DOI 10.12694/scpe.v15i1.963
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 1, pp. 21–31. http://www.scpe.org © 2014 SCPE

LARGE-SCALE VISUALIZATION OF SPARSE MATRICES*

D. LANGR†, I. ŠIMEČEK†, P. TVRDÍK†, AND T. DYTRYCH‡

Abstract.
An efficient algorithm for parallel acquisition of visualization data for large sparse matrices is presented and evaluated both analytically and empirically. The algorithm was designed to be application-independent, i.e., it works with any matrix-processors mapping and with any sparse storage format/scheme. The empirical scalability study of the algorithm was carried on using multiple modern HPC systems. In our largest experiment, we utilized 262144 processors for 73 seconds to gather and store to a file the visualization data for a matrix with $1.17 \cdot 10^{13}$ nonzero elements. Using the proposed algorithm, one can thus visualize large sparse matrices with a minimal runtime overhead imposed on executed HPC codes.

Key words: visualization, sparse matrices, parallel system, distributed algorithm, data acquisition

AMS subject classifications. 65F30, 65F50, 68W15

1. Introduction. Within our previous work, we have addressed weaknesses of common solutions for the problem of visualization of large sparse matrices emerging in HPC applications [9]. There are several reasons that make such a problem difficult to solve. First, matrices exist in memory only for a short and unknown period of time determined by the scheduler of a given HPC system. Second, it is generally impossible to integrate the visualization process directly into the HPC code so that it will automatically produce a final matrix image of a desired quality. Third, very large matrices, due to their sizes, cannot be processed locally on personal computers. Moreover, their storage to a file on an HPC system and their transfer via network would take high amount of time.

We therefore proposed a solution where a matrix is first partitioned into blocks. Then, for each block, visualization information is calculated and stored into a file. Finally, this file is downloaded from the HPC system into a personal computer and processed there, possibly interactively, into a final matrix image. We introduced an algorithm that accomplishes a part of this procedure performed on the side of an HPC system, i.e., the acquisition of visualization data for a given sparse matrix. We also evaluated this algorithm experimentally using up to 1024 processors of a small-scale HPC system.

This paper is an extended version of our previous results [9]. We present an updated version of the algorithm, which uses more efficient approach to the calculation of visualization data. Namely, due to the utilization of an *ordered* associative array, frequent lookup operations with logarithmic runtime complexity were substituted by a single iteration over the ordered records. Each required record is thus available with constant runtime complexity. We also provide more detailed analysis of the computational, space, and communication complexities of the algorithm. The results of large-scale experiments using several modern HPC systems are presented for up to 266144 utilized processors.

Modern HPC systems are typically hybrid shared-distributed memory machines. They consist of shared-memory multicore computational nodes connected by network subsystems. Parallel programming models and corresponding runtime environments allow to use these machines as are, virtualize them as pure shared-memory, or virtualize them as pure distributed-memory. In the context of this paper, we consider the latter case.

*This work was supported by the Czech Science Foundation under Grant No. P202/12/2011, and by the Czech Technical University in Prague under Grant No. SGS14/106/OHK3/1T/18. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

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

‡Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803, USA.

Particularly, we assume the utilization of the MPI parallel programming library and its runtime environment [5, 11], which is motivated by its widespread adoption in the HPC community.

We further call MPI processes involved in algorithm runs simply *processors.* However, we always assume that each MPI process is mapped at runtime to a single CPU core. Therefore, a processor may also refer to a CPU core on which the algorithm runs.

2. Terminology and Notation.

Let $A = (a_{i,j})$ be an $m_A \times n_A$ real or complex matrix. Let $B = (b_{i,j})$ be an $m_B \times n_B$ real matrix, where $m_B \leq m_A$ and

$$n_B = \left\lfloor m_B \cdot \frac{n_A}{m_A} + \frac{1}{2} \right\rfloor. \tag{2.1}$$

We call B the *visualization matrix.*

Let us partition A into $m_B \times n_B$ blocks (submatrices) of same or nearly same sizes as follows:

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,n_B} \\ \vdots & \ddots & \vdots \\ A_{m_B,1} & \cdots & A_{m_B,n_B} \end{bmatrix}, \quad A_{k,l} = \begin{bmatrix} a_{r_k,c_l} & \cdots & a_{r_k,c'_l} \\ \vdots & \ddots & \vdots \\ a_{r'_k,c_l} & \cdots & a_{r'_k,c'_l} \end{bmatrix}$$

where

$$r_k = \left\lceil (k-1) \cdot \frac{m_A}{m_B} + \frac{1}{2} \right\rceil, \qquad r'_k = \left\lceil k \cdot \frac{m_A}{m_B} - \frac{1}{2} \right\rceil, \tag{2.2a}$$

$$c_l = \left\lceil (l-1) \cdot \frac{n_A}{n_B} + \frac{1}{2} \right\rceil, \qquad c'_l = \left\lceil l \cdot \frac{n_A}{n_B} - \frac{1}{2} \right\rceil. \tag{2.2b}$$

