Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 479–486

Storing Sparse Matrices to Files in the Adaptive-Blocking Hierarchical Storage Format

Daniel Langr, Ivan Šimeček, Pavel Tvrdík
Czech Technical University in Prague
Faculty of Information Technology
Thákurova 9, 160 00, Praha, Czech Republic
Email: langrd@fit.cvut.cz

***Abstract*—When there is a need to store a sparse matrix into a file system, is it worth to convert it first into some space-efficient storage format? This paper tries to answer such question for the adaptive-blocking hierarchical storage format (ABHSF), provided that the matrix is present in memory either in the coordinate (COO) or in the compressed sparse row (CSR) storage format. The conversion algorithms from COO and CSR to ABHSF are introduced and the results of performed experiments are then presented and discussed.**

I. Introduction

SPARSE matrices are commonly present in computer memory in storage formats that provide high performance of the matrix-vector multiplication operation. Considering a generic sparse matrix without any particular pattern of its nonzero elements, such storage formats are usually not space-optimal [1]–[3]. If we need to store such a matrix in a file, we have two options:

1) either to store the matrix in its *in-memory storage format* (IMSF), in which is the matrix stored in a computer memory;
2) or to store it in some *space-efficient storage format* (SESF), which additionally requires to perform the conversion between these formats.

Question 1. *Which of these two options will take less time?*

The second option should result in a smaller file and hence its faster store operation. However, the price paid for that is the overhead of the conversion algorithm.

Let S_{IMSF} and S_{SESF} denote the amount of memory required to store a matrix in particular IMSF and SESF, respectively. The time that will be saved when storing the matrix in a file system in SESF instead of IMSF will be

$$t_{\text{saved}} = \frac{S_{\text{IMSF}} - S_{\text{SESF}}}{\text{file system I/O bandwidth}}. \quad (1)$$

Let further t_{overhead} denote the running time of the conversion algorithm between IMSF and SESF. Storing a matrix in SESF will pay off if $t_{\text{saved}} > t_{\text{overhead}}$.

The answer to Question 1 is especially important for high performance computing (HPC) applications where matrices are distributed among P processors of a massively parallel computer system (MPCS). Let L_{IMSF} and L_{SESF} denote the amount of memory required to store a local part of a matrix in IMSF and SESF, respectively, on a particular processor. If the distribution of matrix nonzero elements among processors is well-balanced, then $L_{\text{IMSF}} \approx S_{\text{IMSF}}/P$ and $L_{\text{SESF}} \approx S_{\text{SESF}}/P$, and we can rewrite (1) to

$$t_{\text{saved}} \approx \frac{L_{\text{IMSF}} - L_{\text{SESF}}}{\text{file system I/O bandwidth}} \times P. \quad (2)$$

As the size of a computational problem, and therefore the size of a given sparse matrix, varies, then:

- L_{IMSF} (and hence L_{SESF} as well) is more or less constant, since it is limited by the amount of physical memory available to a single processor on a given MPCS.
- $\boldsymbol{t_{\text{overhead}}}$ **is approximately constant**, since the IMSF-to-SESF conversion algorithm is executed independently by all processors on their local parts of the matrix (of size L_{IMSF}).
- The file system I/O bandwidth—at least its listed maximum value—is constant.
- The number of processors P varies.
- $\boldsymbol{t_{\text{saved}}}$ **varies**, according to (2), **proportionally to** $\boldsymbol{P}$.

Thus, we may expect that as the size of a computational problem grows, beyond some point it will be faster to store a sparse matrix to a file system in SESF instead of IMSF.

In this paper, we focus on situations where the IMSF is either the *coordinate* (COO) or the *compressed sparse row* (CSR) storage format [4, Section 3.4], [5, Section 4.3.1] and the SESF is the *adaptive-blocking hierarchical storage format* (ABHSF) [2]. These formats are introduced in more details in Section II. We have developed conversion algorithms from COO and CSR to ABHSF, which are presented in Section III. The experiments performed with these algorithms are then described and the results discussed in Section IV.

Note that within the context of this paper, by a *storage format* we mean the way how sparse matrices are stored in computer memory (physical/disk), by a *file format* we mean the way how sparse matrices are stored in files (in a particular storage format), and by a *storage scheme* we mean a storage format at a block level for ABHSF.

