Proceedings of the Federated Conference on Computer Science and Information Systems pp. 311–318
DOI: 10.15439/2018F40
ISSN 2300-5963 ACSIS, Vol. 15

Multithreaded Parallelization of the Finite Element Method Algorithms for Solving Physically Nonlinear Problems

Sergiy Fialko
Tadeusz Kościuszko Cracow University of Technology
ul. Warszawska 24, 31-155 Kraków, Poland
Email: sergiy.fialko@gmail.com

Viktor Karpilowskyi
IT company SCAD Soft
ul. Osvity 3a, office 1, 2, Kiev, Ukraine
Email: kvs@scadsoft.com

***Abstract*—The parallelization of the leading procedures of the finite element method applied to solving physically nonlinear problems of structural mechanics is considered.**

I. Introduction

THE SOLUTION of physically nonlinear problems of structural mechanics by the finite element method requires a large number of calculations. The vast majority of such problems are solved on multi-core shared memory computers - desktops, laptops and multiprocessor workstations with SMP (Symmetric Multiprocessing) architecture. When using the finite element method, the most time-consuming procedures are the assembling of a tangent stiffness matrix, the evaluation of an internal force vector, and the solution of a system of linear algebraic equations with a sparse symmetric matrix. Solver PARDISO [1] from the Intel Math Kernel Library (MKL) [2] or PARFES [3], [4] is used to solve systems of linear algebraic equations with a symmetric sparse tangent stiffness matrix. These solvers have successfully proven themselves on multi-core computers of SMP architecture and demonstrate stable acceleration with an increase in the number of cores during the factorization stage. In addition, forward and backward substitutions are also parallelized. Therefore, this paper considers parallel algorithms for the assembling of a tangent stiffness matrix and the calculation of an internal force vector.

A. Assembling of the stiffness matrix

1) Related works: In [13] is presented a parallel node-by-node assembling approach, when each block of the global stiffness matrix, corresponding to given node, is processed on the same processor. The blocks of an element matrices related to given node are evaluated on this processor. There are no overlapping of blocks in the global stiffness matrix and no communication between processors are needed.

In [14] is considered only banded matrices. The set of consecutive rows are taken as a synchronization region. Such a partitioning the matrix into a sufficient number of synchronization regions allows to avoid simultaneous modification of the same elements by the different threads.

The algorithm [15] assembles the stiffness matrix by groups of rows related to each node of the finite element mesh. Each processor will only assemble the rows related to a specific group of nodes. Therefore, no synchronization is required because each processor updates only their addresses of the memory.

The method [16] creates for each element the list of the processors onto which the associated columns in global stiffness matrix composing this element have been mapped. If this list consists in only one processor, the finite element is totally local to this processor, and non totally local otherwise. The totally local elements are mapped to each processor. The remaining finite elements are processed on several processors and need a synchronization.

In [17] before assembling is precomputed a coloring of the finite elements such that no two elements of the same color share any given degree of freedom. The appropriate heuristic coloring algorithm is presented. The several algorithms realising assembling on GPU, are presented.

We present the procedure of the stiffness matrix assembling (section II.B) which require no synchronization between threads and demonstrates an almost perfect load balance even for different types of finite elements – triangular and quadrilateral shell elements, spatial bar elements and so on. It is the typical situation in structural analysis when design model consists of different types of finite elements, and evaluation of their stiffness matrices requires the quite different computational efforts.

2) Assembling procedure: The procedure for a tangent stiffness matrix assembling is as follows:

$$\mathbf{K}_t = \sum_{e=1}^{Ne} \mathbf{P}_e^T \mathbf{K}_{e,t} \mathbf{P}_e, \tag{1}$$

where $\mathbf{K}_{e,t}$ is a tangent stiffness matrix of the e-th finite element, $\mathbf{P}_e$ is a permutation matrix and N_e is a number of finite elements in the design model. In the case of physically

This work was supported by IT company SCAD Soft

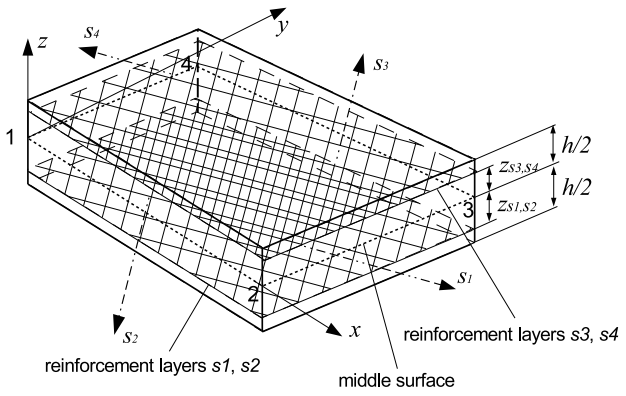

Fig. 1. The flat shell finite element

nonlinear problems, the calculation of the tangent stiffness matrix for the thin plane shell finite element (Fig. 1) is represented as a sum of the following type integrals:

$$\mathbf{K}_e = \int\limits_{\Omega} \mathbf{B}^T(\Omega) \left[\int\limits_{-h/2}^{h/2} f(\Omega, z)\, dz \right] \mathbf{B}(\Omega)\, d\Omega, \qquad (2)$$

where Ω is an in-plane domain ($d\Omega = dxdy$), h is the shell thickness, $\mathbf{B}(\Omega)$ is a deformation matrix [5], $s_1, s_2, ...$ are the axes of the reinforcement layers, $z_{s1}, z_{s2}, ...$ are the distances between reinforcement layers $s_1, s_2, ...$ and middle surface.