We will further write simply $a_{i,j} \in A_{k,l}$ to indicate that the element $a_{i,j}$ belongs to the block $A_{k,l}$. From Eq. (2.2) it follows that $a_{i,j} \in A_{k,l}$ if

$$k = \left\lfloor \left(i - \frac{1}{2}\right) \cdot \frac{m_B}{m_A} \right\rfloor + 1 \quad \text{and} \quad l = \left\lfloor \left(j - \frac{1}{2}\right) \cdot \frac{n_B}{n_A} \right\rfloor + 1. \tag{2.3}$$

Let $\mathbb{D} = \mathbb{R}$ or $\mathbb{C}$ if A is real or complex, respectively. Let $\mathcal{V} : \mathbb{D} \to \mathbb{R}$ be a function that satisfies

$$\mathcal{V}(0) = 0. \tag{2.4}$$

We call $\mathcal{V}$ the *visualization function.* Then, we define the elements of the visualization matrix B as follows:

$$b_{k,l} = \frac{1}{S_{k,l}} \sum_{a_{i,j} \in A_{k,l}} \mathcal{V}(a_{i,j}) = \frac{1}{S_{k,l}} \sum_{\substack{r_k \leq i \leq r'_k \\ c_l \leq j \leq c'_l}} \mathcal{V}(a_{i,j}), \tag{2.5}$$

where

$$S_{k,l} = (r'_k - r_k + 1) \times (c'_l - c_l + 1) \tag{2.6}$$

($S_{k,l}$ equals the number of elements of $A_{k,l}$).

In case of sparse A, condition Eq. (2.4) allows to calculate $b_{k,l}$ by performing the summation only over nonzero elements of $A_{k,l}$. We can thus rewrite Eq. (2.5) as

$$b_{k,l} = \frac{1}{S_{k,l}} \sum_{\substack{a_{i,j} \in A_{k,l} \\ a_{i,j} \neq 0}} \mathcal{V}(a_{i,j}). \tag{2.7}$$

Suppose now that we have an algorithm that calculates B for a sparse matrix A on a given HPC system. Let P denote the number of processors involved in an algorithm run and let $p_1, \dots, p_P$ denote these processors. Then, we can consider A as

$$A = A^{(1)} + \cdots + A^{(P)},$$

where $A^{(q)}$ contains nonzero elements of A stored in the local memory of processor p_q. (We use the (q) superscript frequently in this text to indicate that some entity belongs to processor p_q.) We will further write simply $a_{i,j} \in A^{(q)}$ to indicate that the element $a_{i,j}$ is contained in $A^{(q)}$, therefore in the local memory of processor p_q. Let $|A^{(q)}|$ denote the number of nonzero elements of $A^{(q)}$.

The calculation of $b_{k,l}$ now becomes trickier, since the nonzero elements of $A_{k,l}$ can be distributed among multiple processors. Let $\mathcal{P}_{k,l}$ denote a set of processors each of which contains at least one nonzero element of $A_{k,l}$. Thus,

$$\mathcal{P}_{k,l} = \big\{ p_q \bigm| \text{there exists } a_{i,j} \text{ such that } a_{i,j} \in A^{(q)} \text{ and } a_{i,j} \in A_{k,l} \big\}. \tag{2.8}$$

Conversely, let $\mathcal{A}_q$ denote a set of blocks from each of which processor p_q contains in its memory at least one nonzero element. Thus

$$\mathcal{A}_q = \big\{ A_{k,l} \bigm| \text{there exists } a_{i,j} \text{ such that } a_{i,j} \in A^{(q)} \text{ and } a_{i,j} \in A_{k,l} \big\}.$$

We can now rewrite Eq. (2.7) as

$$b_{k,l} = \sum_{q \in \mathcal{P}_{k,l}} b_{k,l}^{(q)}, \quad b_{k,l}^{(q)} = \frac{1}{S_{k,l}} \sum_{\substack{a_{i,j} \in A_{k,l} \\ a_{i,j} \in A^{(q)} \\ a_{i,j} \neq 0}} \mathcal{V}(a_{i,j}). \tag{2.9}$$

We further say that any processor from $\mathcal{P}_{k,l}$ *contributes* to the calculation of $b_{k,l}$.

Let $b_{k,*}$ denote the k-th row of B. Let $\mathcal{P}_{k,*}$ denote a set of processors that contribute to at least one element of this row, thus:

$$\mathcal{P}_{k,*} = \mathcal{P}_{k,1} \cup \cdots \cup \mathcal{P}_{k,n_B}.$$

We can now define the problem of acquisition of visualization data for a sparse matrix A as follows:

PROBLEM 1. *We are looking for an efficient parallel algorithm that calculates the visualization matrix B, for a given sparse matrix A and a visualization function $\mathcal{V}$, and saves it to a file* `F`. *The additional requirements are:*

Req. 1: The algorithm should not depend on the type of matrix-processors mapping (the type of distribution of the nonzero elements of A among processors).

Req. 2: The algorithm should not depend either on the computer representation of A or on the order in which its nonzero elements are accessible.

Sparse matrices are stored in computer memory in special data structures called *sparse matrix storage formats/schemes* (see, for instance [10, 1, 12]). All these formats have one common feature—they allow to iterate over the nonzero elements. Due to Req. 2, we thus further regard $A^{(q)}, \dots, A^{(P)}$ as a sequence of their nonzero elements with unspecified order. This allows us to work with A independently of the storage format used within an actual HPC code, where the matrix appears. Moreover, this allows us to work with matrices that are even not stored in memory at all (their elements are computed on-thy-fly).

