IMPROVED AND VECTORISED MATLAB-BASED ALGORITHMS FOR SERIAL AND PARALLEL IMPLEMENTATION OF FINITE ELEMENT METHOD IN LINEAR ELASTICITY

Baurice Sylvain Sadjiep Tchuigwa[1], Jan Krmela[1, 2], Jan Pokorny[1], Vladimira Krmelova[2]
[1]University of Pardubice, Czech Republic; [2]Alexander Dubcek University of Trencin, Slovakia
bauricesylvain.sadjieptchuigwa@student.upce.cz, jan.krmela@tnuni.sk, jan.krmela@upce.cz

Abstract. This paper presents two improved algorithms for efficient sequential and parallel implementation of the Finite element method (FEM) for both linear and nonlinear boundary value problems. The proposed algorithms address some weak points, such as the overuse of for-loops and serial computing caused by dependencies in constructing fundamental expressions (global stiffness matrix, mass matrix, global force vector, etc.) resulting from the finite element method. By taking advantage of the concepts of sparse matrix representation, vectorization, and the physical architecture of modern computing resources, the proposed methods are free from mesh partitioning techniques or similar approaches and enable the use of all available CPU cores/threads without synchronization. Moreover, these algorithms are also adapted to deal with meshes involving elements of any order in both 2D and 3D. Two tests from NAFEMS benchmarks are implemented in MATLAB to verify the accuracy and stability of the proposed algorithms in both serial and parallel processing. According to serial and parallel computing results, the proposed algorithms perform better than the standard sparse assembly strategy and behave linearly with the mesh size but at a smaller rate than the latter. In parallel processing, the algorithms are also demonstrated to be accurate and achieve an efficiency of at least 60% in 2D and 70% with two cores/threads when the mesh size is greater than 10,000. Moreover, the simulations revealed that the performance gap between the proposed algorithm and the classical sparse algorithm is more pronounced in 2D than in 3D due to the increase in degrees of freedom.

Keywords: vectorization, sparse algorithm, finite element, parallel, MATLAB.

Introduction

The Finite Element Method (FEM) has come a long way since its introduction in early [1]. Since its inception, the application of advanced optimization techniques has become increasingly widespread in a variety of engineering disciplines, including aerospace, mechanical, and civil engineering. Its efficacy in designing and refining lightweight, durable structures has contributed significantly to the advancement in these fields.

Several algorithms have been suggested for implementing FEM in MATLAB[2] in the literature. Among these, the most frequently used algorithm is the standard one, which involves sparse assembly of a global matrix/force after constructing the triplet (*IndexI*, *IndexJ*, *Kvalues*). This algorithm employs a series of for-loops, which are relatively slow in MATLAB. For tackling this slowness, vectorization is one solution alongside parallelization as investigated in [3; 4]. Although message-passing is a sophisticated technique for communication between cores on distributed architectures. Its efficiency depends on minimizing communication time, which can be difficult. Furthermore, it is subsequent to data pre-processing (domain decomposition into subdomains, renumbering, etc.) [5].

In this paper, we introduce two enhanced algorithms that work for topologies of any order and make use of all CPU cores/threads for serial and parallel computation of finite element global matrices/vectors without domain partitioning. To do so, we first briefly introduce a classical and discretized variational formulation for boundary value problems in elasto-dynamics, examine the shortcomings of the standard method, and subsequently outline the proposed algorithm. Finally, two benchmark tests (FV32 and FV52) by NAFEMS [6] are considered to evaluate the accuracy and efficiency of the proposed methods in serial and parallel execution.

Materials and methods

Let us consider the boundary value problem defined as follows:

$$\begin{cases} div(\bar{\bar{\sigma}}) + \overrightarrow{f_v} = \rho\ddot{\vec{u}} & in\ \Omega_t \\ \bar{\bar{\sigma}}\vec{n} = \vec{t} & on\ \partial\Omega_1 \\ \vec{u} = \vec{u}_p & on\ \Gamma \end{cases} \quad (1)$$

where $\partial\Omega = \partial\Omega_1 \cup \partial\Omega_2 \cup \Gamma$, with Ω a continuum body that initially (at $t = 0$) occupies the domain $\Omega_0 \subset \mathbb{R}^d{}_{|d\in\langle 1,2 \text{ or } 3\rangle}$ in the reference configuration and $\Omega_t \subset \mathbb{R}^d{}_{|d\in\langle 1,2 \text{ or } 3\rangle}$ in the current configuration and is subjected to body forces $\vec{f_v}$ per unit mass and a uniform load $\vec{t}$ per unit surface.

