Paper

Parallel Algorithms for Forward and Back Substitution in Linear Algebraic Equations of Finite Element Method

Sergiy Fialko

Institute of Computer Science, Faculty of Physics, Mathematics and Computer Science,
Cracow University of Technology, Cracow, Poland

https://doi.org/10.26636/jtit.2019.134919

Abstract—This paper considers several algorithms for parallelizing the procedure of forward and back substitution for high-order symmetric sparse matrices on multi-core computers with shared memory. It compares the proposed approaches for various finite-element problems of structural mechanics which generate sparse matrices of different structures.

Keywords— finite element method, multithreaded parallelization, sparse symmetric matrices, triangular solution.

1. Introduction

When dealing with problems of structural and solid mechanics by using finite element method, we have to solve systems of high-order linear algebraic equations with a sparse symmetric matrix. Direct methods are usually used to solve this class of problems, because most design models are poorly conditioned, which leads to slow convergence of iterative solvers [1], [2].

The solution of a system of linear algebraic equations, relying on the use of the direct method, consists of the following stages: matrix factorization and forward/back substitutions. Matrix factorization procedures have already achieved a high level of performance, and are effectively parallelized. Both multifrontal and supernodal solvers have been developed for multi-core computers with shared memory and the symmetric multiprocessing (SMP) architecture [3]–[6], etc. It seems that the factorization procedure has a much greater computational complexity than forward and back substitutions. Therefore, the computation time practically does not depend on how quickly the latter will be performed. However, in practice, problems are often encountered where it is necessary to decompose the matrix once, and then to perform forward and back substitutions with one or several right-hand sides at each iteration. For example, when integrating the equations of motion with the use of the Newmark method or any other implicit method, forward and back substitutions have to be performed once at each time step [7]. This algorithm, as well as the procedure for calculating the internal forces of the system, determine the entire computation time, because all other procedures contain a much smaller number of operations. Parallelization of the procedure for calculating the internal forces is given in [8]. Since it is necessary to perform many time steps, the speed-up of the triangular solution algorithm is one of the key points in improving the performance of the implicit method.

The second scenario in which it is very important to perform forward and back substitutions quickly is the determination of first n eigenvalues and eigenvectors of the lower part of the spectrum of an algebraic generalized eigenvalue problem with a sparse symmetric matrix by applying the block Lanczos method [9] and the subspace iteration method.

The third problem is the solution of a system of linear algebraic equations by the conjugate gradient method with preconditioning obtained on the basis of the incomplete Cholesky factorization [1], [10]. When solving a system of linear algebraic equations for preconditioning, it is necessary to perform forward and back substitutions at each iteration step.

This paper focuses on speeding up the forward and back substitutions algorithm for high-order sparse triangular matrices obtained as a result of factorization of a sparse symmetric stiffness matrix which, in turn, was obtained by applying the finite element method to structural mechanics and solid mechanics problems. It is not by chance that we consider the application range of the approaches proposed in this paper, since the effectiveness of each of them largely depends on the density and non-zero structure of a triangular sparse matrix, which parameters, in turn, are determined by the class of problems characteristic of a given application range. A method which is effective for one class of problems may turn out to be ineffective for another.

We will mention a few previously published papers in order to show the diversity of approaches applied to solving this complex problem.

The parallel triangular solution algorithms for dense and band triangular matrices are presented in [11]. In addition, approach [12] presents a parallel method for solving linear algebraic equations with dense triangular matrices, too.

The algebraic multicolor ordering method for sparse matrices, where the related unknowns are assigned the same color, is considered in [13]. Thus, groups (blocks) of unknowns are formed and assigned different colors. Since blocks with different colors have no data dependency, computations regarding these blocks can be parallelized.

An algorithm implemented in CUDA (cuBLAS and cuSPARSE libraries) for computing performed on a graphics card is presented in [14]. The method is based on the construction of a directed acyclic graph (DAG), representing the dependencies between variables in the original sparse matrix. The algorithm requires separate storage of the upper and lower triangular matrices, which significantly reduces the maximum dimension of the problem that may be stored in the memory of the graphics card. An assessment of the effectiveness of this approach when solving problems of structural mechanics for the design models of high-rise buildings and structures is given in [15].

In [16], the DAG and the structure of levels are also created. Then, independent branches of the solution are searched for, providing the basis for parallelization. The algorithm is used for computing on a GPU. Note that the duration of the preprocessing stage is much longer than that of the numerical phase.

The approach presented in [17] is also developed for usage on a GPU. The method is effective only when the lower triangular matrix is very sparse, so the authors recommend using it for incomplete factorization of the original matrix. The Jacobi method and "block-asynchronous" version of incomplete LU (ILU) factorization are considered as preconditioning.

The technique of dividing a sparse matrix into dense blocks with a subsequent application of the BLAS *dgemm*, *dgemv*, *dtrsv* procedure is described in [18]. This approach aims to improve performance of the triangular solution procedure due to a more efficient use of CPU cache memory and vectorization of calculations. Multithreaded parallelization is not considered.

The reasons behind an insignificant (or lacking) speed-up when solving a sparse system of linear algebraic equations with a sparse triangular matrix on multiprocessor computers with shared memory are described in [19]. It is believed that the significant increase in data transfers from RAM to the CPU cache, as the number of threads increases, is the reason behind the insignificant increase in speed-up. A number of measures are proposed to overcome these difficulties, including reordering the data so as to ensure their space locality. Similar results have been obtained in [20].

The speed-up of the triangular solution procedure in [21] is based on representing the unit lower triangular matrix as a product $\mathbf{L} = \Pi_{i\in[1,n]}(\mathbf{L}_i)$, where $\mathbf{L}$ is a lower triangular matrix, and $\mathbf{L}_i = \mathbf{I} + l_{ij}\mathbf{e}_i$ is the i-th matrix factor, $\mathbf{I}$ is a unit matrix, l_{ij} is an element of the lower triangular matrix $\mathbf{L}$, $\mathbf{e}_i$ is the i-th coordinate vector with all components zero, except for the j-th component which is equal to 1. Then, the matrices $\mathbf{L}_i$ are combined into groups, so that dense blocks are formed. This makes it possible to apply an algorithm for multiplying the matrix by a vector for dense matrices for each combined matrix, and to perform parallelization.

DAG is constructed for a sparse triangular matrix in [22] and level-sets are calculated using the breadth-first search algorithm. Then, the system is permuted symmetrically, so that the rows/columns are in order of the level-sets, while the matrix remains triangular. Parallelization of this algorithm is based on the absence of data dependencies within a given level in the permuted matrix (i.e. there are no edges connecting vertices within a level-set). A similar approach is also used in [23].

Approach [24] is applied in distributed-memory computers and groups the unknowns in such a way that the individual blocks may be calculated independently, on parallel machines.

This analysis of existing studies shows a variety of different approaches to solving the problem of speeding-up the triangular solution algorithm, which can be explained by the difficulty of the task at hand.

