

- The work-loads for threads or processors should be balanced.

Unfortunately, it is hard to satisfy all requirements at the same time, because they are generally in contradiction.

1.5 State-of-art

In this section short survey of state-of-art sparse storage formats is given.
1.5.1 Common sparse storage formats

SSFs describe a way how sparse matrices are stored in a computer memory. The following three SSFs are most common for storing sparse matrices.

1.5.1.1 The Coordinate (COO) Format

The coordinate (COO) format is the simplest SSF (see [4, 29]). The matrix $A$ is represented by three linear arrays $values$, $xpos$, and $ypos$ (see Figure 1.1 b)). 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. COO does not prescribe any order of these arrays. The structure space complexity of the structure of matrix $A$ (the size of the array $values$ is not counted) of this format is

$$S_{COO}(n, N) = N \cdot (2 \cdot S_1 + S_D),$$

if we assume that ”standard” integers are used or

$$S_{COO}(n, N) = 2 \cdot N \cdot S(n),$$

if we assume that integers with minimal bitsize are used.

![Figure 1.1: a) an example of the sparse matrix, b) representation of this matrix in the COO format.](image)

1.5.1.2 The Compressed Sparse Row (CSR) format

The most common SSF is the compressed sparse row (CSR) format (see [B.1,B.3,B.5,B.4, 4, 29] for details). The matrix $A$ stored in the CSR format is represented by three linear
arrays \textit{values}, \textit{addr}, and \textit{ci} (see Figure 1.2 b)). The array \textit{values}[0, \ldots, N - 1] stores the nonzero elements of \textit{A}, the array \footnote{Usually the array \textit{addr} is by one element larger ([0, \ldots, n]), this simplifies many algorithms.} \textit{addr}[0, \ldots, n - 1] contains indexes of initial nonzero elements of rows of \textit{A}; the first nonzero element of the row \textit{j} is stored at index \textit{addr}[j] in array \textit{values}. The row \textit{i} contains \textit{addr}[i+1]–\textit{addr}[i] elements. If row \textit{i} does not contain any nonzero element, then \textit{addr}[i] = \textit{addr}[i+1] and matrix \textit{A} is singular. Hence, all elements of the array \textit{addr} should satisfy the condition \ldots \textit{addr}[i - 1] \leq \textit{addr}[i] \leq \textit{addr}[i+1] \ldots.

The array \textit{ci}[1, \ldots, η] contains column indexes of nonzero elements of \textit{A}. The structural space complexity of the structure of matrix \textit{A} (array \textit{values} is not counted) in this format is

\[ S_{CSR}(n, N) = N(S_I + S_D) + n \cdot S_I, \]

if we assume that ”standard” integers are used or

\[ S_{CSR}(n, N) = N \cdot S(n) + n \cdot S(N), \]

if we assume that integers with minimal bitsize are used.

![](image.png)

Figure 1.2: a) an example of a sparse matrix, b) representation of this matrix in the CSR format.

### 1.5.1.3 The Compressed Sparse Column (CSC) format

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix, so the space complexity of the structure of matrix \textit{A} in this format is the same as for the CSR format.
1.5.1.4 Register blocking formats

Widely-used SSFs are easy to understand, however, sparse operations (like matrix-vector or matrix-matrix multiplication) using these formats are slow (mainly due to indirect addressing). Sparse matrices often contain dense submatrices (blocks), so various blocking SSFs were designed to accelerate matrix operations. Compared to the CSR format, the aim of these formats (like SPARSITY [15] or CARB [B.2,B.5]) is to allow a better use of registers and more efficient computations. But these specialized SSFs have usually large transformation overhead and consume approximately the same amount of memory as the CSR format.

1.5.2 State-of-the-art survey for space-efficient formats

What were the possibilities to manage storing/loading LSMs to/from a distributed file system (before our research)? There were several widely used and well documented text-based file formats for sparse matrices, mainly Matrix Market [5], Harwell-Boeing [6,7], and Matlab (ASCII) [14]. There are, however, reasons why text-based storage formats are not suitable for VLSMs— they must be accessed sequentially and they usually consume much more space than binary formats.

As for binary file formats, there were no satisfactory solutions. Many modern sparse solvers, such as Trilinos [12,13,30] or PETSc [1–3], provide the functionality of storing matrices into a file. However,

1. the matrices must already be loaded into the solver, whereas we might need to store matrices as they are constructed;

2. the binary formats of such files are usually proprietary and poorly or not at all documented, and therefore they cannot be simply used anywhere else.

Just few papers have been published about SSFs in the context of minimization of the required memory (before our research), which is the optimization criterion for a file I/O. Some recent research of hierarchical blocking SSFs, though primarily aimed at SpMV optimization, also addresses optimization of memory requirements [19,21–23].
1.5.3 State-of-art survey for execution-efficient formats

There are many papers discussing the efficiency of SpMV or SpMTV operation mainly in situations when at least one of the following conditions is true:

1. Formats and algorithms are designed for single-threaded execution.
2. Formats and algorithms are designed for banded matrices only.
3. Many authors assume that the matrix $A$ has a known structure.
4. Many authors (such as [16, 36]) overlook the overhead of matrix transformation or design a time-expensive sparse matrix storage transformation (conversion).

The following approaches for acceleration of the SpMV operation are most common:

- Register blocking formats (e.g., [B.2, B.5, 15]): These formats eliminate indirect addressing during the SpMV. Then, vector instructions can be used. These formats are suitable only for matrices with a known structure of nonzero elements. These formats are designed to handle randomly occurring dense blocks in a sparse matrix, but the efficiency of SpMV operation with these formats depends strongly on the presence of these blocks.

- Optimization of cache utilization (e.g., [B.3]): In these papers, we try to find a near-optimal matrix storage format to maximize the performance of the SpMV with respect to matrix transformation overhead and cache parameters.

- Other approaches for increasing the efficiency of the SpMV [15, 24, 32] are based on matrix reordering, sometimes combined with sw-pipelining [27].

Extensive research has been conducted about SSFs in the context of performance optimization of the sparse matrix-vector multiplication (SpMV) algorithms. There are plenty of storage formats aimed at acceleration of the SpMV operation. For example, the Sparse Library version 1.6 from Yzelman (for details see [36]) supports the following storage and multiplication schemes:

- Triplet Scheme (TS, also known as the Coordinate scheme COO),
- Compressed Row Storage (CRS, also known as compressed sparse row; CSR),
• Incremental CRS (ICRS),
• Zig-Zag CRS (ZZ-CRS),
• ZZ-ICRS,
• Sparse vector matrix (SVM),
• Hilbert-curve ordered TS (HTS),
• Bi-directional ICRS (BICRS),
• Hilbert-curve ordered BICRS,
• Hierarchical BICRS (HBICRS),
• Block Hilbert (hard-coded sparse blocking with Hilbert-curve ordering on blocks and HBICRS),
• Bisection Hilbert (as above, but with adaptive sparse blocking),
• Compressed BICRS (CBICRS),
• Vectorised BICRS (vecBICRS, includes compression),
• Dense diagonal scheme (DD Matrix).

The currently supported parallel schemes are following:

• Block CO-H+ (alike to Block Hilbert, but parallelised),
• Row-distributed block CO-H (alike to Block Hilbert, but with explicit 1D partitioning),
• Row-distributed Hilbert (as the above scheme, but without sparse blocking),
• OpenMP CRS (implicit 1D fine-grained parallelisation using openMP).
• Row-distributed Hilbert-compressed block CO-H (only stores a delta array based on 1D Hilbert coordinates, which are unpacked during SpMV multiplication).
CHAPTER 1. THEORETICAL BACKGROUND

The other state-of-the-art research results were published, for example, in [16, 23].

In our papers [RP2, RP3, RP4, RP6] (see Section 2.1), space efficient formats (based on quadtree, binary tree, and arithmetic coding etc.) were discussed. In these papers, we tried to design formats that minimize to reduce space complexity at all costs, these formats were designed only for the LOAD and SAVE operations for the parallel I/O on massively parallel computer systems with distributed memory architectures. Thus, these formats were inefficient in context of the SpMV or SpMTV operation.

As far as we know, the most related papers to our approach (using of space-efficient formats to accelerate SpMV) are [20, 21], but the authors assume a recursive storage scheme and a very complex synchronization during the execution of SpMV. In contrast to them, we try to avoid all complex synchronization operations (mutexes etc.) because we plan to extend this work also for GPUs.
2 Contributions of the thesis

In this chapter, the main contributions of this thesis (description of new formats and algorithms and their results) are summarized.

2.1 Our new space-efficient formats

In this section, new sparse matrix storage formats that minimize the space complexity of information about matrix structure are proposed and evaluated.

2.1.1 The entropy-based (EB) and arithmetical-coding-based (ACB) format

2.1.1.1 The main idea

The space complexity of any sparse matrix storage format depends strongly on its structural pattern. If the sparsity pattern of a matrix is completely known (for example, if a matrix is tridiagonal) then the space complexity for storing the information on its structure is zero. If a random distribution of nonzero elements is assumed, then it is equal to the value of the entropy of a bit vector of size $M$, in which $N$ bits are set to 1 and $M - N$ bits are set to 0. Thus, such a format is denoted as entropy-based. The number of these vectors is $C = \binom{M}{N}$, so its entropy is

$$S_{EBF}(n, N) = \log_2 C = \sum_{i=M-N+1}^{M} \log i - \sum_{i=1}^{N} \log i.$$  

(2.1)

Recall that $M = n^2$. Unfortunately, formula (2.1) is hard to compare with complexity formulae of other storage formats. In [RP3] it was approximated using the Stirling formula\(^1\): The final approximation was then:

$$S_{EBF}(n, N) \approx N \cdot \left(\frac{1}{\ln 2} + 2 \cdot \log_2 n - \log_2 N\right).$$  

(2.2)

\(^1\)There exist more precise approximations like [26], but they lead to the same results and our goal is not the most accurate approximation.
Unfortunately, the EB format is very difficult to compute, thus it serves only for comparison and no practical algorithm to achieve this space complexity was given. In [RP6], the arithmetical-coding-based (ACB) format was introduced:

The probability $p_0$ that a given bit in $B$ is equal to 0 is $\frac{M-N}{M}$. In the arithmetical coding (see [35]), one can encode this information using $-\log_2 p_0$ bits. The probability $p_1$ that a given bit in $B$ is equal to 1 is $\frac{N}{M}$. In the arithmetical coding, one can encode this information using $-\log_2 p_1$ bits. Since a random distribution of nonzero elements is assumed, the vector $B$ is considered to be an order-0 source (each bit is selected independently on other bits). The total number of bits to encode vector $B$ is equal to the value of binary entropy of vector $B$, thus EBF and ACB formats have the same space complexity.

2.1.1.2 Transformation algorithms

In [RP6], the following idea of transformation of the matrix $A$ structure to the ACB format was presented: create $n \times n$ bitmap (with $N$ 1’s) from matrix $A$ structure. Then, compress this bitmap as a bitstream using the arithmetical coding. The representation of matrix $A$ structure in the ACB format the compressed bitstream.

2.1.1.3 Results and applicability

A comparison to common SSF was done in [RP2, RP3, RP6]. This format is suitable for matrices without any locality. A drawback of the ACB format is its computational complexity. Since each bit of vector $B$ is encoded in time $\Theta(1)$, the complete vector $B$ (representation of sparse matrix $A$) is encoded in time $\Theta(n^2)$. This is too much for sparse matrices with a constant number of nonzero elements per row (i.e., $N \in \Theta(n)$).

2.1.2 Minimal quadtree (MQT) format

2.1.2.1 The main idea

**Definition 2.1.1** The Quadtree ($QT$) is a tree data structure in which all inner nodes have exactly four child nodes.

A big drawback of the some QT formats (e.g., advanced quadtree format, see Section 2.2.2) from the viewpoint of space complexity is a larger data overhead (caused by pointers
CHAPTER 2. CONTRIBUTIONS OF THE THESIS

up_left, up_right, lo_left, lo_right) compared to the COO and CSR formats. Since our aim is to minimize the space complexity of QT-based formats, in [RP4] a new QT format called minimal quadtree (MQT) format is proposed that extends ideas of the standard QT format as follows:

- All nodes in the MQT are stored in one array. Since we can compute locations of all child nodes, we can omit pointers up_left, up_right, lo_left, lo_right. We lose the advantage of the possibility to easily modify the QT, but it is not an important property for our application area.

- Instead of pointers, each node of the MQT contains only 4 flags (i.e., 4 bits only) indicating whether given subquadtrees are nonempty.

So, the space complexity of every MQT node is only 4 bits.

2.1.2.2 Transformation algorithms

The idea of transformation of the matrix $A$ structure to the the minimal quadtree format was described in [RP4]. The completely new (bottom-up) algorithm was proposed in [RP7]. Experiments on the real parallel system were performed and these experiments proved that the proposed algorithm allows a more efficient conversion.

2.1.2.3 Results and applicability

A space complexity comparison to common SSF was done in [RP4]. The derivation of lower and upper bounds for this format was also included. Experiments proved that this format minimize space complexity of the sparse matrix structure.

2.1.3 The minimal binary tree (MBT) format

2.1.3.1 The main idea

The full binary tree (FBT) is a widely used data structure in which all inner nodes have exactly two child nodes. Binary trees especially those used for binary space partitioning can also be used for storing sparse matrices. The idea of binary space partitioning is not new, but as far as we know, the use of these formats for efficient storing sparse matrices
was not described in literature. In standard implementations, every node in a FBT is represented by structure `standard_BT_struct` consisting of the following items:

- two pointers (`left`, `right`) to child nodes,
- (only for leaves) the value of a nonzero element.

If a FBT is used as a basis for SSF, it describes a partition of the sparse matrix into submatrices and each node in the FBT represents a submatrix. Equally as in k-d trees, see [28], the decomposition is performed in alternating directions: first horizontally, then vertically, and so on. In other words, nodes in an odd depth represent a partition of the submatrix into two halves along the the $x$-axis (left/right), nodes in an even depth represent a partition of the submatrix into two halves along the $y$-axis (upper/lower). From the viewpoint of space efficiency, a drawback of the standard FBT representation is the overhead caused by pointers `left`, `right`. It causes that the standard FBT-based SSF may have worse space complexity than the CSR format.

To eliminate this drawback, we propose a new k-d-tree-based SSF. Each tree node represents again a submatrix, but we modify the standard representation of the FBT and we call this data structure the `minimal binary tree` (MBT) format. The idea is very similar to that in the MQT format.

- All nodes of a MBT are stored in one array (or stream). Since the size of the input matrix is given, we can compute locations of all child nodes, we can omit pointers `left`, `right`.

- All nodes of a MBT contain only two flags (it means only two bits). Each of them is set to 1 if the corresponding half of the submatrix (left/right or upper/lower) contains at least one nonzero element, otherwise it is set to 0.

So, the space complexity of every MBT node is only 2 bits.

### 2.1.3.2 Transformation algorithms

The idea of transformation of the matrix $A$ structure to the the minimal binary format was described in [RP6]. The completely new one inspired to that proposed in [RP7] (bottom-up) algorithm for the MQT format can be implemented.
2.1.3.3 Results and applicability

A space complexity comparison to common SSF was done in [RP6]. The derivation of lower and upper bounds for this format was also included. Space complexities using this format are comparable to MQT format (see Section 2.1.2).

2.1.4 Minimal compressed formats

2.1.4.1 The main idea

The space complexity of MBT and MQT formats (see Section 2.1.2 and 2.1.3) can be further reduced by compression as was discussed in [RP6]. The MBT and MQT formats have minimal space complexity only if we assume fixed number of bits for each node (2 bits for MBT and 4 bits for MQT). We can relax this assumption to achieve more space efficient formats.

Lemma 2.1.1 Every node in the MBT (or in MQT) format (except for the root node for the zero matrix $A$) has got at least one bit equal to 1.

The proof of Lemma 2.1.1 for the MBT format can be done by contradiction: if both bits in a MBT node $X$ are zero, then this submatrix does not contain any nonzero element, so in the parent’s node of $X$ the corresponding bit is set to 0 and node $X$ is not included in the output stream and this is a contradiction with the initial assumption.

Similar proof can be done for the MQT format. Q.E.D.

Since we assume only nonempty matrices, the only allowed values in every MBT node are: 01, 10, and 11 (value 00 is not possible as a result of Lemma 2.1.1). So, if the first bit is 0, then the second bit must 1. This redundant information can be excluded from the output stream. We call this case the hidden one. Based on this idea, we propose another new format, called compressed binary tree (CBT). Similarly in the MQT format, the value 0000 is not possible as a result of Lemma 2.1.1, so if the first three bits are 0, then the fourth bit must 1. Again, this redundant information can be excluded from the output stream, which allowed us to construct another new compressed quadtree (CQT) format. It is obvious that the probability of hidden one is higher in the MBT format than in the MQT format.
CHAPTER 2. CONTRIBUTIONS OF THE THESIS

2.1.4.2 Transformation algorithms

The idea of transformation of the matrix $A$ structure to the the minimal binary format was described in [RP6]. There are two approaches to transform a LSM to the CBT format:

1. Transform the input matrix to the MBT format (it creates output stream $S$) and then remove from $S$ all hidden ones.

2.1.4.3 Results and applicability

A space complexity comparison to common SSF is done in [RP6]. It also includes the derivation of lower and upper bounds for this format.

2.1.5 COOCOO256 and COOCSR256 formats

2.1.5.1 The main idea

The high memory requirements for the COO format are caused by two arrays of size $N$. We cannot reduce the size of these arrays, but we can try to reduce the number of bytes for every row/column index. The idea is to partition the matrix into square blocks of size $r \times c$ rows/columns. In [RP2], these parameters were fixed ($c = r = 256$), thus these formats were denoted as COOCOO256 or COOCSR256. Every such block can be identified by block row and block column indices of size $S([n/256])$ bytes. Let $K$ denotes the number of nonzero blocks for our matrix $A$ (nonzero block is a block that contains at least one nonzero value). Suppose nonzero matrix elements stored in the COO format. If we store nonzero blocks in the coordinate storage format, we need for each one its block row/column index of size $S([n/256])$ bytes, and a pointer into its data (an index into the original arrays of row/column indices and values) that it therefore an index of size $S(N)$ bytes. Now, for each nonzero element, we need only 1-byte local row/column indices valid within a block instead of $S(n)$-byte row/column indices valid within the whole matrix. Finally, instead of two arrays of $N$ row/column indices of size $S(n)$, we have five arrays:

- two arrays of size $N$ containing 1-byte local row/column indices within a block,
• two arrays of size \( K \) containing \( S(\lceil n/256 \rceil) \)-byte block row/column indices within a matrix,

• one array of size \( K \) containing \( S(N) \)-bytes indices to the elements of each block.

### 2.1.6 Basic hierarchical (BH) formats

#### 2.1.6.1 The main idea

In [RP3], we relaxed the assumption of the fixed block size. The idea was generalized to partition the matrix into square disjoint blocks of size \( 2^c \times 2^c \) rows/columns, where \( c \in \mathbb{N}^+ \) is a formal parameter. Coordinates of the upper left corners of these blocks are aligned to multiples of \( 2^c \). So, indexes of nonzero elements are separated in two parts, indexes of blocks and indexes inside the blocks. Every such a block has block row and block column indexes of size \( S(\lceil n/2^c \rceil) \) bits. Let \( B(c) \) denote the number of nonzero blocks for matrix \( A \). A nonzero block is a block that contains at least one nonzero matrix element. The minimal number of nonzero blocks is equal to

\[
B(c)_{\text{min}} = \left\lceil \frac{N}{2^c} \right\rceil
\]

if all nonzero blocks contain only nonzero elements (i.e., are 100\% dense). The maximal number of nonzero blocks is equal to

\[
B(c)_{\text{max}} = \min \left( N, \left\lceil \frac{n}{2^c} \right\rceil^2 \right),
\]

if each nonzero block contains exactly one nonzero element or if the whole matrix \( A \) is covered by nonzero blocks.

For storing information about the blocks and elements inside the blocks, we can use the COO or CSR format, which results in four combinations of these formats.

#### 2.1.6.2 Transformation algorithms

The idea of transformation of the matrix \( A \) structure to the the basic hierarchical format was described in [RP3]. A drawback of presented algorithm is that it must be called repeatedly for different values of parameter \( c \) from 1, \ldots, \log n \) to find the most space-
efficient format. The completely new (bottom-up) algorithm inspired to that proposed in [RP7] for the MQT format can be implemented and this overhead can be eliminated.

2.1.6.3 Results and applicability

A space complexity comparison to common SSF was done in [RP6]. The derivation of lower and upper bounds for this format was also included.

2.1.7 The advanced hierarchical (AH) format

2.1.7.1 The main idea

Another type of a hierarchical format (introduced in [RP3]) is a format that combines a bitmap at the top level and the COO format at the lower level. The COO format is used due to small number of elements inside each block. So, this format consists of:

- One bitmap (in this bitmap each bit=pixel represents a block of \( s \times s \) elements in matrix \( A \). If this block is nonempty, then the corresponding bit in the bitmap is set to 1 and vice versa. Obviously, the size of the bitmap is \([n/s] \times [n/s]\).

- A list of nonzero elements inside nonzero blocks. For each such an element, we need \( S^{\text{MIN}}(s^2) \) bits for the in-block row and column coordinates and 1 bit for the flag if it is the first nonzero element in the block.

So, the size of the bitmap format (denoted by \( f \)) is

\[
f(n, N, s) = \left( \frac{n}{s} \right)^2 + N \left( 2 \cdot S^{\text{MIN}}(s) + 1 \right).
\]

In [RP3] the optimal value of parameter \( s \) was discussed, it is equal to

\[
s = \sqrt{\frac{n^2 \cdot \ln 2}{N}}.
\]

The space complexity of this format is

\[
N \left( 1 + \frac{1}{\ln 2} + \log_2(\ln 2) + 2 \log_2 n - \log_2(N) \right).
\]
2.1.7.2 Transformation algorithms

The idea of transformation of the matrix $A$ structure to this format is not described in our papers, but it is very similar to the algorithm for BH format [RP6], thus it can be proceed in parallel.

2.1.7.3 Results and applicability

A space complexity comparison to common SSF was done in [RP3,RP6]. The derivation of lower and upper bounds for this format was also included. By comparing results from AH and ACB formats, we see that the AH format is only slightly less efficient (about 0.47 bit per non-zero element) than the ACB format (see Section 2.1.1).

2.2 Execution-efficient formats

In this section, we propose new execution-efficient sparse matrix storage formats (with corresponding algorithms) that accelerate some matrix operations.

2.2.1 Multilevel hierarchical formats

2.2.1.1 The main idea

In [RP8], a modification COOCOO format (see [RP3] or Section 2.1.6) was introduced. The idea of COOCOO format was generalized into formats with multiple levels. We call these formats multilevel hierarchical and denote them by (COO$_k$)$_l$, where:

- $k$ is the number of bits per level,
- $l$ is the number of levels.

In the ”classical” COOCOO format, there is a memory overhead of pointers and linked lists. To make the resulting format space-efficient we eliminate this overhead by merging all items into two arrays:

- the array $a_{nnz}$ that contains the number of nonzero regions in this branch (subtree) and
• the array $a_{xy}$ that contains the part of coordinates of nonzero regions in this branch.

In [B.7] or [RP3] (see Section 2.1.6), similar two-level hierarchical format were described and the optimal value of bits for each level were computed. To avoid this initial computational overhead and also bitwise manipulations during the SpMV, in the paper [RP8] we studied only the format $(\text{COO}_8)^4$.

2.2.1.2 Transformation algorithms

The idea of transformation of the matrix $A$ structure to multilevel hierarchical format can be based on the algorithm for BH formats (see [RP3] or Section 2.1.6).

2.2.1.3 Results and applicability

A space complexity comparison to common SSF was shown in [RP8]. In this paper, the performance comparison for two operations was also done:

1. SpMV operation

2. a pair of operations SpMV and SpMTV. This pair is denoted as *fused multiplication*.

Comparison of SpMV performance in the COO format with $(\text{COO}_8)^4$ format: the latter one is always better (more execution efficient). This follows from theoretical assumptions (better space complexity).

Comparison of SpMV performance in the CSR format with $(\text{COO}_8)^4$ format: the latter one better (more execution efficient) only for 8% of testing matrices. This follows from the facts that the hierarchical format is usually more space efficient but the the code for the SpMV is more complex (e.g., uses more registers) than for the CSR format.

Comparison of fused SpMV and SpMTV performance in the COO format with $(\text{COO}_8)^4$ format: the latter one is better (more execution efficient) for 70% of testing matrices. This follows from theoretical assumptions (better space complexity). On the other hand, the operations in the $(\text{COO}_8)^4$ format suffers from the larger amount of conflict writes (coherence misses).

Comparison of fused SpMV and SpMTV performance in the CSR format with $(\text{COO}_8)^4$ format: the latter one better (more execution efficient) for 61% of testing matrices. This
follows from the facts that the hierarchical format is usually more space efficient and all writes have better spatial locality (smaller amount of thrashing misses).

2.2.2 Advanced quadtree format

2.2.2.1 The main idea

The QT is used to describe a partition of a sparse matrix into submatrices (see Section 2.1.2). Each QT node represents a submatrix. There are different implementations of the QT format. In the standard implementation (for details, see [9, 33, 34]), every node in the QT is represented by structure `standard_QT_struct` consisting of the following items:

- four pointers (`up left, up right, lo left, lo right`) to the child nodes,
- (only for leaves) values of nonzero elements.

Great advantages of the quadtree are the following:

- Easy and fast conversion from common sparse matrix storage formats like CSR or COO.
- Modifications (adding or removing nonzero elements) of the quadtree are relatively easy and fast in comparison to common formats.
- The recursive style of programming and recursive style of storage ("Divide and Conquer" approach) leads to codes with a surprising performance due to the better cache memory utilization.

A big drawback of the quadtree structure is a larger control and data overhead compared to standard formats. The standard quadtree implementation leads to a space (and execution) inefficiency. To remove inefficiencies, (in [RP1]) the additional types of leaves were used: modified versions of the COO and the CSR formats. The modification means that we express all coordinates relatively to the beginning of the submatrix (node). We call "COO" and "CSR" respectively this type of node.

The second improvement described in this paper was the elimination of "Empty" nodes, because they do not contain any useful information. They were simply represented by the NULL pointer.
2.2.2.2 Transformation algorithms

The idea of transformation of the matrix A structure to the advanced quadtree format was described in [RP1]. The completely new (bottom-up) algorithm for transformation was proposed in [RP7].

2.2.2.3 Results and applicability

A performance comparison to common SSF was done in [RP1]. Algorithms using this data format show significant speedups for some matrix operations (between 5% and 80% in the case of multiplication of a sparse matrix by a sparse matrix and about 700% in case of getting a value of a given element). It results from the fact that the quadtree data structure combines advantages of sparse and dense data formats.
CHAPTER 3. AUTHOR’S RELEVANT PAPERS

3 Author’s relevant papers

This chapter describes author’s papers which dealt with new storage formats and algorithms for sparse matrices. The ideas introduced in these papers were presented at various international conferences and published in proceedings of these conferences or in international journals.


CHAPTER 3. AUTHOR’S RELEVANT PAPERS


All papers included in this chapter are in their original form. Page numbers are also unmodified. Before each paper a short description is given. The relationships of the problems being solved in the mentioned papers is also described. These papers are sorted chronologically, but all listed papers can be divided into two groups (but these groups are not disjoint):

The first group contains 5 papers (RP2, RP3, RP4, RP6, RP8) that deals with the design of space-efficient formats for sparse matrix storage, i.e., they are aimed at space complexity.

The second group contains 4 papers (RP1, RP5, RP7, RP8) that deals with the design of execution-efficient algorithms for sparse matrices, i.e., they are aimed at performance.

At the end of this chapter, the achieved results given in this papers are discussed and analyzed. The relationships of issues covered by individual papers are shown.

3.1 RP1: Sparse Matrix Computations Using the Quadtree Storage Format

In this paper (RP1), we tried to renew a interest for quadtree format. This format have been formerly used for storing monochrome pictures. As far as we know, we were the first who use the quadtree format for sparse matrix storage. Some improvements to the quadtree storage format (combination of quadtree and common storage formats) were introduced. Algorithms using the resulting format (denoted as advanced quadtree format) show significant speedups for the following matrix operations:

- to get an value at the given location in the sparse matrix,
- the transposition of the sparse matrix,
- the multiplication of a sparse matrix by a dense vector,
- the multiplication of a sparse matrix by a sparse matrix.
These routines are often used in libraries for the numerical linear algebra. They represent building blocks for more complicated operations.

We have tested the influence of three architecture-dependent parameters ($t$ille$_size$, dense$_limit$, and $XY_limit$) on the performance and on the quadtree datasize, but the parameter $XY_limit$ doesn’t have any significant effect in our experiments simply because the leafs of type ”$XY$” (COO) occur rarely. Figures 1 and 2 illustrate the fact that larger values of parameters $t$ille$_size$ and dense$_limit$ make quadtree data structure more space efficient. Figures 3-10 illustrate the impact of these parameters on the performance.

The speedup is gained due to the fact that the quadtree data structure combines advantages of sparse and dense data formats.
Sparse Matrix Computations Using the Quadtree Storage Format

Ivan Šimeček  
Department of Computer Science and Engineering,  
Czech Technical University, Prague  
Email: xsimecek@fel.cvut.cz

Abstract—Computations with sparse matrices are widespread in scientific projects. Used data format affects strongly the performance. Efficient formats for storing sparse matrices are still under development, since the computation using widely-used formats (like XY or CSR) is slow and specialized formats (like SPARSITY or CARB) have a large transformation overhead.

In this paper, we represent some improvements to the quadtree storage format. We also compare the performance during the execution of some basic routines from the linear algebra using widely-used formats and the quadtree storage format.

I. INTRODUCTION

The performance of mathematical operations with sparse matrices depends strongly on the used matrix storage format. In this paper, we present a variant of the quadtree storage format implementation. We extend the ideas presented in [6], [5], [1] and prove that the idea of sparse computations using the quadtree storage format implementation is viable.

II. COMMON SPARSE MATRIX FORMATS

In the following text, we assume that A and B are real sparse matrix of order n. Let NZ_A, NZ_B be the total number of nonzero elements in A or in B respectively.

A. The Coordinate (XY) Format

The coordinate (XY) format is the most simplest sparse format. The matrix A is represented by three linear arrays Elem, X, and Y. The array Elem[1,...,NZ_A] stores the nonzero values of A, arrays X[1,...,NZ_A] and Y[1,...,NZ_A] contain X- and Y-positions, respectively, of the elements with the nonzero value.

B. The Compressed Sparse Row (CSR) format

The most common format (see [4]) for storing sparse matrices is the compressed sparse row (CSR) format. A matrix A stored in the CSR format is represented by three linear arrays Elem, Addr, and Ci. The array Elem[1,...,NZ_A] stores the nonzero elements of A, the array Addr[1,...,n] contains indexes of initial nonzero elements of rows of A, and the array Ci[1,...,NZ_A] contains column indexes of nonzero elements of A. Hence, the first nonzero element of row j is stored at the index Addr[j] in array Elem.

C. Register blocking formats

Widely-used formats are easy to understand, however sparse operations (like matrix-vector or matrix-matrix multiplication) using these formats are slow (mainly due to indirect addressing). Sparse matrices often contain dense submatrices (blocks), so various blocking formats were designed to accelerate matrix operations. Compared to the CSR format, the aim of these formats (like SPARSITY[2] or CARB[3]) is to consume less memory and to allow a better use of registers and the vectorization of the computation. Algorithms using these formats are very fast, because they are tuned for a target architecture. But these specialized and efficient formats have also some drawbacks. These formats:

- suffer from a large transformation overhead,
- are designed only for limited set of operations,
- doesn’t support fast adding or removing nonzero elements.

D. Quadtree data format

1) Definition: Quadtree (for details see [6], [5], [1]) is the recursive tree data structure. Such a tree represents a partition of the matrix into submatrices ("nodes" in the graph terminology). There are different types of nodes in the tree. Inner nodes of the quadtree are divided into "Mixed" or "Empty" nodes. Leafs of the quadtree are divided into "Full" or "Empty" nodes.

Great advantages of the quadtree are the following:

- Easy and fast conversion from common sparse matrix storage formats like CSR or XY (see Algorithm Transf(A)).
- Modifications (adding or removing nonzero elements) of the quadtree are relatively easy and fast in comparison to common formats.
- The recursive style of programming and recursive style of storage ("Divide and Conquer" approach) leads to codes with a surprising performance due to the better cache memory utilization.

2) Our quadtree extensions: A big drawback of the quadtree structure is a larger control and data overhead compared to standard formats. The standard quadtree implementation leads to a space (and execution) inefficiency. To remove inefficiencies, we use the additional types of leaves: modified versions of the XY and the CSR formats. The
modification means that we express all coordinates relatively to the beginning of the submatrix (node). We call "XY" and "CSR" respectively this type of node.

Our second improvement is the elimination of "Empty" nodes, because they do not contain any useful information. They are simply represented by the NULL pointer.