At any time in the cartesian coordinates, any point M of the body is described according to the repeated indices rule by $u = u_i E_i$ where $i \in \{1,2,3\}$ and E_i are unit orthogonal vectors of the basis. We consider a linear elasticity constitutive law given by $\sigma = \mathcal{C}:\varepsilon$. Where $\mathcal{C}$ and ε are the fourth-order tensor elasticity tensor and small strain tensor, respectively. Let V be the space of kinematically admissible displacement and defined as

$$V = \{u \in H^1(\Omega) \mid u = u_p \text{ on } \Gamma\} \tag{2}$$

Using a virtual displacement $\delta u \in V$, the variational formulation of the boundary value problem is given by

$$\left\{ find\ u_h \in V_h\ such\ that \int_{\Omega_h} \rho \ddot{u}_h \cdot \delta u_h\, dV + \int_{\Omega_h} \sigma_h : \delta\varepsilon_h\, dV = \int_{\Omega_h} f \cdot \delta u_h dV + \int_{\partial\Omega_1} t \cdot \delta u_h\, dS \right. \tag{3}$$

Now considering the discretization of the domain into finite elements, the approximate global displacement can be written in terms of the nodal displacements $\bar{u}_{i\in<1,..,n>}$ and shape functions $N_{i\in<1,..,n>}$ as $u_h = \sum_1^n N_i \bar{u}_i = N\bar{u}_e$. By replacing this expression in Eq. (3), we end up with

$$\begin{aligned} \bigcup_{e=1}^{n} \left\{ \int_{\Omega_e} \rho {N_e}^T N_e\, dV \right\} \ddot{u}_e + \bigcup_{e=1}^{n} \left\{ \int_{\Omega_e} {B_e}^T \mathcal{C} B_e\, dV \right\} \bar{u}_e = \\ = \bigcup_{e=1}^{n} \left\{ \int_{\Omega_h} {N_e}^T f\, dV \right\} + \bigcup_{e=1}^{n} \left\{ \int_{\partial\Omega_1} {N_e}^T t\, dS \right\} \end{aligned} \tag{4}$$

Or simply in the global form

$$M_g \ddot{U} + K_g U = F_{vol} + F_{\partial\Omega_1} \tag{5}$$

where $M_g, K_g, F_{vol}, F_{\partial\Omega_1}$ and U – respectively global mass matrix, stiffness matrix, volume force vector, applied force vector and displacement vector.

For an extensive detail about this type of variational formulation, the reader is referred to research in [9; 10]. In the common computer-based calculation [8], of K and M, the so-called sparse assembly is employed and consists in, first of all, calculating the triplet (*IndexI*: row indices, *IndexJ*: column indices, *K_values*: element values) of all the elements and then call the function sparse to assemble the stiffness/mass matrix as illustrated in Algorithm 1 (see Fig. 1 and 2).

Algorithm 1 (base) standard algorithm for computing Kg and Mg

```
Require: FE_model                                   ▷ FE_model is a structure array
function [K_g, M_g] = STANDARDG LOBALM ATRICES(FE_model, ...)
    [Gq, weight, q] ← GaussQuadrature(FE_model)
    [NI, dN_dθ] ← ShapeFunction(FE_model)
    ce ← (ndof * nn)^2 ▷ ndof: number of degrees of freedom per node
    IndexI ← IndexJ ← int32(zeros(ce * Nel, 1))
    k_values ← M_values ← zeros(ce * Nel, 1); p ← 1
    for e ← 1 to FE_model.Nel do                    ▷ Nel: number of elements
        [ke, Me] ← LocalMatrices(FE_model, ...)
        me ← FE_model.connect(e, :)
        for j ← 1 to nn do                          ▷ nn: number of nodes per element
            for α ← 1 to ndof do
                for i ← 1 to nn do
                    for β ← 1 to ndof do
                        IndexI(p) ← ndof * (me(i) − 1) + α
                        IndexJ(p) ← ndof * (me(j) − 1) + β
                        s ← ndof * (i − 1) + α
                        t ← ndof * (j − 1) + β
                        k_values(p) ← ke(s, t)
```

Fig.1. **Standard algorithm (part one)**

```
19:                     M_values(p) ← Me(s,t)
20:                     p ← p + 1
21:                 end for
22:             end for
23:         end for
24:     end for
25: end for
26: K_g ← sparse(IndexI, IndexJ, k_values)
27: M_g ← sparse(IndexI, IndexJ, M_values)
28: end function
```

Fig. 2. **Standard algorithm (part two)**

This procedure is barely the same for the force vector, with the only difference that the column indices is a vector with 1 as components. Though, this algorithm is relatively efficient for implementation on one core (serial), it is actually not possible to use it in a parallel pool without partitioning the discretized domain, since there is the dependency in line 19 of Algorithm 1, which prevents this loop from being parallelizable.