This paper proposes two parallel algorithms for calculating forward and back substitutions, obtained as a result of block factorization of a symmetric sparse matrix. The algorithms are designed for the purpose of solving problems of structural and solid mechanics by the finite element method on multicore desktops and multiprocessor workstations with shared memory. The goal of this work is to implement efficient, multithreaded parallelization algorithms and cache memory blocking, since all other high-performance techniques related to vectorization of calculations, blocking YMM registers and maximum support for CPU pipelining, are already implemented in the *dgemm* and *dtrsm* procedures from the Intel MKL library [25].

2. Problem Formulation

2.1. LSL^T Block Factorization of a Symmetric Sparse Matrix

The PARFES parallel supernodal solver, designed to solve finite-element problems on multi-core computers with shared memory, is used in this approach. It has been selected because in the case of the finite element analysis, supernodal solvers demonstrate better performance than multifrontal ones at the factorization stage on multi-core computers with shared memory [4], [5]. Unlike the PARDISO solver which is presented by the Intel MKL library and has successfully proven itself on multi-core computers relying on SMP architecture, PARFES uses disk memory when RAM memory runs low. Therefore, it may be used as the main finite element method solver. PARFES performs block $\mathbf{LSL}^T$ decomposition, where $\mathbf{L}$ is a block sparse lower triangular matrix, and $\mathbf{S}$ is a sign diagonal, which allows applying the method not only to positive definite matrices, but to their indefinite counterparts as well.

In this article, we confine ourselves to the scenario in which the factorized lower triangular matrix $\mathbf{L}$ is in the RAM, because when using a disk, the problem of optimized readout of large amounts of data from the disk comes first, and the efficiency of computation acceleration based on multithreading parallelization becomes much lower.

After completing the ordering procedure to reduce the number of fillings, PARFES creates an elimination tree and makes the transition to the supernodal tree. Each supernode in a sparse matrix corresponds to a block column with a dense block at the main diagonal. As a result, the sparse matrix is divided into block columns, and each block column consists of dense blocks. This allows us to pass from scalar-vector procedures of low performance to high-performance matrix procedures [4], [5].

The structure of a typical block-column is shown in Fig. 1. Here, $\pounds_{jb}$ is the non-zero structure of the block-column jb, defined by the position pointer $Pos\,[jb]$ of the first non-zero block in the block-column jb, counting from the diagonal block $\mathbf{L}_{jb,jb}$, which is always dense.

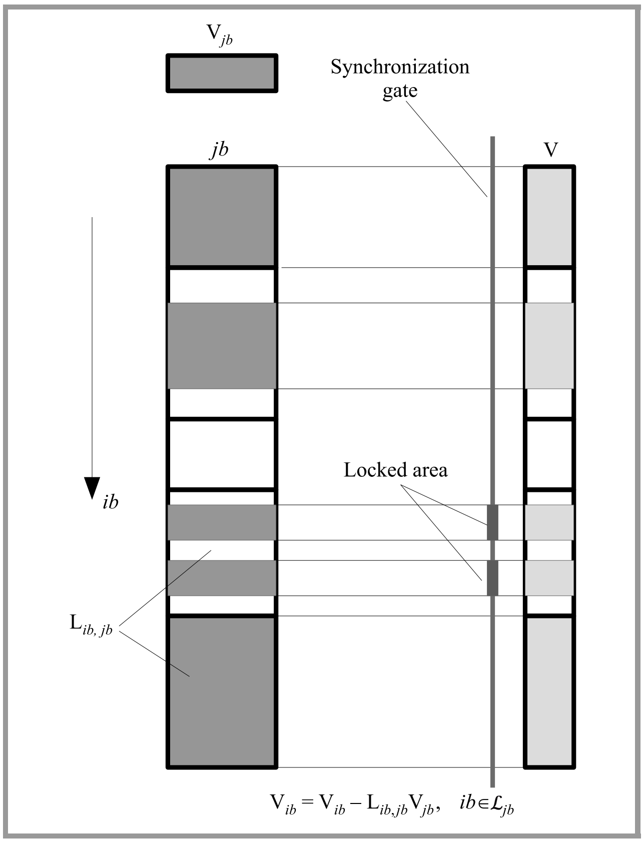
V_{jb}
Synchronization gate
jb
V
Locked area
ib
$L_{ib,jb}$
$V_{ib} = V_{ib} - L_{ib,jb}V_{jb}, \quad ib \in \pounds_{jb}$

Fig. 1. Block-column jb updates vector V.

A $\mathbf{L}_{ib,jb}$ block with at least one non-zero row is considered to be non-zero. Zero rows are not involved in the calculations, and no memory is allocated for them. There can be several submatrices with non-zero rows in a non-zero block $\mathbf{L}_{ib,jb}$ (penultimate block in Fig. 1). Details are given in [4], [5].

2.2. Sequential Triangular Solution Algorithm

The Algorithm 1 presents the sequential forward substitution procedure. To compactly record pseudo-code, the increment $++$ and decrement $--$ operators are used, similar to the corresponding operators of the C programming languages. The increment operator $a++$ or $++a$ means $a = a+1$, and the decrement operator $a--$ or $--a$ means $a = a-1$.

Algorithm 1. Forward reduction for a sparse lower triangular matrix. Sequential algorithm.

```
for $jb = 1; jb \le N; ++jb$ do
    $\mathbf{L}_{jb,jb} \leftarrow Diag\,[jb]\,.L_p$;
    $\mathbf{L}_{jb,jb}\mathbf{V}_{jb} = \mathbf{V}_{jb} \rightarrow \mathbf{V}_{jb}$;
    for $p = Pos\,[jb]\,; p < Pos\,[jb+1]\,; ++p$ do
        $ib = ind[p]$;
        $\mathbf{L}_{ib,jb} \leftarrow Space\,[p]\,.L_p$;
        $\mathbf{V}_{ib} = \mathbf{V}_{ib} - \mathbf{L}_{ib,jb}\mathbf{V}_{jb}$;
    end for
end for
```

The first **for** loop runs through the block-columns from left to right. Here, N_b is the number of block-columns. A system of linear algebraic equations with a dense lower triangular matrix $\mathbf{L}_{jb,jb}$ is solved for each current block-column, and a vector block is determined (a matrix block if there is more than one right-hand side) as $\mathbf{V}_{jb}$. Diagonal blocks do not contain empty rows, and pointers containing the addresses of the first elements of diagonal blocks are stored in the *Diag* array. Expression $\mathbf{L}_{jb,jb} \leftarrow Diag[jb].L_p$ returns the pointer to the first element of the block $\mathbf{L}_{jb,jb}$. In the second **for** loop, non-zero blocks of the block-column jb are selected. Here, p is the position number of the current block in a one-dimensional vector *Space*, which stores pointers to non-zero off-diagonal blocks arranged column by column. Expression $\mathbf{L}_{ib,jb} \leftarrow Space[p].L_p$ returns the pointer to the first element of the block $\mathbf{L}_{ib,jb}$. Array *ind* stores the block-row number for the current non-zero block $\mathbf{L}_{ib,jb}$.