The information we want to visualize is determined by $\mathcal{V}$. For instance, if we want to visualize the structure (pattern) of the nonzero elements of A, the visualization function might be defined simply as follows:

$$\mathcal{V}_1(a_{i,j}) = \begin{cases} 1 & \text{if } a_{i,j} \neq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

Then, B would contain the *density* of the nonzero elements of blocks of A. Other visualization functions might be defined, e.g., as follows:

$$\mathcal{V}_2(a_{i,j}) = \begin{cases} |a_{i,j}| & \text{if } a_{i,j} \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathcal{V}_3(a_{i,j}) = \begin{cases} \big|\mathrm{Re}(a_{i,j})\big| & \text{if } a_{i,j} \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathcal{V}_4(a_{i,j}) = \begin{cases} \big|\mathrm{Im}(a_{i,j})\big| & \text{if } a_{i,j} \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the usage of $\mathcal{V}_2$, $\mathcal{V}_3$, and $\mathcal{V}_4$ results in the visualization of the average magnitude of the elements of each block, the average magnitude of their real parts, and the average magnitude of their imaginary parts, respectively.

The visualization matrix B represents the intermediate visualization data that will be saved into `F` and from which the final matrix image will be constructed. We call `F` the *visualization file.* Recall that `F` is supposed to be downloaded from a parallel system to a user's personal computer. This, in effect, limits the size of `F` to some extent (presumably, to hundreds of megabytes or gigabytes nowadays), which consequently limits the size of B.

Suppose that `F` is a binary file and that we save the elements of B into `F` using a b-byte floating-point data type `T` (e.g., $b = 4$ for `T` being the IEEE 754 single-precision floating point data type [6]). The memory requirements in bytes for saving B in `F` are then

$$S(B) = m_B \cdot n_B \cdot b. \tag{2.12}$$

Let $S(\mathtt{F})$ denote the file size of F. Actually, it might be slightly higher than $S(B)$, since there might be some space overhead given by the used file format as well as by any supplemental saved information. In practice, such overhead should not exceed several kilobytes, therefore we consider it as negligible for further analysis and set

$$S(\mathtt{F}) = S(B). \tag{2.13}$$

Since our goal is to acquire as much visualization data as possible, we are looking for the answer to the following question: What is the maximum size of the visualization matrix B to be stored in `F` of size at most $S(\mathtt{F})$ bytes? Combining Eqs (2.1), (2.12), and (2.13), we can find the solution as

$$m_B = \left\lfloor \left(\frac{1}{b} \cdot \frac{m_A}{n_A} \cdot S(\mathtt{F}) \right)^{1/2} + \frac{1}{2} \right\rfloor \tag{2.14}$$

together with Eq. (2.1).

2.1. Example. Let A be a square matrix. Let us limit the size of the visualization file to 4 GB, which implies $S(B) = 2^{32}$. Let us use the IEEE 754 single-precision floating-point data type for storing elements of B into `F`, thus $b = 4$. Then, $m_B = n_B = 32768$. We can thus partition A into $32768 \times 32768 \approx 10^9$ blocks and save the visualization data in a form of a single floating-point number for each block into a file of approximate size 4 GB. From this file, we can then generate a matrix image up to the size of 1 gigapixel.

In the text below, the capitalized word "Algorithm" followed by a number always refers to a pseudocode presented by a corresponding floating text environment (as are figures and tables). Within pseudocode, we write references to corresponding equations in the end-of-line comments.

3. Methodology and Algorithm.

To solve Problem 1, we need to develop an algorithm for the calculation of the visualization matrix B according to Eq. (2.9), and its storage into the visualization file `F`. First, note that we can translate Eq. (2.9) into pseudocode as follows:

Algorithm 9 Visualization data acquisition: computationally expensive solution

```
 1: for $k \leftarrow 1$ to $m_B$ do
 2:     for $l \leftarrow 1$ to $n_B$ do
 3:         construct $\mathcal{P}_{k,l}$                                           ▷ (2.8)
 4:         for all processors in $\mathcal{P}_{k,l}$ do in parallel
 5:             $b_{k,l}^{(q)} \leftarrow 0$
 6:             for all local nonzero elements $a_{i,j}$ do
 7:                 if $a_{i,j} \in A_{k,l}$ then $b_{k,l}^{(q)} \leftarrow b_{k,l}^{(q)} + \mathcal{V}(a_{i,j})$          ▷ (2.9)
 8:             end for
 9:             $b_{k,l}^{(q)} \leftarrow b_{k,l}^{(q)} / S_{k,l}$                          ▷ (2.9), (2.6), (2.2)
10:             perform parallel reduction of $b_{k,l}^{(q)}$ to $b_{k,l}$ using + operator    ▷ (2.9)
11:             write $b_{k,l}$ into F
12:         end for
13:     end for
14: end for
```

```
for all processors $p_q$ in $\mathcal{P}_{k,l}$ do in parallel
    $b_{k,l}^{(q)} \leftarrow 0$
    for all local nonzero elements $a_{i,j} \in A_{k,l}$ do $b_{k,l}^{(q)} \leftarrow b_{k,l}^{(q)} + \mathcal{V}(a_{i,j})$
    $b_{k,l}^{(q)} \leftarrow b_{k,l}^{(q)} / S_{k,l}$
    perform parallel reduction of $b_{k,l}^{(q)}$ to $b_{k,l}$ using + operator
end for
```

Since this process needs to be performed for all possible combinations of k and l, a pseudocode that would solve Problem 1 might look like Algorithm 9.