3) The quadtree transformation algorithm: The following transformation algorithm converts a matrix $A$ into the quadtree format. The algorithm uses these three parameters:

- $\text{tile\_size}$ = the maximum size of a submatrix in one node (except "Empty" nodes).
- $\text{dense\_limit}$ = the threshold between "Full" and "CSR" nodes.
- $\text{XY\_limit}$ = the threshold between "CSR" and "XY" nodes.

```
Algorithm Transf(A)

(* Transformation alg. to the quadtree format *)
/* Input: A = the matrix for the transformation*/
/* Output: the pointer for the root of the quadtree*/
NZ' = the number of nonzero elements in matrix $A$; 
n' = the order of matrix $A$;
if (NZ' == 0) then return NULL;
if (n' > tile\_size) then
divide $A$ into submatrices $A_1, A_2, A_3, A_4$;
create $M$ - the leaf of type "Mixed";
$M_{\text{up\_left}}, M_{\text{up\_right}}, M_{\text{lo\_left}},$ and $M_{\text{lo\_right}}$;
$M_{\text{up\_left}} \leftarrow \text{Transf} (A_1)$;
$M_{\text{up\_right}} \leftarrow \text{Transf} (A_2)$;
$M_{\text{lo\_left}} \leftarrow \text{Transf} (A_3)$;
$M_{\text{lo\_right}} \leftarrow \text{Transf} (A_4)$;
return $M$;
else
if (NZ' > dense\_limit) then
transform $A$ to the leaf $F$ of type "Full";
return $F$;
else
if (NZ' > XY\_limit) then
transform $A$ to the leaf $C$ of type "CSR";
return $C$;
else
transform $A$ to the leaf $X$ of type "XY";
return $X$;

```

The parameters of the $\text{Transf}(A)$ algorithm ($\text{tile\_size}, \text{dense\_limit},$ and $\text{XY\_limit}$) have a great impact on the space complexity of the quadtree data structure and also on the performance of operations. Therefore they can be used for the optimization of the computation.

III. EVALUATION OF THE RESULTS

In this section, we will compare the data size and the performance of several basic operations with sparse matrices using the CSR and the quadtree format.

A. HW and SW configuration

B. Experimental configuration 1

All results were measured on Intel Pentium Celeron M420 at 1.6 GHz, 2 GB of the main memory at 333 MHz, running OS Windows XP Professional SP3 with the following cache parameters:

- L1 cache is 32 KB data, L2 cache is 1 MB data cache.
- Microsoft Visual Studio 2003
- Intel compiler version 9.0 with switches:
  /O3 /Og /Oa /Oy /Ot /Qpc64
  /QxP /Qipo /Qsalign16 /Zp16

C. Experimental configuration 2

All results were measured on Intel Core 2 Quad Q8200 (only one core was used) at 2.33 GHz, 4 GB of the main memory at 400 MHz, running OS Windows XP Professional SP3 with the following cache parameters:

- L1 cache is 32 KB data cache, L2 cache is 2 MB data cache.
- Microsoft Visual Studio 2003
- Intel compiler version 10.1 with switches:
  /O3 /Og /Oa /Oy /Ot /Qpc64
  /QxT /Qipo /Qsalign16 /Zp16

D. Test applications

We have implemented three very basic routines from the linear algebra:

- to get an value at the given location in the sparse matrix (operation $\text{GetXY}$),
- the transposition of the sparse matrix (operation $\text{Transp}$),
- the multiplication of a sparse matrix by a dense vector (operation $SpM \times V$),
- the multiplication of a sparse matrix by a sparse matrix (operation $SpM \times M$).

These routines are often used in libraries for the numerical linear algebra. They represent building blocks for more complicated operations.

E. Implementation of operations for the CSR format

1) Operation $\text{GetXY}$: Operation $\text{GetXY}$ is implemented as a linear search inside the given row. The maximal complexity is $O(NZ)$, but average complexity is $\Theta(\frac{NZ}{n})$. 

\[ /O3 /Og /Oa /Oy /Ot /Qpc64/ /QxP /Qipo /Qsalign16 /Zp16 /QxT /Qipo /Qsalign16 /Zp16 \]
2) **Operation Transp**: The complexity of this operation is $O(NZ + n)$.

```plaintext
Pseudocode Transp_CSR()
(* Std impl. of the Transp for the CSR format *)
(* n = the order of matrix $A$ *)
(* NZ = the number of nonzero el. in matrix $A$ *)
for $i = 1$ to $n$ do
  Count[k] = 0;
for $i = 1$ to $NZ$ do
  $k = ci[i]$;  Count[$k$] += 1;
Count2[0] = 0;
for $i = 1$ to $n$ do
  Count2[$i$] = Count2[$i-1$] + Count[$i$];
for $i = 1$ to $n$ do
  Out_addr[$i$] = Count2[$i$];
for $i = 1$ to $n$ do
  low = Addr[$i$] + 1;  up = Addr[$i$] + 1;
  for $j = low$ to $up$ - 1 do
    $k = ci[j]$;  $m = Count[k] + 1$;
    Out_s[$m$] = $i$;  Out_elem[$m$] = Elem[$j$];
return Out_addr[$i$], Out_s[$i$], Out_elem[$i$];
```

3) **Operation SpM$\times V$**: The complexity of this operation is $O(NZ + n)$.

```plaintext
Pseudocode spMV_CSR()
(* Std impl. of the SpM$\times V$ for the CSR format *)
(* n = the order of matrix $A$ *)
low = Addr[1];
for $i = 1$ to $n$ do
  $s = 0.0$;  up = Addr[$i$] + 1;
  for $j = low$ to $up$ - 1 do
    $k = ci[j]$;  $s += Elem[j] \times C[i][k]$;
  end for
  y[$i$] = $s$;  low = up;
return y[:];
```

4) **Operation SpM$\times M$**: The maximal complexity is $O(NZ_A \cdot n)$, but average complexity is $O(NZ_A + NZ_E)$.

```plaintext
Pseudocode spMV_CSR()
(* Std impl. of the SpM$\times M$ for the CSR format *)
(* n = order of matrix $A$ *)
for $y = 0$ to $n - 1$ do
  for $i = Addr_A[y]$ to $Addr_A[y + 1] - 1$ do
    $x = ci[i]$;
    for $j = Addr_B[x]$ to $Addr_B[x + 1] - 1$ do
      $x^2 = ci[j]$;
      $c_i[i][x^2] = Elem_A[i] \times Elem_B[j]$;
  end for
return c_i[];
```

F. **Test data**

We have used 32 real matrices from various technical areas from the MatrixMarket and Harwell sparse matrix test collection. Graphs in this section represent results obtained from one of measured matrices: non-symmetric banded matrix ($n = 10^4$, $NZ = 5 \cdot 10^6$, bandwidth = 1500).

G. **Influence of architecture-dependent parameters**

We have tested the influence of three architecture-dependent parameters ($tile_size$, $dense_limit$, and $XY_limit$) on the performance and on the quadtree datasize, but the parameter $XY_limit$ doesn’t have any significant effect in our experiments simply because the leaves of type "XY" occur rarely.

H. **Experimental results**

1) Results of our improvements to the quadtree data format: Our extensions (described in Section II-D2) increase the code complexity, but also decrease significantly the quadtree datasize. To be more specific, in our measured set the datasizes drop to 97% to 13% compared to the standard implementation.

2) Results for the quad data size: Figures 1 and 2 illustrate the fact that larger values of parameters $tile_size$ and $dense_limit$ make quadtree data structure more space efficient.

![Figure 1](image)

**Figure 1.** The influence of $tile_size$ on the size of quadtree.

3) Results for the $GetXY$ operation: Figures 3 and 4 show great speedups in quadtree implementation of the $GetXY$ operation. For larger values of the parameter $tile_size$ is the operation faster. It follows from the fact that the complexity of this operation is $O(\log(n/tile_size) + dense_limit)$.

4) Results for the $Transp$ operation: Figures 5 and 6 show great speedups in quadtree implementation of the $Transp$ operation. For larger values of the parameter $tile_size$ is the operation faster. It follows from the recursive memory access pattern.
5) Results for the SpM × V operation: Figures 7 and 8 illustrate the performance for the matrix-vector multiplication. For the quadtree format, the multiplication of sparse matrix by a dense vector is slower due to a large control and storage overhead. This operation is very simple, the data structure is read only once so a better cache locality in the quadtree format doesn’t result in a higher performance.

6) Results for the SpM × M operation: Figures 9 and 10 show speedups in the quadtree implementation of the SpM × M operation. For larger values of the parameter tile_size is the operation faster. The exact value of the speedup depends strongly on the structure (locations of nonzero elements) of used matrix. There are two main reasons for speedup:

- The quadtree format reduces indirect addressing.
- The quadtree data structure is read repeatedly and the cache locality increases due to the recursive memory access pattern.

IV. CONCLUSIONS

We have implemented some very basic routines from the linear algebra using an unusual data structure called a
in the case of multiplication of a sparse matrix by a sparse matrix and about 700% in case of getting value at the given position). It results from the fact that the quadtree data structure combines advantages of sparse and dense data formats.

V. FUTURE WORKS

• We should optimize some routines and deeply measure the performance on various platforms.

• We should measure the cache behavior and derive an analytical model of the cache behavior.

• We should implemented another routines from the LA using quad-tree and investigate possibilities of multi-threaded version of routines.

ACKNOWLEDGEMENT

This research has been supported by MŠMT under research program MSM6840770014.

REFERENCES


3.2 RP2: Space-efficient sparse matrix storage formats with 8-bit indices

In this paper (RP2), our first space-efficient storage formats for storing very large sparse matrices suitable for parallel I/O systems were introduced. The first two formats were (basic) hierarchical (2-level) formats with COO format on the first level and COO or CSR format on the second level. The size of indices on the second level was fixed to 8 bits, thus these formats are denoted as COOCOO256 and COOCSR256. The last format proposed in this paper was space optimal format for matrices without any locality (with random distribution of nonzero elements).

The motivation of our work were applications with very large sparse matrices that must be processed on massively parallel computer systems and must be loaded/stored from/to a distributed file system using parallel I/O. In our approach, we try to reduce time for LOAD/SAVE operations by reducing the amount of data to be loaded/stored. The results are summarized in Table 1, some of testing matrices consumes only about half of space needed for storing using common formats. Important observation is that the space complexity of the format that uses the CSR format on the lower level is more sensitive to the number of nonzero blocks.

Our measurements show that proposed formats can significantly reduce amount of data needed for storing these matrices.
Space-efficient sparse matrix storage formats with 8-bit indices

I. Šimeček, D. Langr
Department of Computer Systems, Faculty of Information Technologies, Czech Technical University, Prague

1 Introduction

The paper is aimed at space-efficient storage formats for very large sparse matrices (VLSMs). By VLSMs, we mean matrices that because of their size must be stored and processed by massively parallel computer systems with distributed memory architectures consisting of tens or hundreds of thousands of processor cores (MPCs).

1.1 Applicability of new formats

Our research addresses computational problems satisfying the following conditions:

1. Construction of VLSMs is slow.
2. Construction of VLSMs is memory-intensive for auxiliary data structures.
3. The computation performed with VLSMs implies collisions of data structures.
4. The computational tasks with VLSMs need checkpointing and recovery from failures.

If some of these conditions are met, we might need to store VLSMs into a file system. And since the file system access is of orders of magnitudes slower comparing to the memory access, we want to store matrices in a way that minimizes their storage space.

1.2 State-of-the-art survey

There are several widely used and well documented text-based file formats for sparse matrices, mainly Matrix Market, Harwell-Boeing, and Matlab (ASCII). There are, however, many reasons why text-based storage formats are not suitable for VLSMs: they must be accessed sequentially and they consume more space than binary formats.

As for binary file formats, there are no satisfactory solutions. Many modern sparse solvers, such as Trilinos or PETSc, provide the functionality of storing matrices to a file in their proprietary (and usually undocumented) file formats, but this does not satisfy our requirements.

1.3 Our assumptions and requirements for new formats

- $A$ is a very large sparse matrix of order $n$ with $k$ number of nonzero elements.
- We assume that $1 \gg k \gg N = n^2$. 

1
• We assume that there is no pattern of nonzero elements in $A$, or it is unknown.

In this paper, we focus only on the compression of the structure of matrices. Values of nonzero elements are unchanged, because their compression depends strongly on the application. The requirements for the formats proposed are as follows:

1. They should be space-efficient, because we try to keep resulting files sizes as low as possible.
2. They can be proceeded by parallel I/O.

1.4 Representing indices in binary codes

Let us have an array $Y$ of $\xi$ elements indexed from 0 to $\xi - 1$. The minimum number of bits of unsigned data type for indexing $Y$ is $S_0(\xi) = \lceil \log_2 \xi \rceil$, and the minimum number of bytes is thus $S_1(\xi) = \lceil S_0(\xi)/8 \rceil$. Since we usually use data types padded to the nearest power of 2 bytes, the actual number of bytes is hence $S(\xi) = 2^\eta$, where $\eta = \lceil \log_2 S_1(\xi) \rceil$.

2 Sparse storage formats

Sparse storage formats (SSFs) describe a way how are sparse matrices stored in a computer memory. Extensive research has been conducted about SSFs in context of performance optimization of sparse matrix-vector multiplication (SpMV) algorithms. However, few research results have been published about SSFs in context of minimization of the required memory, which is the optimization criterion for file I/O. Some recent research of hierarchical blocking SSFs, though primarily aimed at SpMV optimization, also addresses optimization of memory requirements.

2.1 The coordinate (COO) format

The matrix $A$ is represented by three linear arrays $Elem$, $X$, and $Y$. The array $Elem[1,\ldots,k]$ stores the nonzero values of $A$, the arrays $X[1,\ldots,k]$ and $Y[1,\ldots,k]$ contain column and row positions, respectively, of the elements with the nonzero value. Space requirements for indices of this format is $S_{COO}(n,k) = 2 \cdot k \cdot S(n)$ bytes.

2.2 The compressed sparse row (CSR) format

The matrix $A$ stored in the CSR format is represented by three linear arrays $Elem$, $Addr$, and $Ci$. The array $Elem[1,\ldots,k]$ stores the nonzero elements of $A$, the array $Addr[1,\ldots,n]$ contains indices of initial nonzero elements of rows of $A$, and the array $Ci[1,\ldots,k]$ contains column indices of nonzero elements of $A$. Space requirements for indices of this format is $S_{CSR}(n,k) = k \cdot S(n) + n \cdot S(k)$ bytes.

2.3 Hierarchical formats

The high memory requirements for the COO format are caused by two arrays of size $k$. We cannot reduce the size of these arrays, but we can try to reduce the number of bytes for every row/column index. The idea is to partition the matrix into square blocks of size $256 \times 256$.
rows/columns. Every such block can be identified by block row and block column indices of size $S(\lceil n/256 \rceil)$ bytes. Let $K$ denotes the number of nonzero blocks for our matrix $A$ (nonzero block is a block that contains at least one nonzero value).

### 2.3.1 COOCOO256 format

Suppose nonzero matrix elements stored in the COO format. If we store nonzero blocks in the coordinate storage format, we need for each one its block row/column index of size $S(\lceil n/256 \rceil)$ bytes, and a pointer into its data (an index into the original arrays of row/column indices and values) that it therefore an index of size $S(k)$ bytes. Now, for each nonzero element, we need only 1-byte local row/column indices valid within a block instead of $S(n)$-byte row/column indices valid within the whole matrix. Finally, instead of two arrays of $k$ row/column indices of size $S(n)$, we have five arrays:

- two arrays of size $k$ containing 1-byte local row/column indices within a block,
- two arrays of size $K$ containing $S(\lceil n/256 \rceil)$-byte block row/column indices within a matrix,
- one array of size $K$ containing $S(k)$-bytes indices to the elements of each block.

For this storage format named COOCOO256, the indices hence occupy

$$S_{\text{COOCOO256}}(n, k, K) = 2 \cdot k + 2 \cdot K \cdot S(\lceil n/256 \rceil) + K \cdot S(k)$$

bytes.

### 2.3.2 COOCSR256 format

If we compare memory requirements of indices for the COO and CSR storage formats, we see that the advantage of the CSR storage format is caused by using only one array of size $k$. We can apply the same idea for COOCOO256 by using compressed sparse row storage format for each nonzero block’s data. The only difference is that we throw away the array of local row indices, and use new array with 256 pointers for each nonzero block that indicates where the data of each of its row begins. Since we already know where begins data of each nonzero block, we now need only 2-byte pointer increments in this new array (the maximum number of nonzero values for a block is $256 \times 256 = 65536$ and hence we need $S(65536) = 2$ bytes increments).

For this storage format named COOCSR256, the indices hence occupy

$$S_{\text{COOCSR256}}(n, k, K) = k + K \cdot 256 \cdot 2 + 2 \cdot K \cdot S(\lceil n/256 \rceil) + K \cdot S(k)$$

bytes.

### 2.4 Space optimal format

Minimal amount of data (denoted by $S_{\text{min}}$) needed for storing location informations can be enumerated by following idea: The value of $S_{\text{min}}$ is the value of entropy of bit vector of size $N$ in which $k$ bits are set to 1 and $N - k$ bits are set to 0. The number of these vectors is $C = \binom{N}{k}$,
so the entropy is

\[ S_{\text{min}} = \log_2 C = \frac{N \sum_{i=N-k+1}^{i=N} \log_2 i}{k \sum_{i=1}^{i=k} \log_2 i}. \]

Unfortunately, this expression is hard to compare with storage formats. So, we approximate this using Stirling’s formula:

\[ C = \binom{N}{k} \approx \frac{1}{\sqrt{2\pi}} \cdot \sqrt{\frac{N}{(N-k)k}} \cdot \frac{N^N}{k^k \cdot (N-k)^{N-k}}, \]

which gives us

\[ S_{\text{min}} = \log_2 \left( \binom{N}{k} \right) \approx N \log_2 N - (N-k) \log_2(N-k) - k \log_2 k. \]

Please recall that \( k \ll N \). We can use this approximation for very small \( x \): \( \ln(1+x) \approx x \), so \( \ln(N-k) = \ln N - k/N \). The final approximation is then:

\[ S_{\text{min}} \approx k/\ln 2 + k \log N - k^2/(N \cdot \ln 2) - k \log k \approx k(\ln 2 - k/N + \log_2 N - \log_2 k). \]

## 3 Results

We performed experiments with described COOCOO256 and COOCSR256 storage formats and compared them with widely used CSR. The results are shown in Table 1. We can clearly see that for some matrices, when minimizing the size of indices, hierarchical blocking SSFs can spare significant amount of memory.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>H,A12</th>
<th>nlpkkt120</th>
<th>ldoor</th>
<th>TSOPF_RS.b2383</th>
<th>mouse_gene</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>4.49 \cdot 10^5</td>
<td>3.54 \cdot 10^6</td>
<td>9.52 \cdot 10^5</td>
<td>3.81 \cdot 10^4</td>
<td>4.51 \cdot 10^4</td>
</tr>
<tr>
<td>( k )</td>
<td>4.56 \cdot 10^5</td>
<td>5.02 \cdot 10^7</td>
<td>2.37 \cdot 10^7</td>
<td>1.62 \cdot 10^7</td>
<td>1.45 \cdot 10^7</td>
</tr>
<tr>
<td>( K )</td>
<td>840105</td>
<td>74027</td>
<td>109509</td>
<td>1321</td>
<td>15753</td>
</tr>
<tr>
<td>pCOOCOO256</td>
<td>50.00</td>
<td>46.98</td>
<td>48.96</td>
<td>99.56</td>
<td>99.71</td>
</tr>
<tr>
<td>pCOOCSR256</td>
<td>27.42</td>
<td>41.33</td>
<td>81.92</td>
<td>51.88</td>
<td>77.75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Trel_6Li</th>
<th>bmw7st_1</th>
<th>amazon0312</th>
<th>thread</th>
<th>gupta2</th>
<th>c-29</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>1.98 \cdot 10^5</td>
<td>1.41 \cdot 10^5</td>
<td>4.01 \cdot 10^5</td>
<td>2.97 \cdot 10^4</td>
<td>6.21 \cdot 10^4</td>
<td>5.03 \cdot 10^3</td>
</tr>
<tr>
<td>( k )</td>
<td>3.94 \cdot 10^6</td>
<td>3.74 \cdot 10^6</td>
<td>3.20 \cdot 10^6</td>
<td>2.25 \cdot 10^6</td>
<td>2.16 \cdot 10^6</td>
<td>2.44 \cdot 10^4</td>
</tr>
<tr>
<td>( K )</td>
<td>26172</td>
<td>6638</td>
<td>902901</td>
<td>1334</td>
<td>1202</td>
<td>67</td>
</tr>
<tr>
<td>pCOOCOO256</td>
<td>48.87</td>
<td>48.52</td>
<td>94.58</td>
<td>97.60</td>
<td>94.71</td>
<td>83.35</td>
</tr>
<tr>
<td>pCOOCSR256</td>
<td>106.39</td>
<td>46.41</td>
<td>3294.17</td>
<td>63.73</td>
<td>50.99</td>
<td>100.44</td>
</tr>
</tbody>
</table>

Table 1: Comparison of storage requirements for proposed formats, where \( p_* \) denotes the percentage ratio between * and CSR.

**Acknowledgements:** This research has been supported by GACR under grant P202/12/2011, by CESNET Development Fund under project 390/2010, and by Prague CUDA Teaching Centre (PCTC).
3.3 RP3: Space-efficient sparse matrix storage formats for massively parallel systems

In this paper (RP3), ideas from previous paper [RP2] were extended. We proposed and evaluated new storage formats for sparse matrices that minimize the space complexity of information about matrix structure. In more details:

- We precised the idea of space optimal format for matrices without any locality (EB format), but no practical algorithm for transformation was given.
  - Tables II and III illustrate the fact that the space complexity of the testing matrices using common storage formats (COO and CSR) is significantly higher than using the EB format (independently on the padding) and apparently, they are not suitable for our purposes.
  - Table V shows the ratio of the space complexity of the AH format and the EB format. We can make the observation that the AH format is very efficient. There is only a small difference between the AH format and the EB format.

- We proposed more general (basic) hierarchical (2-level) formats (BH), all combination of COO or CSR formats were evaluated. The size of indexes on the second level was variable.
  - The graphs on Figure 3 illustrate that the value of parameter $c$ strongly influences the space complexity of the BH formats.
  - The graphs on Figures 4 and 5 compare relative space complexities of the COOCSR formats with respect to the EB format. The impact of the format chosen on the block level is negligible for higher values of parameter $c$ due to small number of nonzero blocks.
  - Table IV shows the ratio of the space complexity of the BH formats for $c = 8$ and of the EB format. From this table, we can make these observations:
    * The space complexity of the format that uses the CSR format on the lower level is more sensitive to the number of nonzero blocks.
    * For some testing matrices, the space complexity in the BH formats is less than in the EB format. This is caused by the fact that the EB format is optimal only if the distribution of nonzero elements is random (it means
without any locality in the matrix). In some matrices some kind of locality exists and the BH formats can exploit it.

- We proposed an advanced hierarchical (2-level) format that combines a bitmap at the top level and the COO format at the lower level (AH format).
  - We found AH format comparable to space optimal format (ACB), the second one is slightly more space-efficient, but the corresponding transformation algorithm is very complex.

Comparison of widely used COO or CSR formats with our new formats were performed. From Table VI we can conclude that there are only 2 matrices among 11 ones (\texttt{mouse.gene} and \texttt{amazon0312}) for which the BH format is not more efficient than the EB format. The drawback of this format is that the optimal value of parameter $c$ is unknown and must be computed. Table VII shows the ratios of our formats. We can conclude that the AH and BH formats, proposed in this paper, are:

- usually more memory efficient than the EB format (only 2 matrices are slightly larger),
- from 2 to 5 times more memory efficient than the most common CSR format,
- from 6 to 20 times more memory efficient than the source file in the Matrix Market format,
- from 1.5 to 4 times more memory efficient than the gzipped Matrix Market format.

These experiments also show that for matrices without any locality are ACB and AH formats the right choice, otherwise BH should be used. These experiments also prove that our new formats can significantly reduce the space complexity of these matrices. Low space complexity of these formats makes them good candidates for storing very large sparse matrices using parallel I/O systems.
Abstract—In this paper, we propose and evaluate new storage formats for sparse matrices that minimize the space complexity of information about matrix structure. The motivation of our work are applications with very large sparse matrices that due to their size must be processed on massively parallel computer systems consisting of tens or hundreds of thousands of processor cores and that must be stored in a distributed file system using parallel I/O. The parallel I/O is typically the main performance bottleneck and reading or writing such matrices from/to distributed file system can take significant amount of time. We try to reduce this time by reducing the amount of data to be processed.

I. INTRODUCTION

The paper investigates memory-efficient storage formats for very large sparse matrices (VLSMs). By VLSMs, we mean matrices that due to their size must be stored and processed by massively parallel computer systems (MPCSs) with distributed memory architecture consisting of tens or hundreds of thousands of processor cores.

A. Applicability of the new formats for storing sparse matrices

Definition A matrix of order \( n \) is dense if it contains \( \Theta(n^2) \) nonzero elements and it is sparse otherwise.

In practice, a matrix is considered sparse if the ratio of nonzero elements drops below some threshold.

Our research addresses computations with VLSMs satisfying at least one of the following conditions:

1) The VLSM is used repeatedly and computation of its elements is slow and it takes more time than its later reading from a file system.
2) Construction of a VLSM is memory-intensive. It needs significant amount of memory for auxiliary data structures, typically of the same order of magnitude as the amount of memory required for storing the VLSM itself.
3) A solver requires the VLSM in another format than is produced by a matrix generator and the conversion between these formats cannot be performed effectively on-the-fly.
4) Computational tasks with VLSMs need check-pointing and recovery from failures of the MPCSs. We assume that a distributed-memory parallel computation with a VLSM needs longer time. To avoid recomputations in case of a system failure, we need to save a state of these long-run processes to allow fast recovery. This is especially important nowadays (and will be more in the future) when MPCSs consist of tens or hundreds of thousands of processor cores.

If some of these conditions are met, we might need to store VLSMs into a file system. And since the file system access is of orders of magnitude slower compared to the memory access, we want to store matrices in a way that minimizes their memory requirements.

B. State-of-the-art survey

What are the nowadays possibilities to manage storing/loading VLSMs to/from a distributed file system? There are several widely used and well documented text-based file formats for sparse matrices, mainly Matrix Market [1], Harwell-Boeing [2], [3], and Matlab (ASCII) [4]. There are, however, reasons why text-based storage formats are not suitable for VLSMs—they must be accessed sequentially and they usually consume much more space than binary formats.

As for binary file formats, there are no satisfactory solutions. Many modern sparse solvers, such as Trilinos [5]–[7] or PETSc [8]–[10], provide the functionality of storing matrices into a file. However,

1) the matrices must already be loaded into the solver, whereas we might need to store matrices as they are constructed;
2) the binary formats of such files are usually proprietary and poorly or not at all documented, and therefore they cannot be simply used anywhere else.

C. Our assumptions for the design of matrix formats

In the following text, we assume that:

- The matrix \( A \) is a VLSM of order \( n \). The number of its nonzero elements is denoted by \( N \).
- We assume that \( 1 \ll N \ll M = n^2 \).
- There is none or unknown pattern of nonzero elements in \( A \).

D. Requirements for a new format

The requirements for a new format are as follows:

1) This format should be space-efficient, because we try to keep the resulting file sizes as low as possible.
2) It can be processed by a parallel I/O subsystem. By a parallel I/O, we mean a concurrent file I/O over a network or a parallel file system, accessible to all processes of a parallel program.

3) The amount of data read by each part of the parallel I/O should be similar (read operation should be balanced).

4) It should be possible to access the representation of the format linearly, with minimal amount of skipping. Since nowadays parallel I/O consists of thousands of hard discs, the minimal amount of skipping leads to maximal efficiency of reading.

5) For the designed format, there must also be fast and space-efficient algorithm for remapping from/to the common storage formats. Due to this assumption, the time of remapping is negligible in comparison to I/O operations.

Note: In this paper, we focus only on the compression of the structure of matrices. That is, for the space complexity of a sparse matrix storage format, we consider only information on locations of nonzero elements. We do not take into account the space complexity of storing the values of the nonzero elements, because it does not depend on the format and the compression of the values of nonzero elements depends strongly on the application. For some application areas, only the data about the structure of nonzero elements in the matrix A is needed, as, for example, unweighed graphs.

E. Representing indexes in binary codes

Let us consider an array of \( \xi \) elements indexed from 0 to \( \xi - 1 \). The minimum number of bits of an unsigned indexing data type is

\[
S^{\text{MIN}}(\xi) = \lceil \log_2 \xi \rceil.
\]

The value \( S^{\text{MIN}} \) is the minimum number of bits, but it is usually padded to whole bytes (\( S^{\text{BYTE}} \) bits)

\[
S^{\text{BYTE}}(\xi) = 8 \cdot \lceil S^{\text{MIN}}(\xi)/8 \rceil,
\]

or it is padded to the nearest power-of-2 bytes (\( S^{\text{POW}} \) bits)

\[
S^{\text{POW}}(\xi) = 2^n, \quad \text{where } n = \left\lceil \log_2 S^{\text{MIN}}(\xi) \right\rceil.
\]

When we describe a format, we use simply \( S(\xi) \) instead of \( S^{\text{MIN}}(\xi) \).

II. COMMON SPARSE STORAGE FORMATS

Sparse storage formats (SSFs) describe a way how sparse matrices are stored in a computer memory. Extensive research has been conducted about SSFs in the context of performance optimization of the sparse matrix-vector multiplication (SpMV) algorithms. The state-of-the-art research results were published, for example, in [11]–[13]. However, few research results have been published about SSFs in the context of minimization of the required memory, which is the optimization criterion for a file I/O. Some recent research of hierarchical blocking SSFs, though primarily aimed at SpMV optimization, also addresses optimization of memory requirements [14], [15].

A. The Coordinate (COO) Format

The structure of a sparse matrix \( A \) is represented by two linear arrays \( X[1, \ldots, N] \) and \( Y[1, \ldots, N] \) of column and row indexes, respectively. The space complexity of indexes of this format is

\[
S_{\text{COO}}(n, N) = 2 \cdot N \cdot S(n).
\]

B. The Compressed Sparse Row (CSR) format

The most common format for storing sparse matrices is the compressed sparse row (CSR) format (see [16], [17]). The structure of a sparse matrix \( A \) stored in the CSR format is represented by array \( \text{Addr}[1, \ldots, n] \) that contains indexes of initial nonzero elements of rows of \( A \) and array \( C[i][1, \ldots, N] \) that contains column indexes of nonzero elements of \( A \). The space complexity of indexes of this format is

\[
S_{\text{CSR}}(n, N) = N \cdot S(n) + n \cdot S(N).
\]

In the further text, we distinguish dynamic complexity (that depends on the number of nonzero elements)

\[
S_{\text{CSRdyn}}(n, N) = N \cdot S(n)
\]

and fixed complexity (that does not depend on the number of nonzero elements)

\[
S_{\text{CSRfixed}}(n, N) = n \cdot S(N).
\]

Obviously,

\[
S_{\text{CSR}}(n, N) = S_{\text{CSRfixed}}(n, N) + S_{\text{CSRdyn}}(n, N).
\]

C. The entropy-based (EB) format

The space complexity of any sparse matrix storage format depends strongly on its structural pattern. If the sparsity pattern of a matrix is completely known (for example, if a matrix is tridiagonal) then the space complexity for storing the information on its structure is zero. If we assume a random distribution of nonzero elements, then it is equal to the value of the entropy of a bit vector of size \( M \), in which \( N \) bits are set to 1 and \( M - N \) bits are set to 0. We call such a format entropy-based. The number of these vectors is \( C = \binom{M}{N} \), so its entropy is

\[
S_{\text{EBF}}(n, N) = \log_2 C = \sum_{i=M-N+1}^{M} \log i - \sum_{i=1}^{N} \log i. \quad (1)
\]

Recall that \( M = n^2 \). Unfortunately, formula (1) is hard to compare with complexity formulae of other storage formats. So, we approximate it using the Stirling formula\(^3\):\

\[
C = \binom{M}{N} \\
\approx \frac{\sqrt{2\pi M} \left(\frac{M}{e}\right)^M}{\sqrt{2\pi(M-N)} \left(\frac{M-N}{e}\right)^{M-N}} \\
\approx \frac{1}{\sqrt{2\pi}} \cdot \frac{M^{M}}{(M-N)^{N} \cdot N^{M-N}} \\
\]

\(^3\)There exist more precise approximations like [18], but they lead to the same results and our goal is not the most accurate approximation.
So,

\[ S_{\text{EBF}}(n, N) = \log_2 \left( \frac{M}{N} \right) \]

\[ \approx \frac{1}{2} \log_2 M + M \log_2 M - \frac{1}{2} \log_2 (2\pi N (M - N)) \]

\[ \approx - N \log_2 N - (M - N) \log_2 (M - N) \]

Recall that \( N \ll M \). For a very small \( x \), we can approximate \( \ln(1 + x) \approx x \), so \( \ln(M - N) \approx \ln M - N/M \). The final approximation is then:

\[ S_{\text{EBF}}(n, N) \approx \frac{N}{\ln 2} + N \log_2 M - \frac{N^2}{M \ln 2} - N \log_2 N \]

\[ \approx N \left( \frac{1}{\ln 2} + \log_2 M - \log_2 N \right) \cdot \left( \frac{n}{2^c} \right) + 2 \cdot \log_2 n - \log_2 N \].

III. THE PROPOSAL OF NEW FORMATS

A. Basic hierarchical (BH) formats

The memory requirements for usual sparse matrix formats are given by two index arrays of size \( N \) in case of the COO format and by one array of size \( N \) in case of the CSR format. Since explicit index arrays cannot be eliminated, we cannot reduce the number of elements of these index arrays, but we can reduce the number of bits per one row/column index. The idea is to partition the matrix into square disjoint blocks of size \( 2^c \times 2^c \) rows/columns, where \( c \in \mathbb{N} \) is a formal parameter. Coordinates of the upper left corners of these blocks are aligned to multiples of \( 2^c \). So, indexes of nonzero elements are separated in two parts, indexes of blocks and indexes inside the blocks. Every such a block has block row and block column indexes of size \( S([n/2^c]) \) bits. Let \( B(c) \) denote the number of nonzero blocks for matrix \( A \). A nonzero block is a block that contains at least one nonzero matrix element. The minimal number of nonzero blocks is equal to

\[ B(c)_{\text{min}} = \left\lceil \frac{N}{2^c} \right\rceil \]

if all nonzero blocks contain only nonzero elements (i.e., are 100% dense). The maximal number of nonzero blocks is equal to

\[ B(c)_{\text{max}} = \min \left( N, \left\lceil \frac{n}{2^c} \right\rceil \right) \]

if each nonzero block contains exactly one nonzero element or if the whole matrix \( A \) is covered by nonzero blocks.

