DOI 10.12694/scpe.v15i1.962
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 1, pp. 1–20. http://www.scpe.org © 2014 SCPE

TREE-BASED SPACE EFFICIENT FORMATS FOR STORING THE STRUCTURE OF SPARSE MATRICES [*]

I. ŠIMEČEK[†] D. LANGR[†] AND P. TVRDÍK[†]

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 (MPCSs) 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 bellow 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 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 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

[*]This research has been supported by GACR grant No. P202/12/2011.

[†]Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00, Praha, Czech Republic. E-mail: mailto:xsimecek,langrd@fit.cvut.cz.

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 *avg_per_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 $nnz(B)$.
- 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 ξ elements indexed from 0 to $\xi - 1$. The minimum number of bits of an *unsigned* indexing data type is

$$S^{\mathrm{MIN}}(\xi) = \left\lceil \log_2 \xi \right\rceil.$$

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

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

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

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

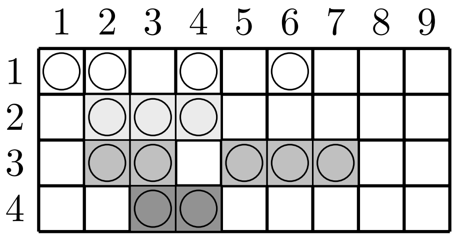
When we describe a matrix storage format, we use simply $S(\xi)$ instead of $S^{\mathrm{MIN}}(\xi)$.

2. State-of-the-art.

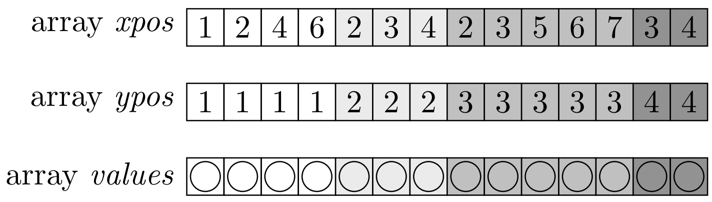
2.1. The Coordinate (COO) Format.

The coordinate (COO) format is the simplest SSF (see [19, 3]). The matrix A is represented by three linear arrays *values*, *xpos*, and *ypos* (see Fig. 2.1 (b)). The array *values*$[1, \ldots, N]$ stores the nonzero values of A, arrays *xpos*$[1, \ldots, N]$ and *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 *values* is not counted) of this format is

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



(a) An example of the sparse matrix

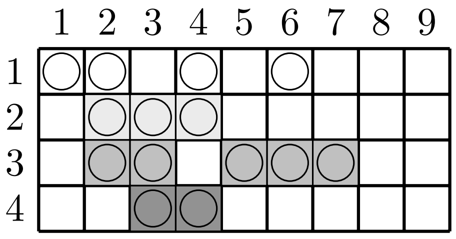

array *xpos* 1 2 4 6 2 3 4 2 3 5 6 7 3 4
array *ypos* 1 1 1 1 2 2 2 3 3 3 3 3 4 4
array *values*

(b) Representation of this matrix in the COO format

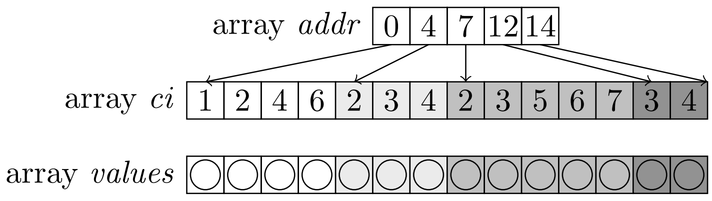
FIG. 2.1. *An example of representation of sparse matrix in the COO format*

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 *values*, *addr*, and *ci* (see Fig. 2.2 (b)). The array $values[1, \ldots, N]$ stores the nonzero elements of A, the array $addr[1, \ldots, n+1]$ contains indexes of initial nonzero elements of rows of A; if row i does not contain any nonzero element, then $addr[i] = addr[i+1]$. It is obvious that $addr[1] = 1$ and $addr[n+1] = N$. The array $ci[1, \ldots, N]$ contains column indexes of nonzero elements of A. Hence, the first nonzero element of the row j is stored at index $addr[j]$ in array *values*. The space complexity of the structure of matrix A (array *values* is not counted) in this format is

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



(a) An example of the sparse matrix



(b) Representation of this matrix in the CSR format

FIG. 2.2. *An example of representation of sparse matrix in the CSR format*

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 SPARSITY [6] or [16] or CARB [20, 10]) 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.

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].