2.3. Parallel Algorithm I

Algorithm 2 presents the first parallel approach (parallel algorithm I) relied upon to perform the forward reduction. Here, np is the number of threads and np queues Q_{ip}, $ip \in [0,\ np-1]$ are created before entering the parallel region for each block-column. Each queue element stores an index p and a number of the block-row ib. The procedure $Prepare\ Q_{ip}, \quad ip \in [0,\ np-1]$ uses algorithm 3 to improve load balance over the threads. Each block $\mathbf{L}_{ib,jb}$ is assigned a weight $weight\_p = M \cdot N \cdot NoRhs$, where M is the number of non-zero rows in the block, N is the number of columns, $NoRhs$ is the number of right-hand sides. The element of the array $sumweight[ip]$ contains the sum of weights of all blocks mapped onto thread ip, $ip \in [0,\ np-1]$. A thread min_ip which currently has the minimum sum of weights ($Find\ min\_ip$), is found for each non-zero block $\mathbf{L}_{ib,jb}$ of

Algorithm 2. Forward reduction for a sparse lower triangular matrix. Parallel algorithm I.

```
for $jb = 1; jb \leq N; ++jb$ do
    $\mathbf{L}_{jb,jb} \leftarrow Diag[jb].L_p$;
    $\mathbf{L}_{jb,jb}\mathbf{V}_{jb} = \mathbf{V}_{jb} \rightarrow \mathbf{V}_{jb}$;
    $Prepare \quad Q_{ip}, \quad ip \in [0, \quad np-1]$;
    parallel region
    $ip = omp\_get\_thread\_num()$;
    while $Q_{ip}$ is not empty do
        $(ib,p) \leftarrow Q_{ip}, \quad Q_{ip} \leftarrow Q_{ip}/(ib,p)$
        $\mathbf{L}_{ib,jb} \leftarrow Space[p].L_p$;
        $\mathbf{V}_{ib} = \mathbf{V}_{ib} - \mathbf{L}_{ib,jb}\mathbf{V}_{jb}$;
    end while
    end of parallel region
end for
```

the current block-column jb, and the given block is mapped onto this thread. The sum of weights is corrected for the given thread ($sumweight[min\_ip] += weight\_p$), and the element (ib,p) is added to the queue Q_{min_ip} : ($Q_{min\_ip} \leftarrow (ib,p)$).

Algorithm 3. Preparation of queues for block-column jb.

```
$sumweight[ip] \leftarrow 0, \; ip \in [0, \quad np-1]$;
for $p = Pos[jb]; p < Pos[jb+1]; ++p$ do
    $Find \; min\_ip$;
    $ib = ind[p]$;
    $sumweight[min\_ip] += weight\_p$;
    $Q_{min\_ip} \leftarrow (ib,p)$;
end for
```

In the parallel region (Algorithm 2) each thread runs a while loop until the Q_{ip} queue is empty. The nearest element is selected from the Q_{ip} queue and removed at each iteration: $(ib,p) \leftarrow Q_{ip}, \; Q_{ip} \leftarrow Q_{ip}/(ib,p)$. Thus, the ip thread gets access to the $\mathbf{L}_{ib,jb}$ block. Since each thread has access only to its individual queue, it can freely select and delete the elements of this queue. The results are written in the $\mathbf{V}_{ib}$ vector block, and the values of the indices ib are always different for different threads. No synchronization is needed here, which is an advantage of this algorithm. However, the columns of sparse matrices have a relatively small number of non-zero blocks in the given class of problems. Therefore, load balance is achieved not for all block-columns, which is a disadvantage of this algorithm.

2.4. Parallel Algorithm II

The idea behind algorithm II is probably close to that of algorithms using DAG to determine independent blocks of the solution vector $\mathbf{V}$, which can be processed in parallel. However, instead of forming and analyzing a complex DAG structure, the proposed algorithm is based on the analysis of a dynamic data structure – a *dependency vector*. As an example, let us consider the forward substitution procedure with a lower triangular matrix, corresponding to the example given in [5], Fig. 1:

$$\begin{pmatrix} \mathbf{L}_{11} & & & & & \\ \mathbf{L}_{21} & \mathbf{L}_{22} & & & & \\ 0 & 0 & \mathbf{L}_{33} & & & \\ 0 & 0 & 0 & \mathbf{L}_{44} & & \\ 0 & 0 & \mathbf{L}_{53} & 0 & \mathbf{L}_{55} & \\ \mathbf{L}_{61} & \mathbf{L}_{62} & \mathbf{L}_{63} & \mathbf{L}_{64} & \mathbf{L}_{65} & \mathbf{L}_{65} \end{pmatrix} \begin{pmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \mathbf{V}_3 \\ \mathbf{V}_4 \\ \mathbf{V}_5 \\ \mathbf{V}_6 \end{pmatrix} = \begin{pmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \mathbf{V}_3 \\ \mathbf{V}_4 \\ \mathbf{V}_5 \\ \mathbf{V}_6 \end{pmatrix} . \quad (1)$$

The solution procedure is as follows:

$$\begin{aligned}
&\mathbf{L}_{11}\mathbf{V}_1 = \mathbf{V}_1 \\
&\mathbf{L}_{22}\mathbf{V}_2 = \mathbf{V}_2 - \mathbf{L}_{21}\mathbf{V}_1 \\
&\mathbf{L}_{33}\mathbf{V}_3 = \mathbf{V}_3 \\
&\mathbf{L}_{44}\mathbf{V}_4 = \mathbf{V}_4 \\
&\mathbf{L}_{55}\mathbf{V}_5 = \mathbf{V}_5 \qquad\qquad\qquad\quad - \mathbf{L}_{53}\mathbf{V}_3 \\
&\mathbf{L}_{66}\mathbf{V}_6 = \mathbf{V}_6 - \mathbf{L}_{61}\mathbf{V}_1 - \mathbf{L}_{62}\mathbf{V}_2 - \mathbf{L}_{63}\mathbf{V}_3 - \mathbf{L}_{64}\mathbf{V}_4 - \mathbf{L}_{65}\mathbf{V}_5 \ .
\end{aligned}$$

We can immediately find the blocks $\mathbf{V}_1$, $\mathbf{V}_3$, $\mathbf{V}_4$, therefore the degree of dependency of these solution vector (matrix) blocks on other blocks is zero. The degree of dependency is 1 for blocks $\mathbf{V}_2$, $\mathbf{V}_5$, because $\mathbf{V}_2$ has to be corrected by $\mathbf{V}_1$, and $\mathbf{V}_5$ has to be corrected by $\mathbf{V}_3$. The $\mathbf{V}_6$ block has a degree of dependency of 5, because 5 blocks, $\mathbf{V}_1$ – $\mathbf{V}_5$, are involved in its correction. Therefore, the dependency vector is:

$$depend\_vect = (0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 5)^T \ . \qquad (2)$$

In order to be able to determine the block $\mathbf{V}_{jb}$, $jb \in [1, N_b]$, from the solution of a system of equations with a lower triangular matrix:

$$\mathbf{L}_{jb,jb}\mathbf{V}_{jb} = \mathbf{V}_{jb} \ , \qquad (3)$$

the element of the vector has to be $depend\_vect[jb] = 0$, and the corresponding element of the vector $depend\_vect$ is reduced by 1 at each correction. As soon as the $\mathbf{V}_{jb}$ block is determined from the solution of the system of equations (3), it has to be used in the correction (removal of dependency) of other blocks. After determining the $\mathbf{V}_{jb}$ block from Eq. (3) we set $depend\_vect[jb] = -1$. By applying these simple rules and modifying $depend\_vect$ at each step, we obtain the result given in Table 1 for this example.