For storing information about the blocks and elements inside the blocks, we can use the COO or CSR format, which results in four combinations of these formats.

1) The COOCOO format: The main idea of this format is depicted on Figure 1. Since we store nonzero blocks in the COO format, we need for each one its \( 2 \cdot c \)-bit block row/column index into arrays \( bX/bY \) and a pointer into its data, implemented as an index into the arrays of row/column indexes \( inX/inY \) and values (not shown in Figure 1). Hence, it takes \( S(N) \) bits. For each nonzero block, we need only \( c \)-bit long local row/column indexes valid within a block instead of \( S(n) \) row/column indexes valid within the whole matrix. Together, instead of two arrays of \( N \) row/column indexes of size \( S(n) \), we have five arrays:

- two arrays \( inX \) and \( inY \) of size \( N \), each containing \( c \) bits of local row/column indexes within a block,
- two arrays \( blX \) and \( blY \) of size \( B(c) \) each containing \( S([n/2^c]) \) bits of block row/column indexes,
- one array \( blS \) of size \( B(c) \) each containing \( S(N) \) bits indexes.

In the COOCOO format, the indexes need \( S_{\text{COOCOO}}(n, N, B(c), c) \) bits, where

\[ S_{\text{COOCOO}}(n, N, B(c), c) = S_{\text{COO}}([n/2^c], B(c)) + S_{\text{CSRoff}}(2^c, N) + B(c) \cdot S(N) \]

\[ = 2 \cdot B(c) \cdot S([n/2^c]) + 2 \cdot N \cdot c + B(c) \cdot S(N). \]

2) The COOCOSR format: If we compare the memory requirements of indexes for the COO and CSR storage formats, we see that the advantage of the CSR format is caused by using only one array of size \( N \). We can apply the same idea as for the COOCOO format by using the CSR format for each nonzero block. The only difference is that we do not use the array of local row indexes, but array \( InAddr \) with \( 2^c \) pointers for each nonzero block that indicates where the data of each of its row begin. Since we already know where the data of each nonzero block begin, we only need \( 2 \cdot c \)-bit long offset in array \( InAddr \) (the maximum number of nonzero values for a block is \( 2^c \cdot 2^c = 2^{2c} \) and we need \( S(2^{2c}) = 2c \)-bit offsets).

Hence, in the COOCOSR format, the indexes need \( S_{\text{COOCOSR}}(n, N, B(c), c) \) bits, where

\[ S_{\text{COOCOSR}}(n, N, B(c), c) = S_{\text{COO}}([n/2^c], B(c)) + S_{\text{CSRffwd}}(2^c, N) + B(c) \cdot S(N) \]

\[ = 2 \cdot B(c) \cdot S([n/2^c]) + N \cdot c + 2 \cdot c \cdot B(c) \cdot 2^c + B(c) \cdot S(N) \]

\[ = N \cdot c + B(c) \left( 2 \cdot S([n/2^c]) + S(N) + 2 \cdot c \cdot 2^c \right). \]
3) The CSR COO format: In the CSRCOO format, the information about blocks is stored using the CSR format and information about elements inside blocks are stored using the COO format. Hence, the indexes need $S_\text{CSRCOO}(n, N, B(c), c)$ bits, where

$$S_\text{CSRCOO}(n, N, B(c), c) = S_{\text{CSR}}([n/2^c], B(c)) + S_{\text{COO}}(2^c, N) + B(c) \cdot S(N)$$

$$= B(c) \cdot S([n/2^c]) + [n/2^c] \cdot S(B(c)) + 2 \cdot N \cdot c + B(c) \cdot S(N).$$

4) The CSRCSR format: In the CSRCSR format, both the information about blocks and the information about elements inside blocks are stored using the CSR format. The indexes need $S_\text{CSRCSR}(n, N, B(c), c)$ bits, where

$$S_\text{CSRCSR}(n, N, B(c), c) = S_{\text{CSR}}([n/2^c], B(c)) + S_{\text{CSRDyn}}(2^c, N)$$

$$+ B(c) \cdot (S_{\text{CSRFixed}}(2^c, 2^c) + S(N))$$

$$= B(c) \cdot S([n/2^c]) + [n/2^c] \cdot S(B(c)) + N \cdot c + 2 \cdot B(c) \cdot c \cdot 2^c + B(c) \cdot S(N)$$

$$= N \cdot c + [n/2^c] \cdot S(B(c)) + B(c) \cdot S([n/2^c]) + 2 \cdot c \cdot 2^c + S(N).$$

B. The advanced hierarchical (AH) format

Another type of a hierarchical format is a format that combines a bitmap at the top level and the COO format at the lower level. The COO format is used due to small number of elements inside each block. So, this format consists of:

- One bitmap (in this bitmap each bit-pixel represents a block of $s \times s$ elements in matrix $A$. If this block is empty, then the corresponding bit in the bitmap is set to 1 and vice versa. Obviously, the size of the bitmap is $\lfloor n/s \rfloor \times \lfloor n/s \rfloor$.

- A list of nonzero elements inside nonzero blocks. For each such an element, we need $S_{\text{MIN}}(s^2)$ bits for the in-block row and column coordinates and 1 bit for the flag if it is the first nonzero element in the block.

So, the size of the bitmap format (denoted by $f$) is

$$f(n, N) = (\frac{n}{s})^2 + N(2 \cdot S_{\text{MIN}}(s) + 1).$$

An important question is how to derive a suitable value of parameter $s$?

1) The first idea: The first idea is based on the maximal entropy in the bitmap, so the probability $p$ that the given bit is set to zero is equal to $1/2$. We assume a random distribution of the nonzero entries, so $p = \left(1 - \frac{1}{N}\right)^{\frac{n}{2}}$. Since $s \ll n$, we can approximate $1 + x \approx 1 + x \cdot y$, and then $p \approx s^2 \cdot N/n^2$. It results in

$$\frac{1}{2} = \frac{s^2 \cdot N}{n^2}.$$  

Hence, the suitable value of $s$ is

$$s = \sqrt{\frac{n^2 \cdot \log 2}{2N}}.$$  

(4)

If we combine (3) and (4), we get that in this storage scheme, the indexes occupy $S_{\text{AHF}}(n, N)$ bits, where

$$S_{\text{AHF}}(n, N) = N(2 + 2 \log_2 n - \log_2 N).$$  

(5)

2) The second idea: The second idea is based on finding the local minimum of the function $f(\partial f/\partial s = 0)$. We can derive that the suitable value of $s$ is

$$s = \sqrt{\frac{n^2 \cdot \log 2}{2N}}.$$  

(6)

From (3) and (6), we get that in this storage scheme, the indexes together occupy $S_{\text{AHF}}(n, N)$ bits, where

$$S_{\text{AHF}}(n, N) = N(1 + 1/\log 2 + 2 \log_2 n - 2 \log_2 n) - \log_2(N).$$  

(7)

By comparing the results from (5) and (7), we see that the AH format using value $s$ from (6) is slightly more efficient (about 0.09 bit per non-zero element). By comparing results from (7) and (2), we see that the AH format is only slightly less efficient (about 0.47 bit per non-zero element) than the EB format.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$n$</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM9</td>
<td>5.56 \times 10^{4}</td>
<td>5.95 \times 10^{4}</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>3.54 \times 10^{6}</td>
<td>5.02 \times 10^{6}</td>
</tr>
<tr>
<td>ldoor</td>
<td>9.52 \times 10^{4}</td>
<td>2.37 \times 10^{6}</td>
</tr>
<tr>
<td>TSOFF_Res2383</td>
<td>3.81 \times 10^{4}</td>
<td>1.62 \times 10^{4}</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>4.51 \times 10^{4}</td>
<td>1.45 \times 10^{7}</td>
</tr>
<tr>
<td>t2em</td>
<td>9.25 \times 10^{4}</td>
<td>4.59 \times 10^{6}</td>
</tr>
<tr>
<td>bmw7at_1</td>
<td>1.41 \times 10^{4}</td>
<td>3.74 \times 10^{6}</td>
</tr>
<tr>
<td>amazon0312</td>
<td>4.01 \times 10^{4}</td>
<td>3.20 \times 10^{6}</td>
</tr>
<tr>
<td>thread</td>
<td>2.97 \times 10^{4}</td>
<td>2.25 \times 10^{6}</td>
</tr>
<tr>
<td>gupta2</td>
<td>6.21 \times 10^{4}</td>
<td>2.16 \times 10^{6}</td>
</tr>
<tr>
<td>c-29</td>
<td>5.03 \times 10^{4}</td>
<td>2.44 \times 10^{4}</td>
</tr>
</tbody>
</table>

**TABLE I**

**Characteristics of the testing matrices.**

IV. RESULTS

A. Testing matrices

We have used 11 testing matrices from various fields from the University of Florida Sparse Matrix Collection [19]. Table I shows the characteristics of the testing matrices.
TABLE II
THE RATIO OF THE SPACE COMPLEXITIES OF THE COO FORMAT USING DIFFERENT PADDINGS AND OF THE EB FORMAT.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>COO, $S^\text{MIN}$</th>
<th>COO, $S^\text{BYTE}$</th>
<th>COO, $S^\text{POW}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>2.25</td>
<td>2.35</td>
<td>3.13</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>2.27</td>
<td>2.48</td>
<td>3.30</td>
</tr>
<tr>
<td>ldoor</td>
<td>2.40</td>
<td>2.88</td>
<td>3.84</td>
</tr>
<tr>
<td>TSOFF_RS_b2383</td>
<td>4.04</td>
<td>4.04</td>
<td>4.04</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>3.73</td>
<td>3.73</td>
<td>3.73</td>
</tr>
<tr>
<td>t2em</td>
<td>2.11</td>
<td>2.53</td>
<td>3.38</td>
</tr>
<tr>
<td>bmw7st_1</td>
<td>2.61</td>
<td>3.47</td>
<td>4.63</td>
</tr>
<tr>
<td>amazon0312</td>
<td>2.23</td>
<td>2.81</td>
<td>3.75</td>
</tr>
<tr>
<td>thread</td>
<td>2.98</td>
<td>3.18</td>
<td>3.18</td>
</tr>
<tr>
<td>gupta2</td>
<td>2.61</td>
<td>2.61</td>
<td>2.61</td>
</tr>
<tr>
<td>c-29</td>
<td>2.27</td>
<td>2.79</td>
<td>2.79</td>
</tr>
</tbody>
</table>

TABLE III
THE RATIO OF THE SPACE COMPLEXITIES OF THE CSR FORMAT USING DIFFERENT PADDINGS AND OF THE EB FORMAT.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CSR, $S^\text{MIN}$</th>
<th>CSR, $S^\text{BYTE}$</th>
<th>CSR, $S^\text{POW}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>1.24</td>
<td>1.32</td>
<td>1.71</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>1.23</td>
<td>1.36</td>
<td>1.77</td>
</tr>
<tr>
<td>ldoor</td>
<td>1.26</td>
<td>1.52</td>
<td>2.00</td>
</tr>
<tr>
<td>TSOFF_RS_b2383</td>
<td>2.03</td>
<td>2.03</td>
<td>2.03</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>1.87</td>
<td>1.87</td>
<td>1.88</td>
</tr>
<tr>
<td>t2em</td>
<td>1.30</td>
<td>1.5</td>
<td>2.03</td>
</tr>
<tr>
<td>bmw7st_1</td>
<td>1.36</td>
<td>1.80</td>
<td>2.40</td>
</tr>
<tr>
<td>amazon0312</td>
<td>1.28</td>
<td>1.58</td>
<td>2.11</td>
</tr>
<tr>
<td>thread</td>
<td>1.52</td>
<td>1.62</td>
<td>1.63</td>
</tr>
<tr>
<td>gupta2</td>
<td>1.36</td>
<td>1.36</td>
<td>1.38</td>
</tr>
<tr>
<td>c-29</td>
<td>1.40</td>
<td>1.68</td>
<td>1.68</td>
</tr>
</tbody>
</table>

B. Results on the common sparse storage formats

Tables II and III illustrate the fact that the space complexity of the testing matrices using common storage formats (COO and CSR) is significantly higher than using the EB format (independently on the padding) and apparently, they are not suitable for our purposes.

C. Results on the BH formats

The graphs on Figure 3 and 4 compare relative space complexities of the COOCSR ($S^\text{COOCSR}(n, N, B(c), c)$) and CSRCSR ($S^\text{CSRCSR}(n, N, B(c), c)$) formats with respect to the EB format ($S^\text{EBF}(n, N)$). The impact of the format chosen on the block level is negligible for higher values of parameter $c$. It is due to small number of nonzero blocks.

Table IV shows the ratio of the space complexity of the BH formats.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$B(c)$</th>
<th>COO COOCO</th>
<th>COO CSR</th>
<th>CSRCO</th>
<th>CSR CSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>161107</td>
<td>0.79</td>
<td>0.79</td>
<td>0.79</td>
<td>0.79</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>74027</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>ldoor</td>
<td>109509</td>
<td>0.97</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td>TSOFF_RS_b2383</td>
<td>1321</td>
<td>2.02</td>
<td>2.02</td>
<td>2.02</td>
<td>2.02</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>15753</td>
<td>1.87</td>
<td>1.87</td>
<td>1.87</td>
<td>1.87</td>
</tr>
<tr>
<td>t2em</td>
<td>22631</td>
<td>1.3</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td>bmw7st_1</td>
<td>6638</td>
<td>1.16</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>amazon0312</td>
<td>902901</td>
<td>1.47</td>
<td>68.74</td>
<td>68.48</td>
<td>68.48</td>
</tr>
<tr>
<td>thread</td>
<td>1334</td>
<td>1.59</td>
<td>1.59</td>
<td>1.59</td>
<td>1.59</td>
</tr>
<tr>
<td>gupta2</td>
<td>1202</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>c-29</td>
<td>67</td>
<td>1.40</td>
<td>1.40</td>
<td>1.40</td>
<td>1.40</td>
</tr>
</tbody>
</table>

TABLE IV
THE RATIO OF THE SPACE COMPLEXITIES OF THE BH FORMATS FOR $c = 8$ AND OF THE EB FORMAT ($S^\text{EBF}(n, N)$).
formats for $c = 8$ and of the EB format. From this table, we can make these observations:

- The space complexity of the format that uses the CSR format on the lower level is more sensitive to the number of nonzero blocks.
- For some testing matrices, the space complexity in the BH formats is less than in the EB format. This is caused by the fact that the EB format is optimal only if the distribution of nonzero elements is random (it means without any locality in the matrix). In some matrices some kind of locality exists and the BH formats can exploit it.

## D. Results for the AH format

Table V shows the ratio of the space complexity of the AH format and the EB format. We can make the observation that the AH format is very efficient. There is only a small difference between the AH format and the EB format.

### E. Discussion

Let us define $\text{minBHF}(n, N)$ as the minimum of $S_{\text{COOCO}}(n, N, B(c), c)$, $S_{\text{COOCSR}}(n, N, B(c), c)$, $S_{\text{CSRCSR}}(n, N, B(c), c)$, and $S_{\text{CSRCSR}}(n, N, B(c), c)$ for optimal values of $c$. Table VI shows the ratio of $\text{minBHF}(n, N)$ and the EB format size ($S_{\text{EB}}(n, N)$). There are only 2 matrices among 11 ones (mouse_gene and amazon0312) for which the BH format is not more efficient than the EB format. The drawback of this format is that the optimal value of parameter $c$ is unknown and must be computed. Table VII shows the ratios of our formats. We can conclude that the AH and BH formats, proposed in this paper, are:

- usually more memory efficient than the EB format (only 2 matrices are slightly larger),
- from 2 to 5 times more memory efficient than the most common CSR format,
- from 6 to 20 times more memory efficient than the source file in the Matrix Market format,
- from 1.5 to 4 times more memory efficient than the gzipped Matrix Market format.

### V. Conclusions

This paper deals with the design of new formats for storing very large sparse matrices suitable for parallel I/O systems. We performed experiments with our new formats and compared them with widely used COO or CSR formats. These experiments prove that our new formats can significantly reduce the space complexity of these matrices. High space complexity of these formats makes them good candidates for storing very large sparse matrices using parallel I/O systems.

The proposed formats are generic, i.e., they may be applied to sparse matrices of any structure. We performed experiments with matrices arising in many different application areas. Since these matrices are available in public data repositories only in “smaller” sizes, we did not processed the truly very large sparse matrices (of the memory size of the order of terabytes) in parallel. However, when processing such very large matrices on a massively parallel computer system, every processor has its own part of a matrix, which itself can be treated as a standalone matrix of a smaller size. Every processor can apply one of the proposed formats to its own matrix independently. Hence, the proposed formats can be utilized on massively parallel computer systems the very same way as in sequential computations.
ACKNOWLEDGMENT

This research has been supported by GACR grant No. P202/12/2011, and by CESNET Development Fund project 390/2010.

REFERENCES


3.4 RP4: Minimal Quadtree Format for Compression of Sparse Matrices Storage

In this paper (RP4) ideas from previous papers [RP1, RP2] were extended and a new storage format called the Minimal quadtree (MQT) was presented. This format extends ideas of the standard QT format as follows:

- All nodes in the MQT are stored in one array. Since we can compute locations of all child nodes, we can omit pointers. We lose the advantage of the possibility to easily modify the QT, but it is not an important property for our application area.

- Instead of pointers, each node of the MQT contains only 4 flags (i.e., 4 bits only) indicating whether given subquadtrees are nonempty.

Consequently, the space complexity of every MQT node is only 4 bits. The main drawback of other space-efficient BH formats is that the optimal size of indexes on the second level must be evaluated. For MQT format, this computation is omitted.

In this paper, algorithms for converting matrices from common storage formats to the MQT format were also mentioned. The lower and upper bound for structure space complexity for MQT format were also derived:

- The minimal size of the MQT format is \( \approx 4 \cdot \left( \frac{N}{3} + \log_4 \left( \frac{n^2}{N} \right) \right) \).

- The maximal size of the MQT format is \( \approx 4 \cdot N \left( \frac{1}{3} + \log_4 \left( \frac{n^2}{N} \right) \right) \).

We also derived upper bound of time complexity for “classical” transformation algorithm: the total time complexity of the transformation is

\[ \Theta(N(1 + n/\sqrt{N}) \cdot \log_2 \text{avg}_\text{per}_\text{row}). \]

A very usual case is \( N = \Theta(n) \), it means matrices with constant number of nonzero elements per row. For this case the time complexity is \( \Theta(n^{3/2}) \).

From the Table II, we can make the following observations: The MQT format

- is from two to five times more efficient than the most common CSR format.
• is from 6 to 45 times more efficient than the space complexity of the text-based Matrix Market format.

• uses from 12.8% to 64.7% of space complexity of the zipped Matrix Market format.

We can conclude that this format is very space-efficient.
Minimal Quadtree Format for Compression of Sparse Matrices Storage

I. Šimeček, D. Langr, P. Tvrdík
Department of Computer Systems, Faculty of Information Technology,
Czech Technical University in Prague Prague, Czech Republic Email: xsimecek,langrd,pavel.tvrdik@fit.cvut.cz

Abstract—Computations with sparse matrices are widespread in scientific projects. Commonly used storage formats (such as COO or CSR) are not suitable for I/O file operations with sparse matrices due to their high space complexities. Memory-efficient formats are still under development.

In this paper, we present a new storage format called the Minimal quadtree (MQ) as well as algorithms for converting matrices from common storage formats to the MQ format. We compare the space complexity of common storage formats and of the MQ format and prove that the idea of using the quadtree as the data structure for sparse matrices is viable.

I. INTRODUCTION

The space complexity of representation of sparse matrices depends strongly on the used matrix storage format. In this paper, we present a new quadtree-based storage format implementation. We extend the ideas presented in [1]–[4].

Definition A matrix of order \( n \) is dense if it contains \( \Theta(n^2) \) nonzero elements and it is sparse otherwise.

In practice, a matrix is considered sparse if the ratio of nonzero elements drops below some threshold.

The paper studies space efficient storage formats for large sparse matrices (LSMs) that allow more efficient I/O file operations. Since the file system access latency is of orders of magnitude greater than the memory access latency, we want to store matrices in a way that minimizes their storage requirements, i.e., in a way that compresses the LSM as much as possible.

In this paper, we focus only on the compression of the information describing the structure of LSMS (i.e., the locations of nonzero elements). The values of the nonzero elements are assumed to be unchanged, because their compression depends strongly on the application. For some application areas, the values of nonzero elements are implicit and only the information about the structure of a LSM is stored (for example, incident matrices of unweighed graphs). Alternatively, we can interleave computations with reading of nonzero elements. For example, we can divide the process of a sparse matrix factorization into these steps:

1) read the matrix structure,
2) do in parallel: perform the symbolic factorization and read the values of nonzero elements of the matrix,
3) perform the numeric factorization.

A. State-of-the-art survey

Let us review the nowadays possibilities to perform efficiently the I/O file operations with LSMs. There are several widely used and well documented text-based file storage formats for sparse matrices, mainly Matrix Market [5], Harwell-Boeing [6], [7], and Matlab (ASCII) [8]. There are, however, reasons why text-based storage formats are not suitable for our purposes: they usually consume much more space than binary formats.

As for the binary file storage formats, there are no satisfactory solutions. Many modern sparse solvers, such as Trilinos [9]–[11] or PETSc [12]–[14], provide functionality of storing matrices into a file. However, the binary formats of such files are usually proprietary and poorly or not at all documented, and therefore they cannot be simply used anywhere else.

B. Our assumptions and requirements for a new storage format

- We consider a large sparse matrix \( A \) of order \( n \). The number of its nonzero elements is denoted by \( N \) and the average number of nonzero elements per row is denoted by \( \text{avg}_{\text{per}_{\text{row}}} \). Obviously, \( \text{avg}_{\text{per}_{\text{row}}} = N/n \).
- We assume that \( 1 \ll N \ll M = n^2 \).
- The pattern of nonzero elements in \( A \) is unknown or random.

The requirements for a new storage format are as follows:

1) This format should be space-efficient, because we try to keep resulting file sizes as low as possible.
2) We want to access LSMs files linearly. Since nowadays I/O file operations are processed by hard discs, linear access with minimal amount of seek operations leads to a maximal efficiency of reading.
3) For the designed format, there must also exist a space-efficient algorithm with small algorithmic complexity for remapping from/to the common storage formats. Due to this assumption, the time complexity of remapping is negligible in comparison to I/O file operations complexity.

C. Representing indexes in binary codes

Let us have an array of \( \xi \) elements indexed from 0 to \( \xi - 1 \). The minimum number of bits of an unsigned indexing data type is

\[
S^{\text{MIN}}(\xi) = \left\lceil \log_2 \xi \right\rceil.
\]
The value \( S^{MIN} \) is the minimum number of bits, but it is usually padded to whole bytes (\( S^{BYTE} \) bits)
\[
S^{BYTE}(\xi) = 8 \cdot \lceil S^{MIN}(\xi)/8 \rceil,
\]
or it is padded to the nearest power of 2 bytes (\( S^{POW} \) bits)
\[
S^{POW}(\xi) = 2^{\eta}, \quad \text{where} \quad \eta = \lceil \log_2 S^{MIN}(\xi) \rceil.
\]
When we describe a matrix storage format, we use simply \( S(\xi) \) instead of \( S^{MIN}(\xi) \).

II. COMMON SPARSE STORAGE FORMATS

Sparse storage formats (SSFs) describe a way how sparse matrices are stored in computer memory. Extensive research has been conducted about SSFs in the context of performance optimization of the sparse matrix-vector multiplication (SpMV) algorithms. The state-of-the-art research results were published, for example, in [15]–[17]. However, few research results have been published about SSFs in the context of minimization of the required memory, which is the optimization criterion for file I/O file operations. Some recent research of hierarchical blocking SSFs, though primarily aimed at the SpMV optimization, also addresses optimization of memory requirements [18], [19].

A. The Coordinate (COO) Format

It is the simplest and most straightforward SSF. The matrix \( A \) is represented by three linear arrays \( Elem, X, \) and \( Y \). The array \( Elem[i, \ldots, N] \) stores the nonzero elements of \( A \), arrays \( X[1, \ldots, N] \) and \( Y[1, \ldots, N] \) contain column and row indexes, respectively, of these nonzero values.

1) Time complexity.: The time complexity of a conversion from/to the COO format is \( \Theta(N + n) \).

2) Space complexity.: The space complexity of the structure of matrix \( A \) (the array \( Elem \) is not counted) is
\[
S_{COO}(n, N) = 2 \cdot N \cdot S(n).
\]

B. The Compressed Sparse Row (CSR) format

It is the most common SSF (see [20]–[22]). The matrix \( A \) is represented by three linear arrays \( Elem, Addr, \) and \( Ci \). The array \( Elem[i, \ldots, N] \) stores the nonzero elements of \( A \), the array \( Addr[i, \ldots, n] \) contains indexes of initial nonzero elements of rows of \( A \). If row \( i \) does not contain any nonzero element, then \( addr[i] = addr[i + 1] \). The array \( Ci[1, \ldots, N] \) contains column indexes of nonzero elements of \( A \). Hence, the first nonzero element of row \( j \) is stored at index \( Addr[j] \) in the array \( Elem \).

1) Time complexity.: The time complexity of a conversion from/to the COO format is \( \Theta(N + n) \).

2) Space complexity.: The space complexity of the structure of matrix \( A \) (array \( Elem \) is not counted) is
\[
S_{CSR}(n, N) = N \cdot S(n) + n \cdot S(N).
\]

C. Quadtree-based sparse matrix storage formats

Definition The Quadtree (QT) is a tree data structure in which all inner nodes have exactly four child nodes.

The QT is used to describe a partition of a sparse matrix into submatrices. Each QT node represents a submatrix. There are different implementations of the QT format. In the standard implementation (for details, see [1]–[3]), every node in the QT is represented by structure \( \text{standard}_Q T \_struct \) consisting of the following items:

- four pointers (\( up\_left, up\_right, lo\_left, lo\_right \)) to the child nodes,
- (only for leaves) values of nonzero elements.

Recently, we have proposed an advanced implementation (for details, see [4]). It has an additional parameter \( tile\_size \) used to stop the recursive partitioning. There are different types of QT nodes. Inner QT nodes can be “Mixed” or “Empty”. QT leaves can be “Full” or “Sparse”. Every node in the advanced QT is represented by structure \( \text{advanced}_Q T \_struct \) consisting of the following items:

- node type (“Empty”, “Mixed”, “Sparse” or “Full”),
- four pointers (\( up\_left, up\_right, lo\_left, lo\_right \)) to the child nodes,
- (only for nodes of type “Sparse” or “Full”) values of nonzero elements.

Great advantages of the QT-based SSFs (both standard and advanced) are the following:

- Modifications (adding or removing nonzero elements) of the QT structure are relatively easy and fast in comparison to the COO or CSR formats.
- The recursive style of programming and recursive style of storage (“Divide and Conquer” approach) leads to codes with surprising performance due to the better cache memory utilization (see [4]).

1) The quadtree transformation algorithm.: Algorithm 3 converts a matrix \( A \) stored in the CSR format into the advanced QT-based format. Empty submatrices (that do not contain any nonzero elements) are represented by the NULL pointers. The algorithm uses parameter \( tile\_size \) as the maximum size of a submatrix in one node (except “Empty” nodes).

Algorithm 1 Enumeration of the number of nonzero elements in a given submatrix of \( A \) stored in the CSR format

1: \textbf{procedure} NNIS(A,\( x_1,y_1,x_2,y_2 \))
\textbf{Input:} \( A = \) the matrix in the CSR format
\textbf{Input:} \( x_1,y_1,x_2,y_2 \) = coordinates of the submatrix
\textbf{Output:} the number of nonzero elements in the submatrix of \( A \)
2: \( \text{number} \leftarrow 0; \)
3: \textbf{for} \( i \leftarrow A.Addr[y_1], A.Addr[y_2 + 1] - 1 \) \textbf{do}
4: \( x \leftarrow A.ci[i]; \)
5: \textbf{if} \( (x \geq x_1) \& (x \leq x_2) \) \textbf{then}
6: \( \text{number} \leftarrow \text{number} + 1; \)
7: \textbf{return} \( \text{number} \);
Algorithm 2 Test if the given submatrix of $A$ stored in the CSR format is nonempty

1: procedure INES($A, x_1, y_1, x_2, y_2$)
2: Input: $A$ = the matrix in the CSR format
3: Input: $x_1, y_1, x_2, y_2$ = coordinates of the submatrix
4: Output: logical value denotes if the given submatrix is nonempty
5: 
6: for $y ← y_1, y_2$ do
7:   $low ← A.addr[y]$; $high ← A.addr[y + 1] - 1$;
8:   $i ←$ BINARY SEARCH(in array $A.ci$);
9:   $\triangleright$ within indexes from $\langle low...high \rangle$
10:   $\triangleright$ to find minimal $i$ such that $A.c[i] ≥ x_1$
11:   if $C.c[i] ≤ x_2$ then
12:     return true;
13: return false;

Algorithm 3 Transformation algorithm from the CSR format to the advanced QT format

1: procedure TRANSF$(A, x_1, y_1, x_2, y_2)$
2: Input: $A$ = the matrix in the CSR format
3: Input: $x_1, y_1, x_2, y_2$ = coordinates of the submatrix to be transformed
4: Output: the pointer to the root of the QT representation of $A$
5: 
6: if INES$(A, x_1, y_1, x_2, y_2) = false$ then
7:   return NULL;
8: $n' ← max(x_2 - x_1, y_2 - y_1)$;
9: if $n' > tile.size$ then
10:   $mx ← (x_1 + x_2)/2$; $my ← (y_1 + y_2)/2$;
11:   create $M$ - the node of QT of type “Mixed”
12:   $M.up.left ←$ TRANSF$(A, x_1, y_1, mx, my)$;
13:   $M.up.right ←$ TRANSF$(A, mx + 1, y_1, x_2, my)$;
14:   $M.lo.left ←$ TRANSF$(A, x_1, my + 1, mx, y_2)$;
15:   $M.lo.right ←$ TRANSF$(A, mx + 1, my + 1, x_2, y_2)$;
16: $\triangleright$ previous four commands transform disjoint submatrices of the given submatrix of $A$
17: return $&M$
18: else
19: $N' ←$ NNIS$(A, x_1, y_1, x_2, y_2)$;
20: $F ←$ representation of the input submatrix $A$
21: $\triangleright$ in some common storage format
22: return $&F$;

2) Time complexity.: The time complexity of the procedure NNIS$(A,x_1,y_1,x_2,y_2)$ (Algorithm 1) is

$$\Theta_{avg}(avg\ per\ row \cdot (y_2 - y_1 + 1)).$$

The time complexity of the procedure INES$(A,x_1,y_1,x_2,y_2)$ (Algorithm 2) is

$$O(log_2 avg\ per\ row \cdot (y_2 - y_1 + 1)).$$

The time complexity of the procedure TRANSF$(A,x_1,y_1,x_2,y_2)$ depends on the value of parameter $tile.size$ and on the criteria for creation of leaves (see the results of experiments in [4]).

3) Space complexity.: Similarly, the space complexity of this format also depends on the value of parameter $tile.size$ and on the criteria for creation of leaves (see also [4]).

III. THE NEW MINIMAL QT BASED FORMAT

A big drawback of the previous QT formats is a larger data overhead (caused by pointers $up.left, up.right, lo.left, lo.right$) compared to the COO and CSR formats. Since our aim is to minimize the space complexity of QT-based formats, we propose a new QT format called minimal quadtree (MQT) format that extends ideas of the standard QT format as follows:

- All nodes in the MQT are stored in one array. Since we can compute locations of all child nodes, we can omit pointers $up.left, up.right, lo.left, lo.right$. We lose the advantage of the possibility to easily modify the QT, but it is not an important property for our application area.
- Instead of pointers, each node of the MQT contains only 4 flags (i.e., 4 bits only) indicating whether given subquadtrees are nonempty.

So, the space complexity of every MQT node is only 4 bits. Algorithm 4 transforms a matrix $A$ from the CSR format into the MQT format, represented by an output array. Since arrays are accessed in linear order, the output array can be viewed as a bitstream $S$.

A. Space complexity:

Let us assume a very small example of a sparse matrix with $n = 8$ and $N = 4$. For common storage formats, the space complexity is given by Eq. (1) or (2), so $S_{COO}(n,N) = 24[\text{bits}]$ and $S_{CSR}(n,N) = 28[\text{bits}]$. For the MQT, the exact size of the output bitstream $S$ (it means the size of the MQT format) cannot be derived from these global parameters, because it depends on the exact locations of nonzero elements. It ranges from 12 to 36 bits (see Figures 1 and 2). The derivation of the lower and upper bounds on the size of the MQT format in a general case is the following.

1) Lower bound: We consider the best case: the QT with the minimal number of nodes, i.e., the number of leaves is equal to $N$ (see Figure 3). It is obviously a generalized idea from Figure 1.

