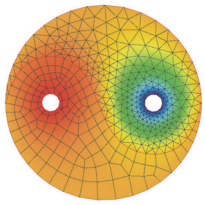
Publishing House
AKAPIT

COMPUTER METHODS IN MATERIALS SCIENCE

Informatyka w Technologii Materiałów

Vol. 16, 2016, No. 3



USING THE SYSTEM OF GRAPH GRAMMAR FOR GENERATION OF QUASI OPTIMAL ELEMENT PARTITION TREES IN TWO DIMENSIONS

ANNA PASZYŃSKA[1]*, IWONA ŚWIDERSKA[1], MACIEJ WOŹNIAK[2], KONRAD JOPEK[2], MACIEJ PASZYŃSKI[2], EWA GRABSKA[1], ANDREW LENHART[3], DONALS NGUYEN[3], KESHAV PINGALI[3]

[1] Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, ul. St. Łojasiewicza 11, 30-348 Krakow, Poland
[2] Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Cracow, Poland
[3] Institute for Computational and Engineering Science, The University of Texas in Austin, USA
**Corresponding author: anna.paszynska@uj.edu.pl*

Abstract

The paper presents a graph grammar based approach for *h*-adaptive finite element method and multi-frontal solver algorithm. The multi-frontal solver is used for solving systems of linear equations created by finite element method. The multi-frontal solver is controlled by so-called ordering. The quality of ordering influences hardly the solver effectiveness. In our approach, the finite element mesh is represented by means of a hypergraph and corresponding element partition tree. The finite element operations like mesh generation or *h*-adaptation are modeled by graph grammar production. Additionally graph grammar productions have corresponding productions for the construction of the element partition tree. The element partition trees are transformed into the ordering that controlls execution of the solver algorithm. We show that the ordering resulting from our element partititon tree results in better performance of the parallel solver than the state of the art nested-dissection ordering available through MUMPS interface on the class of grids refined towards singularities.

Key words: Graph grammar, Automatic *hp* adaptivity, Finite Element Method, optimal element partition tree, multi-frontal solver

1. INTRODUCTION

Finite element method is the most popular method of solution of several engineering problems (Demkowicz, 2006; Hughes, 2000; Pilat, 2004). The multi-frontal solver (Duff & Reid, 1983; Duff & Reid, 1984; Geng & Oden: & van de Geijn, 2006) is the state-of-the-art algorithm for LU factorization of matrices resulting from adaptive finite element method computations (Demkowicz, 2006). The input to classical multi-frontal solvers like MUMPS solver (MUMPS) is the global matrix obtained by assembling element frontal matrices. The computational complexity of the multi-frontal solver algorithm depends on the quality of the constructed ordering (Liu, 1990).

The multi-frontal solver operating time is strongly dependent on the so-called ordering given on the input and specifying the manner and sequence of operations carried out on the matrix elements.

The problem of finding the optimal ordering for the multi-frontal solver by exhaustive search is an NP-hard problem (Yanakakis, 1981). In (Paszyńska et al., 2015) a greedy algorithm called *area and neighbors* for finding of the ordering for multi-frontal solver is presented. Area and neighbors is a greedy algorithm constructing element partition

 ISSN 1641-8581

trees. The element partition tree is actually called the elimination trees in (Paszyńska et al., 2015) solver, but to avoid conflict with the elimination tree constructed from the orderings inside the classical solvers like e.g. MUMPS solver we choose the name element partition tree. The algorithm takes as the input the finite element mesh and constructs the good quality element partition tree. The algorithm for n-elements mesh starts with a forest consisting of n one node trees. Each tree corresponds to one element of a mesh. The algorithm creates element partition tree in n-1 steps. In each step it takes *two* trees $T1$ and $T2$ from a forest, remove them from the forest and adds new tree (newly created root with two offspring - $T1$, $T2$) to the forest. The algorithm chooses trees $T1$ and $T2$ in a greedy way in such a way, that minimizes the number of floating points operation performed by multi-frontal solver with constructed element partition tree. The algorithm is described in details in (Paszyńska et al., 2015). The element partition trees can be transformed to the orderings, as it is described in chapter 8 of the book (Paszynski, 2016).