This work was supported by the Czech Science Foundation under Grant No. P202/12/2011. We acknowledge the Aerospace Research and Test Establishment in Prague, Czech Republic, for providing HPC resources.

II. Data Structures

Let A be an $m \times n$ sparse matrix with z nonzero elements. The COO storage format consist of 3 arrays of size z that contain row indexes, column indexes, and values of the nonzero elements of A. We can thus define a data structure `COO` that stores A in the COO storage format as follows:

structure `COO` := {
- m, n: matrix size;
- z: number of nonzero elements;
- $rows[]$: row indexes of nonzero elements;
- $cols[]$: column indexes of nonzero elements;
- $vals[]$: values of nonzero elements;

}.

Note that we accompany a data name with [] if the data is meant to be an array.

The advantages of the COO storage format are its clear concept, simple usage, and no requirement for the order of matrix nonzero elements. Its drawback is relatively high amount of memory needed for storing A, i.e., high S_{COO}.

If we order matrix nonzero elements according to the increasing row index, we can modify the COO storage format such that we substitute the array of row indexes by the array of positions of each row data in the remaining two arrays. Such approach results in the CSR storage format, which can be defined by the following data structure:

structure `CSR` := {
- m, n: matrix size;
- z: number of nonzero elements;
- $colinds[]$: column indexes of nonzero elements;
- $vals[]$: values of nonzero elements;
- $rowptrs[]$: offsets of places where data of each row in the $colinds$ and $vals$ arrays start;

}.

Since the array $rows[]$ of `COO` is of size z and the array $rowptrs[]$ of `CSR` is of size m (usually $m + 1$ for the sake of simpler implementation), and since $z \gg m$ usually holds for real-world sparse matrices, the CSR storage format typically requires considerably less amount of memory for storing A when compared with COO, i.e., $S_{\text{CSR}} < S_{\text{COO}}$.

ABHSF is a two-level hierarchical storage format (see Figure 1) based on partitioning a matrix into $\lceil m/s \rceil \times \lceil n/s \rceil$ blocks of size $s \times s$ and storing each nonzero block in its space-optimal storage scheme. This approach can considerably reduce the memory requirements for storing sparse matrices when compared not only with COO and CSR but also with other fixed-scheme hierarchical storage formats.

We consider the following storage schemes within this text:

1) **dense:** all block elements are stored including zeros,
2) **bitmap:** only nonzero block elements are stored and their structure is defined by a bit map,
3) **COO:** equivalent of the COO storage format at a block level,
4) **CSR:** equivalent of the CSR storage format at a block level.

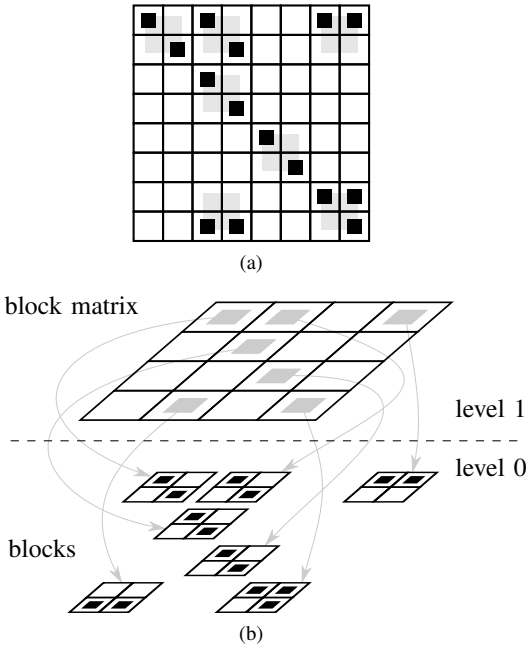

Fig. 1: An 8×8 matrix (a) represented as a hierarchical data structure with 2×2 blocks (b).

The optimal block size s for a particular matrix is application dependent, however, block sizes between 64 and 256 provide best results in general.