Proposed algorithms for serial and parallel computing

In this section, we introduce two improved algorithms (alg.Opt in Fig. 3 and alg.Vect in Fig. 4 and 5) that perform well in series and can be parallelized as well without mesh partitioning.

Algorithm 2 (alg.Opt) proposed algorithm 1 for computing $\mathcal{K}_g$ and $\mathcal{M}_g$

```
Require: FE_model                                   ▷ FE_model is a structure array
1: function [K_g, M_g] = OPTGLOBALMATRICES(FE_model, ...)
2:     [Gq, weight, q] ← GaussQuadrature(FE_model)          ▷ Get Gauss quadrature data
3:     [NI, dN_dθ] ← ShapeFunction(FE_model)    ▷ Get shape functions and their derivatives
4:     IndexNode ← SparseCount(FE_model)        ▷ Get vector of nodal degrees of freedom
5:     ce ← (ndof * nn)^2 ▷ ndof: number of degrees of freedom per node
6:     IndexI ← IndexJ ← int32(zeros(ce, Nel))
7:     k_values ← M_values ← zeros(ce, Nel)
8:     for e ← 1 to FE_model.Nel do                 ▷ Nel: number of elements
9:         [ke, Me] ← LocalMatrices(FE_model, ...)
10:        k_values(:, e) ← ke(:)
11:        M_values(:, e) ← Me(:)
12:        eNodes ← reshape(IndexNode(:, connect(e, :)), [ ], 1)
13:        IndexI(:, e) ← reshape(repmat(eNodes, 1, ndof * nn), [ ], 1)
14:        IndexJ(:, e) ← reshape(repmat(eNodes', ndof * nn, 1), [ ], 1)
15:    end for
16:    K_g ← sparse(IndexI(:), IndexJ(:), k_values(:));
17:    M_g ← sparse(IndexI(:), IndexJ(:), M_values(:));
18: end function
```

Fig. 3. **Proposed optimized algorithm**

Algorithm 3 (alg.Vect) proposed algorithm 2 for computing $\mathcal{K}_g$ and $\mathcal{M}_g$

```
Require: FE_model                                   ▷ FE_model is a structure array
1: function [K_g, M_g] = VECTGLOBALMATRICES(FE_model, ...)
2:     [Gq, weight, q] ← GaussQuadrature(FE_model)          ▷ Get Gauss quadrature data
3:     [NI, dN_dθ] ← ShapeFunction(FE_model)    ▷ Get shape functions and their derivatives
4:     IndexNode ← SparseCount(FE_model)        ▷ Get vector of nodal degrees of freedom
5:     ce ← (ndof * nn)^2 ▷ ndof: number of degrees of freedom per node
6:     k_values ← M_values ← zeros(ce, Nel)
7:     Matrix_dof ← int32(zeros(Nel, nn * ndof))
```

Fig. 4. **Proposed optimized algorithm (part one)**

```
    for e ← 1 to FE_model.Nel do                ▷ Nel: number of elements
        [ke, Me] ← LocalMatrices(FE_model, ...)
        k_values(:, e) ← ke(:)
        M_values(:, e) ← Me(:)
    end for
    for k ← 1 to nn do                          ▷ nn: number of nodes per element
        for β ← 1 to ndof do
            Matrix_dof(ndof * (k − 1) + β, :) ← IndexNode(β, connect(:, k))
        end for
    end for
    IndexI ← repmat(Matrix_dof, ndof * nn, 1)
    IndexJ ← repmat(Matrix_dof(:)', ndof * nn, 1)
    K_g ← sparse(IndexI(:), IndexJ(:), k_values(:));
    M_g ← sparse(IndexI(:), IndexJ(:), M_values(:));
end function
```

Fig. 5. **Proposed optimized algorithm (part two)**

Results and Discussion

In this section, we assess the accuracy and performance of the proposed schemes in both serial and parallel processing.

Example 1: NAFEMS Test FV32.