Performed tests have shown that the multi-frontal solver algorithm with orderings resulting from the element partition tree found by area and neighbors algorithm for the grid with point and edge singularity runs faster than MUMPS with nested-dissection, PORD and AMD algorithm (Paszyńska et al., 2015). For a uniform grid the algorithm of the multi-frontal solver based on the area and neighbors algorithm is comparable with the nested dissection that has been proved to be the best on uniform grids (Paszyńska et al., 2015). The main idea presented in the paper is constructing of the quasi optimal element partition tree, similar to the trees created by the area and neighbors algorithm, during the mesh generation and refinement process. The element partition tree can be constructed step by step, by graph-grammar productions, at the same time when the mesh is refined.

The paper is an extension of the results presented in (Ślusarczyk & Paszyńska, 2012) in which the hypergraphs grammar was used to model the mesh generation process. Each rule, which performs graph transformations suitable for mesh transformations (for example h-adaptation) will be extended in order to perform suitable transformations on the element partition tree being generated. As the result, we obtain graph modeling the adapted mesh and the corresponding quasi-optimal element partition tree. The tree will be an input to the multi-frontal solver.

The graph grammar model has been already utilized for both expressing the adaptive mesh transformations (Strug et al., 2013) and the multi-frontal solver algorithm itself (Paszynski & Shaefer, 2010). However, in our previous work we utilized CP-graph grammars, and in this paper we propose the usage of the hypergraph grammar (Ślusarczyk & Paszyńska, 2012). Our extended hyper-graph grammar construct the finite element mesh and element partition trees. The trees are processed by our multi-frontal solver (Paszyńska et al., 2015) implemented in GALOIS environment (Pingali et al., 2011), that translates the element partition tree into an ordering. We compare the execution time, efficiency and speedup of our solver (Paszyńska et al., 2015) with those provided by state-of-the-art MUMPS solver (MUMPS).

The paper is organized as follows. We start from definining the hypergraph, hypergraph grammar and tree grammar. Next, the hypergraph grammar for modeling of the finite element mesh and tree grammar for modeling of the element partition trees are introduced. Finally, we send our element partition trees to the GALOIS solver (Paszyńska et al., 2015) and compare its efficiency with MUMPS solver.

2. HYPERGRAPHS AND HYPERGRAPH GRAMMAR

In the paper, the hypergraph grammar productions are used to model the mesh transformation while the tree based productions are used to create the element partition trees. The hypergraphs consist of nodes (labelled, attributed) and so-called hyperedges with sequences of source and target nodes assigned to them. For each hypergraph a sequence of external nodes is specified. The number of external nodes determines the type of graph, e.a. the hypergraph with k external nodes has the type k. An exemplary hypergraph of type 3 is presented in figure 1.

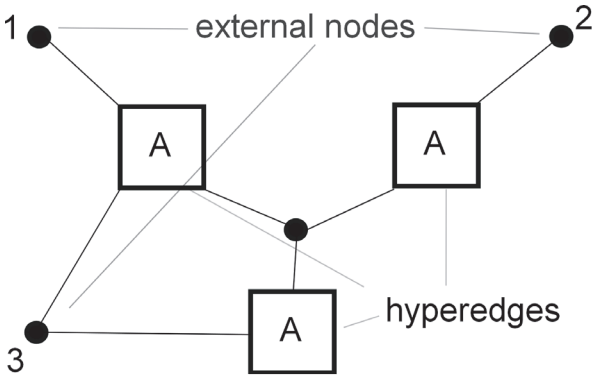


***Fig. 1**. An exemplary hypergraph of type 3.*

Definition 1. An undirected attributed labelled hypergraph over label alphabet C and attribute set A is defined as a system $G=(V,E,t,l,at)$, where:

- V is a finite set of nodes,
- E is a finite set of hyperedges,
- $t:E\rightarrow V^*$ is a mapping assigning sequences of target nodes to hyperedges, where V^* denotes a set of sequences of graph nodes
- $l:V\cup E\rightarrow C$ is a node and hyperedge labelling function,
- $at:V\cup E\rightarrow 2^A$ is a node and hyperedge attributing function.