The drawback of this approach is its computational complexity $O\big(|A^{(q)}| \cdot m_B \cdot n_B\big)$ for processor p_q, where $|A^{(q)}|$ might be very high in HPC applications. However, by rearranging Algorithm 9 as described below, we can reduce the computational complexity to $O\big(|A^{(q)}| \cdot \log_2 |\mathcal{A}_q| + m_B \cdot n_B\big)$. Since $|\mathcal{A}_q| \le m_B \cdot n_B$,

$$|A^{(q)}| \cdot \log_2 |\mathcal{A}_q| + m_B \cdot n_B \le |A^{(q)}| \cdot \log_2(m_B \cdot n_B) + m_B \cdot n_B \ll |A^{(q)}| \cdot m_B \cdot n_B,$$

when $|A^{(q)}|$, m_B, n_B are not all extremely small numbers.

The idea of this rearrangement is to split the solution of Problem 1 into two phases:

1. Within Phase1, processor p_q iterates over all its local nonzero elements $a_{i,j} \in A^{(q)}$. In each iteration, the contribution of $a_{i,j}$ to $b_{k,l}^{(q)}$ is calculated, where k and l are given by Eq. (2.3). This process can be performed by all processors in parallel with no communication costs.

2. Within Phase2, the local contributions $b_{k,l}^{(q)}$ are reduced in parallel to $b_{k,l}$, which is then written to the visualization file F.

The price for such a solution is the additional need to temporarily store the local contributions $b_{k,l}^{(q)}$ in memory of processor p_q. Usage of a *plain two-dimensional array* would imply the algorithm space complexity $O(m_B \cdot n_B)$ for each processor. Consequently, it would limit the size of B by the minimum of the available amounts of memory of all processors. An alternative is to store the contributions in an *associative array* (commonly also known as a *map* or a *dictionary*) of type $(k, l) \to \mathbb{R}$, where k and l are integers. The algorithm space complexity then changes to $O\big(|\mathcal{A}_q|\big)$ for processor p_q. In the *worst case*, processor p_q contributes to all elements of B, which turns this space complexity into $O(m_B \cdot n_B)$ as well (moreover, the hidden constants will be higher here, since the memory overhead of an associative array is higher than of a plain array). However, matrices are in practice mapped to processors typically according to some one- or two-dimensional partitioning scheme. In the *best case*, nonzero elements are distributed in matrices evenly, which implies $|\mathcal{A}_q| \approx m_B \cdot n_B / P$. The space complexity of an associative array-based algorithm then becomes $O(m_B \cdot n_B / P)$. (We observed such

Algorithm 10 Visualization data acquisition

```
    Input: $A = A^{(q)}, \ldots, A^{(P)}$: $A^{(q)}$ is located on $p_q$        ▷ input sparse matrix
    Input: $m_A$, $n_A$                                                        ▷ input matrix size
    Input: $P$                                                                 ▷ number of processors
    Input: $q$                                                                 ▷ actual processor number
    Input: $\mathcal{V}$                                                       ▷ visualization function
    Input: T                                  ▷ data type used for visualization data (elements of $B$)
    Input: $S(\mathtt{F})$                             ▷ required size of the visualization file in bytes
    Output: F                                                                  ▷ visualization file
    Data: ≎⌢⌜                                                                  ▷ ordered associative array
    Data: $m_B$, $n_B$, $b$                                                    ▷ auxiliary variables
 1: for all processors $p_1, \ldots, p_P$ do in parallel
 2:     $b \leftarrow$ byte size of the element of type T
 3:     $m_B = \left\lfloor \left(1/b \cdot m_A/n_A \cdot S(\mathtt{F})\right)^{1/2} + 1/2 \right\rfloor$     ▷ (2.14)
 4:     $n_B = \lfloor m_B \cdot n_A/m_A + 1/2 \rfloor$                          ▷ (2.1)
 5:     execute Phase1                                   ▷ calculate local contributions $b_{k,l}^{(q)}$
 6:     execute Phase2                                   ▷ reduce them to $b_{k,l}$ and store into F
 7: end for
```

a behaviour in experiments [9].)

In HPC applications, computational problems are often solved as big as the available resources allow. Therefore, the minimum of available amounts of memory of processors can be a relatively small value. Since we do not want it to limit the size of B, we further consider, for the algorithm design, the storage of local contributions in associative arrays.

Moreover, we assume this associative array to be *ordered* lexicographically by the (k, l) key. This allows to iterate over the contributions $b_{k,l}^{(q)}$ of processor p_q in Phase2 only once, which is much more efficient than calling the lookup operation for each needed combination of k and l. We further denote an instance of the defined ordered associative array by ≎⌢⌜ and its record with the (k, l) key by ≎⌢⌜$[k, l]$.