- The height of the QT on Figure 1 is $h = h_1 + h_2 = \log_2 n - 1$, where $h_2 = \log_4 N - 1$ and $h_1 = \log_2 n - \log_4 N$.
- All nodes with depth $< h_1$ (in upper $h_1$ levels) contain exactly one 1 (they have one child node). The number of nodes in these levels is $h_1$,

$$h_1 = \log_2(n^2/N).$$

- All nodes with height $> h_1$ (in lower $h_2$ levels) are full of 1’s (they have four child nodes). The number of nodes
Algorithm 4 Transformation from the CSR to the MQT format

1: procedure TR2MQT(A,x1,y1,x2,y2)
2: Input: A = the matrix in the CSR format to be transformed
3: S ← (∅);
4: TRANSF2(A, l, A.n, A.n);
5: return S;
6: procedure TRANSF2(A, x1, y1, x2, y2)
7: Input: A = the matrix in the CSR format to be transformed
8: S ← (∅);
9: TRANSF2(A, l, A.n, A.n);
10: return S;

in these levels is approximately
\[
\sum_{i=h+1}^{\log_2 n} 4^{i-(h+1)} \approx N/3.
\]
So, the minimal size of the MQT format is
\[
\approx 4 \cdot \left(\frac{N}{3} + \log_4(n^2/N)\right).
\]

2) Upper bound: We consider the worst case: the quadtree with the maximal number of nodes, i.e., the number of leaves is equal to \(N\) (see Figure 4). Again, it is a generalized idea from Figure 2).

- The height of this tree is \(h = h1 + h2 = \log_2 n - 1\), where \(h1 = \log_4 N\).
- All nodes with depth < \(h1\) (in upper \(h1\) levels) are full of 1’s (they have four child nodes), \(h1 = \log_4 N\). The number of nodes in these levels is approximately
\[
\sum_{i=0}^{h-1} 4^i \approx N/3.
\]
- All nodes with height \(\geq h1\) (in lower \(h2\) levels) contain exactly one 1 (they have one child node). The number of nodes in these levels is
\[
N \cdot h2 = N \cdot (\log_2 n - \log_4 N) = N \cdot \log_4(n^2/N).
\]

So, the maximal size of the MQT format is
\[
\approx 4 \cdot N(1/3 + \log_4(n^2/N))
\]

B. Time complexity of the transformation from the CSR format:

We consider the worst case (similar ideas as for derivation of the space complexity): the QT with the maximal number of nodes, the number of leaves is equal to \(N\) (see Figure 4). We assume that the time complexity of procedure APPENDTOBITSTREAM is \(\Theta(1)\). Procedure INES(A,x1,y1,x2,y2) is called for every node in the MQT in the output stream \(S\) four times.

- For nodes with depth \(= h1\): The number of these nodes is \(N\), the expression \((y2 - y1 + 1)\) is equal to \(1 + n\sqrt{N}\). The time complexity of the transformation for all nodes with this depth is \(T_{h1} = N \cdot (1 + n\sqrt{N}) \cdot \log_2 \text{avg\_per\_row}\).
- For nodes with depth \(< h1\): The number of these nodes is \(N/4\) and the expression \((y2 - y1 + 1)\) is equal to \(1 + 2n\sqrt{N}\). So, the total time complexity of the transformation for all nodes with depth \(\leq h1\) (in upper \(h1\) levels) is \(T_{upper} \approx \sum_{i=h1}^{n} T_{h1}/2^{i-1} \approx 1 + \log_2 \text{avg\_per\_row}\).
- For nodes with depth \(> h1\): The time complexity of the transformation for all these nodes (for the lower \(h2\) levels) is \(T_{lower} \approx \sum_{i=h1+1}^{n} T_{h1}/2^{i-1} \approx \Theta(N(1 + n\sqrt{N}) \cdot \log_2 \text{avg\_per\_row})\).

So, the total time complexity of the transformation is
\[
\Theta(N(1 + n\sqrt{N}) \cdot \log_2 \text{avg\_per\_row})
\]

A very usual case is \(N = \Theta(n)\), it means matrices with constant number of nonzero elements per row. For this case the time complexity is \(\Theta(n^{3/2})\).

![Diagram](image.png)

IV. RESULTS

A. Testing matrices

We have used 11 testing matrices from various application domains from the University of Florida Sparse Matrix Collection [23]. Table I shows the characteristics of the testing matrices. For our purposes, we have excluded all temporary information from the source Matrix Market files (like comments and values of nonzero values). For the comparison of
Fig. 2. a) The original sparse matrix A. Node Q1 is the root of the quadtree. This matrix with 4 nonzero elements is represented by 9 quadtree nodes = 36 bits. The output bitstream is "1111 1000 0100 0010 0001 1000 0100 0010 0001".

Fig. 4. Quadtree with the maximal number of nodes (the number of leaves is N).

... compression ratios, we have used the PKZIP program with the option for maximal compression.

B. HW and SW configuration

All results were measured on a dual-core Intel i3-370M at 2.4 GHz, 4 GB of the main memory at 1333 MHz, running OS W7 Home, hard disc Hitachi HTS545050B9A300 (500GB) with measured top read speed 44MBs⁻¹ (measured by program CrystalDiskMark 3.01).

Fig. 3. Quadtree with the minimal number of nodes (the number of leaves is N).

C. Results for the MQT format

For comparison of the formats, we define the parameter ratio of space complexity for format X as the ratio of space complexity of the MQT format and space complexity of the X format. Table II shows ratios of space complexity of the MQT format. From this table, we can make the following observations: The MQT format

- is from two to five times more efficient than the most common CSR format.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$S_{MQT}$ [bits]</th>
<th>$T_r$ [s]</th>
<th>$B_t$ [MBs⁻¹]</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>3.23 · 10⁶</td>
<td>3.64</td>
<td>39.6</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>1.93 · 10⁸</td>
<td>2.39</td>
<td>51.8</td>
</tr>
<tr>
<td>ldoor</td>
<td>5.69 · 10⁷</td>
<td>0.78</td>
<td>70.1</td>
</tr>
<tr>
<td>TSOFF_RS_b2383</td>
<td>4.3 · 10⁷</td>
<td>0.44</td>
<td>58.6</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>1.62 · 10⁸</td>
<td>2.6</td>
<td>3.03</td>
</tr>
<tr>
<td>t2em</td>
<td>2.6 · 10⁸</td>
<td>0.28</td>
<td>38.1</td>
</tr>
<tr>
<td>bmw7st_1</td>
<td>1.05 · 10⁸</td>
<td>0.14</td>
<td>50.1</td>
</tr>
<tr>
<td>amazon0312</td>
<td>5.85 · 10⁷</td>
<td>0.89</td>
<td>1.37</td>
</tr>
<tr>
<td>thread</td>
<td>5.4 · 10⁷</td>
<td>0.0779</td>
<td>4.65</td>
</tr>
<tr>
<td>gupta2</td>
<td>9.6 · 10⁷</td>
<td>0.16</td>
<td>20.9</td>
</tr>
<tr>
<td>c-29</td>
<td>1.43 · 10⁵</td>
<td>0</td>
<td>NA</td>
</tr>
</tbody>
</table>

TABLE III

Ratios of space complexity, $S_{MQT}$ denotes the size of the MQT format in bits, $T_r$ denotes the time to convert from the MQT format to (CSR, $S^{CSR}$). $B_t$ denotes the bandwidth threshold. The parameter $T_r$ for matrix c-29 is under precision of the used timer.
• is from 6 to 45 times more efficient than the space complexity of the text-based Matrix Market format.
• uses from 12.8% to 64.7% of space complexity of the zipped Matrix Market format.

We can conclude that this format is very efficient.

D. Testing matrices

We have also tested a practical usability of the MQT format. In real application, a matrix is transferred from/to a file using the maximal bandwidth (denoted by $B$). When we use the CSR format, the time to read (denoted by $T_{\text{readCSR}}$) is equal to $T_{\text{readCSR}} = S_{\text{CSR}}/B$. When we use the MQT format as a storage format and the CSR format as an inner format, the total time has two components:

- the time to read (denoted by $T_{\text{readMQT}}$) is equal to $T_{\text{readMQT}} = S_{\text{MQT}}/B$.
- the time to convert from the MQT format to the CSR format (denoted by $T_c$).

The MQT format pays off only if $T_{\text{readCSR}} > T_{\text{readMQT}} + T_c$. This is equivalent with

$$B < B_t = \frac{S_{\text{CSR}} - S_{\text{MQT}}}{T_c},$$

where $B_t$ is the bandwidth threshold. Said otherwise, if the used bandwidth $B$ is less than $B_t$, then the MQT format pays off and vice versa. Table III shows the measured values for $T_c$ and $B_t$ for HW ans SW configuration described in Section IV-B. We can conclude that for this testing configuration, the MQT format pays off for 5 of 11 testing matrices, but it is true only for this scenario with one hard disc dedicated to one processor core. In more realistic scenarios, where one hard disc is shared by multiple processor cores or where parallel I/O is shared by many cores of a supercomputer, the situation changes and the MQT format pays off for almost all testing matrices.

V. CONCLUSIONS

This paper deals with the design of a new sparse matrix storage QT-based format, called the MQT format, that minimizes space complexity of the LSM structure. We performed experiments with the MQT format and compared it with other common COO or CSR formats. These experiments proved that the MQT format can significantly reduce amount of data needed for storing LSMS, so it allows efficient I/O file operations.

ACKNOWLEDGMENT

This research has been supported by GACR grant No. P202/12/2011 and by CTU grant SGS12/097/0H3/1T/18.

REFERENCES


3.5 **RP5: The study of impact of matrix-processor mapping on the parallel sparse matrix-vector multiplication**

In this paper (RP5), the efficient parallelization of SpMV in distributed memory environment was discussed. In general, there is only one way how to parallelize SpMV:

1. Partitioning of matrix $A$ into $n_R$ disjoint nonempty regions denoted by $R_i$.
2. Every processor executes multiplication of region by (part of) vector $\vec{x}$ in parallel. More exactly, every processor $p_i$ proceeds SpMV with region $R_i$ and part of vector $\vec{x}$ and computes temporal result (vector $\vec{t}$).
3. Reduce temporal results (vectors $\vec{t}$) into final results (vectors $\vec{y}$).

Obviously, the performance and scalability depends strongly on the partitioning, matrix-to-processors mapping, and used matrix storage format. In this paper, the following simplifications were assumed:

- The regions are rectangular submatrices containing at least one nonzero element.
- Each processor holds exactly one region.

The following common matrix-processor mappings are discussed:

1. row-wise 1D block partitioning (static size),
2. row-wise 1D block partitioning (dynamic size),
3. checkerboard 2D block partitioning (static size),
4. checkerboard 2D block partitioning (dynamic size),
5. adaptive 2D block partitioning.

The new mappings were also described: 2D block partitioning based on k-d tree (version 1 and 2). These mappings have a huge impact on the performance of the parallel SpMV execution on massively parallel systems. To precise whether the mapping is good or not, we proposed four quality measures of a matrix-to-processors mapping:
CHAPTER 3. AUTHOR’S RELEVANT PAPERS

1. Good workload-balancing means that every processor will execute approximately the same number of instructions. For SpMV, it means that processors should contain approximately the same number of nonzero elements. This parameter \( q_1 \) represents "fairness" of distribution of elements.

2. It is important that every processor consumes approximately the same amount of memory for storing all required data. More exactly, memory requirement of every processor cannot exceed the some threshold otherwise long-lasting swapping in virtual memory occurs. This aspect is represented by parameter \( q_2 \).

3. Time of transformation. This represents time complexity of transformation algorithm for redistribution of data from input mapping to the target (given) mapping. In some situations, this parameter is not important because matrix is generated directly in the target format.

4. Time of SpMV. This represents time complexity of SpMV including all necessary communication and synchronization.

Unfortunately, it is hard to optimize all quality aspects of a mapping at the same time, because they are in contradiction. The lower and upper bounds of quality measures for different mappings were derived. Algorithms for redistribution were also shown. In the new mapping, the performance and the overhead of the required transformation are balanced. Figures 2, 3, and Table II show the values of the parameters \( q_1 \) and \( q_2 \) for testing matrices. We can conclude that:

- The mapping 6 is the best load-balancing strategy because it has the lowest values of \( q_1 \). The mappings 2, 5, and 7 also achieve very good results. This follows from theoretical assumptions, because these mapping are based on balancing of number of nonzero elements.

- The mapping 7 has the lowest memory requirements because of the lowest values of \( q_2 \). The results of other mapping are not very good. This follows from theoretical assumptions.

So, we can declare mapping 7 as a winner, because it is a trade-off between good load-balancing strategy and low memory requirements. From these results we can conclude that the SpMV algorithm using our new mapping is scalable for almost all matrices arising from various technical areas.
The study of impact of matrix-processor mapping on the parallel sparse matrix-vector multiplication

I. Šimeček, D. Langr, E. Srnec
Department of Computer Systems, Faculty of Information Technologies,
Czech Technical University in Prague, Prague, Czech Republic Email: xsimecek,langrd@fit.cvut.cz

Abstract—Sparse matrix-vector multiplication (shortly SpM×V) is one of the most common subroutines in the numerical linear algebra. The parallelization of this task looks easy and straightforward, but it is not optimal in general case. This paper discusses some matrix-processor mappings and their impact on parallel SpM×V execution on massively parallel systems. We try to balance the performance and the overhead of the required transformation. We also present algorithms for redistribution. We propose four quality measures and derive lower and upper bound for different mappings. Our SpM×V algorithms are scalable for almost all matrices arising from various technical areas.

Keywords—parallel execution; sparse matrix-vector multiplication; sparse matrix representation; matrix-processor mapping; scalability

I. INTRODUCTION AND TERMINOLOGY

We consider a matrix $A$ of order $n \times n$, $A = (a_{i,j})$. The number of its nonzero elements is denoted by $N$. The average number of nonzero elements per row is denoted by $\gamma$, thus $\gamma = N/n$. Matrix $A$ is considered sparse if the ratio of nonzero elements drops below some threshold. This paper is aimed at algorithms for SpM×V with very large sparse matrices (VLSMs). By VLSMs, we mean matrices that because of their size must be stored and processed by massively parallel computer systems (MPCs) with distributed memory architectures.

A. General notation

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 $\eta(B)$.
- Elements of vectors and matrices are of some floating-point datatype and $S_F$ denotes the size of this floating-point datatype.
- All indexes are of type integer and $S_I$ denotes the size of this datatype.

B. Banded matrices

Citing from [1]:

Definition I: 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 Compressed Sparse Row (CSR) format

The most common format for storing sparse matrices is the compressed sparse row (CSR) format (see [2], [3], [4], [2], [5]). In the following text, we assume that $\vec{x}$, $\vec{y}$ are vectors of size $n$. We consider $SpM \times V$ operation $\vec{y} \leftarrow A \vec{y}$. The matrix $A$ stored in the CSR format is represented by three linear arrays: values, addr, and ci. The array $values[1, \ldots, N]$ stores the nonzero elements of $A$, the array $addr[1, \ldots, n]$ contains indexes of initial nonzero elements of rows of $A$; if row $i$ does not contain any nonzero element then $addr[i] = addr[i + 1]$. The array $ci[1, \ldots, N]$ contains column indexes of nonzero elements of $A$. The space complexity of the structure of matrix $A$ in this format is

$$S_{CSR}(n, N) = N(S_I + S_F) + n \cdot S_I.$$ 

D. SpM×V for the CSR format

Consider a sparse matrix $A$ represented in CSR format. The representation contains linear arrays values, addr, and ci as they are defined in Section I-C and a vector $\vec{x}$ represented by dense array $x[1, \ldots, n]$. A standard sequential pseudocode for the sparse matrix-vector multiplication $\vec{y} \leftarrow A \vec{x}$ can be described by the Algorithm 1.

Algorithm 1 SpM×V for the CSR format

1: procedure $\spMV_{CSR}$(in $A$, $x$; out $y$)
2: Input: $A =$ matrix in the CSR format
3: Input: $x =$ array representing vector $\vec{x}$
4: Output: $y =$ array representing vector $\vec{y}$
5: 1: for $j \leftarrow 1$, $n$ do
6: 2: $\text{sum} \leftarrow 0$;
7: 3: for $i \leftarrow A.addr[j]$, $A.addr[j + 1] - 1$ do
8: 4: $\text{sum} + = A.values[i] \ast x[A.ci[i]]$;
9: 5: $y[j] \leftarrow \text{sum}$;
10: 6: return $y$;

Time complexity of $\spMV_{CSR}$ is $\Theta(n + N)$. Assuming $n \leq N$, we can simplify it to $\Theta(N)$. 

E. Parallel execution

- Let $P$ be the number of processors. The matrix $A$ is partitioned among $P$ processors $p_1, \ldots, p_P$ of a given MPCS.
- We assume that $P \ll n$.
- We assume that the architecture of the given distributed system uses the communication topology $G$, and that three basic communication operations are supported:
  - Send operation: its communication complexity is denoted by $SEND_G(x, y)$, where $x$ is a message size and $y$ is a message distance.
  - Broadcast operation: its communication complexity is denoted by $BC_G(x)$, where $x$ is a message size.
  - Parallel reduction operation: its communication complexity is denoted by $PR_G(x)$, where $x$ is a message size.

In the further text, we derive time complexities of algorithms using these general functions.

F. State-of-art

Other papers (for example [6], [7], [8], [9], [10]) analyse and evaluate the efficiency of $SpM \times V$ operation in situations when at least one of the following conditions is true:

- the number of processors ($P$) is small,
- nonzero elements are distributed in $A$ uniformly,
- authors focused only on minimization of the communication complexity,
- matrix $A$ is banded.

There is a large number of ad-hoc solutions for very specific kind of matrices (for example [11]). Sometimes, $n$ is so large that a single processor cannot hold even the whole vector $\vec{x}$. As far as we know, there is no study of optimal mapping of matrix to processors for $SpM \times V$ in general case.

II. PARALLELIZATION OF $SpM \times V$

A. The main idea

In general, there is only one way how to parallelize $SpM \times V$:

1) Partitioning of matrix $A$ into $n_R$ disjoint nonempty regions denoted by $R_i$.
2) Every processor executes multiplication of region by (part of) vector $\vec{x}$ in parallel. More exactly, every processor $p_i$ proceeds $SpM \times V$ with region $R_i$ and part of vector $\vec{x}$ and computes temporal result (vector $\vec{t}$).
3) Reduce temporal results (vectors $\vec{t}$) into final results (vectors $\vec{y}$).

Obviously, the performance and scalability depends strongly on the partitioning, matrix-to-processors mapping, and used matrix storage format.

B. Our assumptions

1) Partitioning matrix into regions: We can divide matrix $A$ into disjoint nonempty regions so that each processor holds all data accessed during the partial $SpM \times V$ within each region (step 2). In this paper, we assume for the sake of simplicity that the regions $R_i$ are rectangular submatrices with $r_i$ rows and $c_i$ columns containing at least one nonzero element. We also assume that each processor holds exactly one region.

2) Choosing a suitable storage format for the regions: The second important decision is to choose an efficient format for data inside each region. Each region presents one submatrix from the entire matrix. Therefore, we can use common sparse matrix formats for storing nonzero elements inside them. In this paper, we assume that every region in stored in the CSR format.

C. Mappings of sparse matrices

- Every processor $p_i$ contains a region $R_i$ of size $r_i \times c_i$.
- The number of different values of $x$-coordinate in $R_i$ is denoted by $Xrange(R_i)$, so $Xrange(R_i) = |K|$, where $K = \{ x; (x, y) \in R_i \}$. This value is difficult to compute, so we can approximate it by the interval of $x$-coordinates in $R_i$. Hence,

$$Xrange(R_i) \approx |\{g_i, \ldots, h_i\}| = h_i - g_i + 1,$$

where $g_i = \min \{ x; (x, y) \in R_i \}$, $h_i = \max \{ x; (x, y) \in R_i \}$.

- For practical usage, we approximate it further and assume that $Xrange(R_i) = c_i$.
- Definition of $Yrange(R_i)$ for $y$-coordinates is very similar, so we assume that $Yrange(R_i) = r_i$.
- The minimal number of nonzero elements in one region is denoted by $\eta = \min_{i} \eta(R_i)$.
- The maximal number of nonzero elements in one region is denoted by $\beta = \max_{i} \eta(R_i)$.

D. Requirements for mappings

We consider a simple requirement for $SpM \times V$: it should be an efficient and scalable due to good matrix-to-processors mapping. To precise whether the mapping is good or not, we propose four quality measures of a matrix-to-processors mapping:

1) Good workload-balancing aspect is denoted by parameter $q_1$. Good workload-balancing means that every processor will execute approximately the same number of instructions. For $SpM \times V$, it means that processors should contain approximately the same number of nonzero elements. This parameter represents “fairness” of distribution of elements. To quantify this aspect, we define

$$q_1 = \frac{\beta - \alpha}{N/P}.$$ 

The optimal value is zero, so low values of $q_1$ means a good load-balancing.

2) It is important that every processor consumes approximately the same amount of memory for storing all required data. More exactly, memory requirement of every processor cannot exceed the some threshold otherwise long-lasting swapping in virtual memory occurs. These
memory requirements are defined by the corresponding matrix-to-processors mapping and for each processor it consists of following parts:

- **nonzero elements in region**:
  \[ S_F \cdot \eta(R_i) , \]
- **additional memory requirement of CSR format**:
  \[ S_I \cdot \eta(R_i) + S_I \cdot r_i , \]
- **corresponding part of vector \( \vec{x} \)**:
  \[ S_F \cdot Xrange(R_i) = S_F \cdot c_i , \]
- **corresponding part of vector \( \vec{f} \)**:
  \[ S_F \cdot Yrange(R_i) = S_F \cdot r_i . \]

So, the requirements for one region \( R_i \) processing, denoted by \( M(R_i) \), can be estimated by

\[
M(R_i) = (S_F + S_I)\eta(R_i) + (S_F + S_I)r_i + S_F \cdot c_i .
\]

In an optimal case, processors hold only disjoint parts of nonzero elements and vectors, so

\[
M_{opt} = \frac{(S_F + S_I)N + (2S_F + S_I)n}{P} .
\]

To estimate how a given mapping is close to this optimal data distribution, we define parameter \( q_2 \) as

\[
q_2 = \frac{\max_i M(R_i)}{M_{opt}} .
\]

Optimal value of this parameter is equal to one. If the value of \( q_2 \) is close to 1, then the mapping is space efficient and vice versa.

3) **Time of transformation.** This represents time complexity (denoted by \( q_3 \)) of transformation algorithm for redistribution of data from input mapping to the target (given) mapping. There are two basic steps:

- **Step 1:** Computing of target locations for nonzero elements. The complexity of this step depends on the partitioning of the nonzero elements among processors in input mapping and on target mapping. For data-insensitive mapping, this step is skipped.

- **Step 2:** Redistribution of nonzero elements: nonzero elements are redistributed between processors according to newly found locations. The complexity depends on the input mapping and on the computed locations in the target mapping. In worst-case scenario, processor \( p_i \) sends all of its nonzero elements (\( \beta \)) to \( p_j \), and the maximal distance between \( p_i \) and \( p_j \) is \( P - 1 \). So, the maximal complexity of this step (all-to-all scatter) is \( O(SEND_C(\beta, P - 1)) \).

In some situations, this parameter is not important because matrix is generated directly in the target format.

4) **Time of SpM\( \times V \).** This represents time complexity of SpM\( \times V \) (denoted by \( q_4 \)) including all necessary communication and synchronization. For some mappings (1 and 2 in Section II-E), there is no need for reduction for final results. In some mappings (3, 4, and 5 in Section II-E), the final reduction can be done among independent groups, it can strongly reduce message size. This time complexity is also influenced by execution of partial SpM\( \times V \). Since number of nonzero elements is not balanced, only the execution time of processor with the maximal number of nonzero elements with time complexity \( \Theta(\beta) \) is taken into account.

Unfortunately, it is hard to optimize all quality aspects of a mapping at the same time, because they are in contradiction.

E. Possible mappings

We assume following target mappings of nonzero elements of matrix \( A \):

- **Mapping 1** = Row-wise 1D block partitioning (static size): The simplest mapping is row-wise 1D block partitioning (see [6]). Matrix \( A \) is divided into \( P \) row blocks of the same size (see Figure 1 a).

- **Mapping 2** = Row-wise 1D block partitioning (dynamic size): This is improved version of row-wise 1D block partitioning, because we relax the assumption of the same size. In this mapping, matrix \( A \) is divided into \( P \) row blocks, each of them contains approximately the same number of nonzero elements (see Figure 1 b).

- **Mapping 3** = Checkerboard 2D block partitioning (static size): Matrix \( A \) is divided into \( P \) blocks with the same size \( \sqrt{\frac{N}{P}} \times \sqrt{\frac{N}{P}} \) (see Figure 1 c).

- **Mapping 4** = Checkerboard 2D block partitioning (dynamic size): This is improved version of row-wise 1D block partitioning, because we relax the assumption of the same size. In this mapping, matrix \( A \) is divided into \( \sqrt{P} \) row blocks \( RB_i \), each of them contains approximately the same number of nonzero elements. Then, matrix \( A \) is divided into \( \sqrt{P} \) column blocks \( CB_j \), each of them contains approximately the same number of nonzero elements. The intersection of \( RB_i \) and \( CB_j \) is assigned to processor \( p_{i\cdot j} \) (see Figure 1 d).

- **Mapping 5** = Adaptive 2D block partitioning: In this mapping, matrix \( A \) is divided into \( \sqrt{P} \) row blocks \( RB_i \), each of them contains approximately the same number of nonzero elements. Then, each of these row blocks is divided (independently on other row blocks) into \( \sqrt{P} \) blocks (see Figure 1 e).

- **Mapping 6** = 2D block partitioning based on k-d tree (version 1): This mapping is based on 2D k-d tree. Each node in this tree represents a submatrix. The binary tree is used to describe a partition of the sparse matrix into submatrices. The decomposition is done in different directions: firstly horizontally, then vertically and so on. In other words, nodes with odd depth represent a partition of the submatrix into two halves along \( x \)-axis (left/right), nodes with even depth represent a partition of the submatrix into two halves along \( y \)-axis (upper/lower).
Fig. 1. An example of partition of the sparse matrix: a) Row-wise 1D block partitioning (static size), b) Row-wise 1D block partitioning (dynamic size), c) Checkerboard 2D block partitioning (static size), d) Checkerboard 2D block partitioning (dynamic size), e) Adaptive 2D block partitioning, f) 2D block partitioning based on k-d tree.

(see Figure 1 f). In this version, each of parts contains approximately the same number of nonzero elements.

- Mapping 7 = 2D block partitioning based on k-d tree (version 2): Similar to previous one (see Figure 1 f), but the division criteria is the same amount of memory requirements for each processor. To our best knowledge, this type of mapping has not been described in literature.

III. Bounds of Quality Measures of Proposed Matrix-to-Processors Mappings

Our first task is to derive upper bounds of parameters $q_1$ and $q_2$ for selected mappings. The second task is to find a scalable parallel algorithm to convert the given input mapping of nonzero elements into the given (target) mapping and derive time complexity of them (parameter $q_3$). The third task is to find a parallel algorithm of $SpM \times V$ in given (target) mapping and derive time complexity of them (parameter $q_4$).

A. Mapping 1: Row-wise 1D block partitioning (static size)

Parameter $q_1$: The number of nonzero elements is in $(\alpha, \beta)$, $\alpha = 0$ and $\beta = \min(n^2/P, N)$ because one region can be either full of zeroes or full of nonzero elements.

So, the parameter

$$q_1 = \frac{\min(n^2/P, N)}{N/P} \approx \min(n^2/N, P)$$

(1)

It is obvious that this parameter can be very high that indicates low quality of load-balancing.

Parameter $q_2$:

$$\forall c_i = n \quad \forall r_i = n/P$$

$$M(R_i) = (S_F + S_t)\beta + (S_F + S_t) \cdot n/P + S_F \cdot n$$

$$\approx (S_F + S_t) \min(n^2/P, N) + S_F \cdot n$$

(2)

$$q_2 = \frac{(S_F + S_t) \min(n, \gamma P) + S_F \cdot P}{(S_F + S_t)\gamma + (2S_F + S_t)}$$

(3)

If $1 \ll \gamma$ and $\gamma P < n$ then it can be simplified to $q_2 \approx P$, so with increasing number of processors is also this parameter linearly growing. It means that the memory requirements per processor do not scale with the number of processors!

Parameter $q_3$: This mapping is data-insensitive, so no computation of target locations is needed and time for transformation to this mapping is equal to the time of redistribution of nonzero elements.

Parameter $q_4$: In this mapping, the parallel $SpM \times V$ algorithm is straightforward: computed parts of vector $\vec{t}$ are equal to parts of resulting vector $\vec{y}$. So, the complexity is $O(\beta)$.
B. Mapping 2: Row-wise 1D block partitioning (variable size)

Parameter \( q_1 \): The number of nonzero elements in this mapping is relatively balanced. The greatest difference occurs when one row full of nonzero elements is followed by the row with only one nonzero element and vice versa, so \( \alpha = N/P - n + 1 \) and \( \beta = N/P + n - 1 \)

So, the parameter
\[
q_1 \approx \frac{2 \cdot n}{N/P} = \frac{2 \cdot P}{\gamma}
\]

It is obvious that this parameter can be very high for (small values of \( \gamma \)) which indicates low quality of load-balancing.

Parameter \( q_2 \): In this mapping, the parallel \( SpM \times V \) algorithm is straightforward: computed parts of vector \( i \) are equal to parts of resulting vector \( \tilde{y} \).

C. Mapping 3: Checkerboard 2D block partitioning (static size)

Parameter \( q_1 \): The derivation is similar to III-A, so \( \alpha = 0 \) and \( \beta = \min(n^2/P, N) \)

\[
q_1 = \min(n^2/N, P)
\]

As it was mentioned above, very high value of this parameter indicates low quality of load-balancing.

Parameter \( q_2 \):

\[
\forall c_i = n/\sqrt{P} \quad \forall r_i = n/\sqrt{P}
\]

This value remains relatively small if \( \gamma \approx P \).

Parameter \( q_3 \): The Algorithm 2 is a simple general transformation algorithm to change input matrix mapping to the requested one and can be further optimized according to different input mappings. It computes the values of array \( \text{start}_\text{row} \). In this array, the value \( \text{start}_\text{row}[i] \) is the starting row for row block assigned to processor \( p_i \).

Let us estimate the time complexity of this approach:

- Algorithm 2 line 3: For the balanced input mapping, the complexity of the local computation is \( O(N/P) \), otherwise \( O(\beta) \).
- Algorithm 2 line 4: parallel reduction \( PR_G(n) \).
- Algorithm 2 line 6: local computation: \( O(\log n) \).
- Algorithm 2 line 8: broadcast \( BC_G(1) \).

Algorithm 2: A transformation algorithm to convert input mapping to 1D block partitioning

```plaintext
procedure MapToRowBlocks()
Output: \text{start}_\text{row}[] = \text{y-}  \text{position of starting row of row blocks}
1: for \( i \leftarrow 1, P \) do in parallel
2: for \( j \leftarrow 1, n \) do
3: compute values in array \( a[1..n] \), such that \( a[i] = \eta(1, 1, A, n, i) \);
4: parallel reduction (sum) of each element array \( a \);
5: for \( i \leftarrow 1, P \) do in parallel
6: find minimal \( y \) inside \( (1, \ldots, n) \) such that \( \eta(1, 1, A, n, y) \geq (i \ast N)/P \);
7: \text{start}_\text{row}[i] = y;
8: broadcast \text{start}_\text{row}[i];
```

Parameter \( q_4 \) in this mapping, the parallel \( SpM \times V \) algorithm is straightforward: computed parts of vector \( i \) are equal to parts of resulting vector \( \tilde{y} \).

D. Mapping 4: Checkerboard 2D block partitioning (variable size)

Parameter \( q_1 \): The numbers of nonzero elements in this mapping for row-superblocks or column-superblocks are relatively balanced, but the numbers in intersections are not balanced in general. So, the derivation is similar to III-C, it means \( \alpha = 0 \) and \( \beta = \min(n^2/P, N) \)

and the parameter
\[
q_1 = \min(n^2/N, P)
\]

This mapping may look as a good solution but for some types of matrices (for example banded matrix) parameter \( q_1 \) can be very high that indicates low quality of load-balancing.
Parameter $q_2$: The derivation is similar to III-B
\[
\max c_i = n - \gamma, \quad \max r_i = n - \gamma
\]
\[
M(R_i) = (S_F + S_I)\beta + (2S_F + S_I)(n - \gamma)
\]
\[
q_2 \approx \frac{(S_F + S_I)\min(n, \gamma P) + (2S_F + S_I)P}{(S_F + S_I)\gamma + (2S_F + S_I)}
\]

For some types of matrices (if $\gamma \gg P$), this parameter is close to one. So, memory requirements are optimal.

Parameter $q_2$: Transformation algorithm is similar as for III-B, we find locations of starting row for each row block. Then, we find locations of starting column for each column block therefore complexity is two times higher than for III-B.

Parameter $q_4$: In this mapping similar as for III-C, the parallel $SpM \times V$ algorithm consists of partial $SpM \times V$ and then all temporary results (computed parts of vector $\tilde{y}$) are collected by parallel reduction into final result (vector $\tilde{y}$).

E. Mapping 5: Adaptive 2D block partitioning

Parameter $q_2$: Derivation of $\alpha$ and $\beta$ is similar to III-B. So, $\alpha \approx N/P - 2n$ and $\beta \approx N/P + 2n$.
\[
q_1 \approx \frac{4n}{N/P} \approx \frac{4 \cdot P}{\gamma}
\]

It is easy to see that this parameter can be very high (for small values of $\gamma$) that indicates low quality of load-balancing.