We can define a data structure `ABHSF` that stores A in the ABHSF format as follows:

structure `ABHSF` := {
- m, n: matrix size;
- z: number of nonzero elements;
- s: block size;
- $brows[]$: row indexes of nonzero blocks;
- $bcols[]$: column indexes of nonzero blocks;
- $zetas[]$: the number of nonzero elements of nonzero blocks;
- $schemes[]$: optimal storage scheme tag of nonzero blocks;
- $bitmap[]$: a bit map;
- $lrows[]$: row indexes local to a block;
- $lcols[]$: column indexes local to a block;
- $lrowptrs[]$: offsets of places where data of each row of a block start;
- $vals[]$: values of the elements of nonzero blocks;

}.

More details about ABHSF are beyond the scope of this paper, however, they were presented by Langr et at. [2].

Let `BLOCK` be an auxiliary data structure used for storing data of a single nonzero block, defined as follows:

structure `BLOCK` := {
- $brow$: row index of a block within A;
- $bcol$: column index of a block within A;

$zeta$: a number of block nonzero elements;
$lrows[]$: local row indexes of nonzero elements;
$lcols[]$: local column indexes of nonzero elements;
$lvals[]$: local values of nonzero elements;
}.

III. Algorithms

We further suppose that all indexes are 0-based (as used in the C/C++ programming languages).

A. Conversion from COO to ABHSF

The pseudocode of the conversion process from COO to ABHSF is presented as Algorithm 1. It is based on the successive gathering and processing of data for each nonzero block. To optimize this process, the nonzero elements of A are first sorted, at line 5, according to the key represented by the following quadruple with left-to-right significance of its elements:

$$\Big(\lfloor coo.rows[i]/s \rfloor, \lfloor coo.cols[i]/s \rfloor, coo.rows[i] \bmod s, coo.cols[i] \bmod s\Big). \quad (3)$$

Such ordering puts the nonzero elements of each nonzero block into a continuous chunks within the arrays of the `COO` data structure. Consequently, the conversion can be performed within a single iteration over matrix nonzero elements (lines 7–21).

The PROCESSBLOCK procedure, which stores data of a single nonzero block into the `ABHSF` data structure at line 20, is defined in Section III-C.

B. Conversion from CSR to ABHSF

The pseudocode of the conversion process from CSR to ABHSF is presented as Algorithm 2. It is based on the same principle as Algorithm 1, i.e., on the successive gathering and processing of the data of individual nonzero blocks. However, this process is here more complicated, since we cannot simply reorder the matrix nonzero elements according to (3), as in Algorithm 1. Therefore, instead of iterating over matrix nonzero elements, Algorithm 2 iterates over all blocks (loops at lines 5 and 15) and for each block it tries to obtain its nonzero elements. If there are any, they are then processed by the PROCESSBLOCK procedure as well.

C. Processing Blocks

The PROCESSBLOCK procedure stores data of a single nonzero block into the `ABHSF` data structure. Its pseudocode is shown as Algorithms 3.

We assume that the `ABHSF` data structure represents an open file and that all updates into this data structure will be directly translated into corresponding updates of its file representation. Within the pseudocode, we regard all `ABHSF` arrays as file output streams/virtual dynamic arrays to which the elements are successively *appended*.

The space-optimal storage schemes for blocks are selected at line 1 line by comparing their memory requirements, which were defined by Langr et al. [2] as functions (1a)–(1d). According to the optimal storage scheme, the block data are then stored into the `ABHSF` data structure as follows:

- **dense** (lines 7–17): The procedure iterates over all elements of a block. If the corresponding nonzero element is found, then its value is appended to the $vals[]$ array of the `ABHSF` data structure. Otherwise, 0 is appended instead.
- **bitmap** (lines 19–30): The procedure iterates over all elements of a block. If the corresponding nonzero element is found, then its value is appended to the $vals[]$ array of the `ABHSF` data structure and 1 is appended to the $bitmap[]$ array. Otherwise, 0 is appended to the to the $bitmap[]$ array instead.
- **COO** (lines 32–36): The procedure iterates over nonzero block elements and appends their row/column indexes and values to the corresponding arrays of the `ABHSF` data structure.
- **CSR** (lines 38–50): The procedure iterates over nonzero block elements and appends their column indexes and values to the corresponding arrays of the `ABHSF` data structure, while it also constructs the array $lrowptrs[]$ of positions of data for each row of a block.

Note that to PROCESSBLOCK work properly, the nonzero elements in the input `BLOCK` data structure need to be ordered according to growing row index and for each row according to the growing column index. Both Algorithm 1 and Algorithm 2 conform to this requirement.