In order to evaluate the performance of our algorithms in serial computing, we consider the well-known benchmark Test FV32 by NAFEMS [6]. Basically, this plane stress test investigates the natural vibration of a cantilever tapered membrane (see Fig. 6) with Young's modulus E = 200 GPa, Poisson's ratio = 0.3, density = 8000 kg·m^{-3} and thickness = 0.05 m. A set of QUAD4 (4-node quadrilateral) elements ranging from coarse to fine, were created in GMSH software [8] and imported into MATLAB as topology inputs for the simulation. It should be noted that the hardware resource used to perform sequential and parallel computing of K and M for this test is an Intel® I7 HP1640 laptop (6 physical cores) with 15.9GB RAM and a base clock speed of 2.70GHz.

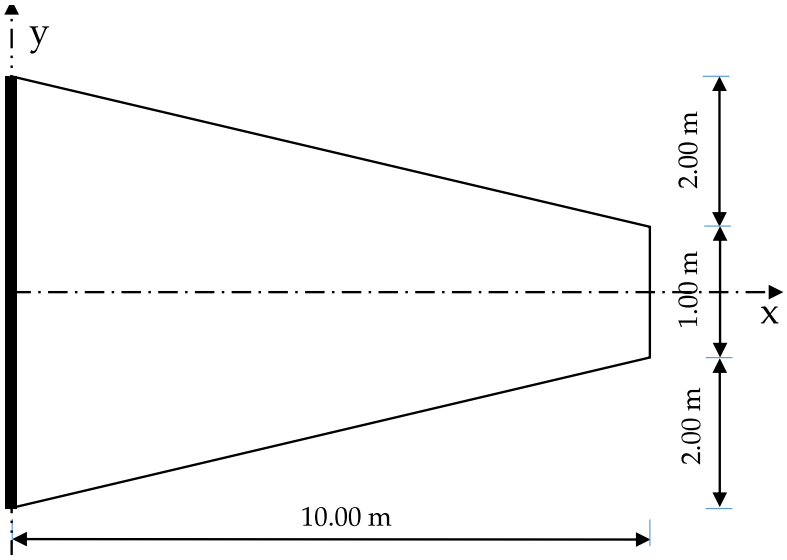

Fig. 6. **Schematic drawing of the membrane**

In serial execution, Fig. 7 shows that both proposed algorithms are slightly faster than the standard algorithm. Also, the alg.Vect outperforms alg.Opt. by a small margin.

4.1.1. Verification of accuracy: The simulations were run on a single core using the present methods on the aforementioned meshes, and the recorded results were identical for both kinds of algorithms. Table 1 reports the six smallest modal frequencies obtained with the present algorithms and compares them to those of NAFEMS test [6] for the set of 4 meshes. We precise that *ndof = 2*.

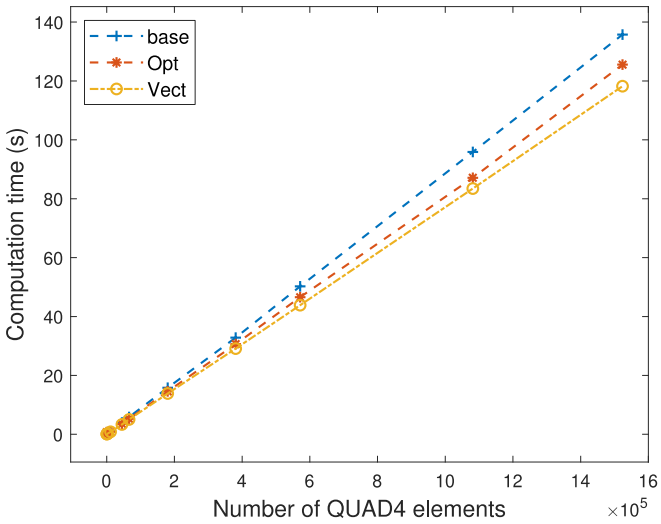

Fig. 7. **CPU time vs number of QUAD4 on 1 core**

Table 1

Comparison of obtained frequencies with reference values

Mode	*Nel*	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6
NAFEMS f_{ref}	-	***44.623***	***130.03***	***162.70***	***246.05***	***379.90***	***391.44***
Proposed algorithms f_{cal}	176	44.83	131.49	162.80	250.45	390.80	392.46
	704	44.67	130.39	162.72	247.14	382.53	391.68
	2816	44.63	130.11	162.70	246.28	380.43	391.48
	1264	44.62	130.03	162.69	246.06	379.89	391.42