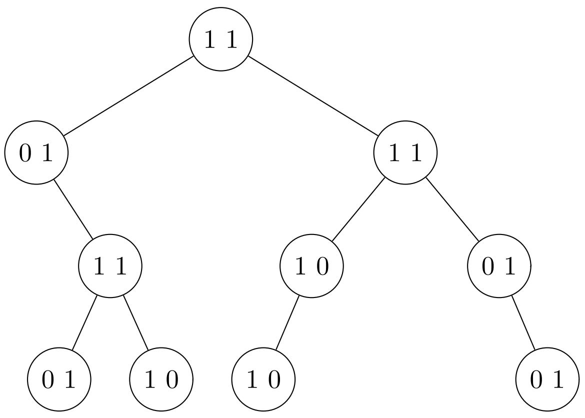


Fig. 3.1. *Visualization of the binary tree from the Example in Sect. 3.2.1.*

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:

$$\begin{aligned} S_{\text{ACB}}(n,N) &= -M \cdot (\frac{M-N}{M} \log_2 \frac{M-N}{M} + \frac{N}{M} \log_2 \frac{N}{M}) \\ &= -(M-N) \log_2 \frac{M-N}{M} - N \log_2 \frac{N}{M} \\ &= -(M-N) \log_2 (M-N) + M \log_2 M - N \log_2 N. \end{aligned}$$

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:

$$\begin{aligned} S_{\mathrm{ACB}}(n,N) &\approx \frac{N}{\ln 2} + N\log_2 M - \frac{N^2}{M\cdot \ln 2} - N\log_2 N \\ &\approx N\cdot\left(\frac{1}{\ln 2} + \log_2 M - \log_2 N\right) \\ &\approx N\cdot\left(\frac{1}{\ln 2} + 2\cdot\log_2 n - \log_2 N\right). \end{aligned}$$

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.

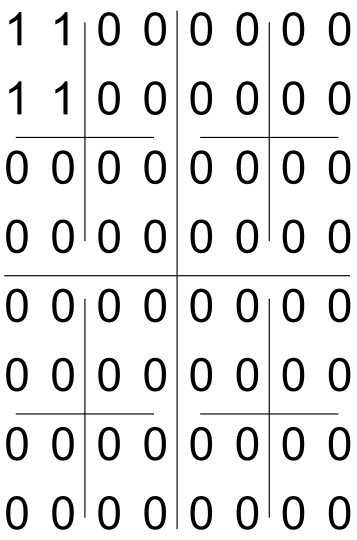
If a FBT is used as a basis for SSF, it describes a partition of the sparse matrix into submatrices and each node in the FBT represents a submatrix. Equally as in k-d trees, see [18], the decomposition is performed in alternating directions: first horizontally, then vertically, and so on. In other words, 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.

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.

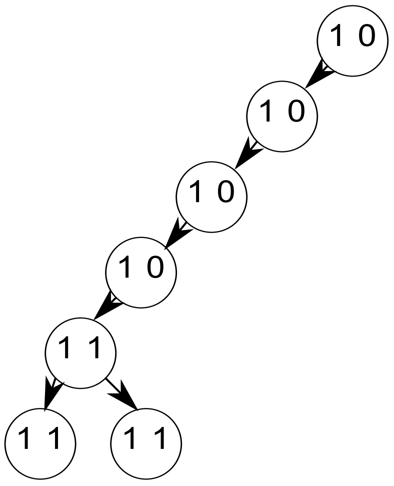
- All nodes of a MBT are stored in one array (or stream). Since the size of the input matrix is given, we can compute locations of all child nodes, we can omit pointers *left*, *right*.
- All nodes of a MBT contain only two flags (it means only two bits). Each of them is set to 1 if the corresponding 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.

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



(a) Original sparse matrix A.



(b) The MBT representation of matrix A.

FIG. 3.2. *The MBT with the minimal number of nodes.*

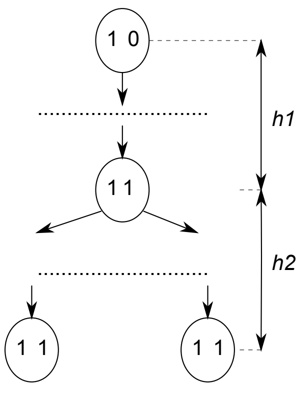
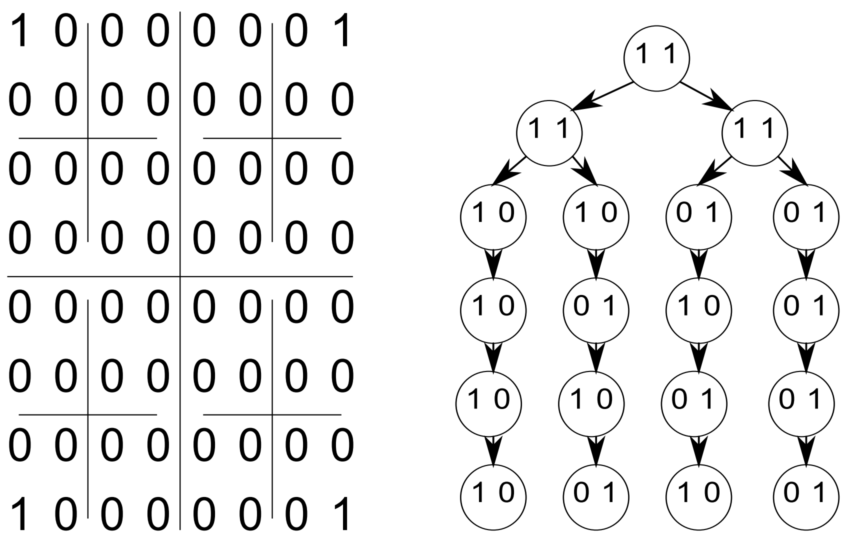