IV. Experiments and Discussion

We have designed and performed experiments to evaluate the suitability of storing sparse matrices in files in ABHSF. Within these experiments, same matrices were stored in the COO, CSR, and ABHSF storage formats into files based on the HDF5 file format [6] so that particular data from the `COO`, `CSR`, and `ABHSF` data structures were stored as HDF5 attributes and data sets.

Within our implementation, the `ABHSF` data structure represented an open file, i.e., all updates to its data were directly translated into updates of corresponding file attributes and data sets. The data types for data sets containing indexes were always chosen to be unsigned integer data types of minimal possible bit width. All floating-point numbers were stored in files in single precision.

For all the experiments, we used the block size $s = 256$, which generally provides reasonable results for a wide range of matrices, as shown by Langr et al. [2]. To achieve maximum performance, we implemented experimental programs so that:

- All HDF5 data sets were chosen to be *fixed-size*. The conversion algorithms hence needed to be executed twice. Within the first *dry run*, the sizes of data sets were computed. Within the second run, data were actually written into them. (Sorting of elements in Algorithm 1 was performed only once within the dry run.)
- All updates of HDF5 data sets were buffered. We used buffers of size 1024 elements for each data set.

Algorithm 1: Conversion from COO to ABHSF

Input: coo: `COO`; s: integer
Output: $abhsf$: `ABHSF`
Data: $block$: `BLOCK`; $k, brow, bcol$: integer

```
1  abhsf.m ← coo.m
2  abhsf.n ← coo.n
3  abhsf.z ← coo.z
4  abhsf.s ← s
5  sort the coo.rows, coo.cols, and coo.vals arrays all at once according to (3)
6  k ← 0
7  while k < coo.z do                                          // iterate over nonzero elements
8      block.brow ← ⌊coo.rows[k]/s⌋
9      block.bcol ← ⌊coo.cols[k]/s⌋
10     block.zeta ← 0
11     while ⌊coo.rows[k]/s⌋ = block.brow and ⌊coo.cols[k]/s⌋ = block.bcol do   // while element is in the actual block
12         block.lrows[block.zeta] ← coo.rows[k] mod s
13         block.lcols[block.zeta] ← coo.cols[k] mod s
14         block.lvals[block.zeta] ← coo.vals[k]
15         block.zeta ← block.zeta + 1
16         k ← k + 1                                           // go to next nonzero element
17         if k ≥ coo.z then break
18
19     end
20     ProcessBlock(block, abhsf)                              // store block data into the ABHSF structure
21 end
```

- All the *bitmap*, *lrows*, *lcols*, and *lrowptrs* arrays from the `ABHSF` data structure were implemented in files as a single data set.

Reasoning for the listed decisions is beyond the scope of this paper. However, they all originated from results of our complementary tests and measurements.

A. File Sizes for Benchmark Matrices

First, we compared sizes of files for different matrices that emerged in real-world scientific and engineering applications. All used *benchmark matrices* were taken from *The University of Florida Sparse Matrix Collection* (UFSMC) [7]. We tried to choose matrices from different computational domains and therefore with different structural properties. Their list together with their characteristics is presented in Table I, where z' denotes the relative number of nonzero elements in percents, i.e., the inverse measure of sparsity of a matrix.

We stored matrices in HDF5-based files in the COO, CSR, and ABHSF storage formats. We further refer to these options as HDF5-COO, HDF5-CSR, and HDF5-ABHSF, respectively. The results are presented in Figure 2 where the file sizes are relative (in percents) to the HDF5-ABHSF option. For comparison, we also included the sizes of compressed (`.mtx.gz`) and uncompressed (`.mtx`) files in the Matrix Market file format [8], in which matrices are originally published in UFSMC.

The main conclusion from these results is that for all benchmark matrices, HDF5-COO resulted in files about twice as big as HDF5-ABHSF and HDF5-CSR resulted in files about 1.4 times bigger than HDF5-ABHSF. Therefore, **if we convert matrices to ABHSF, we can save a considerable amount of file system capacity.**

Unfortunately, we cannot simply compare the results for the text-based Matrix Market file format and the binary-based HDF5 file format, since in the text-based file formats, the floating-point values are generally represented in various precisions. However, note that for some matrices the smallest files were achieved for the compressed Matrix Market file format. This effect was caused by the fact that in these special cases, many matrix elements had identical floating-point values, which led to high efficiency of text compression. HDF5 also allows to compress data, which should reduce the sizes of data sets containing repeated floating-point values. We have, however, not tested this possibility.