Parameter $q_2$:
\[
\max(c_i) = n - \gamma \quad \max(r_i) = n - \gamma
\]
\[
M(R_i) = (S_F + S_I)\beta + (S_F +S_I)(n - \gamma) + S_F \cdot (n - \gamma) 
\approx (S_F + S_I)(N/P + 2n) + (2S_F + S_I)n
\]
\[
q_2 \approx \frac{(S_F + S_I)(\gamma + 2 \cdot P) + (2S_F + S_I)P}{(S_F + S_I)\gamma + (2S_F + S_I)}
\]

Some types of matrices (if $\gamma \gg P$) have this parameter close to one. Again, memory requirements are optimal.

Parameter $q_3$: Transformation algorithm is similar to III-B, we find locations of starting row for each row block. Then, we find locations of starting column for each column block independently therefore complexity is approximately $\sqrt{P}$ higher than for III-B.

Parameter $q_4$: In this mapping, similarly as for III-C, the parallel $SpM \times V$ algorithm consists of partial $SpM \times V$ and then all temporary results (computed parts of vector $\tilde{y}$) are collected by parallel reduction into final result (vector $\tilde{y}$).

F. Mapping 6: 2D block partitioning based on k-d tree (version 1)

Parameter $q_1$: Derivation of $\alpha$ and $\beta$ is similar to III-B. So, $\alpha \approx N/P - n$ and $\beta \approx N/P + n$.
\[
q_1 \approx \frac{2n}{N/P} \approx \frac{2 \cdot P}{\gamma}
\]

It is easy to see that this parameter can be very high (for small values of $\gamma$) that indicates low quality of load-balancing.

Parameter $q_2$:
\[
\max(c_i) = n - \gamma \quad \max(r_i) = n - \gamma
\]
\[
M(R_i) = (S_F + S_I)\beta + (S_F + S_I)(n - \gamma) + S_F \cdot (n - \gamma) 
\approx (S_F + S_I)(N/P + n) + (2S_F + S_I)n
\]
\[
q_2 \approx \frac{(S_F + S_I)(\gamma + P) + (2S_F + S_I)P}{(S_F + S_I)\gamma + (2S_F + S_I)}
\]

For some types of matrices (if $\gamma \gg P$), this parameter is close to one. So, memory requirements are optimal.

Parameter $q_3$: The Algorithm 3 is a simple general algorithm to do that and can be optimize according to different input mapping.

This algorithm has the drawback that the number of processors ($P$) must be a power of 2.

To correct this drawbacks, we improve Algorithm 3 to Algorithm 4 (only new version of procedure $DIVIDEBYX$ is stated, new version of procedure $DIVIDEBYY$ is obvious). In this version, the number of processors are divided into halves of approximately same size, in the same ratio the nonzero elements are divided.

Result of Algorithms 3 or 4 is array $M$ such that $M[i]$ contains coordinates of rectangle (submatrix of matrix $A$) that is assigned to $p_i$.

Time complexity of Algorithm 4 is following: For each processor is called procedure $DIVIDEBYX$ (or $DIVIDEBYY$) once. The complexity of each call equal to complexity of local computation $O(N/P)$ and communication (distributed binary search) with complexity $(BC_G(x) + PR_G(x)) \log n$.

So, total complexity is
\[
N + P(BC_G(x) + PR_G(x)) \log n
\]

Parameter $q_4$: In this mapping, the parallel $SpM \times V$ algorithm consists of partial $SpM \times V$ and then all temporary results (computed parts of vector $\tilde{y}$) are collected by parallel reduction into final result (vector $\tilde{y}$).

G. Mapping 7: 2D block partitioning based on k-d tree (version 2)

Parameter $q_1$: Derivation of parameter $q_1$ is a little bit more complex than in previous mappings. Firstly, we derive that maximal amount of memory requirements ($\max_i M(R_i)$) for one processor is
\[
\frac{N(S_F + S_I)n(2S_F + S_I)}{P} + 2n(S_F + S_I)
\]
Algorithm 3 Transformation algorithm from input mapping to 2D block partitioning based on k-d format

**procedure** K-DTREE\(p_{\text{start}}, p_{\text{end}}, x_1, y_1, x_2, y_2\)

**Input:**
- \(p_{\text{start}}\) = the rank of first processor
- \(p_{\text{end}}\) = the rank of last processor
- \(x_1, y_1, x_2, y_2\) = coordinates of matrix

**Output:**
- \(M\) = mapping to k-d tree

1. DIVIDEBYX\((1, P, 1, 1, A, n, A, n, M)\);
2. return \(M\);
3. **procedure** DIVIDEBYX\((p_{\text{start}}, p_{\text{end}}, x_1, y_1, x_2, y_2, M)\)
4. if \(p_{\text{start}} == p_{\text{end}}\) then
5. \(M[p_{\text{start}}] = (x_1, y_1, x_2, y_2)\);
6. return \(M\);
7. find minimal \(x\) inside \((x_1, x_2)\) such that \(\eta(x_1, y_1, x, y_2) \geq \eta(x_1, y_1, x_2, y_2)\);
8. return \(p_{\text{half}} = (p_{\text{start}} + p_{\text{end}})/2\);
9. return \(M[p_{\text{half}}] = (x_1, y_1, x_2, y_2)\);
10. find minimal \(y\) inside \((y_1, y_2)\) such that \(\eta(x_1, y_1, x, y_2) \geq \eta(x_1, y_1, x_2, y_2)\);
11. return \(p_{\text{half}} = (p_{\text{start}} + p_{\text{end}})/2\);
12. return \(M[p_{\text{half}}] = (x_1, y_1, x_2, y_2)\);
13. return \(M[p_{\text{start}} - 1] = (x_1, y_1, x_2, y_2)\);
14. return \(M[p_{\text{end}} + 1] = (x_1, y_1, x_2, y_2)\);
15. return \(M[p_{\text{half}} + 1] = (x_1, y_1, x_2, y_2)\);
16. return \(M\);

Algorithm 4 Transformation algorithm to the parallel k-d format (improved)

**procedure** DIVIDEBYX2\((p_{\text{start}}, p_{\text{end}}, x_1, y_1, x_2, y_2, M)\)

1. if \((p_{\text{start}} == p_{\text{end}})\) then
2. \(M[p_{\text{start}}] = (x_1, y_1, x_2, y_2)\);
3. return \(M\);
4. \(p_{\text{half}} = (p_{\text{start}} + p_{\text{end}})/2\);
5. \(p_{\text{ratio}} = (p_{\text{end}} - p_{\text{start}} + 1)/(p_{\text{end}} - p_{\text{start}} + 1)\);
6. find minimal \(x\) inside \((x_1, x_2)\) such that \(\eta(x_1, y_1, x, y_2)/\eta(x_1, y_1, x_2, y_2) \geq \text{ratio}\);
7. return \(M[p_{\text{half}}] = (x_1, y_1, x, y_2)\);
8. return \(M[p_{\text{half}} + 1] = (x_1, y_1, x_2, y_2)\);
9. return \(M\);

Minimal amount is \(\frac{(S_F + S_I) + n(2S_F + S_I)}{P} - 2n(S_F + SI)\). From these bounds, we can derive that \(\alpha \approx \frac{N}{P} - 2n\) and \(\beta \approx \frac{N}{P} + \frac{2(S_F + S_I)}{P} + 2n\).

\[
q_1 \approx \frac{4n}{N/P} \approx \frac{4 \cdot P}{\gamma} \quad (19)
\]

It is easy to see that this parameter can be very high (for small values of \(\gamma\)) that indicates low quality of load-balancing.

Parameter \(q_2\):

\[
\max(c_\ell) = n - \gamma \quad \max(r_\ell) = n - \gamma
\]

\[
q_2 \approx 1 + \frac{2P(S_F + S_I)}{(S_F + S_I)\gamma + (2S_F + S_I)} \quad (20)
\]

For some types of matrices (if \(\gamma \approx P\), this parameter is close to one. So, memory requirements are optimal.

Parameter \(q_3\): The Algorithm 3 can be used also for this mapping, so the complexity is the same as in III-F.

Parameter \(q_4\): In this mapping similarly as for III-F, the parallel \(SpM \times V\) algorithm consists of partial \(SpM \times V\) and then all temporary results (computed parts of vector \(\vec{f}\)) are collected by parallel reduction into final result (vector \(\vec{y}\)).

<table>
<thead>
<tr>
<th>Matrix</th>
<th>(n)</th>
<th>(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit</td>
<td>5.36 \cdot 10^4</td>
<td>5.95 \cdot 10^4</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>3.54 \cdot 10^4</td>
<td>5.62 \cdot 10^4</td>
</tr>
<tr>
<td>ldoor</td>
<td>9.52 \cdot 10^4</td>
<td>2.37 \cdot 10^7</td>
</tr>
<tr>
<td>TSOPF_RS_b2383</td>
<td>3.81 \cdot 10^4</td>
<td>1.62 \cdot 10^7</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>4.51 \cdot 10^4</td>
<td>1.45 \cdot 10^6</td>
</tr>
<tr>
<td>t2em</td>
<td>9.25 \cdot 10^4</td>
<td>4.59 \cdot 10^6</td>
</tr>
<tr>
<td>bma7at5_1</td>
<td>1.41 \cdot 10^6</td>
<td>3.74 \cdot 10^6</td>
</tr>
<tr>
<td>amazon0312</td>
<td>4.01 \cdot 10^6</td>
<td>3.20 \cdot 10^6</td>
</tr>
<tr>
<td>thread</td>
<td>2.97 \cdot 10^4</td>
<td>2.25 \cdot 10^6</td>
</tr>
<tr>
<td>gupta2</td>
<td>6.21 \cdot 10^4</td>
<td>2.16 \cdot 10^6</td>
</tr>
<tr>
<td>c=29</td>
<td>5.03 \cdot 10^4</td>
<td>2.44 \cdot 10^4</td>
</tr>
</tbody>
</table>

**Table I**

Characteristics of the testing matrices.

IV. Evaluation of the Results

**Testing configuration:** We have used the values \(S_F = 8\) and \(S_I = 4\) for testing.

**Testing matrices:** We have used 11 testing matrices from various application domains from the University of Florida Sparse Matrix Collection [12]. Table I shows the characteristics of the testing matrices.

**Evaluation of measured parameters:** Figures 2, 3, and Table II show the values of the parameters \(q_1\) and \(q_2\) for testing matrices. We can conclude that:

- The mapping 6 is the best load-balancing strategy because it has the lowest values of \(q_1\). The mappings 2, 5, and 7 also achieve very good results. This follows from theoretical assumptions, because these mapping are based on balancing of number of nonzero elements.
- The mapping 7 has the lowest memory requirements because of the lowest values of \(q_2\). The results of other mapping are not very good. This follows from theoretical assumptions.

So, we can declare mapping 7 as a winner, because it is a trade-off between good load-balancing strategy and low memory requirements.

Another important question is whether changing of input mapping to required mapping pay-offs? It is very difficult to answer this question in general, but there are some important notes:
The overhead of a redistribution of a matrix from one mapping to another one depends strongly on characteristics of matrix and on capabilities of used MPCS communication subsystem.

In often cases, the same matrix $A$ is multiplied with large number of different vectors, e.g., in iterative linear solvers. In this case, a large number of $SpM \times V$ can amortize the overhead of a redistribution.

The used mapping does not influence only the performance of parallel $SpM \times V$, but also the possibility of execution of concrete $SpM \times V$ on the given MPCS. If the memory requirements for at least one node exceeds some threshold then the processing of $SpM \times V$ is not possible (or execution time of $SpM \times V$ is unacceptable due to long-lasting page swapping).

V. CONCLUSIONS

We have presented the new general quality measures for parallel sparse matrix-vector multiplication. We have discussed some common matrix-processor mappings and describe also a new one. We also have described algorithms for the redistribution. We have derived the upper bounds of quality measures. From these results we can conclude that the $SpM \times V$ algorithm using our new mapping is scalable for almost all matrices arising from various technical areas. This conclusion was proved by measurements.

REFERENCES

3.6 RP6: Tree-based Space Efficient Formats for Storing the Structure of Sparse Matrices

In this paper (RP6), the ideas from previous papers [RP3, RP4] were extended. We proposed and evaluated new storage formats for sparse matrices that minimize the space complexity of information about matrix structure.

The first one is based on arithmetic coding (ACB format). The main drawback of the ACB format is its computational complexity. The representation of sparse matrix $A$ is encoded in time $\Theta(n^2)$. This is too much for sparse matrices with a constant number of nonzero elements per row.

The second presented format is based on binary tree format (MBT format). This format extends ideas of the standard binary tree format as follows:

- All nodes in the MBT are stored in one array. Since we can compute locations of all child nodes, we can omit pointers.
- Instead of pointers, each node of the MBT contains only 2 flags (i.e., 2 bits only) indicating whether given subtrees are nonempty.

Thus, the space complexity of every MBT node is only 2 bits.

The minimal size of the MBT format is

$$2 \cdot (N - 1 + \log_2(n^2/N)).$$

The maximal size of the MBT format is

$$\approx 2 \cdot N(1 + \log_2(n^2/N)).$$

For MBT and MQT format, even more space-efficient variants (CBT and CQT formats) were introduced. We compared the space complexity of common storage formats and our new formats and proved that the latter ones are considerably more space efficient.

Parallel algorithms for distributed memory environment are also mentioned (see Section 4). For parallel execution, we assumed that nonzero elements are stored using a distributed version of a common SSF. This initial distribution we called an input mapping. Main idea of parallelization of space-efficient formats follows. The proposed formats are generic, i.e.,
they may be applied to sparse matrices of any structure. When processing large sparse
matrices on a massively parallel computer system, every processor has its own part of
a matrix, which itself can be treated as a stand-alone matrix of a smaller size. Every
processor can apply one of the proposed formats to its own matrix independently. Hence,
the proposed formats can be utilized on massively parallel computer systems the very same
way as in sequential computations. Representation of a large sparse matrix consists of local
submatrices (each processor owns disjoint submatrix) and binary tree (or quadtree) that
defines partitioning of the matrix among processors. The ACB format is space optimal,
but only if the distribution of nonzero elements is random (i.e., without any locality in the
matrix). Due to this fact, we use this format as the reference format. Table 5.2 compares
the ratios of the matrix space complexities. Table 5.3 shows ratios of space complexities
of the four tree-based formats studied in this paper to the ACB format. From this table,
we can observe that the CBT format:

- has usually smaller space complexity than the ACB format. There was only one
  exception among the 11 testing matrices: (mouse_gne).
- has similar space complexity as the MQT or CQT formats.

We can conclude that the CBT format is very space efficient.
TREE-BASED SPACE EFFICIENT FORMATS FOR STORING THE STRUCTURE OF SPARSE MATRICES \(^\dagger\)

I. ŠIMEČEK\(^1\), D. LANGR\(^1\), AND P. TVRDÍK\(^1\)

Abstract. Sparse storage formats describe a way how sparse matrices are stored in a computer memory. Extensive research has been conducted about these formats in the context of performance optimization of the sparse matrix-vector multiplication algorithms, but memory efficient formats for storing sparse matrices are still under development, since the commonly used storage formats (like COO or CSR) are not sufficient. In this paper, we propose and evaluate new storage formats for sparse matrices that minimize the space complexity of information about matrix structure. The first one is based on arithmetic coding and the second one is based on binary tree format. We compare the space complexity of common storage formats and our new formats and prove that the latter are considerably more space efficient.

Key words. sparse matrix representation; parallel execution; space efficiency; arithmetical-coding-based format; minimal binary tree format; minimal quadtree format;

AMS subject classifications. 68M14, 68W10, 68P05, 68P20, 94A17

1. Introduction. The paper investigates memory-efficient storage formats for very large sparse matrices (LSMs). By LSMs, we mean matrices that due to their sizes must be stored and processed by massively parallel computer systems (MPCs) with distributed memory architecture consisting of tens or hundreds of thousands of processor cores.

Within our previous work |9, 12, 11, 8, 7|, we have addressed weaknesses of previously developed solutions for space-efficient formats for storing of large sparse matrices. The space complexity of the representation of sparse matrices depends strongly on the used matrix storage format. A matrix of order \(n\) is considered to be sparse if it contains much less nonzero elements than \(n^2\). Some alternative definitions of sparse matrix can be found in |22|. In practice, a matrix is considered sparse if the ratio of nonzero elements drops below some threshold. Our research addresses computations with LSMs satisfying at least one of the following conditions:

1. The LSM is used repeatedly and the computation of its elements is slow and it takes more time than its later reading from a file system.
2. Construction of a LSM is memory-intensive. It needs significant amount of memory for auxiliary data structures, typically of the same order of magnitude as the amount of memory required for storing the LSM itself.
3. A solver requires the LSM in another format than is produced by a matrix generator and the conversion between these formats cannot be performed effectively on-the-fly.
4. Computational tasks with LSMs need check-pointing and recovery from failures of the MPCs. We assume that a distributed-memory parallel computation with a LSM needs longer time. To avoid recomputations in case of a system failure, we need to save a state of these long-run processes to allow fast recovery. This is especially important nowadays (and will be more in the future) when MPCs consist of tens or hundreds of thousands of processor cores.

\(^\dagger\)This research has been supported by GACR grant No. P202/12/2011.

\(^\dagger\)Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00, Praha, Czech Republic. E-mail: mailto:xsimecek.langr01@fit.cvut.cz

1
If at least one of these conditions is met, we might need to store LSMs into a file system. And since the file system access is of orders of magnitude slower compared to the memory access, we want to store matrices in a way that minimizes their memory requirements.

In this paper, we focus only on the compression of the information describing the structure of LSMs (i.e., the locations of nonzero elements). The values of the nonzero elements are unchanged, because their compression depends strongly on the application. For some application areas, the values of nonzero elements are implicit and only the information about the structure of a LSM is stored (for example, incident matrices of unweighed graphs). Alternatively, we can interleave computations with reading of nonzero elements. For example, we can divide the process of a sparse matrix factorization into these steps:

1. read the matrix structure,
2. do in parallel: perform the symbolic factorization and read the values of nonzero elements of the matrix,
3. perform the numeric factorization.

This paper is an extended version of our previous results [9]. We present updated versions of the algorithms and derivation of lower and upper bounds of space and computational complexity. We also provide more detailed analysis of the computational space, and communication complexities of parallel implementation of conversion to the new MBT format.

1.1. Terminology and notation. We consider a LSM $A$ of order $n$. The number of its nonzero elements is denoted by $N$, the average number of nonzero elements per rows is $N/n$ and it is denoted by $\text{avg}_{\text{per}}_{\text{row}}$.

- We assume that $1 \ll N \ll M = n^2$.
- The pattern of nonzero elements in $A$ is unknown or random.
- Indexes of all vectors and matrices start from zero.
- The number of nonzero elements in submatrix $B$ of matrix $A$ is denoted by $\text{nnz}(B)$.
- Let $P$ be the number of processors. The matrix $A$ is partitioned among $P$ processors $p_1, \ldots, p_P$ of a given massive parallel computer system (MPCS).
- The MPCS uses some variant of parallel I/O that allows to read/write a separate file for each process independently. Parallel I/O is a bottleneck of typical MPCS. Therefore we require that the new format should be space-efficient, in order to keep resulting file sizes as low as possible.
- We assume that nonzero elements are stored using a distributed version of a common sparse storage format (SSF). This initial distribution we called an input mapping.

This work is inspired by some real applications, for example ab initio calculations of medium-mass atomic nuclei (for future details see [1, 2]).

1.2. Representing indexes in binary codes. Let us have an array of $\xi$ elements indexed from 0 to $\xi - 1$. The minimum number of bits of an unsigned indexing data type is

$$S_{\text{MIN}}(\xi) = \left\lceil \log_2 \xi \right\rceil.$$
The value $S^{\text{MIN}}$ is the minimum number of bits, but it is usually padded to whole bytes ($S^{\text{BYTE}}$ bits)

$$S^{\text{BYTE}}(\xi) = 8 \cdot \left\lceil S^{\text{MIN}}(\xi)/8 \right\rceil,$$

or it is padded to the nearest power of 2 bytes ($S^{\text{POW}}$ bits)

$$S^{\text{POW}}(\xi) = 2^n, \quad \text{where } n = \left\lceil \log_2 S^{\text{MIN}}(\xi) \right\rceil.$$

When we describe a matrix storage format, we use simply $S(\xi)$ instead of $S^{\text{MIN}}(\xi)$.

### 2. State-of-the-art.

#### 2.1. The Coordinate (COO) Format.

The coordinate (COO) format is the simplest SSF (see [19, 3]). The matrix $A$ is represented by three linear arrays $\text{values}$, $\text{xpos}$, and $\text{ypos}$ (see Fig. 2.1 (b)). The array $\text{values}[1, \ldots, N]$ stores the nonzero values of $A$, arrays $\text{xpos}[1, \ldots, N]$ and $\text{ypos}[1, \ldots, N]$ contain column and row indexes, respectively, of these nonzero values. The space complexity of the structure of matrix $A$ (the size of the array $\text{values}$ is not counted) of this format is

$$S_{\text{COO}}(n, N) = 2 \cdot N \cdot S(n). \quad (2.1)$$

![Sparse Matrix and COO Format](image)

(a) An example of the sparse matrix

(b) Representation of this matrix in the COO format

#### 2.2. The Compressed Sparse Row (CSR) Format.

The most common SSF is the compressed sparse row (CSR) format (see [19, 3] for details). The matrix $A$ stored in the CSR format is represented by three linear arrays $\text{values}$, $\text{addr}$, and $\text{ci}$ (see Fig. 2.2 (b)). The array $\text{values}[1, \ldots, N]$ stores the nonzero elements of $A$, the array $\text{addr}[1, \ldots, n + 1]$ contains indexes of initial nonzero elements of rows of $A$; if row $i$ does not contain any nonzero element, then $\text{addr}[i] = \text{addr}[i + 1]$. It is obvious that $\text{addr}[1] = 1$ and $\text{addr}[n + 1] = N$. The array $\text{ci}[1, \ldots, N]$ contains column indexes of nonzero elements of $A$. Hence, the first nonzero element of the row $j$ is stored at index $\text{addr}[j]$ in array $\text{values}$. The space complexity of the structure of matrix $A$ (array $\text{values}$ is not counted) in this format is

$$S_{\text{CSR}}(n, N) = N \cdot S(n) + n \cdot S(N). \quad (2.2)$$

#### 2.3. Register blocking formats.

Widely-used SSFs are easy to understand. However, sparse operations (like matrix-vector or matrix-matrix multiplication) using these formats are slow (mainly due to indirect addressing). Sparse matrices often contain dense submatrices (blocks), so various blocking SSFs were designed to accelerate matrix operations. Compared to the CSR format, the aim of these formats (like
2.4. Minimal quadtree (MQT) format. The Quadtree (QT) is a tree data structure in which all inner nodes have exactly four child nodes. Since our aim is to minimize the space complexity of QT-based formats, in [7] we proposed a new QT format called minimal quadtree (MQT) format. Instead of pointers, each node of the MQT contains only 4 flags (i.e., 4 bits only) indicating whether given subquadtrees are nonempty.

2.5. Other state-of-art SSFs. There are several other SSFs specialized for given areas (e.g., compression of text, picture or video). They can be used for compression of sparse matrices, but none of them satisfies all these four requirements:

1. non-lossy compression,
2. possibility of massively parallel execution,
3. space efficiency (high compression rate),
4. high speed compression/decompression

Only few research results have been published about SSFs in the context of minimization of the required memory, which is the optimization criterion for a file I/O of LSMs. Some recent research of hierarchical blocking SSFs, though primarily aimed at optimization of matrix-vector multiplication, also addresses optimization of memory requirements [13, 14, 15]. We have published several papers about space-efficient SSFs suitable for storing sparse matrices [8, 11, 7].

3. Our new space-efficient formats.

3.1. The arithmetical-coding-based (ACB) format. Matrix $A$ can be represented by a bit vector $B$ of size $M$ in which $N$ bits are set to 1 and $M - N$ bits are set to 0. The probability $p_0$ that a given bit in $B$ is equal to 0 is $\frac{M-N}{M}$. In the arithmetical coding (see [21]), we can encode this information using $-\log_2 p_0$ bits. The probability $p_1$ that a given bit in $B$ is equal to 1 is $\frac{N}{M}$. In the arithmetical coding, we can encode this information using $-\log_2 p_1$ bits. Since we assume a random distribution of nonzero elements, the vector $B$ is considered to be an order-0 source (each bit is selected independently on other bits). The total number of bits to encode vector $B$ is equal to the value of binary entropy of vector $B$. This value is

$$S(B) = -M \cdot (p_0 \log_2 p_0 + p_1 \log_2 p_1)$$
Since this expression is hard to compare with other formats, the approximation of the binary entropy of vector $B$ follows:

$$S_{ACB}(n, N) = -M \cdot \left( \frac{M - N}{M} \log_2 \frac{M - N}{M} + \frac{N}{M} \log_2 \frac{N}{M} \right)$$

$$= -(M - N) \log_2 \frac{M - N}{M} - N \log_2 \frac{N}{M}$$

$$= -(M - N) \log_2 (M - N) + M \log_2 M - N \log_2 N.$$

Since we assume that $N \ll M$, we can use the following approximation for very small $x$: $\ln(1 + x) \approx x$. This implies that $\ln(M - N) \approx \ln M - N/M$. The final approximation of the space complexity of the ACB format is then:

$$S_{ACB}(n, N) \approx \frac{N}{\ln 2} + N \log_2 M - \frac{N^2}{M \cdot \ln 2} - N \log_2 N$$

$$\approx N \cdot \left( \frac{1}{\ln 2} + \log_2 M - \log_2 N \right)$$

$$\approx N \cdot \left( \frac{1}{\ln 2} + 2 \cdot \log_2 n - \log_2 N \right).$$

The same space complexity (based on other assumptions) was derived in [8], but it serves only for comparison and no practical algorithm to achieve this space complexity was given. As far as we know, the ACB format has not been described in literature.

The idea of transforming of the matrix $A$'s structure to the ACB format is simple: create $n \times n$ bitmap (with $N$ 1’s) from matrix $A$’s structure. Then, compress this bitmap as a bitstream using the arithmetical coding. The representation of matrix $A$’s structure in the ACB format is given by the compressed bitstream.

A comparison to common SSFs is done in Sect. 5.2. A drawback of the ACB format is its computational complexity. Since each bit of vector $B$ is encoded in time $\Theta(1)$, the complete vector $B$ (representation of sparse matrix $A$) is encoded in time $\Theta(n^2)$. This is too much for sparse matrices with a constant number of nonzero elements per row (i.e., $N \in \Theta(n)$).
3.2. The minimal binary tree (MBT) format. The full binary tree (FBT) is a widely used data structure in which all inner nodes have exactly two child nodes. Binary trees especially those used for binary space partitioning can also be used for storing sparse matrices. The idea of binary space partitioning is not new, but as far as we know, the use of these formats for efficiently storing sparse matrices was not described in literature. In standard implementations, every node in a FBT is represented by a structure standard_BT_struct consisting of the following items:

- two pointers (left, right) to child nodes,
- (only for leaves) the value of a nonzero element.

If a FBT is used as a basis for SSF, it describes a partition of the sparse matrix into submatrices and each node in the FBT represents a submatrix. Equally as in k-d trees, see [18], the decomposition is performed in alternating directions: first horizontally, then vertically, and so on. In other words, nodes in an odd height represent a partition of the submatrix into two halves along the the x-axis (left/right), nodes in an even height represent a partition of the submatrix into two halves along the y-axis (upper/lower). From the viewpoint of space efficiency, a drawback of the standard FBT representation is the overhead caused by pointers left, right. It causes that the standard FBT-based SSF may have worse space complexity than the CSR format.

![Diagram of MBT with minimal number of nodes](image)

(a) Original sparse matrix \( A \)

(b) The MBT representation of matrix \( A \)

Fig. 3.2: The MBT with the minimal number of nodes.

To eliminate this drawback, we propose a new k-d-tree-based SSF. Each tree node represents again a submatrix, but we modify the standard representation of the FBT and we call this data structure the minimal binary tree (MBT) format. The idea is very similar to that in the MQT format:

- All nodes of a MBT are stored in one array (or stream). Since the size of the input matrix is given, we can compute locations of all child nodes, we can omit pointers left, right.
- All nodes of a MBT contain only two flags (it means only two bits). Each
of them is set to 1 if the corresponding half of the submatrix (left/right or upper/lower) contains at least one nonzero element, otherwise it is set to 0. A comparison of the MBT format with other SSFs is done in Sect. 5.3. Let us describe algorithm 2 that generates an output bitstream representing the matrix in the MBT format from the standard CSR format.

![Fig. 3.3: MBT with the minimal number of nodes (the number of leaves is $N/2$).](image)

### 3.2.1. An example of a transformation to the MBT format.

Let us give an example of a construction of matrix representation in the MBT format implemented as a bitstream $S$. The corresponding binary tree is shown in Fig. 3.1.

$$S = MBT(A) = MBT\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$= "11" + MBT\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} +$$

$$+ MBT\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$= "11" + "01" + "11" + MBT\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} +$$

$$+ MBT\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + MBT\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} =$$

$$= "11" + "01" + "11" + "11" + "10" + "01" +$$

$$+ MBT("01") + MBT("10") + MBT("10") + MBT("01") =$$

$$= "11" + "01" + "11" + "11" + "10" +$$

$$+ "01" + "10" + "10" + "01"$$

So, if matrix $A$ is stored in the MBT format, 20 bits are needed for representing its structure.

### 3.2.2. Space complexity.

Let us assume a very small example of a sparse matrix with $n = 8$ and $N = 4$. For common storage formats, the space complexity is given by Eq. (2.1) or (2.2), so $S_{COO}(n,N) = 24[bits]$ and $S_{CSR}(n,N) = 28[bits]$. For
the MQT, the exact size of the output bitstream $S$ (it means the size of the MBT format) cannot be derived from these global parameters, because it depends on the exact locations of nonzero elements. It ranges from 14 to 38 bits (see Figs. 3.2 and 3.4). The derivation of the lower and upper bounds on the size of the MBT format in a general case is the following.

**Lower bound.** We consider the best case: the MBT with the minimal number of nodes, i.e., the number of leaves is equal to $N/2$ (see Fig. 3.3). It is obviously a generalized idea from Fig. 3.2. This matrix with 4 nonzero elements is represented by 7 MBT nodes = 14 bits. Output bitstream is "10 10 10 10 11 11 11".

- The height of the MBT on Fig. 3.2 is $h = h_1 + h_2 = 2 \log_2 n - 1$, where $h_2 = \log_2 N - 1$ and $h_1 = 2 \log_2 n - \log_2 N$. 

Fig. 3.4: The MBT with the maximal number of nodes.

Fig. 3.5: MBT with the maximal number of nodes (the number of leaves is $N/2$).
• All nodes with height \(< h_1\) (in upper \(h_1\) levels) contain exactly one 1 (they have one child node). The number of nodes in these levels is \(h_1\),

\[h_1 = \log_2(n^2/N)\.

• All nodes with height \(\geq h_1\) (in lower \(h_2\) levels) are full of 1’s (they have two child nodes). The number of nodes in these levels is equal to (this part is a full binary tree)

\[\sum_{i=n_1+1}^{\log_2 n-1} 2^{i-(h_1+1)} = N - 1.\]

So, the minimal size of the MBT format is

\[2 \cdot (N - 1 + \log_2(n^2/N)).\]

**Upper bound.** We consider the worst case: the quadtree with the maximal number of nodes, i.e., the number of leaves is equal to \(N/2\) (see Fig. 3.5). Again, it is a generalized idea from Fig. 3.4. This matrix with 4 nonzero elements is represented by 19 MBT nodes = 38 bits. The output bitstream is "11 11 11 10 10 01 01 10 01 01 01 10 01 10 01 01 01 10 01".

• The height of this tree is \(h = h_1 + h_2 = 2 \log_2 n - 1\), where \(h_1 = \log_2 N - 1\).

• All nodes with height \(< h_1\) (in upper \(h_1\) levels) are full of 1’s (they have two child nodes), \(h_1 = \log_2 N - 1\). The number of nodes in these levels is approximately

\[\sum_{i=n}^{n_1-1} 2^i = N - 1.\]

• All nodes with height \(\geq h_1\) (in lower \(h_2\) levels) contain exactly one 1 (they have one child node). The number of nodes in these levels is

\[N \cdot h_2 = N \cdot (2 \log_2 n - \log_2 N) = N \cdot \log_2(n^2/N).\]

So, the maximal size of the MBT format is

\[\approx 2 \cdot N (1 + \log_2(n^2/N)).\]

### 3.2.3. Time complexity of the transformation algorithm

Time complexity of the transformation algorithm 2 is relatively high, because for each node in the MBT, it uses algorithm 1 with complexity \(O(\log_2 n \cdot (y_2 - y_1 + 1))\). Fortunately, the average complexity is much lower (it depends on the value of the parameter \(\text{avg} \_ \text{per} \_ \text{row}\), distribution of nonzero elements, etc.).

We consider the worst case (similar ideas as for derivation of the space complexity in Sect. 3.2.2): the MBT with the maximal number of nodes, the number of leaves is equal to \(N\) (see Fig. 3.5). We assume that the time complexity of procedure \text{APPEXToBitstream} is \(\Theta(1)\). Procedure \text{INES}(A, x_1, y_1, x_2, y_2) is called for every node in the MBT in the output stream \(S\) two times.