FIG. 3.3. *MBT with the minimal number of nodes (the number of leaves is $N/2$).*

shown in Fig. 3.1.

$$
\begin{aligned}
S = \mathrm{MBT}(A) = \mathrm{MBT}\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \\
= \text{"11"} + \mathrm{MBT}\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} + \\
+ \mathrm{MBT}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \\
= \text{"11"} + \text{"01"} + \text{"11"} + \mathrm{MBT}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + \\
+ \mathrm{MBT}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \mathrm{MBT}\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \\
= \text{"11"} + \text{"01"} + \text{"11"} + \text{"11"} + \text{"10"} + \text{"01"} + \\
+ \mathrm{MBT}(\text{"01"}) + \mathrm{MBT}(\text{"10"}) + \mathrm{MBT}(\text{"10"}) + \mathrm{MBT}(\text{"01"}) = \\
= \text{"11"} + \text{"01"} + \text{"11"} + \text{"11"} + \text{"10"} + \\
+ \text{"01"} + \text{"10"} + \text{"10"} + \text{"01"}
\end{aligned}
$$

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



(a) Original sparse matrix A.

(b) The MBT representation of matrix A.

FIG. 3.4. *The MBT with the maximal number of nodes.*

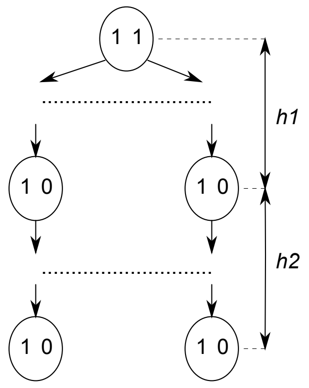


FIG. 3.5. *MBT with the maximal number of nodes (the number of leaves is $N/2$).*

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 $S_{\mathrm{COO}}(n, N) = 24[bits]$ and $S_{\mathrm{CSR}}(n, N) = 28[bits]$. 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 $h = h1 + h2 = 2\log_2 n - 1$, where $h2 = \log_2 N - 1$ and $h1 = 2\log_2 n - \log_2 N$.

- All nodes with height $< h1$ (in upper $h1$ levels) contain exactly one 1 (they have one child node). The number of nodes in these levels is $h1$,
$$h1 = \log_2(n^2/N).$$
- All nodes with height $\geq h1$ (in lower $h2$ 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)
$$\sum_{i=h1+1}^{2\log_2 n-1} 2^{i-(h1+1)} = N-1.$$

So, the minimal size of the MBT format is
$$2 \cdot \bigl(N - 1 + \log_2(n^2/N)\bigr).$$

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 10 01 10 10 01 01 10 01 10 01".

- The height of this tree is $h = h1 + h2 = 2\log_2 n - 1$, where $h1 = \log_2 N - 1$.
- All nodes with height $< h1$ (in upper $h1$ levels) are full of 1's (they have two child nodes), $h1 = \log_2 N - 1$. The number of nodes in these levels is approximately
$$\sum_{i=0}^{h1-1} 2^i = N-1.$$
- 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
$$N \cdot h2 = N \cdot (2\log_2 n - \log_2 N) = N \cdot \log_2(n^2/N).$$

So, the maximal size of the MBT format is
$$\approx 2 \cdot N\bigl(1 + \log_2(n^2/N)\bigr).$$

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 $O(\log_2 n \cdot (y2 - y1 + 1))$. Fortunately, the average complexity is much lower (it depends on the value of the parameter *avg_per_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 AppendToBitstream is $\Theta(1)$. Procedure `INES`(A,$x1$,$y1$,$x2$,$y2$) is called for every node in the MBT in the output stream S two times.

- For nodes with height $= h1$: The number of these nodes is N, the expression $(y2 - y1 + 1)$ is equal to $1 + n/\sqrt{N}$. Time complexity of the transformation for all nodes with this depth is $T_{h1} = N \cdot (1 + n/\sqrt{N}) \cdot \log_2 avg\_per\_row$.
- For nodes with height $= h1 - 1$: The number of nodes is $N/2$ 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 $T_{upper} \approx \sum_{i=0}^{h1} T_{h1}/2^{(i-h1)} = \Theta(N \cdot (1 + n/\sqrt{N}) \cdot \log_2 avg\_per\_row)$.
- For nodes with height $> h1$: The time complexity of the transformation for all these nodes (for the lower $h2$ levels) is $T_{lower} \approx \sum_{i=h1+1}^{h} T_{h1}/2^{(i-h1)} = \Theta(N(1 + n/\sqrt{N}) \cdot \log_2 avg\_per\_row)$.