B. Parallel Experiments

The matrices in UFSMC are of smaller sizes suitable for sequential rather than parallel processing. Since we did not find any suitable scalable HPC application able to generate very large sparse matrices, we simulated such matrices by parallel generation of random block matrices. The developed generating algorithm works as follows:

1) an *imaginary* global matrix is partitioned into P submatrices,
2) each submatrix is further treated by a single processor,
3) each submatrix is partitioned into blocks,

Algorithm 2: Conversion from CSR to ABHSF

Input: csr: CSR; s: integer
Output: $abhsf$: ABHSF
Data: $block$: BLOCK; $k, brow, bcol, firstrow, lastrow, nrows, row, lrow$: integer; $from$, $remains$: integer array

```
1  abhsf.m ← csr.m
2  abhsf.n ← csr.n
3  abhsf.z ← csr.z
4  abhsf.s ← s
5  for brow ← 0 to ⌈csr.m/s⌉ − 1 do                                   // iterate over block rows
6      firstrow ← brow · s
7      if firstrow + s ≤ csr.m then nrows ← s
8      else nrows ← csr.m − firstrow
9
10     lastrow ← firstrow + nrows − 1
11     for row ← firstrow to lastrow do                               // for each row of a block row find out:
12         from[row − firstrow] ← csr.rowptrs[row]                    // position of data to be processed
13         remains[row − firstrow] ← csr.rowptrs[row + 1] − csr.rowptrs[row]   // number of elements to be processed
14     end
15     for bcol ← 0 to ⌈csr.n/s⌉ − 1 do                               // iterate over block columns
16         block.brow ← brow
17         block.bcol ← bcol
18         block.zeta ← 0
19         for row ← firstrow to lastrow do                           // for each row of a block row
20             lrow ← row − firstrow
21             while remains[lrow] > 0 and csr.colinds[from[lrow]] < (bcol + 1) · s do
                                                                      // while elements belong to the actual block
22                 block.lrows[block.zeta] ← lrow
23                 block.lcols[block.zeta] ← csr.colinds[from[lrow]] − bcol · s
24                 block.lvals[block.zeta] ← csr.vals[from[lrow]]
25                 block.zeta ← block.zeta + 1
26                 from[lrow] ← from[lrow] + 1
27                 remains[lrow] ← remains[lrow] − 1
28             end
29         end
30         if block.zeta > 0 then ProcessBlock(block, abhsf)
                                                                      // store block data into the ABHSF data structure
31     end
32 end
```

4) each block becomes nonzero with some probability,
5) each nonzero block contains a random number of nonzero elements,
6) each nonzero element is assigned a random row/column index and a random value.

We further set up the generator so that:

- $L_{\text{COO}} \approx 600$ MB, therefore submatrices took about 600 MB each when stored in COO (the typical amount of physical memory per processor is 1–2 GB on contemporary MPCSs).
- Resulting matrices contained nonzero blocks of various properties, thus various storage schemes were generally space-optimal for them.
- Processors used different seeds for their instances of a pseudorandom number generator to produce different submatrices. However, these seeds were preserved in time, thus each processor generated the very same submatrices through all experiments.

The parallel experiments were carried out in the following steps:

1) Each processor generated a random submatrix and stored it in memory either in COO or in CSR.
2) All processors stored their submatrices to a file system in the original IMSF, which resulted in HDF5-COO or HDF5-CSR files.
3) All processors stored their submatrices to a file system in the ABHSF storage format utilizing either Algorithm 1 or Algorithm 2, which resulted in HDF5-ABHSF files.