The trapezoid method is applied to calculate the integral over the shell thickness. In this case, the shell is divided into 10 - 40 layers through thickness. In addition, reinforcing rods of the same direction form a layer of reinforcement. Usually, the number of reinforcement layers is 4, although it may be arbitrary. The integral over the domain of the finite element Ω is computed by the Gauss method using the 2×2 integration scheme. The components of the stress and strain tensors are calculated at each Gaussian point. The number of such points for a given type of finite element is $2 \times 2 \times$ (the number of layers plus the number of reinforcement layers). The tangent stiffness matrix for other types of finite elements is defined similarly.

Thus, the procedure for the tangent stiffness matrix assembling requires significant computational effort. The tangent stiffness matrix assembling time is of the same order as the factorization time of this matrix.

B. Internal force vector evaluation

The internal force vector is calculated as follows:

$$\mathbf{f}^{int} = \sum_{e=1}^{Ne} \mathbf{P}_e^T \mathbf{r}_e \mathbf{P}_e, \qquad (3)$$

where

$$\mathbf{r}_e = \mathbf{K}_e \mathbf{q}_e, \qquad (4)$$

$\mathbf{q}_e$ is a nodal displacement vector and $\mathbf{r}_e$ is a nodal reaction vector for the e-th finite element. The stiffness matrix $\mathbf{K}_e$ has to be calculated for each finite element in the expression (3). It is not a tangent stiffness matrix, but it is a full stiffness matrix [6], [7], [9].

This paper is devoted to the technique of multithreaded parallelization of problems (1) and (3).

II. Multithreaded Parallelization

A. Internal force vector evaluation

First of all, we consider parallelization of the problem (3), the corresponding Algorithm 1 is presented below.

Algorithm 1 Internal force vector evaluation

1: Initialization.
   $\mathbf{rr}_{ip} \leftarrow 0$, $ip \in [0,\ np-1]$, $\mathbf{f}^{int} \leftarrow 0$

2: **for parallel** $e = 1$ **to** N_e **schedule(dynamic) do**
3: ip = $omp\_get\_thread\_num()$
4: *Compute a transformation matrix* $\mathbf{T}_e$
   $\mathbf{u}_e^{glob} \leftarrow \mathbf{u}$
   $\mathbf{u}_e^{loc} = \mathbf{T}_e \mathbf{u}_e^{glob}$
5: *Compute a nodal reaction vector* $\mathbf{r}_e^{loc}$, *using the constitutive relations.*
   $\mathbf{r}_e^{glob} = \mathbf{T}_e \mathbf{r}_e^{loc}$
   $\mathbf{rr}_{ip} \leftarrow + \mathbf{r}_e^{glob}$
6: **end for**

7: **for** $ip = 0$ **to** $np - 1$ **do**
8: **for parallel** $eqn = 1$ **to** Neq **schedule(dynamic, chunk) do**
9: $\mathbf{f}_{eqn}^{int} += \mathbf{rr}_{ip,eqn}$
10: **end for**
11: **end for**

At the initialization stage (point 1), we dynamically allocate memory for vectors $\mathbf{rr}_{ip}$, where ip is the thread number and np is the number of threads. After Algorithm 1 is finished, vector $\mathbf{f}^{int}$ will store internal forces. Vectors $\mathbf{rr}_{ip}, ip \in [0, np-1]$, and $\mathbf{f}^{int}$ have the dimension Neq - the number of equations in the finite element model. All these vectors are zeroed.

At the second stage (points 2 – 6) we run a parallel loop **for** over the number of finite elements N_e, where e is a number of the current finite element. We obtain the thread number ip (point 3) end evaluate the coordinate transformation matrix $\mathbf{T}_e$ (point 4). Then we put elements of the displacement vector $\mathbf{u}$, corresponding to the degrees of freedom of the finite element e, to vector $\mathbf{u}_e^{glob}$. Vector $\mathbf{u}$ of dimension Neq (number of equation) contains the nodal displacements and rotations in the global coordinate system (CS) for the entire finite element model. Vector $\mathbf{u}_e^{glob}$ of dimension $nstAct$ contains the nodal displacements and rotations of the finite element e. The displacements and rotations in the global CS are transformed into the displacements and rotations in the local CS of the e-th finite element with the help of the coordinate transformation matrix $\mathbf{T}_e$.