Table 1
Evolution of dependency vector on each solution step

Step	[1]	[2]	[3]	[4]	[5]	[6]
0	0	1	0	0	1	5
1	–1	0	–1	–1	0	2
2	–1	–1	–1	–1	–1	0
3	–1	–1	–1	–1	–1	–1

The initial state of $depend\_vect$ is shown at step 0, the indices of its elements are denoted as $[\ldots]$. The numbers of the blocks for which the degree of dependency is 0 are put in the queue Q: $Q = 1,3,4$. Each thread selects the number jb from the queue Q, removes it and adjusts $depend\_vect[jb] = -1$. The system of equations (3) is solved for the given value of jb and the obtained block $\mathbf{V}_{jb}$ is used in the correction (removal of dependency) of the blocks ib:

$$\forall ib \in \pounds_{jb} : \mathbf{V}_{ib} = \mathbf{V}_{ib} - \mathbf{L}_{ib,jb}\mathbf{V}_{jb} \ . \tag{4}$$

Algorithm 4. Forward reduction for a sparse lower triangular matrix. Parallel algorithm II

```
 1: Initialization: $tot\_count = 0; depend\_vect[ib] \leftarrow 0$
 2: for $jb = 1; jb \leq N_b; ++jb$ do
 3:    for $p = Pos[jb]; p < Pos[jb+1]; ++p$ do
 4:       $Space[p].is\_produced = 0;\ depend\_vect[jb]++;$
 5:    end for
 6:    $if(depend\_vect[jb] == 0)$
          $Q \leftarrow jb;$
 7: end for
 8: parallel region
 9: while $tot\_count < N_b$ do
10:    lock
11:       $jb \leftarrow Q; Q \leftarrow Q/jb;$
12:    unlock
13:    if $depend\_vect[jb] == 0$ then
14:       $\mathbf{L}_{jb,jb} \leftarrow Diag[jb].L_p;$
15:       $\mathbf{L}_{jb,jb}\mathbf{V}_{jb} = \mathbf{V}_{jb} \rightarrow \mathbf{V}_{jb};$
16:       lock
17:          $depend\_vect[jb] = -1; tot\_count++;$
18:       unlock
19:    end if
20:    $count\_postponed = 0;$
21:    for $p = Pos[jb]; p < Pos[jb+1]; ++p$ do
22:       if $Space[p].is\_produced$ then
23:          $continue;$
24:       end if
25:       $ib = ind[p];$
26:       if $(first\_eqn\_ib \div last\_eqn\_ib)!locked$ then
27:          lock_gate $(first\_eqn\_ib \div last\_eqn\_ib);$
28:       else
29:          $Space[p].is\_produced = 0;$
30:          $count\_postponed++;\quad continue;$
31:       end if
32:       $\mathbf{L}_{ib,jb} \leftarrow Space[p].L_p;$
33:       $\mathbf{V}_{ib} = \mathbf{V}_{ib} - \mathbf{L}_{ib,jb}\mathbf{V}_{jb};$
34:       unlock_gate$(firs\_eqn\_ib \div last\_eqn\_ib);$
35:       lock
36:       $depend\_vect[ib]--;$
37:       if $depend\_vect[ib] == 0$ then
38:          $Q \leftarrow ib;$
39:       end if
40:       unlock
41:       $Space[p].is\_produced = 1;$
42:    end for
43:    if $count\_postponed$ then
44:       lock; $Q \leftarrow jb$; unlock;
45:    end if
46: end while
47: end of parallel region
```

Here, $\pounds_{jb}$ is a non-zero structure of the block-column jb and $depend\_vect[ib]--$ is adjusted at each correction. As soon as $depend\_vect[ib]$ becomes zero, we put ib in the queue Q – all dependencies are removed from this block, and it can be obtained at the next step.

Following the given algorithm, we calculate the blocks $\mathbf{V}_1$, $\mathbf{V}_3$, $\mathbf{V}_4$, for which the degree of dependency is 0, at the first step. We correct $\mathbf{V}_2$, $\mathbf{V}_5$, $\mathbf{V}_6$ in (2). The degree of dependency of blocks $\mathbf{V}_2$ and $\mathbf{V}_5$ becomes equal to zero, and that of the block $\mathbf{V}_6$ becomes equal to 2. The numbers of block-columns 2 and 5 are added to the queue Q. Blocks $\mathbf{V}_2$ and $\mathbf{V}_5$ are determined at the step 2 and are then involved in the correction of the block $\mathbf{V}_6$. As a result, all dependencies are removed from the block $\mathbf{V}_6$, and index 6 is put in the queue Q. Block $\mathbf{V}_6$ is obtained at the step 3. The presented approach is described in Algorithm 4 (parallel algorithm II).

The number of obtained blocks $\mathbf{V}_{jb}, jb \in [1, N_b]$ is set to zero ($tot\_count = 0$) at the initialization stage (lines 1–7), and each off-diagonal block of the lower triangular matrix is assigned with the "not produced" status ($is\_produced = 0$) – none of the blocks has performed its work on the correction of blocks of the vector $\mathbf{V}_{jb}$. The degree of dependency of each block $\mathbf{V}_{jb}$ – $depend\_vect[jb], jb \in [1, N_b]$ is determined. The numbers of all independent blocks are put in the queue Q ($if(depend\_vect[jb] == 0) \ \ Q \leftarrow jb$).

The number of **while** loops running in the parallel region (lines 8–47) is equal to the number of threads it contains. Each **while** loop runs until the number of found blocks tot_count is equal to N_b – $while(tot\_count < N_b)$.

Then, the current thread in the critical section (lines 10–12) selects the closest number jb and removes it from the queue ($jb \leftarrow Q; Q \leftarrow Q/jb$) at each iteration. If $depend\_vect[jb] == 0$ (lines 13–19), the system of linear algebraic equations with a dense lower triangular matrix is solved $\mathbf{L}_{jb,jb}\mathbf{V}_{jb} = \mathbf{V}_{jb} \rightarrow \mathbf{V}_{jb}$ – the *dtrsm* procedure from the Intel MKL library [25] is used. The following value is set in the critical section (lines 16–18) $depend\_vect[jb] = -1; tot\_count++;$. Off-diagonal blocks which have not yet been involved in the correction of the $\mathbf{V}_{ib}$ block (lines 25–41) are selected in the **for** loop (lines 21–42). Then, a check is performed of whether the region $[first\_eqn\_ib, last\_eqn\_ib]$ is not occupied by another thread, where $first\_eqn\_ib$ and $last\_eqn\_ib$ are the first and last equations in the corrected block $\mathbf{V}_{ib}$, respectively. If the specified region is free, the current thread locks it (lines 26–27) and performs block correction (lines 32–33). In this case the *dgemm* procedure from the Intel MKL library is used. If the specified region is occupied by another thread, then the $\mathbf{L}_{ib,jb}$ block is assigned the "not produced" status, the pending tasks counter $count\_postponed$ is increased by one, and the next iteration is started (lines 29–30).