3.1. Algorithm Pseudocode. We present here an efficient algorithm that solves Problem 1. Its outline is presented by a pseudocode as Algorithm 10. Let all variables introduced by Algorithm 10 have a global scope, i.e., they are available within the pseudocode of phases as well. Moreover, let all auxiliary variables/arrays defined in the "Data" section of each pseudocode be local to processors.

The pseudocode of Phase1 is shown as Algorithm 11. Within, each processor iterates over its nonzero elements and for each one, its contribution to the corresponding element of B is calculated. These contributions are stored in the ≎⌢⌜ data structure such that at the end of Phase1, ≎⌢⌜$[k, l]$ equals $b_{k,l}^{(q)}$ on processor p_q.

In Phase2, local contributions to the elements of B are reduced in parallel to their final values. These are then written to the visualization file F. Performing the parallel reduction $m_B \times n_B$ times, each one for a single element of B, would be inefficient due to the overhead of communication operations. We therefore designed Phase2 such that the reductions are performed for whole rows of B at once, resulting in m_B reductions, each one for n_B elements at once.

To calculate $b_{k,*}$, all the contributions $b_{k,l}^{(q)}$ need to be reduced in parallel from processors $p_q \in \mathcal{P}_{k,*}$. Due to Req. 1 of Problem 1, each processor can generally contribute to any element of B. Consequently, the sets $\mathcal{P}_{k,l}$ and $\mathcal{P}_{k,*}$ can consist of all processors $p_1, \ldots, p_P$. To calculate $b_{k,*}$, all the contributions $b_{k,l}^{(q)}$ need to be reduced in parallel from processors $p_q \in \mathcal{P}_{k,*}$. There are thus two options how to perform such a reduction:

1. from all processors $p_1, \ldots, p_P$, while setting $b_{k,l}^{(q)} = 0$ for $p_q \notin \mathcal{P}_{k,*}$;
2. from $\mathcal{P}_{k,*}$ only, while this set need to be constructed first.

The second option provides no advantage over the first, since the construction of $\mathcal{P}_{k,*}$ would require collective communication between all processors as well. (In terms of MPI, it would require to form a processors group and a corresponding communicator. There has been, in fact, developed even a noncollective communicator

Algorithm 11 Visualization data acquisition: Phase1

```
    Data: $k$, $l$, $r$, $r'$, $c$, $c'$, $S$                              ▷ auxiliary variables
 1: for all local nonzero elements $a_{i,j}$ do
 2:     $k \leftarrow \lfloor (i - 1/2) \cdot m_B/m_A \rfloor + 1$                          ▷ (2.3)
 3:     $l \leftarrow \lfloor (j - 1/2) \cdot n_B/n_A \rfloor + 1$                          ▷ (2.3)
 4:     if record $(k, l)$ does not exist in ≎⌢⌜ then
 5:         insert a record into ≎⌢⌜ with $(k, l)$ key and value $\mathcal{V}(a_{i,j})$     ▷ (2.9)
 6:     else
 7:         ≎⌢⌜$[k, l] \leftarrow$ ≎⌢⌜$[k, l] + \mathcal{V}(a_{i,j})$                       ▷ (2.9)
 8:     end if
 9: end for
10: for all records $(k, l)$ in ≎⌢⌜ do
11:     $r \leftarrow \lceil (k - 1) \cdot m_A/m_B + 1/2 \rceil$                         ▷ (2.2a)
12:     $r' \leftarrow \lceil k \cdot m_A/m_B - 1/2 \rceil$                             ▷ (2.2a)
13:     $c \leftarrow \lceil (l - 1) \cdot n_A/n_B + 1/2 \rceil$                         ▷ (2.2b)
14:     $c' \leftarrow \lceil l \cdot n_A/n_B - 1/2 \rceil$                             ▷ (2.2b)
15:     $S \leftarrow (r' - r + 1) \times (c' - c + 1)$                              ▷ (2.6)
16:     ≎⌢⌜$[k, l] \leftarrow$ ≎⌢⌜$[k, l]/S$                                        ▷ (2.9)
17: end for
```

Algorithm 12 Visualization data acquisition: Phase2

```
    Data: Ⓡ⋙⋍[]                                                    ▷ array of size $n_B$
    Data: $j$, $k$, $l$, $l'$                                         ▷ auxiliary variables
 1: $j \leftarrow 1$
 2: for $k \leftarrow 1$ to $m_B$ do                                       ▷ for all rows of $B$
                                       ▷ gather contributions $b^{(q)}_{k,l}$ for $k$th row:
 3:     for $l \leftarrow 1$ to $n_B$ do Ⓡ⋙⋍$[l] \leftarrow 0$        ▷ ensure that $b^{(q)}_{k,l} = 0$ if $p_q \notin \mathcal{P}_{k,l}$
 4:     while $j \leq$ number of ≎⌢⌜ records and $k' = k$, where $(k', l')$ is the key of the $j$th ≎⌢⌜ record do
 5:         Ⓡ⋙⋍$[l'] \leftarrow$ value of the $j$th ≎⌢⌜ record              ▷ $b^{(q)}_{k,l} \leftarrow$ ≎⌢⌜$[k, l]$
 6:         $j \leftarrow j + 1$
 7:     end while
                                          ▷ reduce to $b_{k,l}$ on $p_1$ and write to F:
 8:     perform parallel reduction of all values of the Ⓡ⋙⋍ array using $+$ operator to processor $p_1$ ▷ from all
    processors
 9:     if $q = 1$ then                                              ▷ if run on processor $p_1$
10:         for $l \leftarrow 1$ to $n_B$ do append Ⓡ⋙⋍$[l]$ into file F         ▷ write $b_{k,*}$ into F
11:     end if
12: end for
```

creation technique, however, it does not perform well for our purposes [4]). We therefore further consider only the first option for algorithm design.