The evaluation of the nodal reaction vector $\mathbf{r}_e^{loc}$ in the local CS is performed according to (4) (point 5). The constitutive relations, reflecting the mechanical rules, are applied to evaluate the full stiffnes matrix $\mathbf{K}_e$. Then, the transformation of the reaction vector, given in the local CS, to the global CS is performed ($\mathbf{r}_e^{glob} = \mathbf{T}_e \mathbf{r}_e^{loc}$). After this, the elements of the reaction vector $\mathbf{r}_e^{glob}$ are added ($\leftarrow +$) to the corresponding elements of the vector $\mathbf{rr}_{ip}$.

At the third stage (points 7 – 11), partially prepared reaction vectors $\mathbf{rr}_{ip}, \quad ip \in [0, np-1]$ are combined in the vector $\mathbf{f}^{int}$ of internal forces. We parallelize an inner loop (points 8 – 9) covering the equations of the entire model. To avoid incoherence in the processor caches when writing data to vector $\mathbf{f}^{int}$ at multithreading, the chunk size **chunk** is assumed to be sixteen. In doing so, we assume that the size of the cache line is 64 or 128 bytes, that is, eight or sixteen double words respectively. It is important to do it to avoid a degradation of performance at multithreading on processors, which are not protected against such incoherence at the hardware level.

B. Stiffness matrix assembling

Unlike the internal forces evaluation algorithm discussed in the previous subsection, the tangent stiffness matrix assembling algorithm stores a large array to memory - a nonzero structure of sparse matrix that comprises only nonzero elements. The symmetric sparse matrix in compressed column format (CCS) is represented in the form of three arrays: $Space[pos], ind[pos]$ and $Pos[j]$. Array $Space[pos]$ comprises nonzero elements located column-by-column from the diagonal element to the last nonzero element of the column. Array $ind[pos]$ stores the i subscript for the corresponding element k_{ij} of matrix $\mathbf{K}_t$ located in $Space[pos]$. Array $Pos[j]$ points to the position pos of the diagonal element k_{jj} of the column j in the arrays $Space$, ind. Here $j \in [1,\ Neq], \quad i \geq j$. For instance, the CCS format for the matrix (5) is presented in (6).

$$\begin{pmatrix} k_{11} & & & & & \\ 0 & k_{22} & & & & \\ 0 & 0 & k_{33} & & & \\ k_{41} & 0 & 0 & k_{44} & & \\ 0 & k_{42} & 0 & 0 & k_{55} & \end{pmatrix} \tag{5}$$

pos	0	1	2	3	4	5	6
$Space$	k_{11}	k_{41}	k_{22}	k_{42}	k_{33}	k_{44}	k_{55}
ind	1	4	2	4	3	4	5
Pos	0	2	4	5	6	7	
j	1	2	3	4	5	–	

(6)

Subscript j is a column number. The last element in Pos is required to properly obtain the number of nonzero elements in the last column using the following expression $nonzero_j = Pos[j+1] - Pos[j]$, where $nonzero_j$ – the number of nonzero entries in the column j counting from the diagonal element to the lowest one. The nonzero structure of a symmetric sparse matrix is presented by the undirected adjacency graph, where vertexes correspond to columns of matrices or to diagonal elements and the edges – to nonzero off-diagonal elements [10].

The reordering of the adjacency graph reduces the number of fill-in in the factorized matrix. The symbolic factorization procedure [10] enables to create a nonzero structure of the factorized sparse matrix without numerical factorization. It is a very fast procedure, which operates only on the adjacency graph of the initial matrix $\mathbf{K}_t$ and obtains a factor-graph. As a result, the arrays Pos and ind are filled. Therefore, the assembling procedure fills the array $Space$. Algorithm 2 demonstrates a trivial solution of a such problem.

Algorithm 2 Assembling of the tangent stiffness matrix with using of critical sections (trivial solution)

```
1:  for parallel e = 1 to N_e schedule(dynamic) do
2:    ip = omp_get_thread_num()
3:    evaluate a finite element matrix K_{e,t} and the list of
      global equation numbers list_glob_eqns
4:    for jeqn = 1 to nstAct do
5:      jglobeqn = list_glob_eqns(jeqn)
6:      prepare the inverse data structure to avoid a search
        procedure
7:      for pos = Pos[jglobeqn] to Pos[jglobeqn+1] - 1
        do
8:        iglobeqn = ind[pos]
          Col[ip][iglobeqn] = pos
9:      end for
        fill Space:
10:     for ieqn = jeqn to nstAct do
11:       iglobeqn = list_glob_eqns(ieqn)
          pos = Col[ip][iglobeqn]
12:       begin_critical_section
13:         Space[pos] += K_{e,t}[ieqn, jeqn]
14:       end_critical_section
15:     end for
16:   end for
17: end for
```

The parallel loop **for** is performed over the number of finite elements (point 1). The thread number ip is defined (point 2) and the finite element tangent stiffness matrix $\mathbf{K}_{e,t}$ with the list of global equation numbers $list\_glob\_eqns$ are obtained (point 3).