As soon as the $\mathbf{V}_{ib}$ block is corrected, the current thread unlocks the locked "synchronization gate" region (line 34) and allows another thread to correct the same $\mathbf{V}_{ib}$ block. Then, in the critical section (lines 35–40), the dependency of the $\mathbf{V}_{ib}$ block is reduced by 1, and if all dependencies are removed for this block ($depend\_vect[ib] == 0$), the number of the block ib is put in the queue Q – $Q \leftarrow ib$. The $\mathbf{L}_{ib,jb}$ block is assigned the "produced" status ($is\_produced = 1$) and the **for** loop proceeds to the next iteration.

Once the **for** loop is over, the pending task counter $count\_postponed$ is checked. If there is at least one postponed task, the block-column jb is added to the queue Q again (lines 43–45), and will later be able to perform its work only for the blocks with the "not produced" status.

Thus, the proposed algorithm enables to perform parallel processing of several block-columns at once if the matrix structure allows it, but it requires synchronization. Firstly, the extraction and removal of block-column numbers from the Q queue, as well as the addition of new block-columns to the Q queue, must be performed by each thread in the critical section, bounded by the **lock** and **unlock** operations. Secondly, the state $depend\_vect$ state has to be changed in the critical section as well. Thirdly, the correction of each $\mathbf{V}_{ib}$ block must be performed by each thread exclusively, in order to eliminate the situation when several threads correct the same $\mathbf{V}_{ib}$ block simultaneously. The so-called "synchronization gate" has been created for this purpose (Fig. 1). As soon as the $\mathbf{V}_{jb}$ block begins to correct the $\mathbf{V}_{ib}$ block, the numbers of the equations corresponding to the corrected rows of the $\mathbf{V}_{ib}$ block are locked (**lock_gate**) with the use of the interlocked functions running at the atomic level, and all other competing threads are forced to skip the correction of this block and proceed to correcting the next block in the list $\forall ib \in £_{jb}$. Once the correction of the $\mathbf{V}_{ib}$ block by the given thread is finished, the "synchronization gate"unlocks the corresponding equation numbers (**unlock_gate**), and another thread gets access to modifying the $\mathbf{V}_{ib}$ block. The threads for which the $\mathbf{L}_{ib,jb}$ blocks do not get access to correcting the $\mathbf{V}_{ib}$ block do not pass to idle but are used to select the next block-column jb from the queue Q and perform other useful tasks.

Back substitution algorithms are not considered here, because they are similar to their forward substitution counterparts.

3. Numerical Results

Numerical results were obtained on a workstation with a 16-core AMD Opteron 6276 processor 2.3/3.2 GHz, 64 GB DDR3 RAM, OS Windows Server 2008 R2 Enterprise SP1, 64-bit. Examples 2 and 3 were also solved on a computer with a 4-core Intel Core i7 7700 CPU 3.60 GHz (4 physical cores, 8 logical cores), 32 GB RAM, Windows 10 Pro OS, 64-bit. Example 1 exceeds the amount of RAM on this computer. The speed-up for PARFES and PARDISO solvers obtained on the second computer with an increase in the number of threads within 4 cores does not practically differ from the results obtained on a computer with the AMD processor, so we limited ourselves to presenting the results obtained on the computer with a 16-core AMD Opteron processor.

Microsoft Visual Studio 2017 IDE with the v141 platform toolset has been applied. The compiler optimization option in the release version is O2 – maximum performance. The "Enable Enhanced Instruction Set" option was selected as /arch:AVX2, which corresponds to vectorization of calculations with a vector length of 256 bytes (4 double words) with supporting of FMA instruction set when compiling the code. However, selection of this option does not exert any significant impact on code performance, since all leading operations with double precision are performed by the *dtrsm* and *dgemm* procedures from the Intel MKL library.

All problems are taken from the collection of real-life problems by SCAD Soft IT company.

3.1. Example 1

Let us consider a finite element model of a multi-storey building with the dimension of 2,989,476 equations (Fig. 2).

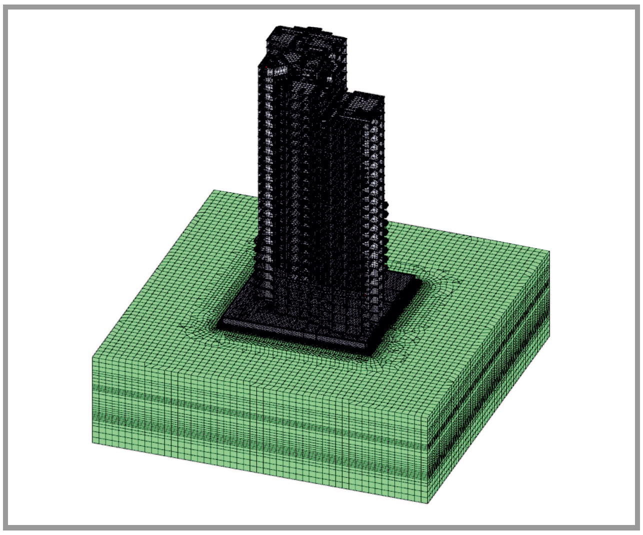

Fig. 2. Finite element model of a multi-storey building.

The soil prism is modeled by solid finite elements, which generates submatrices of relatively high density in a sparse stiffness matrix. The size of the factorized lower triangular matrix is about 36 GB. 30 load cases (30 right-hand sides) are considered, which is a typical number for static analysis of multi-storey buildings.

Table 2 provides a comparison of the duration of forward-back substitutions for different solvers. For comparison, we have selected the PARDISO solver presented by the Intel MKL library [25], which has successfully proven itself on multi-core computers with shared memory, and a sparse, direct looking-left Cholesky solver, which is taken from [26]. This solver ideally bypasses non-zero elements of the sparse matrix, however, the saxpy algorithm related to the first level BLAS is implemented in the inner loop. This method is significantly inferior in performance to other methods

Table 2
Forward – back substitutions [s]

Number of threads	Sparse direct	PARFES alg. I	PARFES alg. II	PARDISO
1	1670	193	203.5	131.7
2	1119	123.8	105.7	101.5
4	874	76.4	54.4	83.5
6	446.8	61.1	38.7	72.5
8	305.4	54.4	31.5	68.4
10	400.7	44.7	33.9	70.1
12	263.5	48.2	33.8	64.4
14	251	36.9	33.8	71.2
16	301	42.2	41.1	63.3

due to the lack of cache memory blocking, register blocking and vectorization of calculations. We have performed multithreaded parallelization of this method. Algorithm 5 presents the sparse triangular solution algorithm for this method.

The lower triangular sparse matrix is stored in the data structure represented by the *Diag* array for storing diagonal elements, the *Space* array for storing non-zero elements of the lower triangular matrix arranged column by column, Sherman's compressed array of the first index *ind* with the position pointer $iPos[j]$ for the first index in the array *ind* for the j column and the position pointer $Pos[j]$ in the *Space* array for the first non-zero element in the j column. Right-hand side vectors are arranged column by column in the **V** array. Here, *Neq* is the number of equations, *NoRhs* is the number of right-hand sides. The back substitution algorithm is similar to Algorithm 5.