• For nodes with height = \(h_1\): The number of these nodes is \(N\), the expression \((y_2 - y_1 + 1)\) is equal to \(1 + n/\sqrt{N}\). Time complexity of the transformation for all nodes with this depth is \(T_{h_1} = N \cdot (1 + n/\sqrt{N}) \cdot \log_2 \text{avg} \_ \text{per} \_ \text{row}\).
Algorithm 1 Procedure to test if the given submatrix is nonempty

1: procedure INES(A,x1,y1,x2,y2)
2: Input: A = an input submatrix in the CSR format
3: Input: x1,y1,x2,y2 = coordinates of the submatrix
4: Output: logical value indicating whether the given submatrix is nonempty
5: for y ← y1, y2 do
6:     low ← A.addr[y]; high ← A.addr[y + 1]
7:     i ← BINARY SEARCH(in array A.ci)
8:         ▷ within indexes from (low...high)
9:         ▷ to find a minimal i such that A.ci[i] ≥ x1
10:     if C.ci[i] ≤ x2 then
11:         return true
12: return false

Algorithm 2 Transformation algorithm to the MBT format

1: procedure Tr2MBT(A)
2: Input: A = the matrix for the transformation in CSR format
3: Output: S = the bitstream representing the input matrix in the MBT format
4: current ← ()
5: enqueue {1, 1, A.n, A.n, 0} into current
6: while current is not empty do
7:     dequeue {x1,y1,x2,y2,h} from current
8:         ▷ x1,y1,x2,y2 = coordinates of submatrix
9:     h = current BFS level, divide rows (h is odd) or columns
10: if h is even then
11:     mx ← x2; my ← (y1 + y2)/2
12:     lx ← x1; ly ← (y1 + y2)/2 + 1
13: else
14:     mx ← (x1 + x2)/2; my ← y2
15:     lx ← (x1 + x2)/2 + 1; ly ← y1
16: l1 ← INES(A, x1, y1, mx, my)
17: l2 ← INES(A, lx, ly, x2, y2)
18: APPEND_TO_BITSTREAM(S, l1)
19: APPEND_TO_BITSTREAM(S, l2)
20: if l1 = true then
21:     enqueue {x1, y1, mx, my, h + 1} into current
22: if l2 = true then
23:     enqueue {lx, ly, x2, y2, h + 1} into current
24: return S

- For nodes with height = h1 - 1: The number of nodes is N/2 and the expression (y2 - y1 + 1) is equal to 1 + 2n/√N. So, the total time complexity of the transformation for all nodes with depth ≤ h1 (in upper h1 levels) is \( T_{\text{upper}} \approx \sum_{i=1}^{h1} T_{h1}/2^{(i-h1)} = \Theta(N \cdot (1 + n/\sqrt{N}) \cdot \log_2 \text{avg}_\text{per}_\text{row}). \)
- For nodes with height > h1: The time complexity of the transformation for all these nodes (for the lower h2 levels) is \( T_{\text{lower}} \approx \sum_{i=h1+1}^{h} T_{h1}/2^{(i-h1)} = \Theta(N(1 + n/\sqrt{N}) \cdot \log_2 \text{avg}_\text{per}_\text{row}). \)
Algorithm 3 Transformation algorithm to the CBT format

1: procedure Tr2CBT(A)
2: Input: A = the matrix for the transformation in CSR format
3: Output: S = the bitstream representing the input matrix in the CBT format
4: current ← ()
5: enqueue {1, 1, A.n, A.n, 0} into current
6: while current is not empty do
7:   dequeue {x1, y1, x2, y2, h} from current
8:   > x1,y1,x2,y2 = coordinates of submatrix
9:   > h = current BFS level, divide rows (h is odd) or columns
10:   if h is even then
11:      mx ← x2;  my ← (y1 + y2)/2
12:     lx ← x1;  ly ← (y1 + y2)/2 + 1
13:   else
14:      mx ← (x1 + x2)/2;  my ← y2
15:     lx ← (x1 + x2)/2 + 1;  ly ← y1
16: l2 ← false
17: l1 ← INES(A, x1, y1, mx, my)
18: APPENDToBitstream(S, l1)
19: if l1 = true then
20:   l2 ← INES(A, lx, ly, x2, y2);
21:   APPENDToBitstream(S, l2)
22: if l1 = true then
23:   enqueue {x1, y1, mx, my, h + 1} into current
24: if l2 = true then
25:   enqueue {lx, ly, x2, y2, h + 1} into current
26: return S

So, the total time complexity of the transformation is

$$\Theta(N(1 + n/\sqrt{N}) \cdot \log_2 \text{avg - per - row}).$$

A very usual case is $N = \Theta(n)$, it means matrices with constant number of nonzero elements per row. For this case the time complexity is $\Theta(n^{3/2})$.

3.3. Compression of minimal formats. The MBT and MQT formats have minimal space complexity only if we assume fixed number of bits for each node (2 bits for MBT and 4 bits for MQT). We can relax this assumption to achieve more space efficient formats.

**Lemma 3.1.** Every node in the MBT (or in MQT) format (except for the root node for the zero matrix A) has got at least one bit equal to 1. The proof of Lemma 3.1 for the MBT format can be done by contradiction: if both bits in a MBT node X are zero, then this submatrix does not contain any nonzero element, so in the parent’s node of X the corresponding bit is set to 0 and node X is not included in the output stream and this is a contradiction with the initial assumption.

Similar proof can be done for the MQT format. Q.E.D.

Since we assume only nonempty matrices, the only allowed values in every MBT node are: 01, 10, and 11 (value 00 is not possible as a result of Lemma 3.1). So, if the first bit is 0, then the second bit must 1. This redundant information can be
excluded from the output stream. We call this case the hidden one. Based on this idea, we propose a new format, called compressed binary tree (CBT). There are two approaches to transform a LSM to the CBT format:

1. Transform the input matrix to the MBT format (it creates output stream $S$) and then remove from $S$ all hidden ones (all 4-tuples are read and transformed values according to Table 3.1 are written).

2. Modify Algorithm 2 to Algorithm 3 that directly create the CBT format.

Similarly in the MQT format, the value 0000 is not possible as a result of Lemma 3.1, so if the first three bits are 0, then the fourth bit must 1. Again, this redundant information can be excluded from the output stream, which allows us to construct another new compressed quadtree (CQT) format. It is obvious that the probability of hidden one is higher in the MBT format than in the MQT format. In the Table 3.1 is the comparison of code-words in MQT, CQT, MBT, and CBT formats. If we assume the same probabilities for all possible code-words, then the average size of code-word is 4 bits in MQT format, 3.93 bits for CQT format, 5.2 bits for MBT format, and 4.47 bits for CBT format. A comparison of these formats with real matrices is done in Sect. 5.3. A transformation algorithm from the CBT format is described by Algorithm 4. This algorithm transforms the input bitstream from the CBT format into the CSR format.

3.3.1. An example of a transformation to the CBT format. For an example, we used the same matrix as in the example in Sect. 3.2.1. Hidden ones are denoted by bold numbers.

$$S = \text{CBT}(A) = \text{CBT} \left( \begin{array}{cccc} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) =$$

$$= "11" + \text{CBT} \left( \begin{array}{cc} 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{array} \right) +$$

$$+ \text{CBT} \left( \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{array} \right) =$$

$$= "11" + "0" + "11" + \text{CBT} \left( \begin{array}{c} 0 \\ 1 \\ 1 \end{array} \right) +$$

$$+ \text{CBT} \left( \begin{array}{c} 1 \\ 0 \\ 0 \end{array} \right) + \text{CBT} \left( \begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right) =$$

$$= "11" + + "0" + "11" + "11" + "10" + "0" +$$

$$+ \text{CBT}("01") + \text{CBT}("10") + \text{CBT}("10") + \text{CBT}("01") =$$

$$= "11" + "0" + "11" + "11" + "10" +$$

$$+ "0" + "0" + "10" + "10" + "0"$$

For storing matrix $A$ in the CBT format only 16 bits are needed (instead of 20 bits in the MBT format).
<table>
<thead>
<tr>
<th>MQT</th>
<th>CQT</th>
<th>MBT</th>
<th>CBT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>N/A</td>
<td>00</td>
<td>N/A</td>
</tr>
<tr>
<td>0001</td>
<td>000</td>
<td>01 01</td>
<td>0 0</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
<td>01 10</td>
<td>0 10</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>01 11</td>
<td>0 11</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>10 01</td>
<td>10 0</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>11 01 01</td>
<td>11 0 0</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>11 01 10</td>
<td>11 0 10</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
<td>11 01 11</td>
<td>11 0 11</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>10 10</td>
<td>10 10</td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
<td>11 10 01</td>
<td>11 10 0</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>11 10 10</td>
<td>11 10 10</td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>11 10 11</td>
<td>11 10 11</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>10 11</td>
<td>10 11</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>11 11 01</td>
<td>11 11 0</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>11 11 10</td>
<td>11 11 10</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>11 11 11</td>
<td>11 11 11</td>
</tr>
</tbody>
</table>

Table 3.1: Transformation table between the MQT, CQT, MBT, and CBT formats. Hidden ones are marked by bold numbers.

**Algorithm 4** Transformation from the CBT format to the CSR format

1: procedure Tr2CSR(S)

Input: $S$ = the input bitstream of the input matrix in the CBT format

Output: $A$ = the output matrix in the CSR format

2: $A \leftarrow$ new empty matrix

3: enqueue \{1, 1, A.n, A.n, 0\} into current

4: while current is not empty do

5: 

6: dequeue \{x1, y1, x2, y2, h\} from current

7: \(\triangleright x_1, y_1, x_2, y_2 = \text{coordinates of submatrix}\)

8: if \(x_1 = x_2 \& y_1 = y_2\) then

9: \(X \leftarrow\) new nonzero element \((x_1, y_1)\)

10: add \(X\) to \(A\)

11: else

12: if \(h\) is even then

13: \(mx \leftarrow x_2; \ my \leftarrow (y_1 + y_2)/2\)

14: \(lx \leftarrow x_1; \ ly \leftarrow (y_1 + y_2)/2 + 1\)

15: else

16: \(mx \leftarrow (x_1 + x_2)/2; \ my \leftarrow y_2\)

17: \(lx \leftarrow (x_1 + x_2)/2 + 1; \ ly \leftarrow y_1\)

18: \(l1 \leftarrow \text{ReadOneBit}(S)\)

19: if \(l1 = \text{false}\) then

20: \(l2 \leftarrow \text{true}\) \(\triangleright\) hidden one

21: else

22: \(l2 \leftarrow \text{ReadOneBit}(S)\)

23: if \(l1 = \text{true}\) then

24: enqueue \(\{x_1, y_1, mx, my, h + 1\}\) into current

25: if \(l2 = \text{true}\) then

26: enqueue \(\{lx, ly, x_2, y_2, h + 1\}\) into current

27: return \(A\)
4. Parallel execution of transformation algorithm.

4.1. Main idea of parallelization. The proposed formats are generic, i.e., they may be applied to sparse matrices of any structure. When processing LSMs on a massively parallel computer system, every processor has its own part of a matrix, which itself can be treated as a stand-alone matrix of a smaller size. Every processor can apply one of the proposed formats to its own matrix independently. Hence, the proposed formats can be utilized on massively parallel computer systems the very same way as in sequential computations. This approach to parallelization is straightforward for the ACB format, but for SSF’s based on trees the parallelization is more complicated.

![Diagram showing parallelization of MBT or CBT transformation]

Fig. 4.1: The main idea of parallelization of MBT or CBT transformation.

![Diagram showing parallelization of MQT or CQT transformation]

Fig. 4.2: The main idea of parallelization of MQT or CQT transformation.

4.2. Parallel transformation of formats based on trees. Consider the MBT format. The proposed Algorithm 2 for transformation is sequential. Let us
now describe its master-slave parallelization. We assume two possible mappings how the matrix $A$ is distributed among processors:

- using general mapping,
- using row-wise 1D block mapping (see [17]). Matrix $A$ is divided into $P$ row blocks of variable size (recall $P$ is the number of processors in a MPCs). This mapping uses array $\text{start.row}$. In this array, the value $\text{start.row}[i]$ is the starting row for the row block assigned to processor $p_i$.

The only difference is that for general mapping we must use a general (unoptimized) procedure PARINES (see Algorithm 5) and for the row-wise 1D block mapping we can use an optimized procedure PARINES2 (see Algorithm 6).

1. In the first step (Algorithm 8 and codelines 1-10 in Algorithm 7), only processor $p_1$ (similarly to Algorithm 2) expands nonempty nodes of a binary tree by BFS until the number of nodes (denoted by $C$) is greater or equal to $k \cdot P$, where $k$ is the chosen constant (see Figs. 4.1 and 4.2, and Algorithm 7). The proper value of parameter $k$ is the trade-off between better work-load balancing (higher values of $k$) and smaller sequential part of transformation (lower values of $k$).

This binary tree defines partitioning of the matrix among processors. This tree (more exactly intervals of coordinates of nodes) is stored into array $B$ and also stored in the MB1 format in a special master file.

2. Initial communication (codeline 11 in Algorithm 7): Processor $p_1$ sends to all other processor blocks of array $B$ using one-to-all scatter operation. The block for $p_i$ starts at index $1 + (i - 1)[C/P]$ and ends at $\min(i[C/P], C)$. Each block contains intervals of coordinates of submatrices that are assigned to the given processor (see Algorithm 6).

3. Redistribution of nonzero elements (codeline 12 in Algorithm 7): nonzero elements of the matrix $A$ are redistributed between processors according to the resulting partitioning (array $B$).

4. Local transformation (codelines 13-20 in Algorithm 7): Every processor transforms assigned submatrices to the required MB1 format independently and stores them into a separate file.

Algorithm 5 Distributed procedure to test if the given submatrix is nonempty

1: \textbf{procedure} PARINES($A,x1,y1,x2,y2$)
Input: $A$ = the input matrix in the distributed CSR format
Input: $x1,y1,x2,y2$ = coordinates of submatrix
Output: logical value indicating whether the given submatrix is nonempty
2: \textbf{one-to-all broadcast} \{$x1,y1,x2,y2$\}
3: $output = false$
4: for $j \leftarrow y1,y2$ do
5: \hspace{1em} for $i \leftarrow A.AAddr[j], A.AAddr[j + 1] - 1$ do
6: \hspace{2em} if $(x1 \geq A.C[i][i] \leq x2)$ then
7: \hspace{2em} $output = true$
8: \hspace{2em} break
9: $po \leftarrow$ parallel reduction of $output$ using logical OR
10: \textbf{return} $po$
Algorithm 6 Distributed procedure to test if the given submatrix is nonempty

1: procedure PARINES2($A, x_1, y_1, x_2, y_2$)  
Input: $A =$ the input matrix in the distributed CSR format  
Input: $x_1, y_1, x_2, y_2 =$ coordinates of submatrix  
Output: logical value indicating whether the given submatrix is nonempty

2: construct $G'$
3: multicast $\{x_1, y_1, x_2, y_2\}$ in $G'$
4: $i \leftarrow$ index of the current processor
5: $si \leftarrow start\_row[i]$
6: $si1 \leftarrow start\_row[i + 1]$
7: if $(si > y_2) \text{ OR } (si1 \leq y_1)$ then
8: output = false
9: end
10: for $y \leftarrow \max(y_1, si), \min(y_2, si1 - 1)$ do
11: low $\leftarrow A.addr[y];$ high $\leftarrow A.addr[y + 1]$
12: $i \leftarrow \text{BINARY SEARCH(in array A.ci)}$
13: $\triangleright$ within indexes from (low . . . high)
14: $\triangleright$ to find a minimal $i$ such that $A.ci[i] \geq x_1$
15: if $C.ci[i] \leq x_2$ then
16: output = true
17: break
18: output = false
19: send predicate output to parallel reduction
20: $po \leftarrow$ parallel reduction of output using logical OR in $G'$
21: return $po$

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$n$</th>
<th>$N$</th>
<th>$\text{avg_per_row}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitN5</td>
<td>$5.56 \cdot 10^6$</td>
<td>$5.95 \cdot 10^7$</td>
<td>$1.93 \cdot 10^{-6}$</td>
</tr>
<tr>
<td>slpkkt120</td>
<td>$5.34 \cdot 10^6$</td>
<td>$5.02 \cdot 10^7$</td>
<td>$4.01 \cdot 10^{-6}$</td>
</tr>
<tr>
<td>ldoor</td>
<td>$9.52 \cdot 10^5$</td>
<td>$2.37 \cdot 10^7$</td>
<td>$2.60 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>TSQFF.RS.b2383</td>
<td>$3.81 \cdot 10^4$</td>
<td>$1.62 \cdot 10^7$</td>
<td>$1.10 \cdot 10^{-2}$</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>$4.51 \cdot 10^4$</td>
<td>$1.45 \cdot 10^7$</td>
<td>$7.10 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>t2em</td>
<td>$9.25 \cdot 10^5$</td>
<td>$4.59 \cdot 10^6$</td>
<td>$5.36 \cdot 10^{-6}$</td>
</tr>
<tr>
<td>bmw7st.1</td>
<td>$1.41 \cdot 10^6$</td>
<td>$3.74 \cdot 10^6$</td>
<td>$1.88 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>amazon0312</td>
<td>$4.01 \cdot 10^4$</td>
<td>$3.20 \cdot 10^6$</td>
<td>$2.00 \cdot 10^{-6}$</td>
</tr>
<tr>
<td>thread</td>
<td>$2.97 \cdot 10^4$</td>
<td>$2.25 \cdot 10^6$</td>
<td>$2.55 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>gupta2</td>
<td>$6.21 \cdot 10^4$</td>
<td>$2.16 \cdot 10^6$</td>
<td>$5.60 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>c-29</td>
<td>$5.03 \cdot 10^3$</td>
<td>$2.44 \cdot 10^4$</td>
<td>$9.64 \cdot 10^{-4}$</td>
</tr>
</tbody>
</table>

Table 4.1: Characteristics of the testing matrices.

5. Results of space efficient formats.

5.1. Testing matrices. We have used 11 testing matrices from various application domains from the University of Florida Sparse Matrix Collection [5]. Table 4.1 shows the characteristics of the testing matrices. For quantification of the reduction in data-representation size produced by a data format, we have used ratio of space complexities. For comparison of the results, we have used these common formats:

- the COO format.
- the CSR format.
Algorithm 7 Parallel transformation algorithm to the MBT format

1: procedure PARTr2MBT(A)

Input: A = the input matrix in the distributed CSR format

Output: S = the local bitstream of the resulting MBT format

2: i ← index of the current processor

3: if i = 1 then ▶ master section

4: S ← ()

5: enqueue \{A, 1, A.n, A.n, 0\} into current

6: while |current| < k ⋅ P do

7: current ← EXPANDLEVEL(S, current)

8: store S in master file

9: convert current to array B

10: barrier

11: one-to-all scatter of B

12: all-to-all scatter of matrix structure

13: S ← ()

14: C ← |current|

15: for j ← 1 + (i − 1)[C/P], \min(i(C/P), C) do

16: \{x1, y1, x2, y2, h\} ← B[j]

17: D ← A[y1...y2][x1...x2]

18: Tr2MBT(D, x1, y1, x2, y2, h)

19: store S in separate file dedicated to \pi

20: return

- the text-based Matrix Market format[4],
- the zipped Matrix Market format (we have used the PKZIP program with the option for maximal compression).

For our purposes, we have excluded all temporary informations from the source Matrix Market files (like comments and values of nonzero values).

<table>
<thead>
<tr>
<th>Matrix</th>
<th>COO S_MIN</th>
<th>COO S_POW</th>
<th>CSR S_MIN</th>
<th>CSR S_POW</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>2.25</td>
<td>3.13</td>
<td>1.24</td>
<td>1.71</td>
</tr>
<tr>
<td>alpckt120</td>
<td>2.27</td>
<td>3.30</td>
<td>1.23</td>
<td>1.77</td>
</tr>
<tr>
<td>ldoor</td>
<td>2.40</td>
<td>3.84</td>
<td>1.26</td>
<td>2.00</td>
</tr>
<tr>
<td>TSOPF_RS.b2383</td>
<td>4.04</td>
<td>4.04</td>
<td>2.03</td>
<td>2.03</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>3.73</td>
<td>3.73</td>
<td>1.87</td>
<td>1.88</td>
</tr>
<tr>
<td>t2em</td>
<td>2.11</td>
<td>3.38</td>
<td>1.30</td>
<td>2.03</td>
</tr>
<tr>
<td>bmw7at_1</td>
<td>2.61</td>
<td>4.63</td>
<td>1.36</td>
<td>2.40</td>
</tr>
<tr>
<td>amazon0312</td>
<td>2.23</td>
<td>3.75</td>
<td>1.28</td>
<td>2.11</td>
</tr>
<tr>
<td>thread</td>
<td>2.98</td>
<td>3.18</td>
<td>1.52</td>
<td>1.63</td>
</tr>
<tr>
<td>gupta2</td>
<td>2.61</td>
<td>2.61</td>
<td>1.36</td>
<td>1.38</td>
</tr>
<tr>
<td>c-29</td>
<td>2.27</td>
<td>2.79</td>
<td>1.40</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Table 5.1: The ratio of the space complexities of matrices in the COO or CSR formats using different paddings and in the ACB format.

5.2. Comparison of space complexities of common and ACB SSFs. Table 5.1 illustrates the fact that the space complexity of storing the structure of these sparse matrices using common storage formats (COO and CSR) are significantly
Algorithm 8 Master section of parallel transformation algorithm to the MBT format

1: procedure ExpandLevel(S, current)
2:  create empty queue new
3:  while current is nonempty do
4:    dequeue \{x_1, y_1, x_2, y_2, h\} from current
5:    if \(h\) is even then
6:      \(mx \leftarrow x_2; \) \(my \leftarrow (y_1 + y_2) / 2\)
7:      \(lx \leftarrow x_1; \) \(ly \leftarrow (y_1 + y_2) / 2 + 1\)
8:    else
9:      \(mx \leftarrow (x_1 + x_2) / 2; \) \(my \leftarrow y_2\)
10:     \(lx \leftarrow (x_1 + x_2) / 2 + 1; \) \(ly \leftarrow y_1\)
11:    \(l1 \leftarrow \text{PARINES}(A, x_1, y_1, mx, my)\)
12:    \(l2 \leftarrow \text{PARINES}(A, lx, ly, x_2, y_2)\)
13:    \text{APPENDToBitstream}(S, l1)
14:    \text{APPENDToBitstream}(S, l2)
15:    if \(l1 = \text{true}\) then
16:      enqueue \{A, x_1, y_1, mx, my, h + 1\} into next
17:    if \(l2 = \text{true}\) then
18:      enqueue \{A, lx, ly, x_2, y_2, h + 1\} into next
19:  return next

greater than in the ACB format (independently on the padding). We can conclude that the common SSFs (COO, CSR) are not suitable for our purposes.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CR1 [%]</th>
<th>CR2 [%]</th>
<th>CR3 [%]</th>
<th>CR4 [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM6</td>
<td>16.5</td>
<td>28.3</td>
<td>4.9</td>
<td>28.8</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>12.8</td>
<td>22.6</td>
<td>3.5</td>
<td>25.6</td>
</tr>
<tr>
<td>ldoor</td>
<td>8</td>
<td>16.1</td>
<td>2.4</td>
<td>15.3</td>
</tr>
<tr>
<td>TSOPF.RS.b2383</td>
<td>15.6</td>
<td>31.5</td>
<td>2.7</td>
<td>14.0</td>
</tr>
<tr>
<td>mouse.gene</td>
<td>73.0</td>
<td>137.0</td>
<td>12.8</td>
<td>53.9</td>
</tr>
<tr>
<td>t2em</td>
<td>16.3</td>
<td>33.0</td>
<td>5.7</td>
<td>26.2</td>
</tr>
<tr>
<td>bmw7test.1</td>
<td>8.5</td>
<td>20.3</td>
<td>2.8</td>
<td>15.7</td>
</tr>
<tr>
<td>amazon0312</td>
<td>53.1</td>
<td>112.1</td>
<td>18.1</td>
<td>67.7</td>
</tr>
<tr>
<td>thread</td>
<td>16.4</td>
<td>26.8</td>
<td>3.0</td>
<td>14.4</td>
</tr>
<tr>
<td>gupta2</td>
<td>28.7</td>
<td>39.7</td>
<td>5.3</td>
<td>23.3</td>
</tr>
<tr>
<td>c-29</td>
<td>31.7</td>
<td>53.4</td>
<td>8.6</td>
<td>27.2</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison of the space complexity of the MBT format with that of other storage schemes.

5.3. Results for the tree-based formats. The ACB format is space optimal, but only if the distribution of nonzero elements is random (i.e., without any locality in the matrix). Due to this fact, we use this format as the reference format. Table 5.2 compares the ratios of the matrix space complexities in the MBT format w.r.t. other storage schemes. CR stands for compression rate. CR1 denotes the ratio of the MBT to the (CSR, S\(^\text{ROW}\)) format space complexities. CR2 denotes the ratio of the MBT to the ACB format space complexities. CR3 denotes the ratio of the MBT format space complexity to space complexity of the text based Matrix Market format. CR4 denotes the ratio of the MBT format space complexities to the zipped Matrix Market
Table 5.3: Comparison of the space complexity of the tree-based SSI’s with that of the ACB format.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>MBT [%]</th>
<th>CBT [%]</th>
<th>MQT [%]</th>
<th>CQT [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>27.4</td>
<td>24.4</td>
<td>26.6</td>
<td>26.5</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>26.1</td>
<td>23.5</td>
<td>19.8</td>
<td>19.8</td>
</tr>
<tr>
<td>ldoor</td>
<td>22.5</td>
<td>21.2</td>
<td>14.4</td>
<td>14.3</td>
</tr>
<tr>
<td>TSOPF.RS.b2383</td>
<td>36.1</td>
<td>35.9</td>
<td>33.6</td>
<td>33.5</td>
</tr>
<tr>
<td>mouse.gene</td>
<td>130.8</td>
<td>111.0</td>
<td>130.4</td>
<td>128.0</td>
</tr>
<tr>
<td>t2em</td>
<td>36.6</td>
<td>31.7</td>
<td>29.9</td>
<td>29.4</td>
</tr>
<tr>
<td>bmw7st.1</td>
<td>26.3</td>
<td>25.0</td>
<td>20.3</td>
<td>20.3</td>
</tr>
<tr>
<td>amazon0312</td>
<td>110.8</td>
<td>89.1</td>
<td>107.1</td>
<td>103.1</td>
</tr>
<tr>
<td>thread</td>
<td>37.3</td>
<td>35.3</td>
<td>23.8</td>
<td>23.8</td>
</tr>
<tr>
<td>gupta2</td>
<td>48.0</td>
<td>42.3</td>
<td>36.3</td>
<td>36.0</td>
</tr>
<tr>
<td>c-29</td>
<td>55.7</td>
<td>48.8</td>
<td>51.3</td>
<td>50.9</td>
</tr>
</tbody>
</table>

Table 5.3 shows ratios of space complexities of the four tree-based formats studied in this paper to the ACB format. From this table, we can observe that the CBT format:

- has usually smaller space complexity than the ACB format. There was only one exception among the 11 testing matrices: (mouse.gene).
- has similar space complexity as the MQT or CQT formats.

We can conclude that the CBT format is very space efficient.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>N</th>
<th>#calls (kP = 10^4)</th>
<th>#calls (kP = 10^5)</th>
<th>#calls (kP = 10^6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuitM5</td>
<td>5.56 · 10^6</td>
<td>5.95 · 10^7</td>
<td>20</td>
<td>278</td>
<td>4248</td>
</tr>
<tr>
<td>nlpkkt120</td>
<td>3.54 · 10^6</td>
<td>5.02 · 10^7</td>
<td>26</td>
<td>452</td>
<td>3852</td>
</tr>
<tr>
<td>ldoor</td>
<td>9.52 · 10^5</td>
<td>2.37 · 10^7</td>
<td>26</td>
<td>240</td>
<td>2204</td>
</tr>
<tr>
<td>TSOPF.RS.b2383</td>
<td>3.81 · 10^6</td>
<td>1.62 · 10^7</td>
<td>20</td>
<td>378</td>
<td>3210</td>
</tr>
<tr>
<td>mouse.gene</td>
<td>4.51 · 10^6</td>
<td>1.45 · 10^7</td>
<td>22</td>
<td>240</td>
<td>2060</td>
</tr>
<tr>
<td>t2em</td>
<td>9.25 · 10^5</td>
<td>4.59 · 10^6</td>
<td>26</td>
<td>426</td>
<td>4906</td>
</tr>
<tr>
<td>bmw7st.1</td>
<td>1.41 · 10^6</td>
<td>3.74 · 10^6</td>
<td>22</td>
<td>232</td>
<td>3368</td>
</tr>
<tr>
<td>amazon0312</td>
<td>4.01 · 10^5</td>
<td>3.20 · 10^6</td>
<td>18</td>
<td>200</td>
<td>2072</td>
</tr>
<tr>
<td>thread</td>
<td>2.97 · 10^5</td>
<td>2.25 · 10^6</td>
<td>26</td>
<td>260</td>
<td>3486</td>
</tr>
<tr>
<td>gupta2</td>
<td>6.21 · 10^4</td>
<td>2.16 · 10^6</td>
<td>28</td>
<td>380</td>
<td>3888</td>
</tr>
<tr>
<td>c-29</td>
<td>5.03 · 10^4</td>
<td>2.44 · 10^6</td>
<td>26</td>
<td>348</td>
<td>3900</td>
</tr>
</tbody>
</table>

Table 5.4: The efficiency of parallel algorithm (the number of calls of PARINES).

5.4. Results for parallelization of the tree-based formats. The most time-consuming part of master section of PARTr2MBT procedure (parallel implementation of conversion to the MBT format) is the blocking call of PARINES (or PARINES2) procedure. To achieve good scalability of this code, the overhead of these calls should be small in comparison to the global parameters (n and N), because these parameters influence the time complexity of parallel section (local transformation) of PARTr2MBT procedure (see Sect. 3.2.3).

Table 5.4 shows the number of calls of PARINES in comparison to global parameters. From this table, we can observe that these numbers of calls are relatively close to the parameter kP, so the parallel conversion to the MBT format is reasonable (and the value of parameter k is suitable) if kP ≪ n.
6. Conclusions. This paper deals with the design of four new SSFs called arithmetical coding based format, minimal binary tree format, compressed binary tree format, and compressed quadtree format. These formats have been designed in order to minimize the space complexity. We performed experiments with these formats and compared them with other common SSFs (COO or CSR) and other schemes used for LSMs in a file. These experiments proved that our new formats can significantly reduce the amount of data needed for storing LSMs. We have also presented a parallel algorithm for transformation of a LSM in the CSR format to one of these newly proposed formats.

REFERENCES


[22] M. Tuma, Overview of direct methods, I. Winter School of SEMINAR ON NUMERICAL ANALYSIS, January 2004, Ostrava, Czech Republic
3.7 RP7: Efficient Converting of Large Sparse Matrices to Quadtree Format

In this paper (RP7), ideas from previous papers [RP4] were extended. A completely new algorithm based on bottom-up approach for converting matrices from common storage formats to the quadtree format was introduced.

The previous approach (that has been presented in [RP4]) was denoted as “classical”. Its main drawback is time complexity in the worst case. Therefore, a completely different approach for the efficient converting of large sparse matrices to quadtree-based format was developed. The algorithm for this approach consists of the following steps:

1. Construction of a sequence of reduced matrices: time complexity of this stage is $O(N \log n + n)$.

2. Construction of the list of QT nodes: time complexity of this stage is $O(N \log n + n)$.

3. Reordering of the list of QT nodes: time complexity of this stage is $O((\log_2 n - 1 - \log_4 N)N \log N)$.

4. Completing of the output: time complexity of this stage is $O(N(1/3 + \log_4(n^2/N)))$.

Similar approach can be used for converting into advanced QT format. This transformation algorithm consists of the following steps:

1. Construction of the matrix $M^{(0)}$,

2. Construction of a sequence of reduced matrices,

3. Construction of the list of QT nodes,

4. Reordering of the list of QT nodes,

5. Completing of the advanced QT format.

Thus, these two algorithms differ only in two steps.

Comparison of performance of sequential algorithms: Table II shows the comparison of measured times for different algorithms for the transformation to the MQT format. We
can conclude that the reducing algorithm is always faster (except two cases) than the “classical” algorithm.

Parallel variants of the “classical” and the new algorithm were also shown. We performed experiments on the real parallel system.

For the optimal value of machine-depended parameter $PLIMIT$, the “classical“ algorithm scales relatively well with the number of threads (see Figure 2). The reducing algorithm achieves the significant speedup only for two threads, for higher number the speedup remains the same. The main bottleneck lies in the step “the list of QT nodes reordering”: if the size of one list $L(i)$ is much higher than others then the thread assigned for sorting of this list is busy for the long time, but other threads are idle, so the speedup is almost independent on the number of threads.

These experiments proved that the proposed algorithm allows an efficient conversion.
Efficient Converting of Large Sparse Matrices to Quadtree Format

I. Šimeček, D. Langr, J. Trdlička
Department of Computer Systems, Faculty of Information Technology,
Czech Technical University in Prague Prague, Czech Republic Email: xsimecek,langrd,jan.trdlicka@fit.cvut.cz

Abstract—Computations with sparse matrices are widespread in scientific projects. Used data format affects strongly the performance and also the space-efficiency. Commonly used storage formats (such as COO or CSR) are not suitable neither for some numerical algebra operations (e.g., the sparse matrix-vector multiplication) due to the required indirect addressing nor for I/O file operations with sparse matrices due to their high space complexities. In our previous papers, we prove that the idea of using the quadtree for these purposes is viable. In this paper, we present a completely new algorithm based on bottom-up approach for the converting matrices from common storage formats to the quadtree format. We derive the asymptotic complexity of our new algorithm, design the parallel variant of the classical and the new algorithm, and discuss their performance.