Complex hypergraphs are derived from the simpler ones by so-called graph grammar productions.

The productions allow for replacing hyperedges by new hypergraphs. This operation is allowed only under the assumption, that for each new hypergraph a sequence of its external nodes is specified. These nodes correspond to target nodes of a replaced hyperedge.

Definition 2. A hypergraph of type k is a system $H=(G,ext)$, where:

- $G=(V,E,t,l,at)$ is a hypergraph over C and A,
- ext is a sequence of specified nodes of V, called external nodes, with $|ext|=k$.

Figure 1 presents an exemplary hypergraph of type 3 with denoted external nodes.

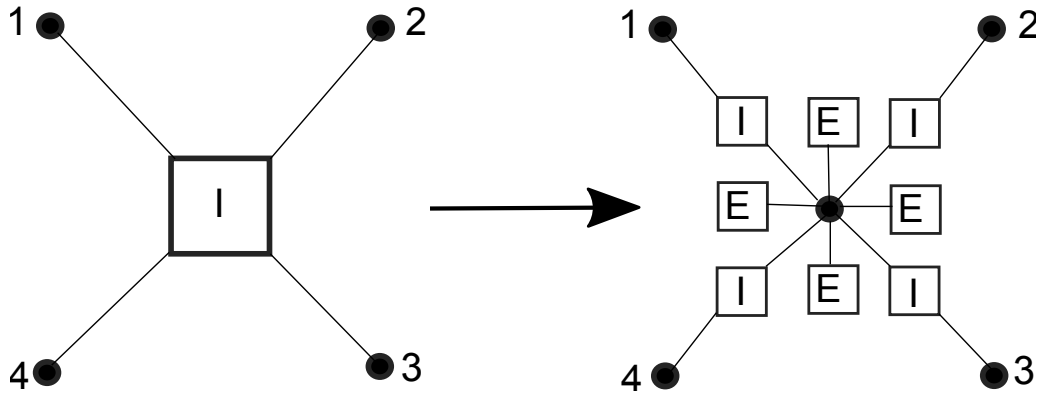


Fig. 2. *An exemplary hypergraph production.*

Definition 3. A hypergraph production is a pair $p=(L, R)$, where both L and R are hypergraphs of the same type.

To apply a production $p=(L,R)$ to a hypergraph H means to replace a subhypergraph h of H isomorphic with L by a hypergraph R and replacing external nodes from graph h with the corresponding external nodes of R.

Figure 2 presents an exemplary hypergraph production.

Figures 3 and 4 present an exemplary starting hypergraph and the graph after application of the production from figure 5.

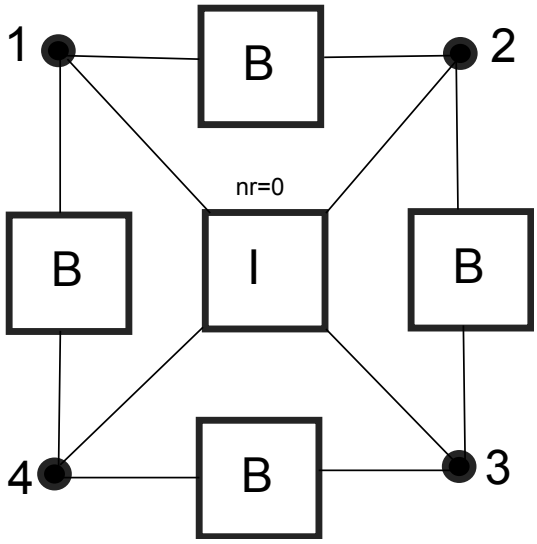


Fig. 3. *An exemplary starting hypergraph with denoted attribute nr.*

Definition 4. A hypergraph grammar is a system $G=(C,P,S)$, where:

- C is a finite set of nodes and edges labels,
- P is a finite set of hypergraph productions of the form $p=(L,R)$, where L and R are hypergraphs of the same type,
- S is an initial hypergraph.