The loop **for** is performed over the local equation number $jeqn$ (point 4 – 15). A column number $jglobeqn$ of the global tangent stiffness matrix $\mathbf{K}_t$ (point 5) is defined for each column number $jeqn$ of the matrix $\mathbf{K}_{e,t}$. Then, the inverse data structure $Col[ip][iglobeqn]$ is prepared to avoid a time-consuming search of the position pos in the array $Space$ (points 7 – 9).

Loop **for** (points 10 – 15) runs along the column $jeqn$ of the matrix $\mathbf{K}_{e,t}$. The position number pos is extracted from $Col[ip][iglobeqn]$, where $ieqn$ is a row number of the matrix $\mathbf{K}_{e,t}$ and $iglobeqn$ is a row number of the global tangent stiffness matrix $\mathbf{K}_t$ (point 11 and a line below). To avoid the situation, when several threads simultaneously modify the

same element of the array $Space[pos]$, a critical section is used (points 12 – 14).

Using a critical section in such a situation is not a good idea, because the speedup with the increasing number of threads begins to degrade very fast. The application of interlocked functions [11] instead of a critical section does not improve the speedup much.

Therefore, we do not use Algorithm 2. The following approach is proposed instead. We divide the finite elements into groups so that each group is simultaneously processed by different threads and writes only in its positions pos of the array $Space$. In other words, a situation, when different threads simultaneously write data to the same addresses of the array $Space$, should be eliminated.

To create such groups, we prepare an adjacency graph for the finite elements of the design model (Fig. 2). The vertices of the graph present the finite elements and the edges – the nodes in which the adjacent finite elements are coupled.

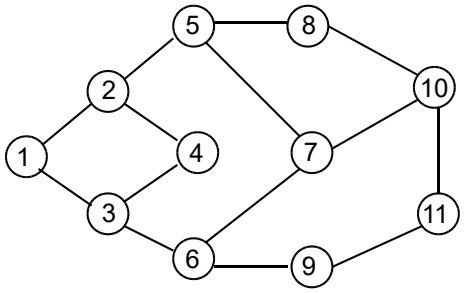

Fig. 2. The finite element adjacency graph. The vertices present the finite elements and edges – the nodes of the design model.

Then, we search for a pseudo peripherical vertex and create a structure of levels with a root in such a vertex [10] (Fig. 3).

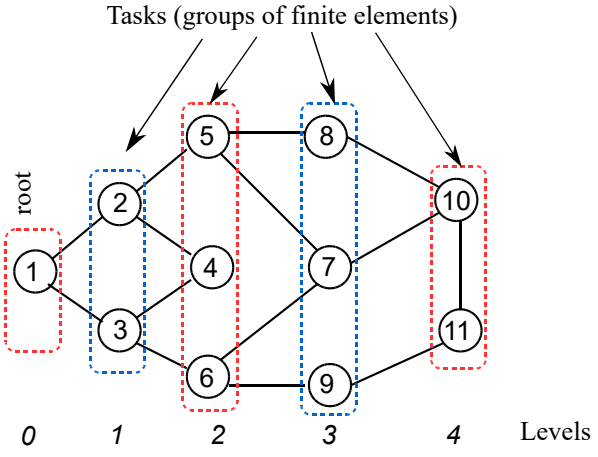

Fig. 3. The structure of levels for the finite element adjacency graph with a root in a (pseudo) peripherical vertex

The root of the level structure is placed in the pseudo peripheral vertex in order for the structure to be maximally elongated (containing as many levels as possible) and to minimize the number of vertices on each level. We call the vertices belonging to a given level a computational task or just a task. If we choose tasks of only even levels, then the vertices of each task are not connected in any way with the vertices of the remaining selected tasks. In other words, the finite elements belonging to each even-level task do not have any common nodes with finite elements belonging to tasks of other even levels. Thus, we guarantee that the finite elements belonging to different even levels make contributions to different degrees of freedom of the design model, that is, the modification of different elements of the $Space$ array is performed. Therefore, each even-level task can be performed simultaneously at multithreaded data processing. In the same way, as in the case of even levels, each task belonging to odd level can be solved independently from the other tasks of odd levels.

To achieve an acceptable load balance between threads, the Algorithm 3 is applied.

Algorithm 3 Mapping tasks onto threads

1: *Prepare finite element adjacency graph*
2: *Find a (pseudo) peripherical vertex*
3: *Create a structure of levels for the finite element adjacency graph with the root in a (pseudo) peripherical vertex*

Prepare the weights for even levels:
4: **for** *lev=0;* **to** *NoLevels-1, lev += 2* **do**
5: $levelWeight_{lev} = \sum_{e \in lev} nstAct_e^2$
6: **end for**

7: *Sort the even levels in descending order of their weights. Map tasks of even levels onto threads:*
$sumWeigh\,[ip] \leftarrow 0, \quad ip \in [0, np-1]$
8: **for** *lev=0;* **to** *NoLevels-1, lev += 2* **do**
9: *Find the thread number* min_ip *having a minimum sum of already mapped weights,* $min\_ip \in [0, np-1]$
10: $queue\,[min\_ip] \leftarrow \forall e \in lev$
$sumWeigh\,[min\_ip] += levelWeight_{lev}$
11: **end for**