I. INTRODUCTION

A. Motivation

Computations with sparse matrices are widespread in scientific projects. Very large sparse matrices are often processed on massively parallel computer systems with distributed memory architectures consisting of tens or hundreds of thousands of processor cores. Efficient formats for storing sparse matrices are still under development, since the computation using widely-used formats (like XY or CSR) is slow and these formats are also space-inefficient.

In [1]–[3], we find about that the formats based on quadformat can be suitable for these applications areas.

B. Efficient numerical algebra operations

Within our previous work [1], we have addressed weaknesses of previously developed solutions for efficient numerical algebra operations (e.g., the sparse matrix-vector multiplication or sparse matrix-matrix multiplication).

In paper [1], we represent some improvements to the quadtree storage format and compare the performance during the execution of some basic routines from the linear algebra using widely-used formats and the quadtree storage format. Algorithms using this data format show significant speedups for some matrix operations (between 5% and 80% in the case of multiplication of a sparse matrix by a sparse matrix and about 700% in case of getting value at the given position). It results from the fact that the quadtree data format combines advantages of sparse and dense data formats.

C. Space-efficient sparse matrix formats

Within our previous work [2]–[4], we have addressed weaknesses of previously developed solutions for space-efficient formats for storing of large sparse matrices. In these papers, we propose and evaluate new sparse matrix storage formats that minimize the space complexity of information about matrix structure. The motivation of our work is in applications with very large sparse matrices that due to their size must be processed on massively parallel computer systems consisting of tens or hundreds of thousands of processor cores and that must be stored in a distributed file system using a parallel I/O subsystem. The parallel I/O subsystem is typically the main performance bottleneck and loading or storing such matrices from/to a distributed file system can take significant amounts of time. We reduce this time by reducing the space complexity of the very large sparse matrices.

In paper [3], we deal with the design of a new sparse matrix storage QT-based format, called the MQT format, that minimizes space complexity of the large sparse matrix structure. We performed experiments with the MQT format and compared it with other common COO or CSR formats. These experiments proved that the MQT format can significantly reduce amount of data needed for storing large sparse matrices, so it allows efficient I/O file operations.

D. Terminology and notation

Definition A matrix of order \( n \) is considered to be sparse if it contains much less nonzero elements than \( n^2 \).

Some alternative definitions of sparse matrix can be found in [5]. In practice, a matrix is considered sparse if the ratio of nonzero elements drops below some threshold.

- We consider a large sparse matrix \( A \) of order \( n \). The number of its nonzero elements is denoted by \( N \) and the average number of nonzero elements per row is denoted by \( \text{avg}_{\text{per row}} \). Obviously, \( \text{avg}_{\text{per row}} = N/n \).
- We assume that \( 1 \ll N \ll M = n^2 \).
- The pattern of nonzero elements in \( A \) is unknown or random.
- Indexes of all vectors and matrices start from zero.
- The number of nonzero elements in submatrix \( B \) of matrix \( A \) is denoted by \( \text{nnz}(B) \). If the submatrix \( B \) does not contain any nonzero elements \( (\text{nnz}(B) = 0) \), it is denoted as empty submatrix.

E. Common sparse storage formats

1) The Coordinate (COO) Format: It is the simplest and most straightforward sparse storage format. The matrix \( A \) is
represented by three linear arrays \( \text{Elem}, X, \) and \( Y \). The array \( \text{Elem}[1, \ldots, N] \) stores the nonzero elements of \( A \), arrays \( X[1, \ldots, N] \) and \( Y[1, \ldots, N] \) contain column and row indexes, respectively, of these nonzero values.

2) The Compressed Sparse Row (CSR) format: It is the most common sparse storage format (see [6], [7]). The matrix \( A \) is represented by three linear arrays \( \text{Elem}, \text{Addr}, \) and \( \text{Ci} \). The array \( \text{Elem}[1, \ldots, N] \) stores the nonzero elements of \( A \), the array \( \text{Addr}[1, \ldots, n] \) contains indexes of initial nonzero elements of rows of \( A \). The array \( \text{Ci}[1, \ldots, N] \) contains column indexes of nonzero elements of \( A \). Hence, the first nonzero element of row \( j \) is stored at index \( \text{Addr}[j] \) in the array \( \text{Elem} \).

F. Quadtree-based sparse matrix storage formats

Definition The Quadtree (QT) is a tree data structure in which all inner nodes have exactly four child nodes.

The QT is used to describe a partition of a sparse matrix into submatrices. Each QT node represents a submatrix. There are different implementations of the QT format. In the standard implementation (for details, see [8]–[10]), every node in the QT is represented by structure \( \text{standard_QT_struct} \) consisting of the following items:

- four pointers \((\text{up\_left, up\_right, lo\_left, lo\_right})\) to the child nodes,
- (only for leaves) values of nonzero elements.

1) Quadtree format for efficient numerical algebra operations: In paper [1], we have proposed an advanced implementation of QT format. It has an additional parameter \( \text{tile\_size} \) used to stop the recursive partitioning. There are different types of QT nodes. Inner QT nodes can be "Mixed" or "Empty". QT leaves can be "Full" or "Sparse". Every node in the advanced QT is represented by structure \( \text{advanced_QT_struct} \) consisting of the following items:

- node type ("Empty", "Mixed", "Sparse" or "Full"),
- four pointers \((\text{up\_left, up\_right, lo\_left, lo\_right})\) to the child nodes,
- (only for nodes of type "Sparse" or "Full") values of nonzero elements.

Great advantages of the QT-based sparse storage formats (both standard and advanced) are the following:

- Modifications (adding or removing nonzero elements) of the QT structure are relatively easy and fast in comparison to the COO or CSR formats.
- The recursive style of programming and recursive style of storage ("Divide and Conquer" approach) leads to codes with surprising performance due to the better cache memory utilization (see [1])

2) Quadtree-based space-efficient sparse matrix formats: A big drawback of the previous QT formats is a larger data overhead (caused by pointers \( \text{up\_left, up\_right, lo\_left, lo\_right} \)) compared to the COO and CSR formats. Since our aim is to minimize the space complexity of QT-based formats, in paper [3] we propose a new QT format called minimal quadtree (MQT) format that extends ideas of the standard QT format as follows:

- All nodes in the MQT are stored in one array. Since we can compute locations of all child nodes, we can omit pointers \( \text{up\_left, up\_right, lo\_left, lo\_right} \). We lose the advantage of the possibility to easily modify the QT, but it is not an important property for our application area.
- Instead of pointers, each node of the MQT contains only 4 flags (i.e., 4 bits only) indicating whether given subquadtrees are nonempty.

So, the space complexity of every MQT node is only 4 bits.

Algorithm 1 Test if the given submatrix of \( A \) stored in the CSR format is nonempty

\begin{verbatim}
1: procedure INES(A,x1,y1,x2,y2)
Input: A = the matrix in the CSR format
Input: x1,y1,x2,y2 = coordinates of the submatrix
Output: logical value denotes if the given submatrix is nonempty
2: for i ∈ A.Addr[y1], A.Addr[y2 + 1] − 1 do
3: x ← A.ci[i];
4: if (x ≥ x1) \& (x ≤ x2) then
5: return true;
6: return false;
\end{verbatim}

Algorithm 2 Test if the given submatrix of \( A \) stored in the CSR format is nonempty

\begin{verbatim}
1: procedure INES2(A,x1,y1,x2,y2)
Input: A = the matrix in the CSR format
Input: x1,y1,x2,y2 = coordinates of the submatrix
Output: logical value denotes if the given submatrix is nonempty
2: for y ← y1, y2 do
3: low ← A.addr[y]; high ← A.addr[y + 1] − 1;
4: i ← \text{BINARY SEARCH}(in array A.ci);
5: \triangleright within indexes from \((\text{low} \ldots \text{high})\)
6: \triangleright to find minimal \( i \) such that \( A.ci[i] \geq x1 \)
7: if A.ci[i] ≤ x2 then
8: return true;
9: return false;
\end{verbatim}

G. The quadtree transformation algorithm

1) Quadtree-based space-efficient sparse matrix formats: Algorithm 3 transforms a matrix \( A \) from the CSR format into the MQT format, represented by an output array. Since arrays are accessed in linear order, the output array can be viewed as a bitstream \( S \). It calls the subroutines \( \text{NNIS}(A,x1,y1,x2,y2) \) for the enumeration of the number of nonzero elements in a given submatrix of \( A \) and \( \text{INES}(A,x1,y1,x2,y2) \) (Algorithm 1) or \( \text{INES2}(A,x1,y1,x2,y2) \) (Algorithm 2) for the test if the given submatrix of \( A \) is nonempty.
### Algorithm 3: Transformation algorithm to the MQT format

1. **procedure** `TR2MQT(A)`

   **Input:** $A$ = the matrix for the transformation in CSR format
   
   **Output:** $S$ = the bitstream representing the input matrix in the MQT format

   2. `current ← ()`
   3. `enqueue \{1, 1, A.n, A.n\} into current`
   4. while `current` is not empty do
      5. `dequeue \{x1, y1, x2, y2\} from current`
      6. if `x1 = x2` then
         7. \(mx ← (x1 + x2)/2;\)
         8. \(my ← (y1 + y2)/2;\)
         9. if `l1 = true` then
            10. `enqueue \{x1, y1, mx, my\} into current`
         11. else
            12. `enqueue \{mx + 1, y1, x2, my\} into current`
      13. if `l2 = true` then
         14. `enqueue \{x1, y1, mx, my\} into current`
      15. else
         16. `enqueue \{mx + 1, y1, x2, my\} into current`
      17. if `l3 = true` then
         18. `enqueue \{x1, y1, mx, my\} into current`
      19. else
         20. `enqueue \{mx + 1, my + 1, x2, y2\} into current`
   21. `return S;`

### Algorithm 4: Transformation algorithm from the CSR format to the advanced QT format

1. **procedure** `TRANSF(A, x1, y1, x2, y2)`

   **Input:** $A$ = the matrix in the CSR format
   
   **Input:** $x1,y1,x2,y2$ = coordinates of the submatrix to be transformed

   **Output:** the pointer to the root of the QT representation of $A$

   2. if `INES(A, x1, y1, x2, y2) = false` then
      3. `return NULL;`
   4. \(n' ← \text{max}(x2 − x1, y2 − y1);\)
   5. if `n' > tile_size` then
      6. \(mx ← (x1 + x2)/2; \quad my ← (y1 + y2)/2;\)
      7. `create M - the node of QT of type "Mixed"`
      8. \(M.\text{up}\_\text{left} ← \text{TRANSF}(A, x1, y1, mx, my);\)
      9. \(M.\text{up}\_\text{right} ← \text{TRANSF}(A, mx + 1, y1, x2, my);\)
      10. \(M.\text{lo}\_\text{left} ← \text{TRANSF}(A, x1, my + 1, mx, y2);\)
      11. \(M.\text{lo}\_\text{right} ← \text{TRANSF}(A, mx + 1, my + 1, x2, y2);\)
      12. \(\triangleright\) previous four commands transform disjoint submatrices of the given submatrix of $A$
      13. `return M;`
   14. else
      15. \(N' ← \text{NNIS}(A, x1, y1, x2, y2);\)
      16. \(F ← \text{representation of the input submatrix } A;\)
      17. \(\triangleright\) in some common storage format
      18. \(\triangleright\) according to value $N'$
      19. `return F;`

#### 2) Quadtree format for efficient numerical algebra operations

Algorithm 4 converts a matrix $A$ stored in the CSR format into the advanced QT-based format. Empty submatrices (that do not contain any nonzero elements) are represented by the NULL pointers. The algorithm uses parameter `tile_size` as the maximum size of a submatrix in one node (except "Empty" nodes).

#### H. Time complexity of the transformation from the CSR format

1) **Quadtree-based space-efficient sparse matrix formats:**

The following derivation was published in [3]. The time complexity of the procedure `INES(A,x1,y1,x2,y2)` is

\[
O\left((y2 - y1 + 1) \frac{\text{avg}_\text{per}_\text{row}}{\text{col}}\right).
\]

The time complexity of the procedure `INES2(A,x1,y1,x2,y2)` (Algorithm 2) is

\[
O\left((y2 - y1 + 1) \log_2 \frac{\text{avg}_\text{per}_\text{row}}{\text{col}}\right).
\]

For Algorithm 3, we consider the usage of `INES2` and the worst case: the quadtree with the maximal number of nodes, i.e., the number of leaves is equal to $N$ (see Figure 1).

- The height of this tree is \(h = h1 + h2 = \log_2 n - 1,\) where \(h1 = \log_4 N.\)
- All nodes with depth < \(h1\) (in upper \(h1\) levels) are full of 1's (they have four child nodes), so \(h1 = \log_4 N.\) The number of nodes in these levels is approximately

\[
\sum_{i=0}^{h1-1} 4^i \approx N/3.
\]
- All nodes with height > \(h1\) (in lower \(h2\) levels) contain exactly one 1 (they have only one child node). The
number of nodes in these levels is
\[ N \cdot h^2 = N \cdot (\log_4 n - \log_4 N) = N \cdot \log_4(n^2/N). \]
We assume that the time complexity of procedure APPEND-ToBitstream is \( \Theta(1) \). Procedure INES(A,x1,y1,x2,y2) is called for every node in the MQT in the output stream \( S \) four times.

- For nodes with depth=1: The number of these nodes is \( N \), the expression \( (y^2 - y^1 + 1) \) is equal to \( 1 + n/\sqrt{N} \). Time complexity of the transformation for all nodes with this depth is \( T_{h1} = N \cdot (1 + n/\sqrt{N}) \cdot \log_2 \text{avg per row} \).
- For nodes with depth\(< h1 \): the number of nodes is \( N/4 \) and the expression \( (y^2 - y^1 + 1) \) is equal to \( 1 + 2n/\sqrt{N} \). So, the total time complexity of the transformation for all nodes with depth\(< h1 \) (in upper \( h1 \) levels) is \( T_{\text{upper}} \approx \sum_{i=0}^{h1} T_{h1}/2^{i} = O(N \cdot (1 + n/\sqrt{N}) \cdot \log_2 \text{avg per row}) \).
- For nodes with depth\(> h1 \): The time complexity of the transformation for all these nodes(for the lower \( h2 \) levels) is \( T_{\text{lower}} \approx \sum_{i=h1+1}^{h} T_{h1}/2^{i} = O(N/(1 + n/\sqrt{N}) \cdot \log_2 \text{avg per row}) \).

So, the total time complexity of the transformation is
\[ O(N(1 + n/\sqrt{N}) \cdot \log_2 \text{avg per row}). \]

A very usual case is \( N = \Theta(n) \), i.e., matrices with constant number of nonzero elements per row. For this case, the time complexity is \( O(nh^2) = O(N^{3/2}) \).

2) Quadtree format for efficient numerical algebra operations: In Algorithm 4 the time complexity of the procedure TRANSF(A,x1,y1,x2,y2) depends on the value of parameter tile_size and on the criteria for creation of leaves (see the results of experiments in [1]), but we can use similar ideas for the derivation as in Section I-H1. We consider the worst case: the quadtree with the maximal number of nodes, i.e., the number of leaves is equal to \( N \) (see Figure 1). The total complexity depends on the value of an additional parameter tile_size:
- If the value tile_size (used to stop the recursive partitioning) is so high than the height of QT \( h \) is less that \( h1 \) in Figure 1. It means
  \[ \log_4(n/tile\_size) < h1 = \log_4 N, \]
  it results in \( tile\_size > n/\sqrt{N} \). In this case \( h = \log_2(n/tile\_size) \), so the total complexity is
  \[ T = \sum_{i=0}^{h} n^{4^i/2^i} \log_2 \text{avg per row} \]
  \[ = n^{(h+1)} - 1 \log_2 \text{avg per row} \]
  \[ = O(n^2/tile\_size \log_2 \text{avg per row}). \]
- Otherwise, the total complexity is the same as for MQT, i.e.,
  \[ O(N(1 + n/\sqrt{N}) \cdot \log_2 \text{avg per row}). \]

I. OpenMP

The OpenMP API specification (for details see [11], [12]) 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 process of parallelization is however not automated, the programmer is responsible for the correct usage of the API and avoidance of race conditions, deadlocks, and other data consistency issues related to shared memory environment.

The core of OpenMP is the fork-join 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. Use of the compiler directives allows a program to be compiled both as sequential or OpenMP parallel by using a compiler command line option. That however doesn’t mean that the program will produce correct results in both versions as this is a responsibility and a choice of programmer.

There are several constructs allowing different ways to implement parallelization, most commonly used are parallel and task and work sharing constructs. The parallel construct causes the thread to create a team of itself and several more additional threads and becoming master thread. An set of implicit tasks (one per thread) is created with the code from parallel construct. Each task is then assigned to one of the threads, tied to it and executed in parallel.

The task construct encountered by a thread generates a new explicit task. Its execution is assigned to one of the threads in current team and it may be executed immediately or deferred until later based on the task scheduling and thread availability. Threads can suspend the current task region to execute a different task and the original task can be finished by another thread if it’s not tied to the original thread.

If a team of threads encounter a work sharing construct (e.g., loop construct), the work inside is divided among the members of the team and executed cooperatively.

To coordinate tasks and data access in parallel regions, synchronization constructs and library routines can be used. These include constructs for critical sections, atomic operations or explicit barriers.

II. RESULTS

A. The reducing method algorithm for the MQT format

The previous approach (represented by Algorithm 4) we denote as "classical", its main drawback is time complexity in the worst case. Therefore, we have developed a completely different bottom-up approach for the efficient converting of large sparse matrices to quadtree-based format. The algorithm for this approach consists of the following steps (they are discussed in detail later):

1) construction of a sequence of reduced matrices,
2) construction of the list of QT nodes,
3) reordering of the list of QT nodes,
4) completing of the output.
The performance comparison with classical algorithms is done in Section III.

1) Construction of a sequence of reduced matrices: In this step, we construct a sequence of matrices by progressively reducing the size of matrices:
- \( M^{(0)} = A \) with order \( k^{(0)} = n \),
- for \( 1 \leq i < \log n \), the matrix \( M^{(i)} \) has order \( k^{(i)} = \lceil k^{(i-1)}/2 \rceil \).

Reducing the size of matrices is done by merging of rows and columns in matrix. **Note:** Construction of \( M^{(\log_2 n)} \) is useless because \( M^{(\log_2 n)} = (1) \) for every matrix that contains at least one nonzero element.

**Example of this step:**
Instead of the values of the matrix elements, we deal only with binary flags indicating the existence of nonzero elements.

\[
M^{(0)} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

The original matrix \( M^{(0)} \) that is equal to input matrix \( A \) is processed by REDUCE operation (merging even and odd rows and then even and odd columns using the binary OR operation). It results in matrix \( M^{(1)} \).

\[
\text{REDUCE}(M^{(0)}) = M^{(1)} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

The next REDUCE operation transforms the matrix \( M^{(1)} \) to matrix \( M^{(2)} \). Then the sequence of matrices (consisting of matrices \( M^{(0)}, M^{(1)}, \) and \( M^{(2)} \)) is completed.

\[
\text{REDUCE}(M^{(1)}) = M^{(2)} = (111)
\]

The Algorithm 5 implements merging of the rows for the CSR format. Modification for the merging of the columns is obvious.

2) Construction of the list of QT nodes: In this step, for each \( i, 0 \leq i < \log_2 n \) from matrix \( M^{(i)} \) the list of QT nodes \( L^{(i)} \) is created. The QT node is quadruple of values \( M_{x,y}^{(i)}, M_{x+1,y}^{(i)}, M_{x,y+1}^{(i)}, \) and \( M_{x+1,y+1}^{(i)} \) with following properties:
- \( x \) and \( y \) are initial coordinates of QT node in the matrix \( M^{(i)} \),
- \( x \) and \( y \) are even,
- at least one value of QT node is nonzero.

All QT nodes are created, but they are not in correct order (given by the QT breadth-first traversal). The QT nodes must be reordered according to the so-called Morton order (for details see [13]) in the next step. Morton ordering is a mapping from an \( n \)-dimensional space onto a linear list of numbers. If you want to convert a certain set of integer coordinates to a Morton code, you 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 Z-curve. For the successful reordering (in the next step), every QT node in list contain also its Morton code. The algorithm for this step is very similar to Algorithm 5.

**Example of this step:**
From the original matrices \( M^{(2)}, M^{(1)}, \) and \( M^{(0)} \) the follow-

---

**Algorithm 5** Reducing of matrix in the CSR format

1: procedure REDUCE(In)

Input: \( In = \) the matrix in the CSR format

Output: \( Out = \) the output matrix in the CSR format

\( \triangleright \) for even number of rows

2: create empty matrix \( Out \)

3: \( Out.adr[0] \) $\leftarrow$ 0; \( index \) $\leftarrow$ 0;

4: for \( y3 \leftarrow 1, In.n/2 \) do

5: \( even \leftarrow 2 \cdot y3; \) \( odd \leftarrow even + 1; \)

6: \( i1 \leftarrow In.addr[even]; \)

7: \( h1 \leftarrow In.addr[even + 1]; \)

8: \( i2 \leftarrow In.addr[odd]; \)

9: \( h2 \leftarrow In.addr[odd + 1]; \)

10: number $\leftarrow 0; \)

11: while \( i1 \leq h1 \) $\&$ \( i2 \leq h2 \) do

12: \( x1 \leftarrow In.ci[i1]/2; \) \( x2 \leftarrow In.ci[i2]/2; \)

13: if \( x1 = x2 \) then

14: \( Out.ci[index] \) $\leftarrow x1; \)

15: \( i1 \leftarrow i1 + 1; \)

16: \( i2 \leftarrow i2 + 1; \)

17: else

18: \( Out.ci[index] \) $\leftarrow x2; \)

19: \( i2 \leftarrow i2 + 1; \)

20: \( index \leftarrow index + 1; \) \( number \leftarrow number + 1; \)

21: while \( i1 \leq h1 \) do

22: \( x1 \leftarrow In.ci[i1]/2; \) \( Out.ci[index] \) $\leftarrow x1; \)

23: \( i1 \leftarrow i1 + 1; \) \( number \leftarrow number + 1; \)

24: \( index \leftarrow index + 1; \)

25: while \( i2 \leq h2 \) do

26: \( x2 \leftarrow In.ci[i2]/2; \) \( Out.ci[index] \) $\leftarrow x2; \)

27: \( i2 \leftarrow i2 + 1; \) \( number \leftarrow number + 1; \)

28: \( index \leftarrow index + 1; \)

29: \( Out.adr[y3 + 1] \) $\leftarrow Out.adr[y3] + number; \)

30: return \( Out; \)
ing lists arise

\[
L^{(2)} = \left( \begin{array}{ll}
1 & 1 \\
1 & 1 \\
\end{array} ; 00 \right)
\]

\[
L^{(1)} = \left( \begin{array}{ll}
0 & 0 \\
1 & 1 \\
0 & 0 \\
\end{array} ; 0000 ; \begin{array}{ll}
0 & 1 \\
0 & 0 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 0100 ; \begin{array}{ll}
0 & 1 \\
0 & 0 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 1100 \right)
\]

\[
L^{(0)} = \left( \begin{array}{ll}
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 010100 ; \begin{array}{ll}
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 010100 ; \begin{array}{ll}
1 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 101100 \right)
\]

3) The list of QT nodes reordering: In this step, for each \( i, 0 \leq i < \log_2 n \) the lists of QT nodes are reordered (sorted) according to the Morton codes.

Example of this step:

\[
L^{(2)} = \left( \begin{array}{ll}
1 & 1 \\
1 & 1 \\
\end{array} ; 00 \right)
\]

\[
L^{(1)} = \left( \begin{array}{ll}
0 & 0 \\
1 & 1 \\
0 & 0 \\
\end{array} ; 0000 ; \begin{array}{ll}
0 & 1 \\
0 & 0 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 0100 ; \begin{array}{ll}
0 & 1 \\
0 & 0 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 1100 \right)
\]

\[
L^{(0)} = \left( \begin{array}{ll}
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 010000 ; \begin{array}{ll}
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 010100 ; \begin{array}{ll}
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array} ; 101100 \right)
\]

4) Completing of the output: This stage is obvious, for each \( i, 0 \leq i < \log_2 n \) (begin with \( i = (\log_2 n) - 1 \)) all QT nodes from the list \( L_i \) are appended into output bitstream \( S \).

\[
S = 111110010100000101000111011101101
\]

B. The reducing method algorithm for the advanced QT format

The approach presented in Section II-A can be used for converting either into advanced QT format. The algorithm consists of the following steps (some of them are discussed in detail later):

1) construction of the matrix \( M^{(0)} \),
2) construction of a sequence of reduced matrices: this step is the same as in Section II-A1,
3) construction of the list of QT nodes: this step is the same as in Section II-A2,
4) reordering of the list of QT nodes: this step is the same as in Section II-A3,
5) completing of the advanced QT format.

1) Construction of the matrix \( M^{(0)} \): To start the next step “construction of a sequence of reduced matrices” we need to construct the initial matrix \( M^{(0)} \). This algorithm can be described by Algorithm 6.

Algorithm 6 Construction of the matrix \( M^{(0)} \)

1: procedure CONSTRUCT(A)
2: input: \( A = \) the matrix in the CSR format
3: output: \( M^{(0)} = \) initial matrix for the next step
4: procedure 1) Construction of a sequence of reduced matrices:
5: \( M^{(0)} = \) initial matrix as zero matrix with order \( [n/tile\_size] \)
6: for \( y \leftarrow 1, n \) do
7: \( low \leftarrow A.addr \[ y \]; \ high \leftarrow A.addr \[ y + 1 \] - 1
8: \( ys = \lceil y/tile\_size \rceil \)
9: for \( i \leftarrow low, high \) do
10: \( x = A.ci \[ i \]; \ xs = \lfloor x/tile\_size \rfloor \)
11: set \( M^{(0)} \)[xs, ys] to 1
12: end for
13: end for
14: \( M^{(0)} \)

2) Completing of the advanced QT format: In this step, the space for all QT nodes are allocated and they are linked together (pointers in QT nodes from the list \( L^{(i+1)} \) are set to QT nodes from \( L^{(i)} \)).

C. Time complexity of the reducing method algorithm

1) Construction of a sequence of reduced matrices: Time complexity of Algorithm 5 for a matrix of order \( n \) with \( N \) nonzero elements is \( \Theta(N+n) \) operations. The resulting matrix contains \( \lceil N/2, \ldots, N \rceil \) nonzero elements. After the merging of columns, the resulting matrix contains \( \lceil N/4, \ldots, N \rceil \) nonzero elements.

So, the total time complexity of this step depends on the number of iterations (\( \log_2 n \)), but the complexity of one iteration depends on the number of nonzero elements in the reduced matrices:

- The best case: The (reduced) matrix \( M^{(i)} \) contains only one quarter of nonzero elements compared to the matrix \( M^{(i-1)} \) so, in this case the complexity of this step is
  \[
  \sum_{i=0}^{\log_2 n} O\left(\frac{N + n}{4^i}\right) = O(N + n)
  \]

- The average case: The (reduced) matrix \( M^{(i)} \) contains approximately half of nonzero elements compared to the original matrix \( M^{(i-1)} \). So, in this case the complexity of this step is
  \[
  \sum_{i=0}^{\log_2 n} O\left(\frac{N + n}{2^i}\right) = O(N + n).
  \]

- The worst case: Every \( 2^g \times 2^g \) submatrix \( D \) of initial matrix \( A \) contains exactly one nonzero element, so the (reduced) matrix \( M^{(0)} \) contains the same number of nonzero elements as the previous matrix \( M^{(i-1)} \).

\[
D = \begin{pmatrix}
1 & 0 & 0 & \cdots \\
0 & 0 & 0 & \cdots \\
0 & 0 & 0 & \cdots \\
\vdots & \vdots & \vdots & \ddots \\
\end{pmatrix}
\]
\[
\text{REDUCE}(D) = \begin{pmatrix} 1 & 0 & 0 & \ldots \\ 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}^{2^{g-1}}
\]

So, the first \( g \) iterations of the merging operation reduce the order of the matrix \( n \), but the value of \( N \) remains the same. Next \((\log_2 n) - g\) iterations of the merging operation behave as in the best case scenario. So the complexity of this step is

\[
gN + \sum_{i=g+1}^{\log_2 n} \frac{N}{4^i} + \sum_{i=0}^{\log_2 n} \frac{n}{4^i} = O(gN + n)
\]

Since \( 1 \ll n \ll n^2 \) and \( N \cdot 2^{2g} = n^2 \), we can derive that:

\[
2^{2g} = \frac{n^2}{N}
\]

\[
g = \log_2 n - \frac{1}{2} \log_2 N
\]

\[
g < \log n. \tag{1}
\]

So, \( g \in O(\log n) \) and the total complexity of this stage is \( O(N \log n + n) \).

2) Construction of the list of QT nodes: The algorithm for this step is very similar to Algorithm 5, so its complexity is the same: \( O(N \log n + n) \).

3) Reordering of the list of QT nodes: In this step, for each \( i \), \( 0 \leq i < \log n \) the lists of QT nodes are sorted according to their Morton codes. We assume the sorting algorithm with complexity \( O(k_i \log k_i) \) for a list length \( k_i \). In the worst case, \( k_j = N \) for all \( h_i \geq j \geq h \), so the complexity of this step is \( \sum_{h_i=1}^{\log_2 n} 4^i \log 4^i + \sum_{i=1}^{h_i} k_i \log k_i = O((\log_2 n - 1 - \log_4 N)N \log N) \).

4) Completing of the output: In this step, the complexity is proportional to the sum of list lengths \( k_i \), i.e. to the maximal size of the MQT format. In [3], it was derived

\[
O(N(1/3 + \log_4(n^2/N))).
\]

D. Discussion about parallelization

1) Parallelization of classical algorithm: In the Algorithm 4, all calls to subroutine INES (or INES2) can be proceed in parallel, but all these calls must be completed before calling of subroutine APPENDTOBITSTREAM. In OpenMP API, this can be done using TASK mechanism and taskwait command for the barrier. Since every creation of the new TASK has some overhead, it is more efficient to limit the number of TASKs. In our implementation, the new TASK is created only if \( (y_2 - y_1) > PLIMIT \), where \( y_1 \) and \( y_2 \) are coordinates of the submatrix and \( PLIMIT \) is some chosen constant (see Table III).

2) Parallelization of reducing algorithm: In this algorithm, the ability for the parallel execution differs:
   - construction of a sequence of reduced matrices: this step can not be done in parallel because one iteration depends on the previous ones.
   - construction of the list of QT nodes: this step can be done in parallel because all matrices \( M_i \) (and corresponding lists \( L_i \)) are independent.
   - reordering of the list of QT nodes: this step can be done in parallel because all lists \( L_i \) are independent.
   - completing of the output: this step can be done in parallel one bitstream from every list \( L_i \), then all created bitstreams are merged together.

III. EXPERIMENTAL RESULTS

A. Testing matrices
We have used 11 testing matrices from various application domains from the University of Florida Sparse Matrix Collection [14]. Table I shows the characteristics of the testing matrices.

B. HW and SW configuration

C. Cluster experiments
We have implemented all algorithms in C/C++ using the OpenMP API for evaluation of its performance and scalability. We have used 32-bit row/column indices and 32-bit floating point values. Experiments on a cluster were performed on the small university cluster called "star". Each node is an IBM BladeCenter module LS22 with the following configuration:

- 2x AMD Opteron 6C Processor Model 2435 2.6GHz/6MB L3 (12 computing cores)
- 8GB RAM PC2-6400 CL6 ECC DDR2 800 VLP RDIMM
- The operating system is Linux (64 bit distribution Gentoo) - version Gentoo 4.4.3-r2 p1.2.
- C compiler (gcc) and C++ compiler (g++) - version 4.4.3 with switches -O3.

D. Evaluation of results

1) Comparison of sequential algorithms: Table II shows the comparison of measured times for different algorithms for the transformation to the MQT format. From this table, we can conclude that version with INES2 subroutine is always faster.
complexity of the sparse matrix structure. We also design the parallel version of this algorithm. We performed experiments on the real parallel system and these experiments proved that the proposed algorithm allows an efficient conversion.

**ACKNOWLEDGMENT**

This research has been supported by GACR grant No. P202/12/2011.

**REFERENCES**


3.8 RP8: Space and execution efficient formats for modern processor architectures

In this paper (RP8), ideas from previous papers [RP1, RP5] were extended. For modern CPU architectures, the main bottleneck of sparse matrix-vector multiplication and transposed sparse matrix-vector multiplication is the limited memory bandwidth. In this paper, a new approach for these routines for modern processor architectures using a space efficient hierarchical format was introduced. This approach can significantly reduce the amount of transferred data from memory for almost all types of matrices arising from various application disciplines. This format represents a trade-off between space and execution efficiency as follows.

The simplest hierarchical format is the COOCAO format (for details see [RP3]). Since we store nonzero regions in the COO format, we need for each one its block row/column index into arrays $rX/rY$ and a pointer into its data, implemented as an index into the arrays of row/column indexes $inX/inY$ and values. In this paper, the idea of COOCAO format was generalized into formats with multiple levels. We call these formats multilevel hierarchical and denote them by $(COO_k)^l$, where:

- $k$ is the number of bits per level,
- $l$ is the number of levels.

In this paper, only 32-bit indexes were assumed, so $k \cdot l = 32$. The efficient multithreaded version of SpMV was discussed. Comparison of space complexity: COO format with $(COO_8)^4$, the latter one is always better (more space efficient). More exactly, the ratio between these complexities ranges from 50% to 80% (56% on the average).

Comparison of space complexity: CSR format with $(COO_8)^4$, the latter one is better for more than 93% of testing matrices. More exactly, the ratio between these complexities ranges from 67% to 109% (81% on the average).