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

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 $\Theta(n^{3/2})$.

Algorithm 1 Procedure to test if the given submatrix is nonempty

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

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.

Algorithm 2 Transformation algorithm to the MBT format

```
 1: procedure TR2MBT($A$)
Input: $A$ = the matrix for the transformation in CSR format
Output: $S$ = the bitstream representing the input matrix in the MBT format
 2:     $current \leftarrow ()$
 3:     enqueue $\{1, 1, A.n, A.n, 0\}$ into $current$
 4:     while $current$ is not empty do
 5:         dequeue $\{x1, y1, x2, y2, h\}$ from $current$
 6:                                         ▷ $x1$,$y1$,$x2$,$y2$ = coordinates of submatrix
 7:                 ▷ $h$ = current BFS level, divide rows ($h$ is odd) or columns
 8:         if $h$ is even then
 9:             $mx \leftarrow x2$;    $my \leftarrow (y1 + y2)/2$
10:             $lx \leftarrow x1$;    $ly \leftarrow (y1 + y2)/2 + 1$
11:         else
12:             $mx \leftarrow (x1 + x2)/2$;    $my \leftarrow y2$
13:             $lx \leftarrow (x1 + x2)/2 + 1$;    $ly \leftarrow y1$
14:         end if
15:         $l1 \leftarrow \mathrm{INES}(A, x1, y1, mx, my)$
16:         $l2 \leftarrow \mathrm{INES}(A, lx, ly, x2, y2)$
17:         APPENDTOBITSTREAM($S, l1$)
18:         APPENDTOBITSTREAM($S, l2$)
19:         if $l1 = true$ then
20:             enqueue $\{x1, y1, mx, my, h + 1\}$ into $current$
21:         end if
22:         if $l2 = true$ then
23:             enqueue $\{lx, ly, x2, y2, h + 1\}$ into $current$
24:         end if
25:     end while
26:     return $S$
27: end procedure
```

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.

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

$$= \text{”11”} + \mathrm{CBT}\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} +$$

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

$$= \text{”11”} + \text{”}\mathbf{0}\text{”} + \text{”11”} + \mathrm{CBT}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} +$$

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

$$= \text{”11”} + \text{”}\mathbf{0}\text{”} + \text{”11”} + \text{”11”} + \text{”10”} + \text{”}\mathbf{0}\text{”} +$$

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

$$= \text{”11”} + \text{”}\mathbf{0}\text{”} + \text{”11”} + \text{”11”} + \text{”10”} +$$

$$+ \text{”}\mathbf{0}\text{”} + \text{”}\mathbf{0}\text{”} + \text{”10”} + \text{”10”} + \text{”}\mathbf{0}\text{”}$$

Algorithm 3 Transformation algorithm to the CBT format

```
 1: procedure TR2CBT($A$)
Input: $A$ = the matrix for the transformation in CSR format
Output: $S$ = the bitstream representing the input matrix in the CBT format
 2:     $current \leftarrow ()$
 3:     enqueue $\{1, 1, A.n, A.n, 0\}$ into $current$
 4:     while $current$ is not empty do
 5:         dequeue $\{x1, y1, x2, y2, h\}$ from $current$
 6:                                 ▷ $x1$,$y1$,$x2$,$y2$ = coordinates of submatrix
 7:             ▷ $h$ = current BFS level, divide rows ($h$ is odd) or columns
 8:         if $h$ is even then
 9:             $mx \leftarrow x2$;    $my \leftarrow (y1 + y2)/2$
10:             $lx \leftarrow x1$;    $ly \leftarrow (y1 + y2)/2 + 1$
11:         else
12:             $mx \leftarrow (x1 + x2)/2$;    $my \leftarrow y2$
13:             $lx \leftarrow (x1 + x2)/2 + 1$;    $ly \leftarrow y1$
14:         end if
15:         $l2 \leftarrow false$
16:         $l1 \leftarrow \text{INES}(A, x1, y1, mx, my)$
17:         APPENDTOBITSTREAM($S, l1$)
18:         if $l1 = true$ then
19:             $l2 \leftarrow \text{INES}(A, lx, ly, x2, y2)$
20:             APPENDTOBITSTREAM($S, l2$)
21:         end if
22:         if $l1 = true$ then
23:             enqueue $\{x1, y1, mx, my, h + 1\}$ into $current$
24:         end if
25:         if $l2 = true$ then
26:             enqueue $\{lx, ly, x2, y2, h + 1\}$ into $current$
27:         end if
28:     end while
29:     return $S$
30: end procedure
```

For storing matrix A in the CBT format only 16 bits are needed (instead of 20 bits in the MBT format).