Algorithm 3: PROCESSBLOCK(b, a)

Input: $block$: BLOCK; $abhsf$: ABHSF
Output: $abhsf$: ABHSF
Data: $scheme$: scheme tag; k, row, col: integer

```
1  scheme ← space-optimal storage scheme for block block      // functions (1a)–(1d) defined by Langr et al. [2]
2  append block.brow to abhsf.brows
3  append block.bcol to abhsf.bcols
4  append scheme to abhsf.schemes
5  append block.zeta to abhsf.zetas
6  if scheme = dense then                                     // optimal scheme is dense
7      k ← 0
8      for row ← 0 to abhsf.s − 1 do                          // iterate over all block elements
9          for col ← 0 to abhsf.s − 1 do
10             if k < block.zeta and block.lrows[k] = row and block.lcols[k] = col then   // if element exists
11                 append block.lvals[k] to abhsf.vals        // store its nonzero value
12                 k ← k + 1
13             else
14                 append 0 to abhsf.vals                     // otherwise store 0
15             end
16         end
17     end
18 else if scheme = bitmap then                               // optimal scheme is bitmap
19     k ← 0
20     for row ← 0 to abhsf.s − 1 do                          // iterate over all block elements
21         for col ← 0 to abhsf.s − 1 do
22             if k < block.zeta and block.lrows[k] = row and block.lcols[k] = col then   // if element exists
23                 append block.lvals[k] to abhsf.vals        // store its nonzero value
24                 append 1 to abhsf.bitmap                   // and 1 to bit map
25                 k ← k + 1
26             else
27                 append 0 to abhsf.bitmap                   // otherwise store 0 to bitmap
28             end
29         end
30     end
31 else if scheme = COO then                                   // optimal scheme is COO
32     for k ← 0 to block.zeta − 1 do                         // iterate over block nonzero elements
33         append block.lrows[k] to abhsf.lrows               // and store them into COO storage scheme
34         append block.lcols[k] to abhsf.lcols
35         append block.lvals[k] to abhsf.vals
36     end
37 else if scheme = CSR then                                   // optimal scheme is CSR
38     row ← 0
39     for k ← 0 to block.zeta − 1 do                         // iterate over block nonzero elements
40         while row ≤ block.lrows[k] do                      // and store them in the CSR storage scheme
41             append k to abhsf.lrowptrs
42             row ← row + 1
43         end
44         append block.lcols[k] to abhsf.lcols
45         append block.lvals[k] to abhsf.vals
46     end
47     while row ≤ abhsf.s do                                 // align final rows if needed
48         append block.zeta to abhsf.lrowptrs
49         row ← row + 1
50     end
51 end
```

Matrix	Domain	m	n	z' [%]	Symmetric
ldoor	structural problem	$9.5 \cdot 10^5$	$9.5 \cdot 10^5$	$2.6 \cdot 10^{-3}$	yes
Freescale1	circuit simulation	$3.4 \cdot 10^6$	$3.4 \cdot 10^6$	$1.6 \cdot 10^{-4}$	no
atmosmodj	computational fluid dynamics	$1.3 \cdot 10^6$	$1.3 \cdot 10^6$	$5.5 \cdot 10^{-4}$	no
cage12	directed weighted graph	$1.3 \cdot 10^5$	$1.3 \cdot 10^5$	$1.2 \cdot 10^{-2}$	no
ohne2	semiconductor device	$1.8 \cdot 10^5$	$1.8 \cdot 10^5$	$3.3 \cdot 10^{-2}$	no
FEM_3D_thermal2	thermal problem	$1.5 \cdot 10^5$	$1.5 \cdot 10^5$	$1.6 \cdot 10^{-2}$	no
bmw7st_1	structural problem	$1.4 \cdot 10^5$	$1.4 \cdot 10^5$	$1.8 \cdot 10^{-2}$	yes
nlpkkt120	optimization problem	$3.5 \cdot 10^6$	$3.5 \cdot 10^6$	$4.0 \cdot 10^{-4}$	yes

TABLE I: The list of the benchmark matrices used for the performed experiments.