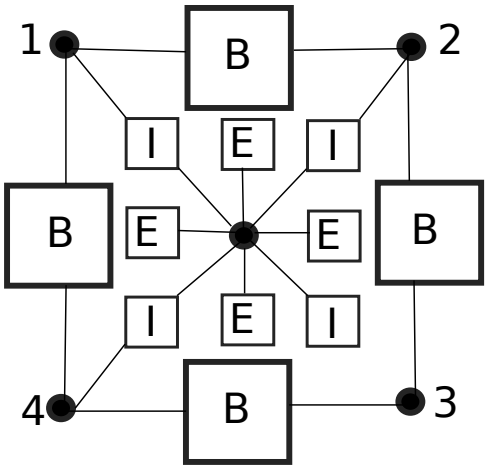


Fig. 4. Graph from figure 3 after application of the production from figure 5.

3. TREE GRAMMAR

Definition 5. A tree is a finite set of one or more nodes such that

- there is a unique node called root,
- the remaining nodes are partitioned into $m \geq 0$ disjoint sets of trees $T_1, T_2, ..., T_m$. The trees $T_1, T_2, ..., T_m$ are called subtrees of T.

Definition 6. A ranked alphabet is a pair (D,r) where D is a finite label alphabet and the rank r is defined as $r \subseteq D \times N$. The rank of a node defines the number of its direct offspring.

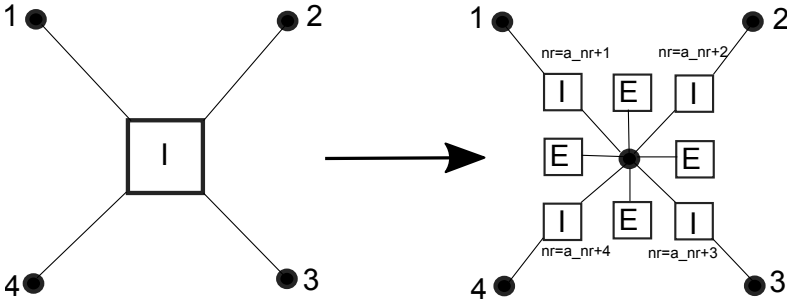


Fig. 5. Production for breaking the interior.

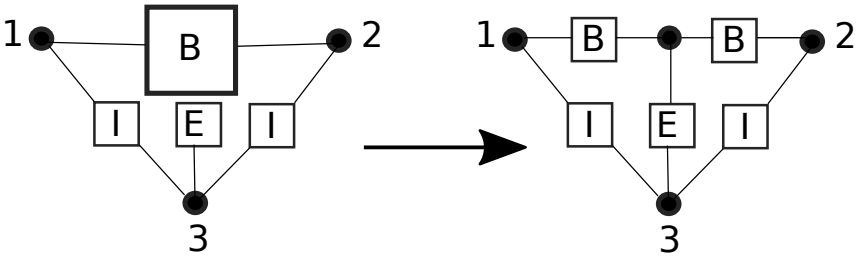


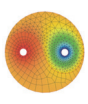
Fig. 6. One of production for Breaking Boundary Edge.

4. GRAPH GRAMMAR AND TREE GRAMMAR FOR *H*-ADAPTIVE FINITE ELEMENT METHOD WITH CONSTRUCTION OF THE ELEMENT PARTITION TREE

In this section an extension of the hypergraph grammar modeling the *h*-adaptive finite method (Ślusarczyk & Paszyńska, 2012) is presented. The extensions concern the construction of the element partition tree by means of tree grammar. Each hypergraph grammar production modeling *h*-adaptation (breaking a finite element) will be followed by corresponding tree grammar production in order to modify element partition tree. In the model, each finite element mesh is represented by means of a hypergraph. The hypergraph will consists of hyperedges representing boundary edges (denoted by *B*), shared edges denoted by *E*, interiors of elements denoted by *I* and vertices of a hypergraph which represents nodes of finite elements. The hypergraph corresponding to one element mesh is presented in figure 3. An example of finite element mesh and the corresponding hypergraph are presented in figures 14 and 15.

In this section only the productions for breaking of the elements and edges are presented, because this productions must be followed by the corresponding tree grammar productions for constructions of the element partition trees. After the description of productions two examples of derivation, including generation of the hypergraph representing the finite element mesh as well as the construction of element partition tree are presented. The examples concern one point singularity and one edge singularity.