The pseudocode of Phase2 is presented as Algorithm 12. Note that the order of processing the local contribution matches the order of records in the ≎⌢⌜ data structure. This allowed us to design the pseudocode such that no ≎⌢⌜ lookup operation is needed, which considerably reduced the overall algorithm complexity. The auxiliary array Ⓡ⋙⋍ serves as a buffer for storing and reducing local contributions for a single row of B. Before reduction of the kth row contributions, Ⓡ⋙⋍$[l] = b^{(q)}_{k,l}$ on processor p_q. After this reduction, Ⓡ⋙⋍$[l] = b_{k,l}$ on processors p_1. The reduction is performed by all processors (see line 1 in Algorithm 10).

3.2. Complexities. Let us analyze complexities of the presented algorithm. Generally, the complexities highly depend on the matrix-processors mapping. We therefore restrict our analysis to the following two extreme cases: In the *worst case*, there is at least one processor that contributes to all $m_B \times n_B$ elements of B. On the other hand, in the *best case*, all processors contribute to at most $\lceil m_B \times n_B / P \rceil$ elements of B.

We define the complexity of the algorithm as its complexity for the most loaded processor (this value determines both the algorithm running time as well as its per-processor memory requirements).

3.2.1. Computational Complexity. For processor p_q, the *computational complexity* of Phase1 is given by two iterative processes defined at lines 1–9 and 10–17 of Algorithm 11. The computational complexity of the first process is given by iterating over $|A^{(q)}|$ nonzero elements, while in each iteration a record is either found or inserted into ≎⌢⌜. Recall, that ≎⌢⌜ is an ordered associative array. These are typically implemented as binary search trees with logarithmic complexity of both search and insert operations (see, for instance [2, Chap. 12]). The computational complexity of the second process is given by iterating over all $|\mathcal{A}_q|$ records of ≎⌢⌜. The computational complexity of Phase1 on processor p_q is thus

$$T^{(q)}_{\text{Phase1}} = O\big(|A^{(q)}| \cdot \log_2 |\mathcal{A}_q| + |\mathcal{A}_q|\big).$$

In the worst case, $|\mathcal{A}_{q'}| = m_B \cdot n_B$ for some processor $p_{q'}$. Therefore, the computational complexity of Phase1 becomes

$$T^{(\text{worst})}_{\text{Phase1}} = O\big(\max_{1 \le q \le P} |A^{(q)}| \cdot \log_2(m_B \cdot n_B) + m_B \cdot n_B\big).$$

In the best case, $|\mathcal{A}_q| \le \lceil m_B \cdot n_B / P \rceil$ for all processors. The computational complexity of Phase1 then equals

$$T^{(\text{best})}_{\text{Phase1}} = O\Big(\max_{1 \le q \le P} |A^{(q)}| \cdot \log_2\big(\lceil m_B \cdot n_B / P \rceil\big) + \lceil m_B \cdot n_B / P \rceil\Big).$$

The computational complexity of Phase2 is, for processor p_1 and thus for the whole algorithm,

$$T_{\text{Phase2}} = O(m_B \cdot n_B)$$

in all cases.

3.2.2. Space Complexity. The space complexity of Phase1 is given by the number of records of the associative array ≎⌢⌜, which equals $|\mathcal{A}_q|$ on processor p_q. The space complexity of Phase2 is given by the auxiliary array Ⓡ⋙⌓ of size n_B. The overall space complexity of the algorithm is thus

$$S^{(\text{worst})} = O(m_B \cdot n_B) \quad \text{and} \quad S^{(\text{best})} = O\big(\lceil m_B \cdot n_B / P \rceil + n_B\big)$$

in the worst case and best case, respectively.

3.2.3. Communication Complexity. The communication complexity depends on the network topology of a given HPC system as well as on the implementation of communication operations by the utilized version of the MPI library. We can therefore only discuss the number of communication operations, instead of analyzing their asymptotic behaviour in terms of running time. In Phase1, there is no communication at all. In Phase2, m_B parallel all-to-one reductions are performed by all processors, each over n_B elements.

4. Experiments. We have performed a weak-scalability study (constant problem size per processor), since it matches the real-world situations, where HPC problems are usually solved as large as available resources allow. On modern hybrid shared-distributed memory HPC systems, the available amount of memory, which limits the size of A, is thus proportional to the number of utilized processors P. In experiments, we always set the size of A such that its number of nonzero elements was approximately $P \cdot 4.6 \cdot 10^7$ (this corresponded to the per-processor memory requirements of A being around 1 GB using the coordinate storage scheme and the single-precision floating point data type for matrix elements).