Algorithm 5. Parallel triangular solution algorithm for a sparse direct solver

```
for $j = 1; j \le Neq; ++j$ do
  for parallel $k = 0; k < NoRhs; ++k$ do
    $ip = omp\_get\_thread\_num();$
    $rhsj = \mathbf{V}[j + k \cdot Neq]/Diag[j];$
    $V[j + k \cdot Neq] = rhsj;$
    $ipos = iPos[j];$
    for $ii = Pos[j]; ii < Pos[j+1]; ++ii$ do
      $i = ind[ipos++];$
      $\mathbf{V}[i + k \cdot Neq] = Space[ii] \cdot rhsj;$
    end for
  end for parallel
end for
```

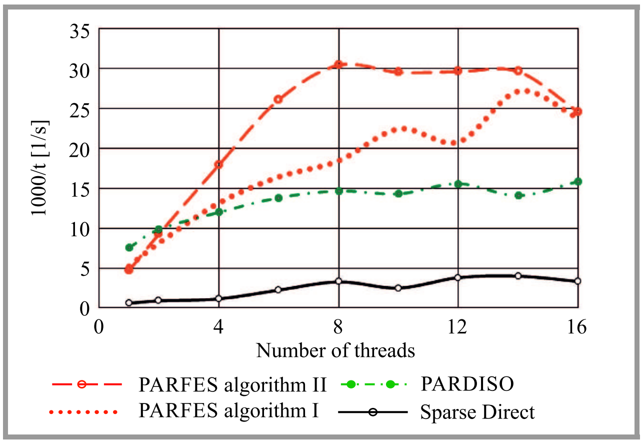

Fig. 3. Example 1: forward and back substitutions – comparison of performance for different solvers.

Figure 3 shows a comparison of performance of different solvers with an increase in the number of threads. Each solver has a different number of non-zero elements in the lower triangular matrix due to different techniques of dividing a sparse matrix into dense blocks, despite the fact that the same ordering method, METIS, was used for all methods [27].

The user is interested in the computation time, but not in the performance of the method itself. Therefore, to obtain a more objective comparison, the use of parameter $1000/t$ is proposed as a measure of performance, which is inversely proportional to the computation time t, as a measure of performance. The proportionality factor is assumed to be the same for all methods, and its value is based on the readability of results. The procedure provided by the Intel MKL library for determining the number of CPU cores works incorrectly on a computer with the AMD Opteron 6276 processor. Therefore, PARDISO treats this CPU as having 8 cores and 16 logical processors in the hyperthreading mode, and if the number of threads exceeds 8, limits the number of threads to 8, and the task manager shows that only 8 threads are running. In fact, this CPU has 16 physical cores and does not support hyper threading. In Table 2, in the PARDISO column, the computing time with the number of threads exceeding 8 does not decrease almost at all.

Dividing the sparse matrix into dense blocks provides for an efficient use of the processor cache and improves performance of the triangular solution procedure by several times. This may be seen from the comparison of curves for solvers using cache blocking and for the sparse direct solver, where the cache memory blocking is not performed. The best result is achieved by the PARFES solver using algorithm II on 8 threads. Further, as the number of threads increases, performance decreases. Starting from 14 threads, both PARFES algorithms show approximately the same performance. The speed-up of forward and back substitutions is much lower for PARDISO than for PARFES.

Since the performance and speed-up of I and II triangular solution algorithms implemented in the PARFES solver turned out to be significantly higher than in PARDISO and the sparse direct solver, in further examples we restrict ourselves to considering only these algorithms.

3.2. Example 2

Let us consider a finite element model of a 5-span arch bridge (Fig. 4).

To perform calculations related to the slow movement of a load, an influence surface represented by a set of nodes

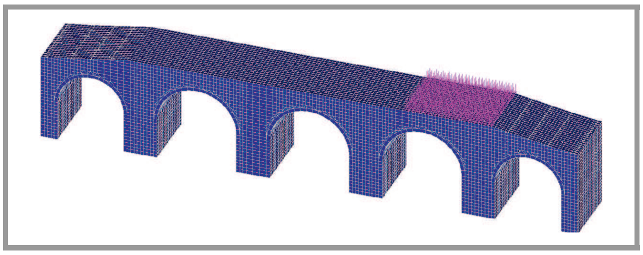

Fig. 4. Finite element model of an arch bridge.

M of the finite element model enveloped by the moving load is generated. A concentrated vertical unit force is applied in each node, and the respective displacements are determined $\mathbf{x}_i, i \in [1, NoRhs]$, where $NoRhs$ is the number of nodes included in M. The displacement vector caused by the moving load $\mathbf{d}(t)$, which occupies a certain position at the current time t, is:

$$\mathbf{d}(t) = \sum_{j=1}^{nrhs} a_j(t)\mathbf{x}_j \ , \qquad (5)$$

where t is the time parameter, $a_j(t)$ is the value of the nodal load in the j-th node determined by the moving load, $nrhs$ is the number of nodes of the influence surface, covered by the moving load at time t ($nrhs \leq NoRhs$).

This formulation of the problem leads to the single factorization of a sparse matrix and to execution of forward and back substitutions with the number of right-hand sides equal to $NoRhs$. There are 162,603 equations and 12,797 right-hand sides in the considered example. Therefore, it is critical to achieve high performance of the triangular solution algorithm and to ensure stable speed-up with an increase in the number of threads. Figure 5 shows a comparison of the speed-up of the two considered PARFES solver algorithms. Due to the large number of right-hand sides, each thread performs significant computational work with relatively few instances of communication between the threads during the forward and back substitutions. Algorithm II turned out to be significantly more efficient than algorithm I.

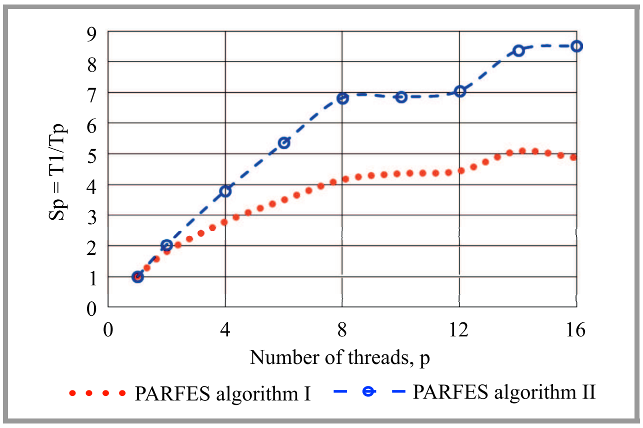

Fig. 5. Example 2: PARFES forward and back substitutions.

3.3. Example 3

Let us consider a FEM model of a multi-storey building (Fig. 6), with three towers on a common podium.