Figure 5 presents production for breaking an element (breaking the interior of an element). The production can be applied only if the value of the attribute *h* is equal to 1. To break an element means to create four new interiors, four new edges and one new node of newly created elements. The corresponding graph will consist of 4 new hyperedges labelled by *I*, 4 new hyperedges labelled by *E* and one new node. The hyperedges labelled by *I* have



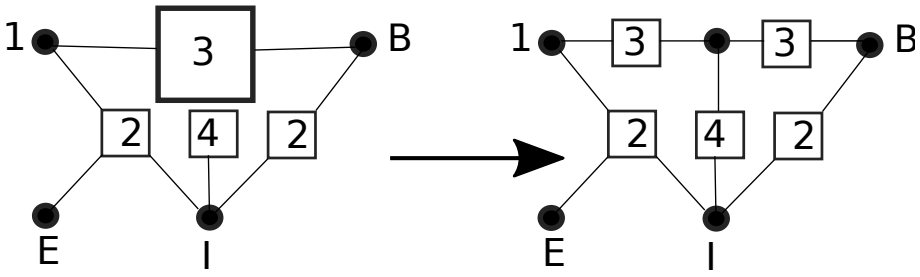


Fig. 7. One of production for Breaking Boundary Edge.

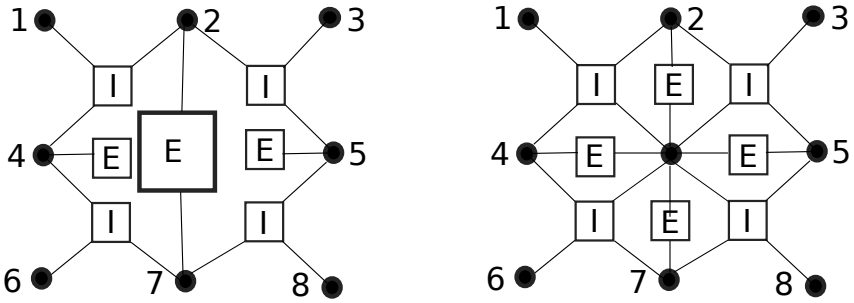


Fig. 8. One of production for Breaking Boundary Edge.

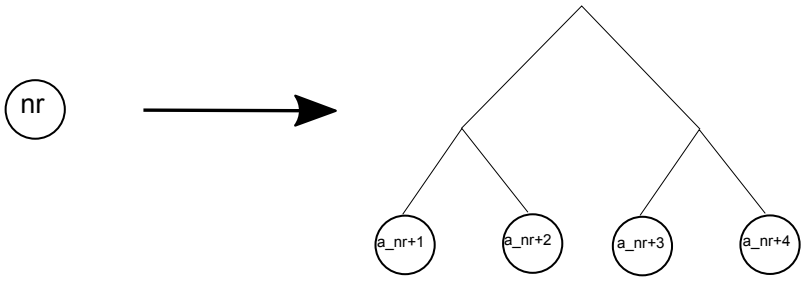


Fig. 9. Production adding new nodes corresponding to newly created finite elements to the element partition tree.

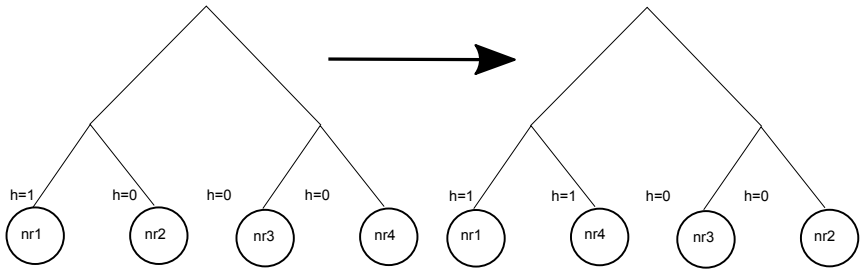


Fig. 10. Production modifying the element partition tree in the case of mesh with edge refinements.

attribute *nr* which denotes the number of the corresponding element. In this production, we use the global numbering of elements. The enumeration starts with 1. The *a nr* denotes the last used value. After breaking of the interior some edges should be broken. An edge can be broken if it is a boundary edge denoted by *B* and is neighboring to one broken element - see figures 6 and 7. A shared edge denoted by *E* can be broken if it is surrounded by two broken interiors (see figure 8).