Prepare the weights for odd levels:
12: **for** *lev=1;* **to** *NoLevels-1, lev += 2* **do**
13: $levelWeight_{lev} = \sum_{e \in lev} nstAct_e^2$
14: **end for**

15: *Sort the odd levels in descending order of their weights. Map tasks of odd levels onto threads:*
$sumWeigh\,[ip] \leftarrow 0, \quad ip \in [0, np-1]$
16: **for** *lev=1;* **to** *NoLevels-1, lev += 2* **do**
17: *Find the thread number* min_ip *having a minimum sum of already mapped weights,* $min\_ip \in [0, np-1]$
18: $queue1\,[min\_ip] \leftarrow \forall e \in lev$
$sumWeigh\,[min\_ip] += levelWeight_{lev}$
19: **end for**

The points 1 – 3 of the presented algorithm have been discussed above. After creating the level structure with a root

at a (pseudo) peripherical vertex, we obtain the weight of each even level (points 4 – 6). Here, the $NoLevels$ is the number of levels of a level structure and $levelWeight_{lev}$ is defined as a sum of weights for all vertices (finite elements) belonging to the level lev. The dimension of the finite element tangent stiffness matrix $\mathbf{K}_{e,t}$ is denoted as $nstAct_e$ and the weight of the vertex is accepted as a number of elements $nstAct_e^2$ in $\mathbf{K}_{e,t}$.

Then, we sort all vertices belonging to the even levels, in descending order of their weights, and zero the sum of weights $sumWeight\,[ip]$ mapped onto thread ip, $\quad ip \in [0, np-1]$ (point 7). For each even level we find a thread min_ip which has a minimum sum of already mapped weights (point 9), put all vertices of the given level to the $queue\,[min\_ip]$ and correct $sumWeight\,[min\_ip]$ (point 10).

Finally, we apply the same approach to all the odd levels and obtain $queue1\,[ip]\,, ip \in [0, np-1]$ (points 12 – 19).

Therefore, all even levels are mapped onto np threads and are presented by $queue\,[ip]\,, \quad ip \in [0, np-1]$ queues. Similarly, all odd levels are presented by $queue1\,[ip]\,, \quad ip \in [0, np-1]$ queues.

Sorting in descending order improves a load balance between threads. A similar approach has been used in [3], [4] to achieve a load balance between threads in solver PARFES and in block incomplete Cholesky factorization solver [8].

The Algorithm 4 demonstrates a tangent stiffness matrix assembling using multithreading without any synchronization allowing a high speedup with the increasing thread number.

In the first parallel region (points 1 – 18) the **while** loop runs for each thread ip until the $queue\,[ip]\,, \quad ip \in [0, np-1]$ is empty. These queues contain parallel tasks for even levels of level structure. In each **while** loop, the nearest finite element number e is retrieved (point 4) and the tangent stiffness matrix $\mathbf{K}_{e,t}$ with the list of global equation numbers are evaluated (point 5). The loop **for** is executed over the columns of the matrix $\mathbf{K}_{e,t}$ (points 6 – 16), where *nstAct* is a dimension of $\mathbf{K}_{e,t}$. The $Col\,[ip]$ array stores a position number *pos* of the nonzero entry $Space\,[pos]$ with the global equation number *iglobeqn* (points 9 – 11). It allows us to avoid a time-consuming search of pos, corresponding to the global equation number $iglobeqn$. The loop **for** (points 12 – 15) fills the $Space\,[pos]$ with elements of the matrix $\mathbf{K}_{e,t}$.

The second parallel region (points 19 – 35) does the same as the previous parallel region, but operates with queues $queue1\,[ip]$, containing parallel tasks for odd levels of a level structure. In contrast to Algorithm 2, Algorithm 4 does not contain any synchronization objects due to the approach presented above. This allows us to hope a high speedup with the increasing thread number, even when the number of threads is large. This approach formed the basis for the master's degree thesis in computer science [12], in which one of the authors, S. Fialko, was a scientific leader.

Algorithm 4 Assembling of the tangent stiffness matrix using the proposed approach

1: **parallel region**
2: $ip = omp\_get\_thread\_num()$
3: **while** $queue\,[ip]\,is \quad not \quad empty$ **do**
4: $e \leftarrow queue\,[ip]$ *retrieve element number* e
5: *evaluate a finite element matrix* $\mathbf{K}_{e,t}$ *and the list of global equation numbers* $list\_glob\_eqns$
6: **for** $jeqn = 1$ **to** $nstAct$ **do**
7: $jglobeqn = list\_glob\_eqns\,[jeqn]$
8: *prepare the inverse data structure to avoid a search procedure*
9: **for** $pos = Pos[jglobeqn]$ **to** $Pos[jglobeqn+1]-1$ **do**
10: $iglobeqn = ind[ieqn]$
$Col\,[ip]\,[iglobeqn] = pos$
11: **end for**
fill Space:
12: **for** $ieqn = jeqn$ **to** $nstAct$ **do**
13: $iglobeqn = list\_glob\_eqns\,[ieqn]$
$pos = Col\,[ip]\,[iglobeqn]$
14: $Space\,[pos] += \mathbf{K}_{e,t}\,[ieqn, jeqn]$
15: **end for**
16: **end for**
17: **end while**
18: **end of a parallel region**