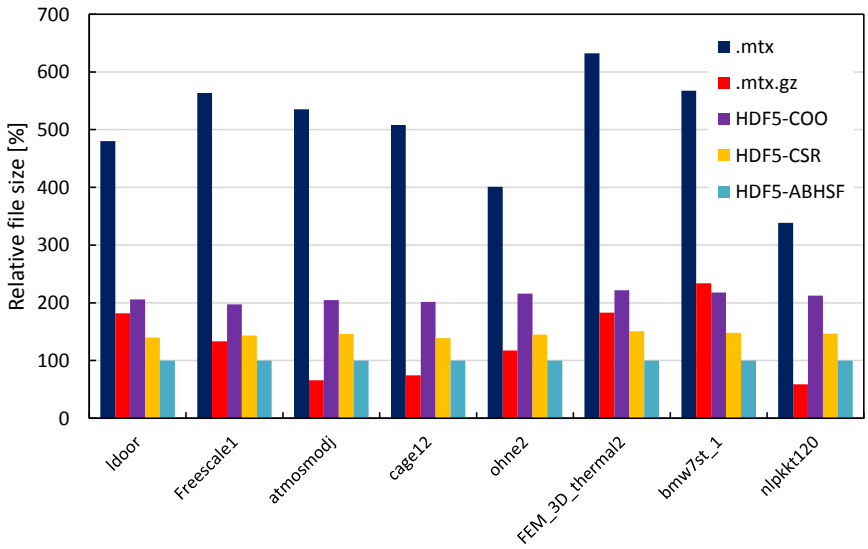

Fig. 2: File sizes in percents relative to HDF5-ABHSF for benchmark matrices and different file/storage formats.

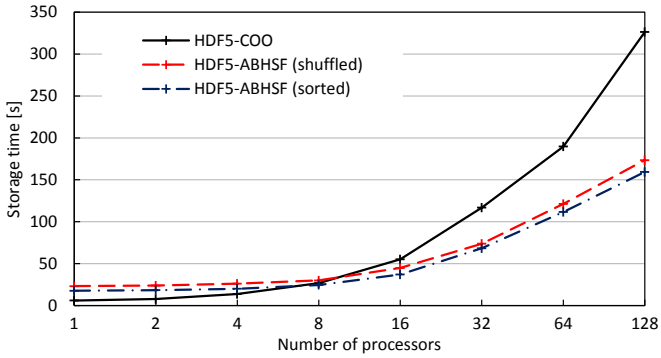

Fig. 3: Storage times for cases when COO was used as IMSF.

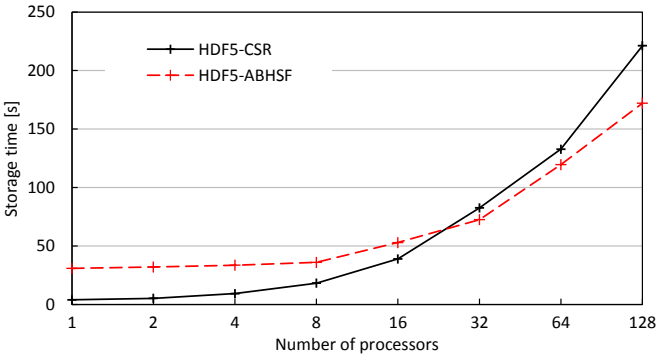

Fig. 4: Storage times for cases when CSR was used as IMSF.

We measured the storage times of steps 2 and 3 for different numbers of processors. Results for the case of using COO and CSR as IMSF are in Figure 3 and Figure 4, respectively. All measurements were performed 3 times and the average values are presented. Since the conversion algorithm from COO to ABHSF contains sorting of elements, we kept nonzero elements in memory in COO in two different orderings to evaluate the influence of the sorting algorithm. In the first case the elements were randomly *shuffled* and in the second case they were *sorted* by their (row index, column index) keys.

The obtained results clearly correspond to our assumption introduced in Section I. In case of ABHSF, there was some constant computational overhead imposed by the conversion algorithms (note that this overhead was considerably higher for the conversion from CSR, which was caused by the higher complexity of the conversion algorithm compared with the

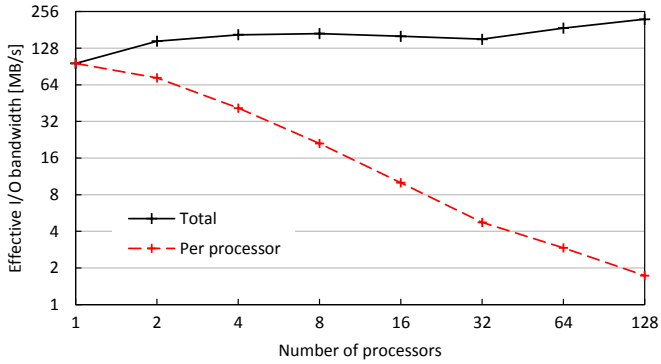

Fig. 5: Measured I/O bandwidth of the file system used for experiments.