Performance analysis: In this part, we investigate the scalability of the proposed algorithms for carrying out CPU-based parallel computation of the stiffness matrix and the mass matrix for 11 QUAD4 meshes. As it can be observed in Fig. 8, both algorithms require almost the same amount of memory as the standard algorithm. Talking about performance (computational cost, speedup and efficiency), Fig. 9, Fig. 10 and Fig. 11 report the behaviour of the scheme with respect to the mesh size number of cores. The workload balancing is depicted in Fig. 12. Meanwhile, we recall that

$$\text{speedup} = \frac{t_1}{t_n} \; and \quad \text{and} \quad \text{efficiency} = \frac{\text{speedup}}{\text{n}_{\text{core}}} * 100 \tag{6}$$

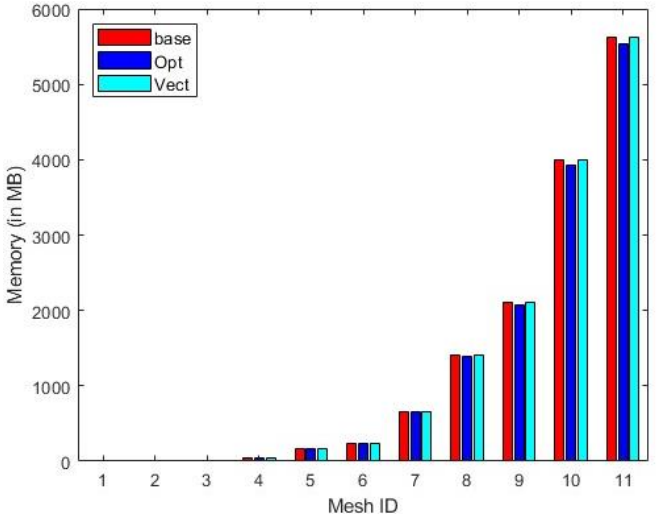

Fig. 8. **Memory (RAM) usage for computing Kg and Mg**

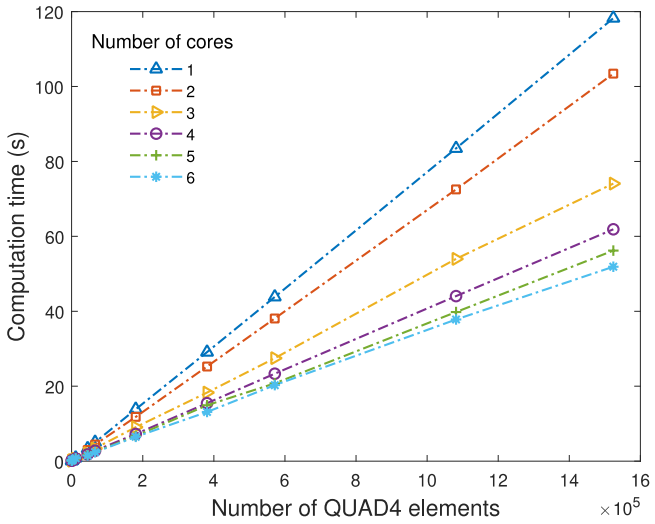

Fig. 9. **CPU time vs number of QUAD4**

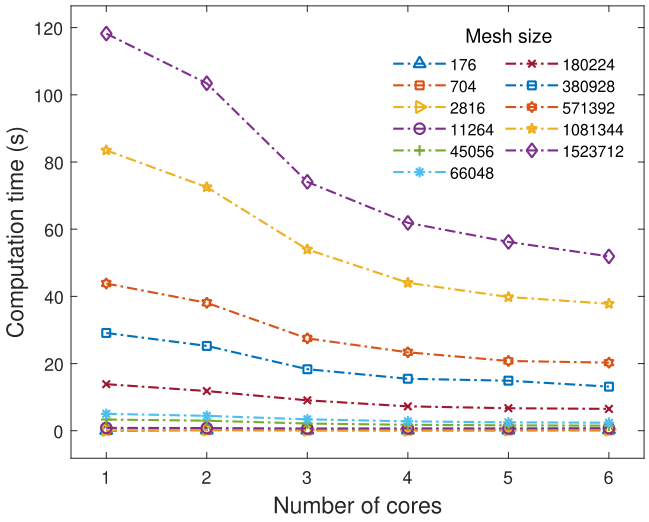

Fig. 10. **CPU time vs number of cores**

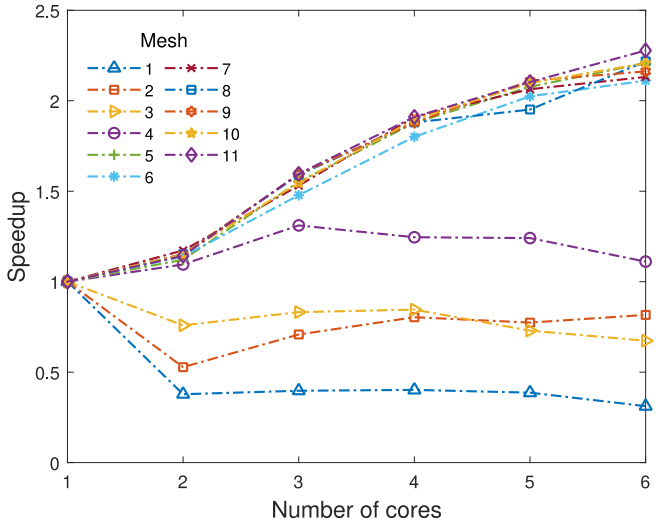

Fig. 11. **Speedup vs number of cores**

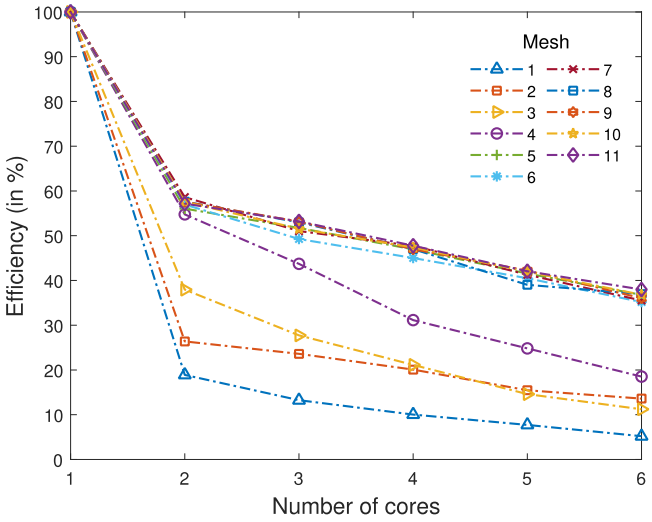

Fig. 12. **Efficiency vs number of cores**

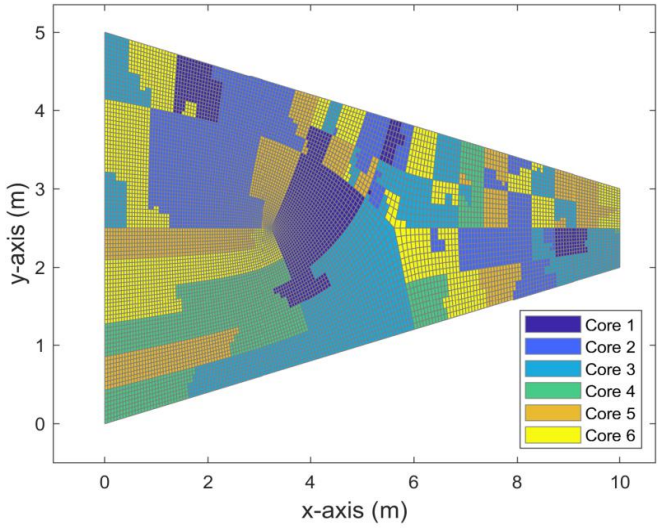

Fig. 13. **Automatic workload balancing for a parallel pool with 6 cores for the mesh with 11264 QUAD4**