MQT	CQT	MBT	CBT
0000	N/A	00	N/A
0001	**000**	01 01	**0** **0**
0010	0010	01 10	**0** 10
0011	0011	01 11	**0** 11
0100	0100	10 01	10 **0**
0101	0101	11 01 01	11 **0** **0**
0110	0110	11 01 10	11 **0** 10
0111	0111	11 01 11	11 **0** 11
1000	1000	10 10	10 10
1001	1001	11 10 01	11 10 **0**
1010	1010	11 10 10	11 10 10
1011	1011	11 10 11	11 10 11
1100	1100	10 11	10 11
1101	1101	11 11 01	11 11 **0**
1110	1110	11 11 10	11 11 10
1111	1111	11 11 11	11 11 11

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:         dequeue $\{x1, y1, x2, y2, h\}$ from $current$
6:                                         ▷ $x1$,$y1$,$x2$,$y2$ = coordinates of submatrix
7:                   ▷ $h$ = current BFS level, divide rows ($h$ is odd) or columns
8:         if $x1 = x2$ & $y1 = y2$ then
9:             $X \leftarrow$ new nonzero element $(x1, y1)$
10:            add $X$ to $A$
11:        else
12:            if $h$ is even then
13:                $mx \leftarrow x2$;   $my \leftarrow (y1 + y2)/2$
14:                $lx \leftarrow x1$;   $ly \leftarrow (y1 + y2)/2 + 1$
15:            else
16:                $mx \leftarrow (x1 + x2)/2$;   $my \leftarrow y2$
17:                $lx \leftarrow (x1 + x2)/2 + 1$;   $ly \leftarrow y1$
18:            end if
19:            $l1 \leftarrow$ READONEBIT($S$)
20:            if $l1 = false$ then
21:                $l2 \leftarrow true$                              ▷ hidden one
22:            else
23:                $l2 \leftarrow$ READONEBIT($S$)
24:            end if
25:            if $l1 = true$ then
26:                enqueue $\{x1, y1, mx, my, h + 1\}$ into $current$
27:            end if
28:            if $l2 = true$ then
29:                enqueue $\{lx, ly, x2, y2, h + 1\}$ into $current$
30:            end if
31:        end if
32:    end while
33:    return $A$
34: end procedure
```

4. Parallel execeution 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 SSFs based on trees the parallelization is more complicated.

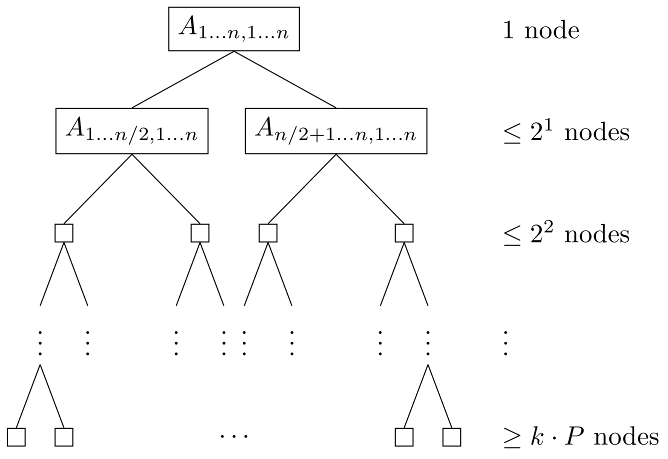


FIG. 4.1. *The main idea of parallelization of MBT or CBT transformation.*

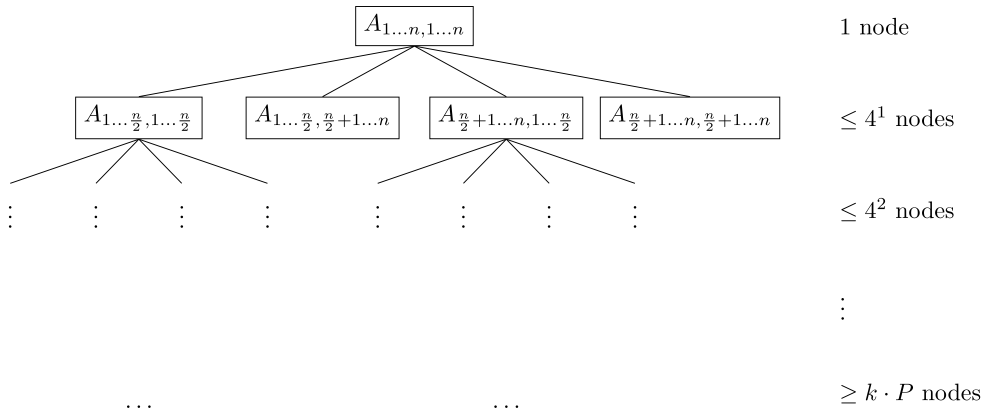

$A_{1\dots n,1\dots n}$
1 node
$A_{1\dots\frac{n}{2},1\dots\frac{n}{2}}$
$A_{1\dots\frac{n}{2},\frac{n}{2}+1\dots n}$
$A_{\frac{n}{2}+1\dots n,1\dots\frac{n}{2}}$
$A_{\frac{n}{2}+1\dots n,\frac{n}{2}+1\dots n}$
$\leq 4^1$ nodes
$\leq 4^2$ nodes
$\geq k \cdot P$ nodes

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 *start_row*. In this array, the value *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 a 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 MBT 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 $1 + (i-1)\lceil C/P \rceil$ and ends at $\min(i\lceil C/P \rceil, C)$. 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 MBT format independently and stores them into a separate file.

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