COO case). For smaller numbers of processors, this overhead dominated the overall storage time ($t_{\text{saved}} < t_{\text{overhead}}$) and therefore it was faster to store matrices to a file system in their original IMSFs. However, **as the matrix size increased** (proportionally to P), **the amount of saved data** ($S_{\text{IMSF}} - S_{\text{SESF}}$) **increased as well, and from some point, it paid off to store matrices in ABHSF.**

From the measured storage times, we have also computed the *total I/O bandwidth* of the used file system. Provided that this total bandwidth is shared by all processors evenly, we can also express the *I/O bandwidth per processor*. These values are shown in Figure 5.

C. Generalization of Results

If we assume that the total I/O file system bandwidth is shared evenly among processors, we may rewrite (2) as

$$t_{\text{saved}} \approx \frac{L_{\text{IMSF}} - L_{\text{SESF}}}{\text{file system I/O bandwidth per processor}}.$$

This implies that **the amount of saved time generally grows inversely proportionally to the file system I/O bandwidth per processor**.

Within our experiments, we utilized a small parallel computer system with the GPFS-based storage subsystem [9]. The total I/O bandwidth of this file system varied, according to Figure 5, approximately from 100 to 200 MB/s. On this system, the point where ABHSF started to provide faster storage of matrices emerged around 16–32 processors, which corresponded to the I/O bandwidth of 4–8 MB/s per processor.

On todays biggest MPCSs, the per-processor I/O bandwidth would be typically much lower for large-scale computations. For instance, the Hopper/NERSC MPCS consists of over 153 thousands processors (CPU cores) and the listed maximum I/O bandwidth of its fastest file system is 35 GB/s. Therefore, we cannot get the I/O bandwidth per processor higher than 0.23 MB/s when utilizing the whole system. In general, for such low I/O rates, we may expect that the ABHSF storage format would be much more superior to the original IMSF when storing matrices to a file systems.

V. Conclusions

The contribution of this paper are new conversion algorithms for sparse matrices from the COO and CSR to the ABHSF storage formats and the evaluation of suitability of storing sparse matrices into file systems in ABHSF using these algorithms, with the focus on the HPC application domain. We showed that as the size of a computational problem grows, and so does the number of processors, there is some point from which it pays off to store matrices to a file system in ABHSF instead of their original IMSF.

Unfortunately, we cannot simply predict this point, since it depends on many factors, such as the I/O bandwidth of the file system, the actual workload of the file system, the clock rate of processors, the bandwidth of memory units, the available amount of physical memory per processor, the quality of the compiler, the quality of the program code, structural properties of the matrix, etc. However, provided that we use a particular MPCS and a particular HPC application that generates matrices with similar properties, many of these factors becomes fixed. Moreover, computational power and compiler capabilities that influence the overhead imposed by the conversion algorithms generally do not differ much across contemporary MPCSs. Then, the suitability of storing matrices to a file system in ABHSF (generally in any SESF) would be determined primarily by the I/O bandwidth of the file system per processor.

References

[1] Ivan Šimeček, Daniel Langr, and Pavel Tvrdík. Space-efficient sparse matrix storage formats for massively parallel systems. In *Proceedings of the 14th IEEE International Conference of High Performance Computing and Communications (HPCC 2012)*, pages 54–60. IEEE Computer Society, 2012.

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

[3] I. Šimeček, D. Langr, and P. Tvrdík. Minimal quadtree format for compression of sparse matrices storage. In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*. IEEE Computer Society, September 2012. Accepted for publication.

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

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

[6] The HDF Group. Hierarchical data format version 5, 2000-2013. http://www.hdfgroup.org/HDF5/ (accessed June 3, 2013).

[7] T. A. Davis and Y. F. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1), 2011.

[8] Ronald F. Boisvert, Roldan Pozo, and Karin Remington. The Matrix Market Exchange Formats: Initial Design. Technical Report NISTIR 5935, National Institute of Standards and Technology, Dec. 1996.

[9] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.