19: **parallel region**
20: $ip = omp\_get\_thread\_num()$
21: **while** $queue1\,[ip]\,is \quad not \quad empty$ **do**
22: $e \leftarrow queue1\,[ip]$ *retrieve element number* e
23: *evaluate a finite element matrix* $\mathbf{K}_{e,t}$ *and the list of global equation numbers* $list\_glob\_eqns$
24: **for** $jeqn = 1$ **to** $nstAct$ **do**
25: $jglobeqn = list\_glob\_eqns\,[jeqn]$
prepare the inverse data structure to avoid a search procedure
26: **for** $pos = Pos[jglobeqn]$ **to** $Pos[jglobeqn+1]-1$ **do**
27: $iglobeqn = ind\,[ieqn]$
$Col\,[ip]\,[iglobeqn] = pos$
28: **end for**
fill Space:
29: **for** $ieqn = jeqn$ **to** $nstAct$ **do**
30: $iglobeqn = list\_glob\_eqns\,[ieqn]$
$pos = Col\,[ip]\,[iglobeqn]$
31: $Space\,[pos] += \mathbf{K}_{e,t}\,[ieqn, jeqn]$
32: **end for**
33: **end for**
34: **end while**
35: **end of a parallel region**

III. Numerical Results

We consider a design model of a reinforced concrete floor slab, comprising 65 117 equations (Fig. 4). The triangular and

quadrilateral finite elements, taking into account the physical nonlinearity, are used. The supports, modeling the walls, are shown in blue color. The uniform normal pressure simulates the dead and operational loads.

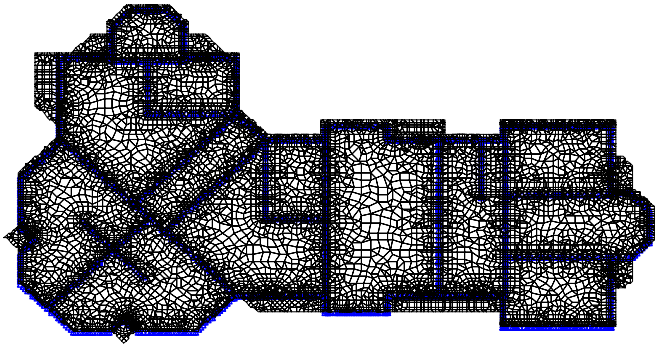

Fig. 4. The design model of a reinforced concrete floor slab

The computer with sixteen-core AMD Opteron 6276 processor, 2.3/3.2 GHz, 64 GB DDR3 RAM, OS Windows Server 2008 R2 Enterprise SP1, 64-bit, is used. Table I depicts a tangent stiffness matrix assembling time (s) during the solution of the entire nonlinear problem for the different number of threads np in the case with no optimization, optimization using a (pseudo) peripherical vertex and optimization using a (pseudo) peripherical vertex and weights of levels.

The "no optimization" case means that a root for the level structure is taken in the vertex 1. The pseudo peripherical vertex is not found. The weights of levels are not used. The queues $queue\,[ip]$ and $queue1\,[ip]$, $ip \in [0, np-1]$ are prepared using a cyclic mapping of levels onto threads.

In the "use a peripherical vertex" case the pseudo peripherical vertex is found, but the weights of levels are not used. The cyclic mapping of levels onto threads is applied. Taking the pseudo peripherical vertex as a root of the level structure results in an increase of the levels from 110 to 114 for the given problem. It should be pointed out that for other problems the choice of a root in a pseudo peripherical vertex has a considerably larger impact on the increase of the level number. Therefore, for the considered problem of the "use a peripherical vertex" option on the reduction of the computing time is not observed.

Algorithm 3 is applied in the "plus the use of the weights of levels" case. The red color means that the load imbalance between threads exceeds 15%. The shortest computing time on a large number of threads is achieved when all optimizations are applied.

Table II shows the distribution of sum of weights among threads for the "no optimization" and "use peripherical vertexes and weights of levels" cases. Here, ip is a thread number, a red color indicates a thread with a maximum computational effort and a green color indicates a thread with a minimum computational effort. The difference between the maximum sum of weights and the minimum one is a measure of imbalance between threads. These results demonstrate that at a maximum number of threads the proposed approach, corresponding to Algorithm 3 and Algorithm 4, has a imbalancee between threads of about 11%. The "no optimization" approach has a imbalance of about 36%. We estimate a imbalance as (maximum sum of weights – minimum sum of weights)/maximum sum of weights in percent. Therefore, the above optimizations play an important role in improving a load balance between threads.