Besides the efficiency reported above, Fig. 13 shows the automatic distribution of mesh elements hosted by the six cores of the parallel pool. This mapping helps identify elements computed by each core.

Example 2: NAFEMS Test FV52 (free vibration of a simply supported "solid" square plate)

The geometry in this test is the four-edge simply supported thick plate depicted in Fig. 14 having for properties the Young's modulus E = 200GPa, Poisson's ratio = 0.3, density = 8000 kg·m^{-3} and thickness = 1.00m, meshed with P_2-tetrahedral elements. The computing resource used here is MATLAB online server (only thread-based pool is supported). Table 2 reports the comparison of the calculated frequencies to the reference value by NAFEMS Test FV52 [6].

Table 2

Comparison of obtained frequencies with reference values

Mode	***1, 2, 3***	**4**	**5**	**6**	**7**	**8**	**9**	**10**
NAFEMS f_{ref}	-	45.90	109.44	109.44	167.89	193.59	206.19	206.19
f_{cal}	-	44.64	108.74	108.77	166.14	194.17	205.63	206.90
Error (%)	-	-2.75	-0.64	-0.62	-1.04	0.30	-0.27	0.35

We notice that the error between the two frequencies is less than 1%. Therefore, we conclude that the rearrangement in the proposed algorithms does not modify the discretized global form in Eq. (6).