As a source of large sparse matrices, we used the parallel scalable generator of benchmark sparse matrices [8]. This generator produces matrices by scalable enlargement of a small fixed *seed* matrix. As the seed matrix, we used the cage12 real square matrix obtained from the University of Florida Sparse Matrix Collection [3].

Table 4.1
Configurations of HPC systems and their run-time environments used for experiments.

System:	Anselm	Edison	Blue Waters
Provider:	IT4Innovations	NERSC	NCSA
Type:	Linux cluster	Cray XC30	Cray XE6
Total CPU cores:	3344	124608	362240
Total memory [TB]:	15	332	1380
Interconnect:	Infiniband	Aries	Gemini
Topology:	Fat tree (nb)	Dragonfly	3D torus
I/O bandwidth [GB/s]:	6	48	1000+
C++ compiler:	Intel icpc 13.1	Intel icpc 14.0	GNU g++ 4.8.1
MPI library:	Intel MPI 4.1	cray-mpich 6.2.0	cray-mpich 6.0.1
HDF5 library version:	1.8.11	Cray 1.8.11	Cray 1.8.11

We set the input algorithm parameters so to generate a matrix image according to Section 2.1. The size of B was thus always set to $m_B = n_B = 32768$, which resulted in possibility of generation of a 1 gigapixel matrix image. As a visualization function, we used $\mathcal{V}_1$ defined by Eq. (2.10).

We have implemented the presented algorithm with C++ and MPI to evaluate its performance and scalability. As an ordered associative array, we used the `std::map` container from the C++ Standard Library [7, Sect. 7.7], which is typically implemented as a red-black tree [2, Chap. 13]. We exploited the HDF5 library [13] for creation of visualization files `F`.

We utilized several modern parallel systems that are listed in Table 4.1, where we show their parameters together with the C++ compilers and the MPI libraries used for compilations of test programs and their runtime execution. For the Blue Waters system, we present parameters of the XE cabinets only (the XK cabinets are intended primarily for GPU-based computations).

In all experiments, we primarily measured algorithm running times. These are presented by Figure 4.1. The results indicate that the majority of the running time is spent in Phase2, which is caused by the execution of communication and I/O operations. However, the algorithm is generally fast and the overall algorithm running time grows only slightly with the growing number of processors.

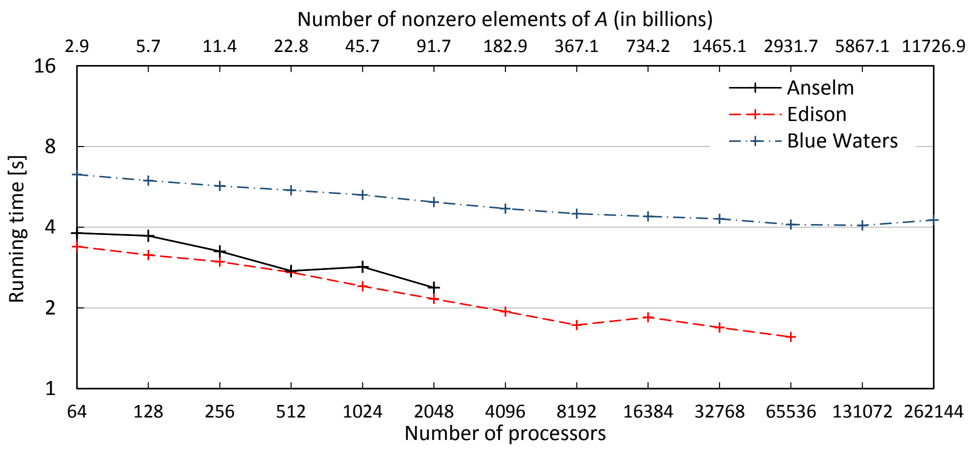
We also measured the estimated memory requirements of the ≎⌢⌜ data structure for all processors. Their maximal and average values are shown in Figure 4.2. Up to some small constant memory overhead, these memory requirements clearly decrease inversely proportionally to the number of processors P. For higher P, the overall memory footprint of the algorithm is thus negligible in practice.

5. Conclusions. This paper extends the work of Langr et al. [9]. It presents an updated version of the algorithm for the parallel acquisition of visualization data for large sparse matrices, along with its more detailed analytical and empirical evaluation. The main characteristic of the algorithm, from a user's point of view, is its application independence. It can be used with any mapping of matrices to processors and any sparse storage formats/schemes. Moreover, it can be used even when matrix nonzero elements are computed on-the-fly. It can be also easily implemented using the MPI parallel programming library, from which it needs to execute only the all-to-one parallel reduction operations.