The Fig. 5 demonstrates a speedup with the increasing thread number in the range of the physical core number for the given processor: $S_{np} = T_1/T_{np}$, where T_1 is a time when using one thread and T_{np} is a time on np threads. The "ideal" curve corresponds to an ideal speedup, passing through the points (0, 0), (1, 1), (2, 2), ... However the given processor has a turbo core mode. When a small number of cores are loaded, the clock frequency is 3.2 GHz. When the number of loaded cores is a maximum, the clock frequency reduces to 2.3 GHz. Therefore, an ideal speedup is unreachable.

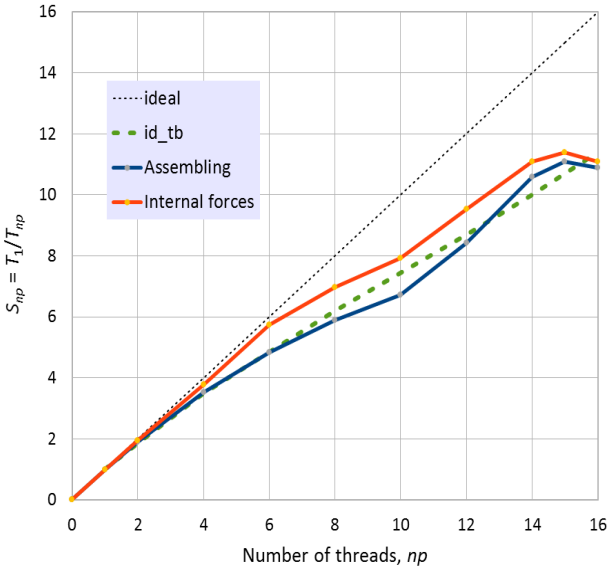

Fig. 5. The speedup with the increasing thread number for the evaluation of the internal forces and the tangent stiffness matrix assembling

The curve "id_tb" approximates an ideal speedup caused by the turbo core mode using a square parabola. The points (0, 0), (1, 1) and (16, x) are used to define such a parabola. The ordinate x is obtained as follows. A processor with sixteen cores without a turbo core mode should work with a clock frequency 3.2 GHz and should achieve a speedup of 16 times. A processor in the turbo core mode works with a clock frequency 2.3 GHz and has a speedup of x times. So, from the proportion, we obtain x = 2.3/3.2*16 = 11.5 times.

The "Internal forces" curve demonstrates a speedup of the internal force vector evaluation with the increasing thread number (Algorithm 1). The "Assembling" curve depicts a speedup of the tangent stiffness matrix assembling procedure with the above optimizations, presented by Algorithms 3

TABLE I
THE TANGENT STIFFNESS MATRIX ASSEMBLING TIME DEPENDING ON THE NUMBER OF THREADS

nos of threads	no optimization	use a peripherical vertex	plus the use of the weights of levels
1	84.7	85.9	86.7
2	46.6	46.2	45.6
4	26.3	26.8	27.6
8	17.7	18.2	16.6
12	13.6	14.4	12.8
16	11.3	11.6	10.4

TABLE II
SUM OF WEIGHTS PER EACH THREAD

thread number ip	no optimization		use a peripherical vertex and weights of levels	
	even levels	odd levels	even levels	odd levels
0	177264	183564	194760	196524
1	184644	187020	195012	197280
2	189108	208404	197820	198252
3	225216	242856	196488	195084
4	254772	248400	197532	206640
5	245304	228528	195840	194076
6	223272	218484	193392	194148
7	193572	174780	208512	210492
8	175464	169524	204840	194652
9	162432	163980	195048	211284
10	164268	168444	216396	205812
11	181224	185976	202140	202428
12	195804	202824	197460	194112
13	219816	229248	201348	193788
14	218160	190908	206784	196596
15	186840	187344	193788	199116

and 4. We obtain an acceptable correlation between "Internal forces" and "Assembling" curves with the "id_tb" curve. A steady increase in the speedup, with the exception of the last point (np = 16), confirms the effectiveness of the proposed approaches.

It should be noted that if we define speedup as $\frac{T_1 \cdot \Delta t_{np}}{T_{np} \cdot \Delta t_1}$, where Δt_1 and Δt_{np} are the time of a single processor's tick when using a single thread and when using the np threads correspondingly, then this ratio is not dependent on the processor clock speed, and the curve describing an algorithm's acceleration can be compared with the ideal speedup (curve "ideal"). However, for users, the definition of speedup as $S_{np} = T_1/T_{np}$, based on real-time execution of the tasks, is more understandable, therefore we use such a definition and above approach.

IV. CONCLUSION

Two different approaches for multithreaded parallelization of similar procedures – internal force vector evaluation and tangent stiffness matrix assembling have been considered.

The first approach requires the allocation of an additional vector of dimension Neq (number of equations) for each thread. Therefore, the amount of additional core memory is $Neq \times np$ words of double. On the other hand, such an approach is relatively simple and fully eliminates incoherences in caches of different processor cores.

The second approach, based on creating a finite element adjacency graph and preparing a level structure, ensures the independence of computational tasks, belonging to only even levels or only odd levels of a level structure, allows us to reject any type of synchronization and obtain a stable speedup with an acceptable correlation in comparison with an ideal speedup, taking into account the turbo core mode. Taking a pseudo peripherical vertex of the adjacency graph as a root of the level structure results in an increase of the levels number, so, the number of computational tasks increases too and each task becomes shorter. Together with a specific mapping-tasks-onto-threads algorithm, using the weights of computational tasks, this approach significantly improves the load balance between threads (Tables I, II) and helps to achieve a stable speedup.