These expected results follow from the fact that the hierarchical format is usually more space-efficient than common storage formats. The only one exception is a random non-banded sparse matrix due to the lack of locality.

Comparison of SpMV performance in the COO format with $(COO_8)^4$: the latter one is always better (more execution efficient). This follows from theoretical assumptions (better space complexity).
Comparison of SpMV performance in the CSR format with (COO₄₈)⁴: the latter one better (more execution efficient) only for 8% of testing matrices. This follows from the facts that the hierarchical format is usually more space efficient but the the code for the SpMV is more complex (e.g., uses more registers) than for the CSR format.

Comparison of fused SpMV and SpMTV performance in the COO format with (COO₄₈)⁴: the latter one is better (more execution efficient) for 70% of testing matrices. This follows from theoretical assumptions (better space complexity). On the other hand, the operations in the (COO₄₈)⁴ format suffers from the larger amount of conflict writes (coherence misses).

Comparison of the performance in the CSR format with (COO₄₈)⁴: the latter one better (more execution efficient) for 61% of testing matrices. This follows from the facts that the hierarchical format is usually more space efficient and all writes have better spatial locality (smaller amount of thrashing misses).

Based on these results, we can conclude that these routines using our format achieve higher performance for majority of testing matrices.
Abstract—Sparse matrix-vector multiplication (shortly SpMV) and transposed SpMV (shortly SpMTV) are the most common routines in the numerical linear algebra. Sparse storage formats describe a way how sparse matrices are stored in a computer memory. Since the commonly used storage formats (like COO or CSR) are not sufficient for high-performance computations, extensive research has been conducted about maximal computational efficiency of these routines. For modern CPU architectures, the main bottleneck of these routines is the limited memory bandwidth. In this paper, we introduce a new approach for these routines for modern processor architectures using a space efficient hierarchical format, which can significantly reduce the amount of transferred data from memory for almost all types of matrices arising from various application disciplines. This format represents a trade-off between space and execution efficiency. The performance of these routines with this format seems to be very close to the hardware limits.

Keywords—parallel execution; sparse matrix-vector multiplication; sparse matrix representation; OpenMP; high performance numerical linear algebra;

I. INTRODUCTION AND TERMINOLOGY

A. Motivation

The paper investigates space and execution efficient formats for modern processor architectures for very large sparse matrices. For the solution of partial differential equations (PDEs), e.g., arising from simulations of some physical phenomenon, iterative solvers are commonly used. The most time-consuming part of many iterative solvers based on the conjugate gradient method [1] is the multiplication \( \vec{y} = A \vec{x} \) of sparse matrix \( A \) by dense vector \( \vec{x} \) (SpMV). For solvers based on the biconjugate gradient method [1], the most time-consuming parts is the multiplication \( \vec{y} = A \vec{x} \) (SpMV) and \( \vec{y} = A^T \vec{x} \) (SpMTV); this pair of operations we denote as \textit{fused multiplication}. Due to matrix sparsity, the memory access patterns in common formats (like CSR) are irregular and the utilization of cache suffers from low spatial and temporal locality, so other formats are used in practice. Due to its overhead, the sparse matrix format transformation achieves good speedups only for large number of executions of SpMV or SpMTV with the same matrix \( A \); such a condition is typically satisfied for iterative solvers.

This paper introduces a new hierarchical format for storing sparse matrices that combines advantages of both space-efficient and execution-efficient formats.

B. General notation

We consider a matrix \( A \) of order \( n \times n \), \( A = (a_{ij}) \). The number of its nonzero elements is denoted by \( N \). Matrix \( A \) is considered \textit{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 \( 1 \ll n \ll N \ll n^2 \).
- The number of nonzero elements in submatrix \( B \) of matrix \( A \) is denoted by \( \eta(B) \), so \( \eta(A) = N \).
- For any submatrix \( B \), if \( \eta(B) = 0 \) then the submatrix \( B \) is called zero submatrix, otherwise it is called nonzero submatrix.
- If not stated otherwise, all space complexities are in bits.
- Elements of vectors and matrices are real number represented in a computer memory by a floating-point datatype, where \( S_F \) denotes the bitsize of this floating-point datatype.
- If not stated otherwise, all indexes are zero-based and represented by an integral data type, where \( S_I \) denotes the bitsize of this datatype. In this paper, we assume 32-bit indexes, so \( S_I = 32 \).
- If \( A \) has the same number of nonzero elements in each row then nonzero elements are distributed in \( A \) uniformly and we denote \( A \) as a \textit{uniform} matrix.
- The average number of nonzero elements per row is denoted by \( \text{avg}_{\text{per}\text{-}\text{row}} \). Obviously, \( \text{avg}_{\text{per}\text{-}\text{row}} = N/n \).
- The parameter \( th \) denotes the number of threads used for the execution of SpMV or SpMTV.

C. Banded matrices

Citing from Golub and Van Loan [2]:

\textit{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.

**D. The Coordinate (COO) Format**

The coordinate (COO) format is the simplest format for storing sparse matrices (see [3], [4]). 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.

\[S_{\text{COO}}(n, N) = N \cdot (2 \cdot S_t + S_F).\]

**E. The Compressed Sparse Row (CSR) Format**

The most common format for storing sparse matrices is the compressed sparse row (CSR) format (see [3], [4], [5], [6], [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 \(A\). The space complexity of the structure of matrix \(A\) in this format is

\[S_{\text{CSR}}(n, N) = N(S_t + S_F) + n \cdot S_t.\]

**F. SpMV algorithms for common formats**

1) **SpMV in the COO format:** Consider a sparse matrix \(A\) represented in COO format. The representation contains linear arrays values, xpos, and ypos as they are defined in Section I-D and a vector \(x\) represented by dense array \(x[0, \ldots, n-1]\). The number of FPU operations in one execution of SpMV in the COO format (spMV_COO) is \(2N\).

2) **SpMV in the CSR format:** Consider a sparse matrix \(A\) represented in CSR format. The representation contains linear arrays values, addr, and ci as they are defined in Section I-E and a vector \(x\) represented by dense array \(x[0, \ldots, n-1]\). The number of FPU operations in one execution of SpMV in the CSR format (spMV_CSR) is \(2N\). If we compare the algorithms for SpMV in the COO format and in the CSR format, the latter one seems to be more efficient due to two facts:
- The CSR format has a smaller space complexity if \(n < N\), so there is a smaller amount of read operations.
- The SpMV in the CSR format requires a smaller amount of write operations (\(N\) for the COO format, \(n\) for the CSR format).

3) **SpMTV in common formats:** The operation SpMTV is very similar to SpMV. Since both of them should be executed in one iteration of Biconjugate gradient method, we will not perform SpMTV separately, but as a fused multiplication. The big advantage of this approach is that the matrix \(A\) is read only once.

**G. Our assumptions and the requirements for a new format**

Our assumptions and the requirements for a new format are as follows:
- In this paper, we aimed at processing of large sparse matrices, therefore we assume that the space complexity (memory footprint) of the sparse matrix \(A\) is higher than the cache size. Consequently, the matrix \(A\) must be reloaded from the main memory for every execution of SpMV or SpMTV.
- For the designed format, there must also be fast and space-efficient algorithm for transformation of matrices from the common storage formats.
- For modern processor architectures, the main bottleneck of these routines is in the limited memory bandwidth. Experimental results [8] led us to modify the storage scheme in order to reduce the indexing overhead and the memory traffic. So, the new format should be space-efficient, because we try to keep the resulting memory footprint as low as possible.

Unfortunately, it is hard to satisfy all requirements at the same time, because they are generally in contradiction.

**H. State-of-art**

There are many papers discussing the efficiency of SpMV operation mainly in situations when at least one of the following conditions is true:

1) Formats and algorithms are designed for single-threaded execution.
2) Formats and algorithms are designed for banded matrices only.
3) Many authors assume that the matrix \(A\) has a known structure.
4) Many authors (such as [9], [10]) overlook the overhead of matrix transformation or design a time-expensive sparse matrix storage transformation (conversion).

The following approaches for acceleration of the SpMV operation are most common:
- Register blocking formats (e.g., [11], [7], [12]): These formats eliminate indirect addressing during the SpMV. Then, vector instructions can be used. These formats are suitable only for matrices with a known structure of nonzero elements. These formats are designed to handle randomly occurring dense blocks in a sparse matrix, but the efficiency
of SpMV operation with these formats depends strongly on the presence of these blocks.

- Optimization of cache utilization (e.g., [6], [13]): In these papers, we try to find a near-optimal matrix storage format to maximize the performance of the SpMV with respect to matrix transformation overhead and cache parameters.
- Other approaches for increasing the efficiency of the SpMV [14], [15], [12] are based on matrix reordering, sometimes combined with sw-pipelining [16].
- As far as we know, the most related papers are [17], [18], but the authors assume a recursive storage scheme and a very complex synchronization during the execution of SpMV. In contrast to them, we try to avoid all complex synchronization operations (mutexes etc.) because we plan to extend this work also for GPUs.

In our recent papers [19], [20], [21], [22], [23], we discussed space efficient formats (based on quadtree, binary tree, and arithmetic coding etc.). In these papers, we tried to design formats that minimize to reduce space complexity at all costs, these formats are designed only for the LOAD and SAVE operations for the parallel I/O on massively parallel computer systems with distributed memory architectures. Thus, these formats are inefficient in context of the SpMV or SpMTV operation.

II. HIERARCHICAL FORMATS

The memory requirements for usual sparse matrix formats are given by two index arrays of size $N$ in case of the COO format and by one array of size $N$ in case of the CSR format. Since explicit index arrays cannot be eliminated, we cannot reduce the number of elements of these index arrays, but we can reduce the number of bits per one row/column index [24]. The idea is to partition the matrix into square disjoint regions of size $2^c \times 2^c$ rows/columns, where $c \in \mathbb{N}^+$ is a formal parameter. Coordinates of the upper left corners of these regions are aligned to multiples of $2^c$. Thus, indexes of nonzero elements are separated in two parts, indexes of regions and indexes inside the regions. Every such a region has region row and region column indexes of size $S(\lceil n/2^c \rceil)$ bits. Let $R(c)$ denote the number of nonzero regions for matrix $A$. The minimal number of nonzero regions is equal to $R(c)_{\text{min}} = \left\lceil \frac{N}{2^c} \right\rceil$, if all nonzero regions contain only nonzero elements (i.e., are 100\% dense). The maximal number of nonzero regions is equal to $R(c)_{\text{max}} = \min \left\{ N, \left\lceil \frac{n}{2^c} \right\rceil^2 \right\}$, if each nonzero region contains exactly one nonzero element or if the whole matrix $A$ is covered by nonzero regions. For storing information about the regions and elements inside the regions, we can use the COO or CSR format. The CSR format looks like a better solution (the reasons are stated in Section I-F2), but we use for every level the COO format.

A. The COOCOO format

The simplest hierarchical format is the COOCOO format (see [24]). Since we store nonzero regions in the COO format, we need for each one its $2 \cdot c$-bit region row/column index into arrays $rX/rY$ and a pointer into its data, implemented as an index into the arrays of row/column indexes in $X/inY$ and values.

We generalize the idea of COOCOO format into formats with multiple levels. We call these formats hierarchical and denote them by $(COO_b)^l$, where:

- $k$ is the number of bits per level,
- $l$ is the number of levels.

In this paper, we assume 32-bit indexes, so $k \cdot l = 32$. In the “classical” COOCOO format, there is a memory overhead of pointers and linked lists. To make the resulting format space-efficient we eliminate this overhead by merging all items into two arrays:

- the array $a_{\text{nnz}}$ that contains the number of nonzero regions in this branch (subtree) and
- the array $a_{\text{xy}}$ that contains the part of coordinates of nonzero regions in this branch.

More exactly, the value stored in the item $a_{\text{nnz}}$ is the number of nonzero regions minus one since the number is always greater or equal to one. The items of arrays $a_{\text{nnz}}$ and $a_{\text{xy}}$ should $2 \cdot k$ bits long. In [21], similar 2-level hierarchical format are used and the optimal value of bits for each level are computed. To avoid this initial computational overhead and also bitwise manipulations during the SpMV, in this paper we study only the format $(COO_b)^4$.

B. Hierarchical format example

Let us assume a very small example of a sparse matrix with $n = 8$ and $N = 11, S_1 = 3$ and $Sp = 32$. Instead of the values of the matrix elements, we deal only with binary flags indicating the existence of nonzero elements.

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

The matrix in the $(COO_3)^3$ format is depicted on Figure 1. The matrix representation in this format consists of 3 levels, each level contains one bit of coordinates of nonzero elements (in array $a_{\text{xy}}$).
C. Space complexity

For common storage formats, the space complexity is given by equations in Sections I-D or I-E, so 
\[ S_{COO}(n, N) = 66 + 11 \cdot S_F \] and 
\[ S_{CSR}(n, N) = 65 + 11 \cdot S_F \] For the hierarchical formats, the exact size of the memory footprint cannot be derived from these global parameters, because it depends on the exact locations of nonzero elements. For the given matrix, the space complexity is 11·2[bits] for array \( a_{nnz} \), 21·2 for array \( a_{xy} \), and 11·S_F for values of nonzero elements.

The derivation of the lower and upper bounds on the size of the hierarchical format in a general case is relatively complex, so we derive them only for the \((COO_1)^3 \) format under assumption that \( n = 2^{32} \) and \( N = 2^{16} \).

1) Lower bound: We consider the best case: all nonzero elements are located in one \( 2^4 \times 2^4 \) region that is aligned to \( 2^4 \). So, the minimal space complexity of the this format (except values of nonzero elements) is \( (3 + 2^{16}) \cdot 16 \) for array \( a_{nnz} \) and \( (3 + 2^{16}) \cdot 16 \) for array \( a_{xy} \).

2) Upper bound: We consider the worst case: there is only one nonzero element in every \( 2^{24} \times 2^{24} \) region, so the maximal space complexity of this format (except values of nonzero elements) is \( (1 + 3 \cdot 2^{16}) \cdot 16 \) for array \( a_{nnz} \) and \( (4 \cdot 2^{16}) \cdot 16 \) for array \( a_{xy} \).

3) \( SpMV \) in the hierarchical format: Consider a sparse matrix \( A \) represented in \((COO_1)^2 \) format. The representation contains linear arrays \( a_{nnz} \) and \( a_{xy} \) as they are defined in Section II-A and a vector \( \bar{x} \) represented by dense array \( x[0, \ldots, n-1] \). A standard sequential algorithm for the sparse matrix-vector multiplication \( \bar{y} \leftarrow A \bar{x} \) can be described by the Algorithm 1.

The number of FPU operations in one execution of \( SpMV \) for the \((COO_1)^2 \) format is also \( 2N \), but in comparison to algorithms for the COO or CSR format it requires to load usually smaller amount of data.

### Algorithm 1 SpMV for the \((COO_1)^2 \) format

#### Input:
- \( A \) = matrix in the \((COO_1)^2 \) format
- \( x \) = array representing vector \( \bar{x} \)

#### Output:
- \( \bar{y} \) = array representing vector \( \bar{y} \)

1: \textbf{procedure} \textit{spMV}_{COO\_16\_2}(in \( A; x; \) out \( y \))
2: \( i_{xy} \leftarrow 0; \ i_{x} \leftarrow 0; \ m_{zz} \leftarrow a_{nnz}[0]; \ i_{nnz} \leftarrow 1; \)
3: \textbf{for} \ \( i_0 \leftarrow 0 \) to \( m_{zz} - 1 \) \textbf{do}
4: \( x_0 \leftarrow a_{xy}[i_{xy}]; x; \ y_0 \leftarrow a_{xy}[i_{xy}]; y; \ i_{xy} \leftarrow i_{xy} + 1; \ m_{zz} \leftarrow a_{nnz}[i_{nnz}]; \)
5: \( i_{nnz} \leftarrow i_{nnz} + 1; \)
6: \textbf{for} \ \( i_1 \leftarrow 0 \) to \( m_{zz} - 1 \) \textbf{do}
7: \( x_1 \leftarrow a_{xy}[i_{xy}]; x; x_t \leftarrow x_0 \cdot 2^{16} + x_1; \ y_t \leftarrow y_0 \cdot 2^{16} + y_1; \ i_{xy} \leftarrow i_{xy} + 1; \)
8: \( y[y_t] \leftarrow \text{values}[i_{val}] \cdot x[x_t]; \ i_{val} \leftarrow i_{val} + 1; \)
9: \textbf{return} \( y[]\)
and Fortran languages. These can be used to create portable parallel programs utilizing shared memory. The OpenMP is based on following two models:

a) Execution model: 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 parallel construct causes the thread to create a team of itself and several more additional threads and becoming master thread.

To coordinate tasks and data access in parallel regions, synchronization constructs and library routines can be used. These include constructs for critical sections, atomic operations or explicit barriers.

b) Memory model: 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.

B. Cache model

The cache model we consider corresponds to the structure of caches in the Intel x86 architecture. An s-way set-associative cache consists of h sets and one set consists of s independent blocks (called lines in the Intel terminology). Let $C_B$ denote the size of the data part of a cache in bytes and $B_S$ denote the cache block size in bytes. Then $C_S = s \cdot B_S \cdot h$.

We distinguish 3 types of cache misses:

1) Compulsory (sometimes called intrinsic) misses that occur when empty cache blocks are loaded with new data.

2) Thrashing misses (also called cross-interference or conflict misses) that occur when useful data are loaded into a cache block, but the cache associativity $s$ is too small, these data are replaced prematurely, and subsequently, they must be reloaded again.

3) Coherency misses that occur when a cache line that would otherwise be present in the thread’s cache has been invalidated by a write from another thread.

Due to the assumption that each new execution of SpMV starts with empty cache (this results from the first assumption in Section I-G), all $N$ elements of arrays $A$ and $c$ and all $n$ elements of arrays $x$, $adr$, $y$ must be loaded into the cache once and the number of compulsory misses is

$$N_{CM}^{comp} = N (S_F + S_I) + n (2 \cdot S_F + S_I)$$

For the execution of SpMV at least $N_{CM} \cdot B_S$ bytes must be transferred from/to main memory. The total number of cache misses is

$$N_{CM} = N_{CM}^{comp} + N_{CM}^{thr} + N_{CM}^{coh}.$$  

C. Parallel SpMV in common formats

1) The main idea: In general, there are the following ways how to parallelize SpMV in shared memory multithreaded environment:

- Assign the computation of each row to exactly one thread.
- Assign the computation of each row to more threads, so every thread computes temporary result. Then all these temporal result are merged into final result (vector $\vec{y}$) by the parallel reduction or by atomic operations.

In this paper, we assume for the sake of simplicity that the regions are consecutive chunks of rows. We also assume that each thread holds exactly one region.

2) Parallel SpMV in COO format: Parallelization of SpMV in COO format looks straightforward, we can simply put OpenMP directive \#pragma omp parallel for with option schedule(static) before main loop, so same-sized ($N/th$) chunks of iterations (rows) are assigned to threads. But in this solution atomic operation must be used (different threads can modify the same element of array $y$). To eliminate this drawback, the chunks of nonzero elements are aligned to beginnings of rows.

3) Parallel SpMV in CSR format: Parallelization of SpMV_CSR looks straightforward, we can simply put OpenMP directive \#pragma omp parallel for with option schedule(static) before loop over rows, so same-sized ($n/th$) chunks of iterations (rows) are assigned to threads. But this solution (we denote it as variant A) has two drawbacks:

1) It can result in a large number of thrashing misses because a cache must hold different parts of vector $x$ (in the worst case $th$-times more elements).

2) If the matrix $A$ is not uniform, then the loads for different threads are not balanced.

The solution of previous drawbacks looks easy, interleave executions of rows by different threads (multiplication of row 0 is executed by thread 0, row
1 is executed by thread 1, row \(th\) is executed by thread 0, etc.). It can be done by OpenMP option schedule(static,1), but this solution doesn’t guarantee a perfect load-balancing. Also so called "false sharing" [26] occurs. False sharing (sometimes called cache line ping-ponging) is the situation, where threads write to different memory addresses but those addresses are close enough in memory that they fall on the same cache line. Thus, the theoretically best solution (we denote it as variant B) is to assign rows to threads using the pragma schedule(dynamic,X), where X should be a multiple of \(BS/SP\) to avoid false-sharing.

4) Parallel SpMV in hierarchical format: The Algorithm 1 is not suitable for the parallel execution, it can be modified using some temporary arrays, but we lost the main advantage of space-efficiency. So, we decide to create this structure for each thread separately. We divide the hierarchical format that is equal to \(2^{k}\)-ary tree (the data structure for the whole matrix \(A\)) into forest (trees for the regions of matrix \(A\), each thread has its own root node).

D. Parallel fused multiplication

Similarly to sequential version (Section I-F3), we fuse SpMV and SpMTV operation. But in parallel version we cannot avoid write conflicts during execution of SpMTV, so these memory operations must be atomic.

IV. EVALUATION OF THE RESULTS

A. Used HW and SW

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

- \(2 \times\) 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().

2) Testing matrices: We have used 235 randomly generated testing matrices of three different types:

- general sparse matrix,
- banded sparse matrix,
- combination of general and banded matrix: each matrix consists of 3 consecutive groups of rows (submatrices): "upper” and "lower" groups are banded submatrices, the "central" group is a general sparse (non-banded) submatrix.

Testing matrices were generated for \(n \in \{3 \cdot 10^3, 1 \cdot 10^3, 3 \cdot 10^3\}\) and \(N \in \{1 \cdot 10^7, 3 \cdot 10^7, 1 \cdot 10^8\}\). It is very difficult to visualize the results for such huge number of testing matrices using graphs, so we present only the most important statistical information.

3) Evaluation of space complexities: If we compare space complexity of the COO format and (COO\(_8\))^4, the latter one is always better (more space efficient). More exactly, the ratio between these complexities ranges from 50% to 80% (56% on the average).

If we compare space complexity of the CSR format and (COO\(_8\))^4, the latter one is better for more than 93% of testing matrices. More exactly, the ratio between these complexities ranges from 67% to 109% (81% on the average).

These expected results follow from the fact that the hierarchical format is usually more space-efficient than common storage formats. The only exception is a random non-banded sparse matrix due to the lack of locality (see Section II-C).

4) Evaluation of SpMV performance: The performance results are shown in Table I. If we compare SpMV performance in the COO format and (COO\(_8\))^4, the latter one is always better (more execution efficient). This follows from theoretical assumptions (better space complexity).

If we compare SpMV performance in the CSR format variant A and B (see Section III-C3), the latter one is better (more execution efficient) for 86% of testing matrices. This follows from the better load-balancing and cache utilization. On the other hand, the variant B suffers from the larger overhead of dynamic scheduling strategy.

If we compare SpMV performance in the CSR format and (COO\(_8\))^4, the latter one better (more execution efficient) only for 8% of testing matrices. This follows from the facts that the hierarchical format is usually more space efficient but the the code for the SpMV is more complex (e.g., uses more registers) than for the CSR format.

5) Comparison with related works: We compare the SpMV performance of our implementation and implementation from Yzelman [9]. This implementation (namely Sparse Library v1.6.0) supports about 15 storage schemes, but only 5 of them are designed for multi-threaded execution. The SpMV performance results are shown in Table II. For some testing cases, our implementation (CSR format, variant B) is the best, in other cases Row-distributed block CO-H format or Hilbert format achieve the higher performance. Thus, we declare both implementations as comparable for SpMV operation. The current version of Sparse Library also supports SpMTV operation, but the fused multiplication is not
The performance results are shown in Table I. If we compare fused SpMV and SpMTV performance in the COO format and \((\text{COO}_8)^4\), the latter one is better (more execution efficient) for 70% of testing matrices. This follows from theoretical assumptions (better space complexity). On the other hand, the operations in the \((\text{COO}_8)^4\) format suffers from the larger amount of conflict writes (coherence misses).

If we compare the performance in the CSR format variant A and B (see Section III-C3), the latter one is better (more execution efficient) for 51% of testing matrices. This follows from the better load-balancing and cache utilization. On the other hand, the variant B suffers from the larger amount of conflict writes (coherence misses).

If we compare the performance in the CSR format and \((\text{COO}_8)^4\), the latter one better (more execution efficient) for 61% of testing matrices. This follows from the facts that the hierarchical format is usually more space efficient and all writes have better spatial locality (smaller amount of thrashing misses).

### 7) Evaluation of practical applicability

The transformation into another format takes some overhead. A useful question is whether the transformation into different formats pays off in real cases. To answer it, we define the parameter Payoff.

\[
\text{Payoff} = \frac{T_{\text{transf}}}{T_{\text{old}} - T_{\text{new}}}, \quad T_{\text{old}} > T_{\text{new}}
\]

It denotes the number of executions of the SpMV to amortize the overhead of the matrix format transformation. Currently we use a very simple transformation algorithm that is efficient only for banded matrices. In the current state of the project, the value of parameter Payoff have impractically large values (from about 100 to 1000).

### V. Conclusions

We have presented a new hierarchical storage format that is designed for parallel sparse matrix-vector multiplication and transposed sparse matrix-vector multiplication. This format combines advantages of the space-efficient formats and execution-efficient formats specialized for these routines. We have measured the performance and space complexity for a large number of testing matrices. Based on the analysis of preliminary results, we can conclude that these routines using our format achieve higher performance for majority of testing matrices.

### Future work

The project is under development, we have identify the following areas for further research:

- Optimization of transformation algorithm: We have already designed a new transformation algorithm for conversion to hierarchical format. Preliminary results show that this algorithm is about one magnitude faster than the previous one, so the value of the parameter Payoff become acceptable.

- Optimization for other processor architectures: We want to extend and optimize this format for other processor architectures (including GPUs and Intel MIC).

- Improve cache utilization: We plan to extend this study further by using so-called Morton order (for details see [27]). If the nodes in the hierarchical formats are reordered according to the Morton order.
order, the spatial and temporal locality should be improved.

ACKNOWLEDGEMENTS

This research has been supported by CTU internal grant SGS No. SGS15/120/OHK3/1T/18.

REFERENCES


3.9 Summary

As was already said, the listed papers are divided into two (not disjoint) groups: The first group contains 5 papers (RP2, RP3, RP4, RP6, and RP8) that deal with design of space-efficient formats for sparse matrix storage. We can divide this group further according to the following criteria:

- the purpose of the format:
  - formats for storing large sparse matrices suitable for parallel I/O systems: ACB, BHF, AHF, MQT, MBT, CQT, CBT formats (described in RP2, RP3, RP4, and RP6).
  - a format for acceleration of basic numeric algebra routines: \((COO_8)^4\) (described in RP8)

- the principle of the format:
  - tree based: MQT, MBT, CQT, CBT formats (described in RP4, RP6)
  - hierarchically based:
    * 2-level: BHF, AHF formats (described in RP2, RP3)
    * multi-level: \((COO_8)^4\) format (described in RP8)
  - arithmetic coding based: the ACB format (described in RP4)

The second group contains 4 papers (RP1, RP5, RP7, and RP8) that deal with design of execution-efficient algorithms for sparse matrices. We can divide this group further according to the following criteria:
• if the paper is focused only on sparse matrix-vector multiplication:
  – no, other operations are also discussed: (other sparse matrix operations for numerical linear algebra in RP1, transformation algorithm to the MQT format in RP7)

• if parallelization of the algorithm is also discussed:
  – no: RP1.
  – yes:
    * parallelization in a shared memory environment (RP7, RP8).
    * parallelization in a distributed memory environment (RP5).

All listed papers represent a significant contribution to fields of sparse matrix formats and related algorithms. Interrelationship of the topics of individual relevant papers as described above is shown in Fig. 3.1.
Figure 3.1: Interrelationship of the topics of individual relevant papers.
4 Conclusions

Large sparse matrices are widely used in high-performance computing. These matrices due to their sizes are usually stored and processed by parallel computer systems. In Chapter 1, “Theoretical background and survey of the state-of-the-art”, two main topics of this thesis were mentioned.

The first main topic was motivated by the fact that a parallel I/O subsystem is typically the main performance bottleneck in computation with large sparse matrices, e.g., loading or storing of large sparse matrices from/to a distributed file system can take significant amounts of time. Weaknesses of the previously developed solutions for space-efficient formats for storage of large sparse matrices were discussed. Reducing the space complexity of the representation of large sparse matrices resulted in reduced time of parallel I/O. New formats for storage of large sparse matrices suitable for parallel I/O systems were designed. In particular, the first new formats were from a large family of hierarchical formats (BH and AH), the next format was arithmetical coding based (ACB) format, one new format was quadtree-based (MQT format), one new format was based on binary tree (MBT format), and the last two formats were compressed variants of the previous two (the CQT and CBT format).

We performed experiments with matrices arising in many different application areas and compared them with widely used COO or CSR formats. These experiments proved that our new formats could significantly reduce the space complexity of these matrices and consequently reduce amount of data needed for storing these matrices. Low space complexity of these formats made them good candidates for storage of large sparse matrices using parallel I/O systems.

The second main topic of this thesis was motivated by the fact that some algorithms for sparse matrices do not utilize CPU resources efficiently, e.g., the memory subsystem bandwidth is typically the main bottleneck for sparse matrix-vector multiplication. In particular, some very basic routines from linear algebra using an advanced quadtree format were implemented. My implementation using this storage format showed significant speedups for some matrix operations (multiplication of a sparse matrix by a sparse matrix or getting a value of a given element and so on). It resulted from the fact that the quadtree data structure combined advantages of sparse and dense data formats.

Another result for this topic was the design of a new algorithm for converting of the sparse
matrices from common formats to the quadtree-based formats. The parallel version of this algorithm was also introduced.

Another result for this topic was a discussion about efficient distributed sparse matrix-vector multiplication (SpMV). The new general quality measures for this operation were also introduced. Some common matrix-processor mappings (including a new one) were discussed. Algorithms for the redistribution and the upper bounds derivation of quality measures were also described. From these results we could have concluded that the SpMV algorithm using our new mapping had been scalable for almost all matrices arising from various technical areas.

The last result for this topic was design of a new hierarchical storage format that was aimed at efficient parallel SpMV and transposed SpMV. This format combined advantages of space-efficient formats and execution-efficient formats specialized for these routines. Algorithms using presented format achieved higher performance for majority of testing matrices than those using common formats.

All advances presented in this work concern both theoretical and practical areas.
Symbols, Acronyms, Code index, etc.

Acronyms

All acronyms are defined when first used in the text, with the exception of frequently used ones. All acronyms (mainly formats) are indexed (mainly at verb ‘format’).

Symbols

\( A, \ A \) sparse matrix of order \( n \)

\( B_S \) cache block size in bytes

\( DC_S \) the size of the data part of a cache in bytes

\( h \) number of sets in the cache

\( n \) order of matrix, vector

\( \text{avg\_per\_row} \) the average number of nonzero elements per row

\( P \) the number of processors

\( p_{1...P} \) processors of a given massive parallel computer

\( s \) the number of independent blocks in one cache set

\( S_D \) the size of type \texttt{double} in bytes

\( S_I \) the size of type \texttt{integer} in bytes

\( th \) the number of used threads

\( \omega(A) \) bandwidth of the matrix

\( \bar{x} \) an \( n \)-vector of unknown values
Index

ACB (arithmetical-coding-based), 16, 69
AH (advanced hierarchical), 23, 42
BH (basic hierarchical), 22, 42
cache, 3
  block, 4
  hit, 4
  hit ratio, 4
  line, 4
  miss, 4
    coherency, 4
    compulsory, 4
    thrashing, 4
  miss ratio, 4
  set, 4
CBT (compressed binary), 20, 69
coherency cache miss, 4
compulsory cache miss, 4
COO (Coordinate), 10
CQT (compressed quadtree), 20, 69
CSC (Compressed Sparse Column), 11
CSR (Compressed Sparse Row), 10
EB (entropy-based), 16, 42

format
  advanced hierarchical, 23, 42
  advanced quadtree, 26, 29
  arithmetical-coding-based (ACB), 16, 69
  basic hierarchical, 22, 42
  compressed binary (CBT), 20, 69
  compressed quadtree (CQT), 20, 69
Compressed Sparse Column (CSC), 11
Compressed Sparse Row (CSR), 10
COOCOO, 22, 42
COOCOO256, 21, 37
COOCSR, 22, 42
COOCSR256, 21, 37
Coordinate (COO), 10
CSRCOO, 22, 42
CSRCSR, 22, 42
entropy-based (EB), 16, 42
hierarchical, 21, 37
minimal binary (MBT), 18, 69
minimal quadtree (MQT), 17, 51
multilevel hierarchical, 24, 102
space optimal, 16, 37
sparse storage (SSF), 6

locality
  spatial, 4
  temporal, 4

LSM (large sparse matrix), 6

matrix
  banded, 2
  diagonal, 2
  large sparse (LSM), 6
  nonzero, 1
  regular, 1
  singular, 1
  sparse, 1
  structural symmetric, 2
  symmetric, 1

118
uniform, 2
zero, 1
MBT (minimal binary), 18, 69
memory
hierarchy, 3
MPCS (massive parallel computer system), 2
MPI (Message Passing Interface), 6
MQT (minimal quadtree), 17, 51
OpenMP, 5
QT (quadtree), 17, 51
spatial locality, 4
spMTV (sparse transposed matrix-vector multiplication), 9
spMV (sparse matrix-vector multiplication), 9
SSF (Sparse storage format), 6
temporal locality, 4
thrashing cache miss, 4
APPENDIX A. RELEVANT PUBLICATIONS OF THE AUTHOR

A Relevant publications of the author


B Refereed co-authored publications


C Unrefereed important publications of the author

APPENDIX D. OTHER REFEREED PUBLICATIONS


D Other refereed publications