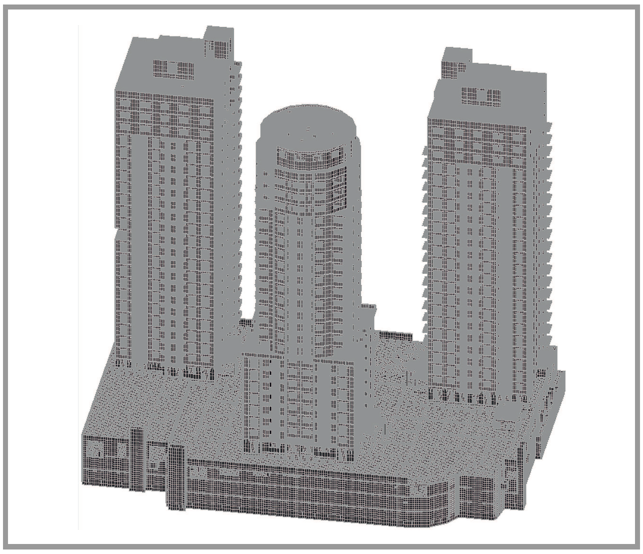

Fig. 6. Finite element model of a multi-storey building with three towers.

The dimension of the design model is 4,262,958 equations. There are 15 right-hand sides, but the solution method performs forward and back substitutions 15 times with only one right-hand side. This sequential mode simulates algorithms used for integrating motion equations or for solving nonlinear problems. It is impossible to combine the right-hand sides into one pack (packed mode) because a right-hand side cannot be generated until the solution for the previous right-hand side is obtained. The sequential mode turns out to be the most unfavorable one, because if there is only one right-hand side, each thread performs minimum computational work, and the number of instances of communication between the threads is the same as if there were many right-hand sides. Moreover, the performance of the *dgemm* procedure (Algorithm 4, line 33), as well as the efficiency of the use of CPU cache are significantly reduced

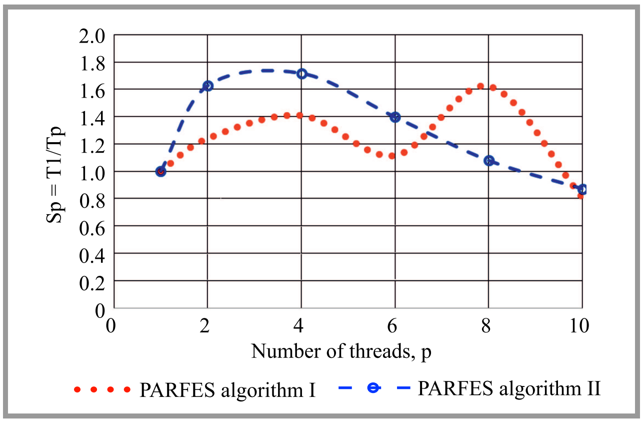

Fig. 7. Example 3: PARFES forward and back substitutions – sequential mode.

when there is one right-hand side only, while the load on RAM memory increases, because the matrix multiplication algorithm (level 3 BLAS) turns into a matrix-vector multiplication algorithm (level 2 BLAS). Figure 7 shows the minimum speed-up with an increase in the number of threads up to 4, and then the parallelization efficiency decreases.

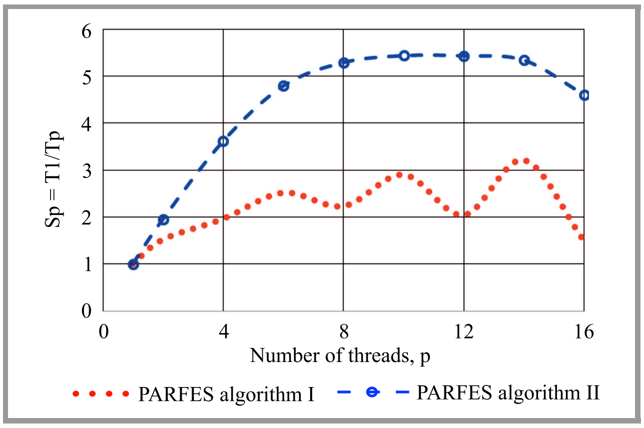

Fig. 8. Example 3: PARFES forward and back substitutions – packed mode.

In scenarios in which it is possible to pack all right-hand sides into one block, the performance and speedup of the triangular solution algorithm become much better than in the case of sequential mode. Figure 8 shows the speed-up with an increase in the number of threads for the packed mode scenario. The transition from sequential mode to packed mode (if possible) reduces the duration of forward-back substitutions by several times (Table 3). Moreover, the speed-up with an increase in the number of threads is significantly greater in the packed mode than in its sequential counterpart.

Table 3
Example 3: Duration of forward - back substitutions [s]. PARFES: algorithm II

Number of threads	PARFES, sequential mode	PARFES, packed mode
1	200.4	64.10
2	132.3	33.35
4	123.1	16.79
6	166.5	13.07
8	211.6	12.03
10	259.3	12.12
12	–	13.21
14	–	14.43
16	–	17.43

4. Conclusions

Parallel algorithm 2 turned out to be more efficient than parallel algorithm 1 in all tests. The maximum speed-up for the most unfavorable sequential mode was obtained with 4 threads. The performance decreases with the further increase in the number of threads. This means that in the case of a single right-hand side, the number of threads has to be limited to 4. For the most common engineering problems with 5–80 right-hand sides, the maximum performance was achieved with 8 threads. In the scenario involving generation of influence surfaces (several thousand right-hand sides), the maximum performance was achieved, during the tests, on a computer with a 16-core AMD Opteron 6276 processor with 16 threads.

Acknowledgements

The author is deeply grateful to SCAD Soft for providing support and for the collection of real-life problems involving design models created by SCAD Office users.

References

[1] S. Yu. Fialko, "Iterative methods for solving large-scale problems of structural mechanics using multi-core computers", *Archiv. of Civil and Mechan. Engin.*, vol. 14, pp. 190–203, 2014 (doi: 10.1016/j.acme.2013.05.009).

[2] A. V. Perelmuter and S. Yu. Fialko, "Problems of computational mechanics relate to finite-element analysis of structural constructions", *Int. J. for Computat. Civil and Struct. Engin.*, vol. 1, no 2, 2005, pp. 72–86 (doi: 10.1615/IntJCompCivStructEng.v1.i2.70).

[3] P. R. Amestoy, I. S. Duff, and J-Y. L'Excellent, "Multifrontal parallel distributed symmetric and unsymmetric solvers", *Comp. Meth. Appl. Mechan. Engin.*, vol. 184, no. 2, pp. 501–520, 2000 (doi: 10.1016/S0045-7825(99)00242-X).

[4] S. Yu. Fialko, "PARFES: A method for solving finite element linear equations on multi-core computers", *Advan. in Engin. Software*, vol. 40, no. 12, pp. 1256–1265, 2010 (doi: 10.1016/j.advengsoft.2010.09.002).

[5] S. Yu. Fialko, "Parallel direct solver for solving systems of linear equations resulting from finite element method on multi-core desktops and workstations", *Comp. and Mathema. with Appl.*, vol. 70, pp. 2968–2987, 2015 (doi: 10.1016/j.camwa.2015.10.009).