On the other hand, the second approach does not guarantee the absence of incoherence in the processor caches. Numerous tests, performed on different computers, demonstrate the reliability of this approach. Moreover, the above example as well as other tests, performed on the AMD Opteron processor, which does not have hardware protection against performance degra-

dation due to incoherence in the processor caches, demonstrate stable speedup with the increasing thread number. This approach could be applied to the multithreaded parallelization of the internal force vector evaluation procedure, but we wanted to compare the efficiency of two different approaches to justify the reliability of the more complicated second method.

ACKNOWLEDGMENT

The authors are deeply grateful to IT company SCAD Soft for the financial support of this research and for providing a collection of problems.

REFERENCES

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

[2] Intel Math Kernel Library Reference Manual. URL: https://software.intel.com/en-us/mkl-developer-reference-c-intel-mkl-pardiso-parallel-direct-sparse-solver-interface (Last access: 17.04.2018).

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

[4] S. Fialko, "PARFES: A method for solving finite element linear equations on multi-core computers", *Advances in Engineering Software,* 40, (12), 2010, pp. 1256–1265. https://doi.org/10.1016/j.advengsoft.2010.09.002

[5] K. J. Bathe, *Finite Element Procedures,* New Jersey: Prentice Hall; 1996.

[6] S. Yu. Fialko, "Quadrilateral finite element for analysis of reinforced concrete floor slabs and foundation plates", *Applied Mechanics and Materials,* 725–726, 2015, pp. 820 – 835, doi: 10.4028/www.scientific.net/AMM.725-726.

[7] S. Yu. Fialko, V. S. Karpilowskyi, "Triangular and quadrilateral fat shell fnite elements for nonlinear analysis of thin-walled reinforced concrete structures in SCAD software," *In: Petraszkiewicz and Witkowski (eds). Shell Structures: Theory and Applications,* V. 4., Taylor and Francis Group, London, 2018, pp. 367–370.

[8] S. Yu. Fialko, V. S. Karpilowskyi, "Block subspace projection preconditioned conjugate gradient method for structural modal analysis", *in Proceedings of the Federated Conference on Computer Science and Information Systems,* ISSN 2300-5963 ACSIS, Vol. 11, pp. 497–506. DOI: 10.15439/2017F64 .

[9] S. Yu. Fialko, *Application of finite element method to analysis of strength and bearing capacity of thin-walled concrete structures, taking into account the physical nonlinearity,* Moscow: Publishing House SCAD SOFT, Publishing House ASV; 2018 (Russian).

[10] A. George, J. Liu, E. Ng, Computer Solution of Sparse Linear Systems, 1994. URL: http://web.engr.illinois.edu/~heath/courses/cs598mh/george_liu.pdf

[11] Interlocked variable access. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684122(v=vs.85).aspx

[12] M. Olczyk, "The procedure of parallel assembling of stiffness matrix in FE analysis for applying to the solution of nonlinear algebraic equation systems", *Master's degree work, Cracow University of Technology,* Cracow, Polish, 2017.

[13] D. Th. Nguyen, *Parallel-Vector Equation Solvers for Finite Element Engineering Applications,* Springer Science+Business Media, LLC: New Yourk; 2002. DOI 10.1007/978-1-4615-1337-7.

[14] Yu.V. Khalevitsky, N.V. Burmasheva, A.V. Konovalov, "An approach to the parallel assembly of the stiffness matrix in elastoplastic Problems", *Mechanics, Resource and Diagnostics of Materials and Structures (MRDMS-2016), AIP Conf. Proc. 1785, pp. 040023-1–040023-4; Published by AIP Publishing. 978-0-7354-1447-1/$30.00* doi: 10.1063/1.4967080.

[15] M. N. De Rezendea, J. B. de Paiva, "A parallel algorithm for stiffness matrix assembling in a shared memory environment", *Computers & Structures,* 76, (5, 15), 2000, pp. 593–602. https://doi.org/10.1016/S0045-7949(99)00181-9.

[16] D. Goudin, J. Roman, *A scalable parallel assembly of irregular meshes based on a block distribution for a parallel direct solver*, In: Applied Parallel Computing, New paradigms for HPC in industry and academia, *5th International Workshop, PARA 2000, Bergen, Norway, June 2000, Proceedings*, Springer, Lecture Notes in Computer Science, V. 1947, pp. 113 – 116. URL: https://link.springer.com/chapter/10.1007/3-540-70734-4_15 (Last access: 13.07.2018)

[17] C. Cecka, A. Lew, E. Darve, "Introduction to Assembly of Finite Element Methods on Graphics Processors", *IOP Conf. Series: Materials Science and Engineering*, 10, 012009, 2010, pp. 1 – 10. doi:10.1088/1757-899X/10/1/012009.