In terms of computational cost, Fig. 15 shows the CPU time of alg.Opt, base and alg.Vect (on 1core) and in parallel (2 or more threads), Fig. 16 and Fig. 17 depict the performance of alg. Vect. We precise that *ndof = 3*, *Ndof = nne*ndof*: is the total number of all degrees of freedom. Only alg.Vect has been studied in parallel since it is a bit faster.

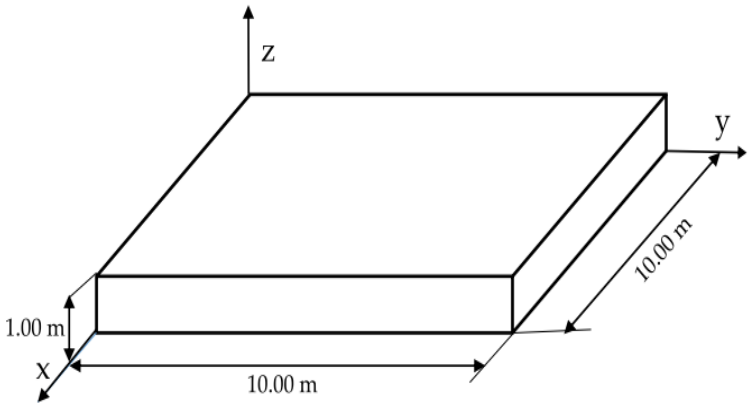

Fig. 14. **Schematic drawing of the 3D plate**

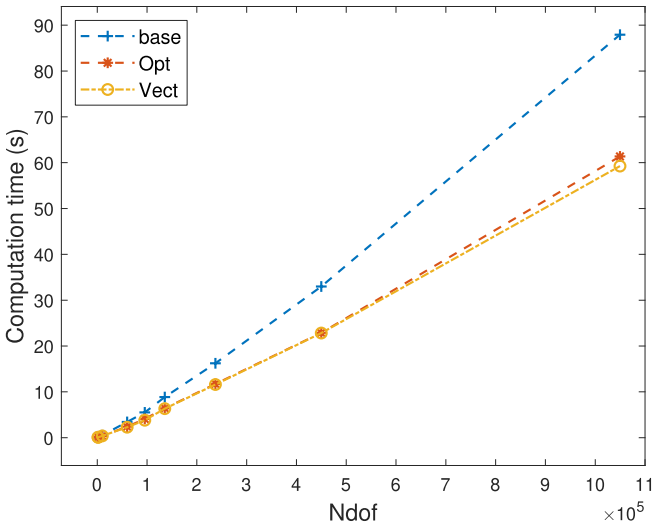

Fig. 15. **CPU time vs Ndof**

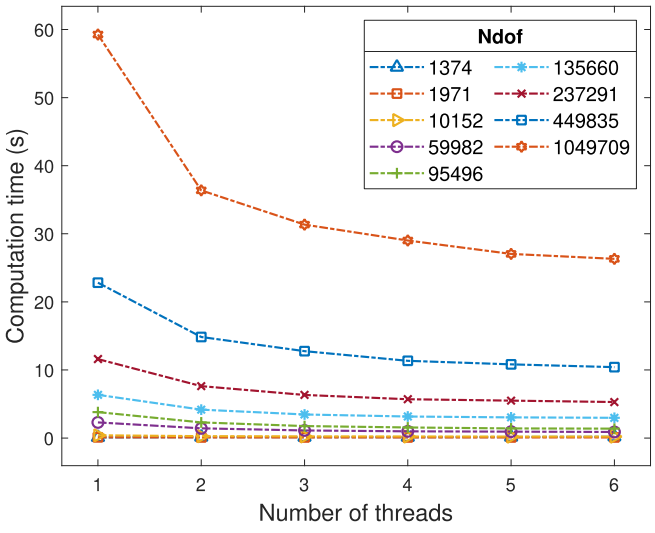

Fig. 16. **CPU time vs number of threads**

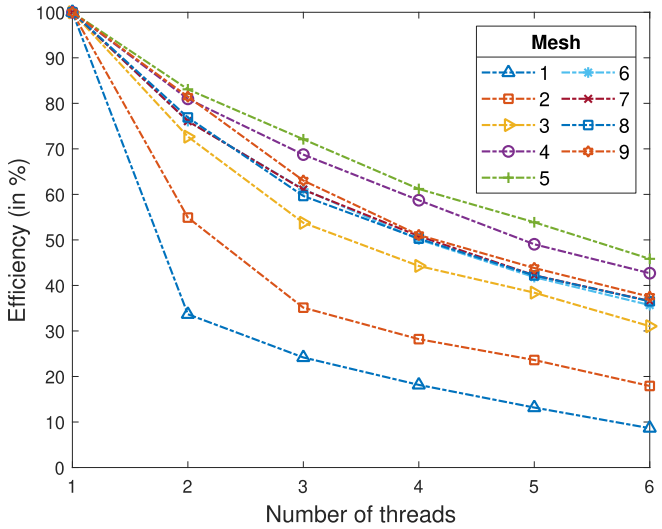

Fig. 17. **Efficiency vs number of threads**

Conclusions

In this paper, we propose two improved algorithms for easy FEM calculation of global matrix/force in serial and parallel processing. These algorithms exploit both good programming practices and MATLAB functionalities to overcome some limitations of the standard algorithm. As demonstrated in NAFEM TEST FV32 and FV 52, they are accurate, memory-efficient, and as fast as the standard algorithm. For meshes with greater than 10 000 elements in 2D and 3D, we achieve with 2 cores/threads an efficiency of about 60% and 70%, respectively. However, when the number of cores/threads is greater, the efficiency drops and nearly stagnates at 40%, this is due to overhead. In forthcoming works, possible directions are the following:

1. improve parallel efficiency using distributed computing;
2. implementation in Julia, Python and C++ languages;
3. extension of the code to nonlinear elasticity and material nonlinearity.

Acknowledgements

The authors acknowledge the funding provided by the University of Pardubice through the grant number SGS 2024 and the Cultural and Educational Grant Agency of the Slovak Republic (KEGA), project No. 003TnUAD 4/2022.

Author contributions