[6] O. Schenk and K. Gartner, "Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems", *Parall. Comput.*, vol. 28, pp. 187–197, 2002 (doi: 10.1016/S0167-8191(01)00135-1).

[7] S. Fialko and V. Karpilovskyi, "Time history analysis formulation in SCAD FEA software", *J. of Measur. in Engin.*, vol. 6, no. 4, pp. 173–180, 2018 (doi: 10.21595/jme.2018.20408).

[8] S. Fialko and V. Karpilovskyi, "Multithreaded parallelization of the finite element method algorithms for solving physically nonlinear problems", in *Proc. of the Federated Conf. on Comp. Sci. and Inform. Syst.*, Poznań, Poland, 2018, vol. 15, pp. 31–318 (doi: 10.15439/2018F40).

[9] S. Yu. Fialko, E. Z. Kriksunov, and V. S. Karpilovskyy, "A block Lanczos method with spectral transformations for natural vibrations and seismic analysis of large structures in SCAD software", in *Proc. 15th Int. Conf. on Comp. Methods in Mechan. CMM-2003*, Gliwice, Poland, 2003, pp. 12–130 [Online]. Available: https://pdfs.semanticscholar.org/046c/c4f0e921c75f6dc081909324de3c31a1f8ea.pdf

[10] S. Fialko and V. Karpilovskyi, "Block subspace projection preconditioned conjugate gradient method for structural modal analysis", in *Proc. of the Federated Conf. on Comp. Sci. and Inform. Syst.*, Praha, Czech Republik, 2017, vol. 11, pp. 497–506 (doi: 10.15439/2017F64).

[11] E. Gallopoulos, B. Philippe, and A. H. Sameh, *Parallelism in Matrix Computations*. New York, London: Springer, 2016 (ISBN: 9789401771870).

[12] C. C. K. Mikkelsen, A. B. Schwarz, and L. Karlsson, "Parallel robust solution of triangular linear systems. Concurrency and computation, practice and experiments", *Concurr. and Comput. Pract. and Exper.*, vol. 30, pp. 1–19, 2018, (doi: 10.1002/cpe.5064).

[13] T. Iwashita and M. Shimasaki, "Algebraic multi-color ordering method for parallelized ICCG solver in unstructured finite element analyses", *IEEE Trans. on Magnet.*, vol. 38, no. 2, pp. 429–432, 2002 (doi: 10.1109/20.996114).

[14] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU", NVIDIA Tech. Rep. NVR-2011-001, June 2011, pp. 1–21 [Online]. Available: https://research.nvidia.com/sites/default/files/pubs/2011-06_Parallel-Solution-of/nvr-2011-001.pdf (accessed on 11.04.2019).

[15] S. Fialko and F. Zeglen, "Preconditioned conjugate gradient method for solution of large finite element problems on CPU and GPU", *J. of Telecommun. and Inform. Technol.*, no. 2, 2016, pp. 26–33 [Online]. Available: https://www.il-pib.pl/czasopisma/JTIT/2016/2/26.pdf

[16] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "Synchronization-free algorithm for parallel sparse triangular solves", in *Euro-Par 2016: Parallel Processing 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, P.-F, Dutot and D. Trystram, Eds. *LNCS*, vol. 9833, pp. 617–630. Springer, 2016 (doi: 10.1007/978-3-319-43659-3_45).

[17] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning", in *Euro-Par 2015: Parallel Processing 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, J. L. Träff, S. Hunold, and F. Versaci, Eds. *LNCS*, vol. 9233, pp. 650–661. Springer, 2015 (doi:: 10.1007/978-3-662-48096-0_50).

[18] R. Vuduc *et al.*, "Automatic performance tuning and analysis of sparse triangular solve", *Semantic Scholar*, 2002 [Online]. Available: https://www.semanticscholar.org/paper/Automatic-Performance-Tuning-and-Analysis-of-Sparse-Vuduc-Kamil/002ed5f20260cb140cd12da352db61daf6bd3984 (accessed on 12.04.2019).

[19] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors impacting performance of multithreaded sparse triangular solve", in *High Performance Computing for Computational Science – VECPAR 2010. 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*, J. M. Laginha M. Palma *et al.*, Eds. *LNCS*, vol. 6449, pp. 32–44. Springer, 2010 (doi: 10.1007/978-3-642-19328-6_6).

[20] E. Rothberg and A. Gupta, "Parallel ICCG on a hierarchical memory multiprocessor-addressing the triangular solve bottleneck", *Parall. Comput.*, vol. 18, no. 7, 1992, pp. 719–741 (doi: 10.1016/0167-819(92)90041-5).

[21] F. L. Alvarado, A. Pothen, and R. Schreiber, "Highly parallel sparse triangular solution", in *Graph Theory and Sparse Matrix Computation*, A. George, J. R. Gilbert, and J. W. H. Liu, Eds. Springer-Verlag, 1993 (ISBN: 9781461383710).

[22] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, "Adapting sparse triangular solution to GPUs", in *Proc. 41st Int. Conf. on Parall. Process. Worksh.*, Pittsburgh, PA, USA, 2012 (doi: 10.1109/ICPPW.2012.23).

[23] S. Marrakchi and M. Jemni, "Fine-grained parallel solution for solving sparse triangular systems on multicore platform using OpenMP interface", in *Proc. Int. Conf. on High Perform. Comput. & Simul. HPCS 2017*, Genoa, Italy, 2017, pp. 659–666 (doi: 10.1109/HPCS.2017.102).

[24] E. Totoni, M. T. Heath, and L. V. Kale, "Structure-adaptive parallel solution of sparse triangular linear systems", *Parall. Comput.*, vol. 40, 2014, pp. 454–470 (doi: 10.1016/j.parco.2014.06.006).

[25] "Developer Guide for Intel Math Kernel Library for Windows", Intel Math Kernel Library [Online]. Available: https://software.intel.com/en-us/mkl-windows-developer-guide (accessed on 20.04.2019).

[26] A. George and J. W. H. Liu, *Computer Solution of Sparse Positive Definite Systems*. New Jersey: Prentice-Hall, 1981 (ISBN: 0131652745).

[27] G. Karypis and V. Kumar, "METIS: Unstructured graph partitioning and sparse matrix ordering system", Tech. Rep., Department of Computer Science, University of Minnesota, Minneapolis, 1995 [Online]. Available: https://dm.kaist.ac.kr/kse625/resources/metis.pdf

Sergiy Fialko studied structural mechanics and obtained his degree from the Institute of Mechanics in Kiev, Ukraine in 1983. He worked at the Steel Structures Research Institute in Kiev, at the RoboBAT software company in Cracow, Poland, and at the National University of Construction and Architecture in Kiev, where he obtained his habilitation degree in 2004. Since 2007, he has been working at the Cracow University of Technology. He has also held, for a number of years now, the position of a scientific consultant at SCAD Soft.

https://orcid.org/0000-0001-7543-5744
E-mail: sergiy.fialko@gmail.com
Institute of Computer Science
Faculty of Physics, Mathematics
and Computer Science
Cracow University of Technology
Warszawska 24
31-155 Cracow, Poland