```
1: 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:     one-to-all broadcast $\{x1, y1, x2, y2\}$
3:     $output = false$
4:     for $j \leftarrow y1, y2$ do
5:         for $i \leftarrow A.Addr[j], A.Addr[j+1] - 1$ do
6:             if $(x1 \geq A.Ci[i] \leq x2)$ then
7:                 $output = true$
8:                 break
9:             end if
10:        end for
11:    end for
12:    send predicate $output$ to parallel reduction
13:    $po \leftarrow$ parallel reduction of $output$ using logical OR
14:    return $po$
15: end procedure
```

Matrix	n	N	*avg_per_row*
circuitM5	$5.56 \cdot 10^6$	$5.95 \cdot 10^7$	$1.93 \cdot 10^{-6}$
nlpkkt120	$3.54 \cdot 10^6$	$5.02 \cdot 10^7$	$4.01 \cdot 10^{-6}$
ldoor	$9.52 \cdot 10^5$	$2.37 \cdot 10^7$	$2.60 \cdot 10^{-5}$
TSOPF_RS_b2383	$3.81 \cdot 10^4$	$1.62 \cdot 10^7$	$1.10 \cdot 10^{-2}$
mouse_gene	$4.51 \cdot 10^4$	$1.45 \cdot 10^7$	$7.10 \cdot 10^{-3}$
t2em	$9.25 \cdot 10^5$	$4.59 \cdot 10^6$	$5.36 \cdot 10^{-6}$
bmw7st_1	$1.41 \cdot 10^5$	$3.74 \cdot 10^6$	$1.88 \cdot 10^{-4}$
amazon0312	$4.01 \cdot 10^5$	$3.20 \cdot 10^6$	$2.00 \cdot 10^{-6}$
thread	$2.97 \cdot 10^4$	$2.25 \cdot 10^6$	$2.55 \cdot 10^{-3}$
gupta2	$6.21 \cdot 10^4$	$2.16 \cdot 10^6$	$5.60 \cdot 10^{-4}$
c-29	$5.03 \cdot 10^3$	$2.44 \cdot 10^4$	$9.64 \cdot 10^{-4}$

Table 4.1
Characteristics of the testing matrices.

Algorithm 6 Distributed procedure to test if the given submatrix is nonempty

```
1: procedure PARINES2($A$,$x1$,$y1$,$x2$,$y2$)
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:     construct $G'$
3:     multicast $\{x1, y1, x2, y2\}$ 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 $(si > y2)$ OR $(si1 \leq y1)$ then
8:         $output = false$
9:     else
10:        for $y \leftarrow \max(y1, si), \min(y2, si1-1)$ do
11:            $low \leftarrow A.addr[y]$;   $high \leftarrow A.addr[y+1]$
12:            $i \leftarrow$ BINARY SEARCH($in\ array\ A.ci$)
13:                                              ▷ within indexes from $\langle low \dots high \rangle$
14:                                              ▷ to find a minimal $i$ such that $A.ci[i] \geq x1$
15:            if $C.ci[i] \leq x2$ then
16:                $output = true$
17:                break
18:            end if
19:        end for
20:        $output = false$
21:    end if
22:    send predicate $output$ to parallel reduction
23:    $po \leftarrow$ parallel reduction of $output$ using logical OR in $G'$
24:    return $po$
25: end procedure
```

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,
- 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).

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 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.

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.

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:     $i \leftarrow$ index of the current processor
3:     if $i = 1$ then                                         ▷ master section
4:         $S \leftarrow ()$
5:         enqueue $\{A, 1, 1, A.n, A.n, 0\}$ into current
6:         while $|current| < k \cdot P$ do
7:             current $\leftarrow$ ExpandLevel($S$, current)
8:         end while
9:         store $S$ in master file
10:        convert current to array $B$
11:    end if
12:    barrier
13:    one-to-all scatter of $B$
14:    all-to-all scatter of matrix structure
15:    $S \leftarrow ()$
16:    $C \leftarrow |current|$
17:    for $j \leftarrow 1 + (i-1)\lceil C/P \rceil, \min(i\lceil C/P \rceil, C)$ do
18:        $\{x1, y1, x2, y2, h\} \leftarrow B[j]$
19:        $D \leftarrow A[y1 \dots y2][x1 \dots x2]$
20:        Tr2MBT($D, x1, y1, x2, y2, h$)
21:    end for
22:    store $S$ in separate file dedicated to $p_i$
23:    return
24: end procedure
```