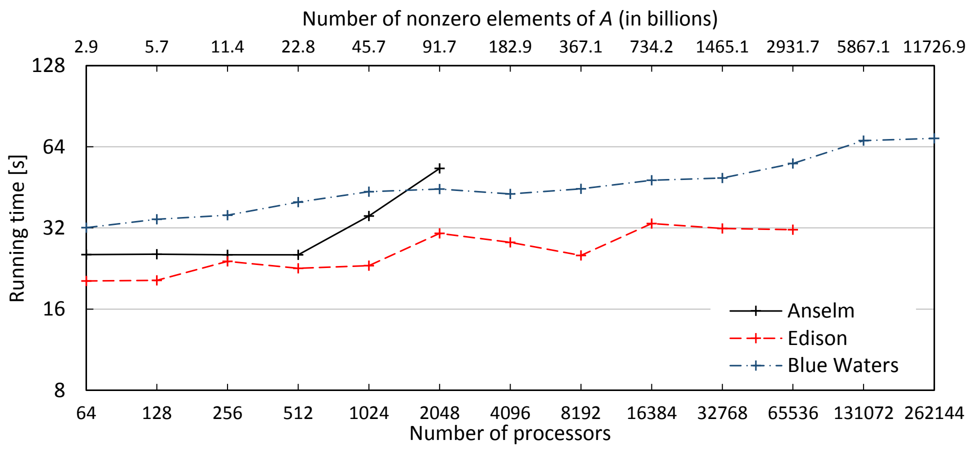
Using this algorithm, one can visualize sparse matrices emerging in an HPC code with minimal runtime and memory overhead. Namely, within the largest presented experiment, 266144 processors gathered and stored visualization data for a sparse matrix with $1.17 \cdot 10^{13}$ nonzero elements with a measured runtime of 73 seconds.

The majority of the algorithm running time takes the collective communication between all processors together with serial I/O operations. Resulting visualization data are appended to output files, which allows, among others, visualization files to be ASCII-based. Due to the application independence of the algorithm, parallel I/O cannot be efficiently employed to increase the I/O performance. This does not seem to be of much importance at the petascale performance level of today's prime HPC systems, as demonstrated by the presented experiments.

However, for emerging exascale and higher-level computing, even faster algorithms might appear to be necessary. In our future work, we want to focus on two promising options how to achieve them. The first one is



(a) Phase1



(b) Phase2

Fig. 4.1. *Running times of both algorithm phases measured on different parallel systems.*

to adapt the presented algorithm for the hybrid MPI+OpenMP parallel programming model, which naturally represents the shared-distributed memory architecture of modern HPC systems. The second option is to give up the application independence and to adapt the algorithm for certain types of matrix-processor mappings, which should allow to reduce the burden of collective communication and to efficiently utilize parallel I/O.

REFERENCES

[1] 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.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, Cambridge, Massachusetts, USA, 3rd ed., 2009.

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

[4] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu, *Noncollective*

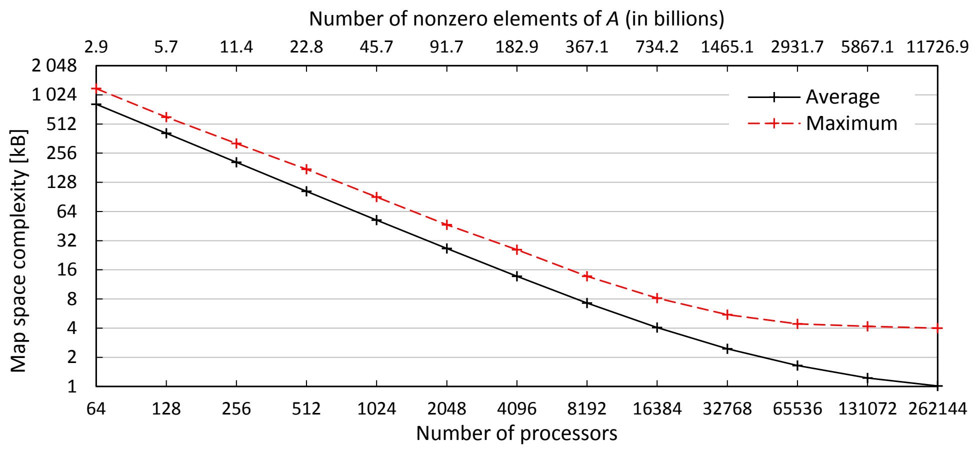


FIG. 4.2. *Statistics of the estimated per-processor memory requirements of the* ≎⌒⌜ *data structure.*

communicator creation in mpi, in Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, eds., vol. 6960 of Lecture Notes in Computer Science, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 282–291.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI—The Complete Reference, Volume 2: The MPI-2 Extensions*, MIT Press, Cambridge, MA, USA, 1998.

[6] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2008, (2008), pp. 1–58.

[7] N. M. Josuttis, *The C++ Standard Library—A Tutorial and Reference*, Addison Wesley Longman, Boston, MA, USA, 2nd ed., 2012.

[8] D. Langr, I. Šimeček, P. Tvrdík, and T. Dytrych, *Scalable parallel generation of very large sparse matrices*, in 10th International Confernce on Parallel Processing and Applied Mathematics (PPAM 2013), Lecture Notes in Computer Science, Springer Berlin Heidelberg. Accepted for publication.

[9] ———, *Parallel data acquisition for visualization of very large sparse matrices*, in Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2013), Los Alamitos, CA, USA, September 2013, IEEE Computer Society, pp. 338–345.

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

[11] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI—The Complete Reference, Volume 1: The MPI Core*, MIT Press, Cambridge, MA, USA, 2nd (revised) ed., 1998.

[12] P. T. Stathis, *Sparse Matrix Vector Processing Formats*, PhD thesis, Technische Universiteit Delft, 2004.

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

Edited by: Teodor Florin Fortiş
Received: Feb 28, 2014
Accepted: Apr 2014