Conceptualization, B.S.S.T., J.K.; methodology, B.S.S.T. and J.K.; software, B.S.S.T., J.K., J.P. and V.K.; validation, B.S.S.T., J.K., J.P. and V.K.; formal analysis, B.S.S.T. and J.K.; investigation, B.S.S.T., J.K., J.P. and V.K.; data curation, B.S.S.T., J.K., J.P. and V.K.; writing – original draft preparation, B.S.S.T. and J.K.; writing – review and editing, B.S.S.T., J.K., J.P. and V.K.; visualization, B.S.S.T., J.K., J.P. and V.K.; project administration, J.K. and J.P.; funding acquisition, B.S.S.T., J.K., J.P. and V.K. All authors have read and agreed to the published version of the manuscript.

References

[1] Liu W.K., Li S., Park H.S. Eighty years of the finite element method: Birth, evolution, and future. Archives of Computational Methods in Engineering. 2022 Oct;29(6): pp. 4431-4453. DOI: 10.1007/s11831-022-09740-

[2] The MathWorks, Inc. MATLAB version: 9.13.0 (R2022b). [12.01.2024] [online] Available at: https://www.mathworks.com

[3] Rahman T., Valdman J. Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements. Applied mathematics and computation. 2013 Mar 1; 219(13): pp. 7151-7158. DOI: 10.1016/j.amc.2011.08.043

[4] Anjam I., Valdman J. Fast MATLAB assembly of FEM matrices in 2D and 3D: Edge elements. Applied Mathematics and Computation. 2015 Sep 15; 267: pp. 252-263. DOI: 10.1016/j.amc.2015.03.105
[5] Jolivet P. Domain decomposition methods. Application to high performance computing. PhD thesis, Université de Grenoble Alpes 2014, p.1- 130. [online] [05.01.2024] Available at : https://tel.archives-ouvertes.fr/tel-01155718
[6] NAFEMS Ltd. The Standard NAFEMS BENCHMARKS TNSB Rev. 3, NAFEMS Ltd, Scottish Enterprise Technology Park, Whitworth Building 1990, East Kilbride, Glasgow, United Kingdom.
[7] Sumets P. Computational Framework for the Finite Element Method in MATLAB® and Python. CRC Press; 2022 Aug 11. DOI: 10.1201/9781003265979
[8] Geuzaine C, Remacle JF. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering 2009,79(11), pp. 1309-1331. [online] [05.01.2024] Available at: https://gmsh.info/
[9] Zienkiewicz O.C., Taylor R.L., Zhu J.Z. The finite element method: its basis and fundamentals. Elsevier 2013,7th ed.p1-714. DOI: 10.1016/C2009-0-24909-9
[10] Wunderlich W., Pilkey W.D. Mechanics of structures: variational and computational methods. CRC press 2002. DOI: 10.1201/9781420041835