Matrix	COO S^{MIN}	COO S^{POW}	CSR S^{MIN}	CSR S^{POW}
`circuitM5`	2.25	3.13	1.24	1.71
`nlpkkt120`	2.27	3.30	1.23	1.77
`ldoor`	2.40	3.84	1.26	2.00
`TSOPF_RS_b2383`	4.04	4.04	2.03	2.03
`mouse_gene`	3.73	3.73	1.87	1.88
`t2em`	2.11	3.38	1.30	2.03
`bmw7st_1`	2.61	4.63	1.36	2.40
`amazon0312`	2.23	3.75	1.28	2.11
`thread`	2.98	3.18	1.52	1.63
`gupta2`	2.61	2.61	1.36	1.38
`c-29`	2.27	2.79	1.40	1.68

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.

other storage schemes. CR stands for *compression rate*. CR1 denotes the ratio of the MBT to the (CSR, S^{POW}) 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 format space complexity.

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`).

Algorithm 8 Master section of parallel transformation algorithm to the MBT format

```
procedure ExpandLevel($S$, $current$)
    create empty queue $new$
    while $current$ is nonempty do
        dequeue $\{x1, y1, x2, y2, h\}$ from $current$
        if $h$ is even then
            $mx \leftarrow x2$;    $my \leftarrow (y1 + y2)/2$
            $lx \leftarrow x1$;    $ly \leftarrow (y1 + y2)/2 + 1$
        else
            $mx \leftarrow (x1 + x2)/2$;    $my \leftarrow y2$
            $lx \leftarrow (x1 + x2)/2 + 1$;    $ly \leftarrow y1$
        end if
        $l1 \leftarrow$ ParINES$(A, x1, y1, mx, my)$
        $l2 \leftarrow$ ParINES$(A, lx, ly, x2, y2)$
        AppendToBitstream$(S, l1)$
        AppendToBitstream$(S, l2)$
        if $l1 = true$ then
            enqueue $\{A, x1, y1, mx, my, h + 1\}$ into $next$
        end if
        if $l2 = true$ then
            enqueue $\{A, lx, ly, x2, y2, h + 1\}$ into $next$
        end if
    end while
    return $next$
end procedure
```

Matrix	CR1 [%]	CR2 [%]	CR3 [%]	CR4 [%]
circuitM5	16.5	28.3	4.9	28.8
nlpkkt120	12.8	22.6	3.5	25.6
ldoor	8	16.1	2.4	15.3
TSOPF_RS_b2383	15.6	31.5	2.7	14.0
mouse_gene	73.0	137.0	12.8	53.9
t2em	16.3	33.0	5.7	26.2
bmw7st_1	8.5	20.3	2.8	15.7
amazon0312	53.1	112.1	18.1	67.7
thread	16.4	26.8	3.0	14.4
gupta2	28.7	39.7	5.3	23.3
c-29	31.7	53.4	8.6	27.2

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

- has similar space complexity as the MQT or CQT formats.

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

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 $kP \ll n$.

Matrix	MBT [%]	CBT [%]	MQT [%]	CQT [%]
circuitM5	27,4	24,4	26,6	26,5
nlpkkt120	26,1	23,5	19,8	19,8
ldoor	22,5	21,2	14,4	14,3
TSOPF_RS_b2383	36,1	35,9	33,6	33,5
mouse_gene	130,8	111,0	130,4	128,0
t2em	36,6	31,7	29,9	29,4
bmw7st_1	26,3	25,0	20,3	20,3
amazon0312	110,8	89,1	107,1	103,1
thread	37,3	35,3	23,8	23,8
gupta2	48,0	42,3	36,3	36,0
c-29	55,7	48,8	51,3	50,9

TABLE 5.3
Comparison of the space complexity of the tree-based SSFs with that of the ACB format.

Matrix	n	N	#calls ($kP = 10^1$)	#calls ($kP = 10^2$)	#calls ($kP = 10^3$)
circuitM5	$5.56 \cdot 10^6$	$5.95 \cdot 10^7$	20	278	4248
nlpkkt120	$3.54 \cdot 10^6$	$5.02 \cdot 10^7$	26	452	3852
ldoor	$9.52 \cdot 10^5$	$2.37 \cdot 10^7$	26	240	2204
TSOPF_RS_b2383	$3.81 \cdot 10^4$	$1.62 \cdot 10^7$	20	378	3210
mouse_gene	$4.51 \cdot 10^4$	$1.45 \cdot 10^7$	22	220	2060
t2em	$9.25 \cdot 10^5$	$4.59 \cdot 10^6$	26	426	4906
bmw7st_1	$1.41 \cdot 10^5$	$3.74 \cdot 10^6$	22	232	3368
amazon0312	$4.01 \cdot 10^5$	$3.20 \cdot 10^6$	18	200	2072
thread	$2.97 \cdot 10^4$	$2.25 \cdot 10^6$	26	260	3486
gupta2	$6.21 \cdot 10^4$	$2.16 \cdot 10^6$	28	380	3888
c-29	$5.03 \cdot 10^3$	$2.44 \cdot 10^4$	26	348	3900

TABLE 5.4
The efficiency of parallel algorithm (the number of calls of PARINES*).*

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

[1] T. Dytrych, K. D. Launey, J. P. Draayer, P. Maris, J. P. Vary, E. Saule, U. Catalyurek, M. Sosonkina, D. Langr, and M. A. Caprio, *Collective Modes in Light Nuclei from First Principles*, PHYSICAL REVIEW LETTERS, 111 (2013).

[2] K. D. Launey, S. Sarbadhicary, T. Dytrych, and J. P. Draayer, *Program in C for studying characteristic properties of two-body interactions in the framework of spectral distribution theory*, COMPUTER PHYSICS COMMUNICATIONS, 185 (2014), pp. 254–267.

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

[4] R. F. Boisvert, R. Pozo, and K. Remington, *The Matrix Market Exchange Formats: Initial Design*, Tech. Report NISTIR 5935, National Institute of Standards and Technology, Dec. 1996.

[5] T. A. Davis, *The university of florida sparse matrix collection*, NA DIGEST, 92 (1994).

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

[7] I. Šimeček, D. Langr, and P. Tvrdik, *Minimal quadtree format for compression of sparse matrices storage*, in 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'2012), SYNASC'2012, Timisoara, Romania, sept. 2012, pp. 359–364.

[8] I. Šimeček, D. Langr, and P. Tvrdík, *Space-efficient sparse matrix storage formats for massively parallel systems*, in High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and

Systems (HPCC-ICESS), HPCC'12, Liverpool, Great Britain, june 2012, pp. 54–60.

[9] I. Šimeček, D. Langr, and P. Tvrdík, *Space efficient formats for structure of sparse matrices based on tree structures*, in Proceedings of 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2013), SYNASC '13, 2013. to be published.

[10] I. Šimeček and P. Tvrdík, *Sparse matrix-vector multiplication - final solution?*, in Parallel Processing and Applied Mathematics, vol. 4967 of PPAM'07, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 156–165.

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

[12] D. Langr, I. Šimeček, and P. Tvrdík, *Storing Sparse Matrices in the Adaptive-Blocking Hierarchical Storage Format*, in Federated Conference on Computer Science and Information Systems (FedCSIS), September 2013, IEEE Xplore Digital Library, pp. 479–486.

[13] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, *On the usage of 16 bit indices in recursively stored sparse matrices*, in Proceedings of the 2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC '10, Washington, DC, USA, 2010, IEEE Computer Society, pp. 57–64.

[14] M. Martone, S. Filippone, S. Tucci, P. Gepner, and M. Paprzycki, *Use of hybrid recursive csr/coo data structures in sparse matrix-vector multiplication*, in Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, Oct 2010, pp. 327–335.

[15] M. Martone, S. Filippone, S. Tucci, and M. Paprzycki, *Assembling recursively stored sparse matrices*, in Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, Oct 2010, pp. 317–325.

[16] M. Martone, M. Paprzycki, and S. Filippone, *An improved sparse matrix-vector multiply based on recursive sparse blocks layout*, in Large-Scale Scientific Computing, vol. 7116 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 606–613.

[17] A. Pinar and C. Aykanat, *Sparse matrix decomposition with optimal load balancing*, in Proceedings of the Fourth International Conference on High-Performance Computing, HIPC '97, Washington, DC, USA, 1997, IEEE Computer Society, pp. 224–.

[18] L. Romero and E. Zapata, *Data distributions for sparse matrix vector multiplication*, Parallel Computing, 21 (1995), pp. 583 – 605.

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

[20] P. Tvrdík and I. Šimeček, *A new diagonal blocking format and model of cache behavior for sparse matrices*, in Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics, vol. 12 of PPAM'05, Poznan, Poland, 2005, Springer-Verlag, pp. 164–171.

[21] I. H. Witten, R. M. Neal, and J. G. Cleary, *Arithmetic coding for data compression*, Commun. ACM, 30 (1987), pp. 520–540.

[22] M. Tuma, *Overview of direct methods*, I. Winter School of SEMINAR ON NUMERICAL ANALYSIS, January 2004, Ostrava, Czech Republic.

Edited by: Teodor Florin Fortiş
Received: Mar 3, 2014
Accepted: Apr 4, 2014