Our goal is to construct element partition tree during the process of *h*-adaptation. Each application of a production which breaks finite element (breaks interior - see figure 5) will be followed by production for tree refining (adding new nodes to the element partition tree) presented in figure 9. In this production, we use the global numbering of elements. The enumeration starts with 1. The *a_nr* denotes the last used value. In the case of adaptation towards an edge additional production which rebuilt the tree is used. In this case application of the production of rebuilding the tree is necessary in order to create the element partition trees according to the area and neighbors algorithm presented in (Paszyńska et al., 2015). The production is shown in figure 10. The production must be performed before the production for tree refining. Attribute *h* of nodes of element partition trees (see figure 10) denotes the kind of adaptation. If attribute *h* of an element equals 1, it means that the element should be broken. In the other case (*h*=0) the element will not be broken. The decisions about making (or not) an adaptation are described in productions for virtual refinement presented in (Ślusarczyk & Paszyńska, 2012).

Figures 11-24 present step by step the mesh, the hypergraph representing it and the corresponding element partition tree for point singularity.

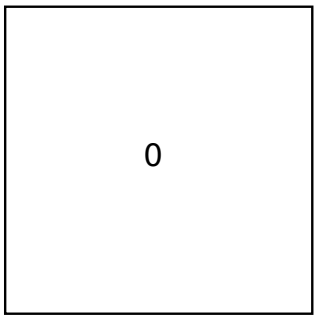


Fig. 11. An exemplary initial mesh.

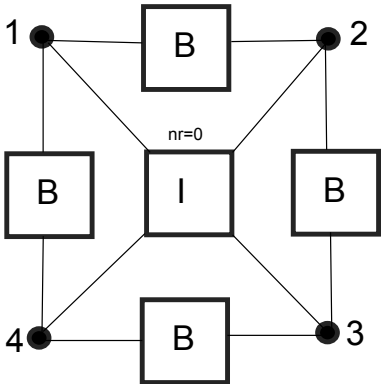


Fig. 12. A hypergraph representing initial mesh from figure 11.

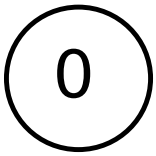


Fig. 13. Element partition tree constructed for mesh from figure 11.

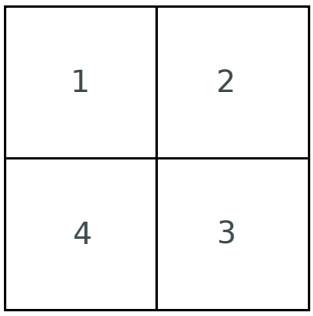


Fig. 14. The mesh from figure 11 after first refinement.

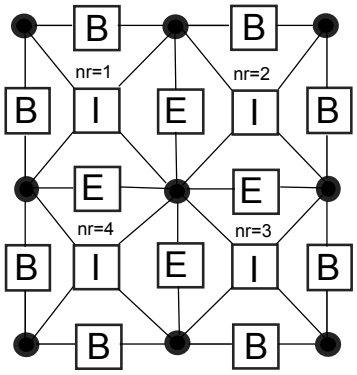


Fig. 15. The hypergraph for exemplary mesh from figure 14.

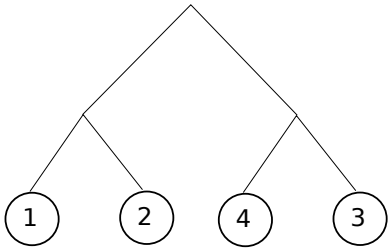


Fig. 16. Element partition tree constructed for mesh from figure 14.

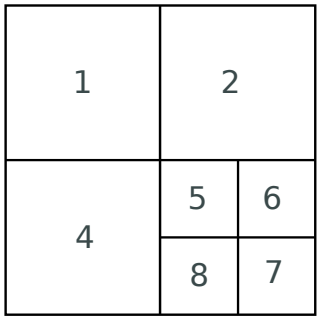


Fig. 17. The mesh from figure 11 after second refinement towards the point.

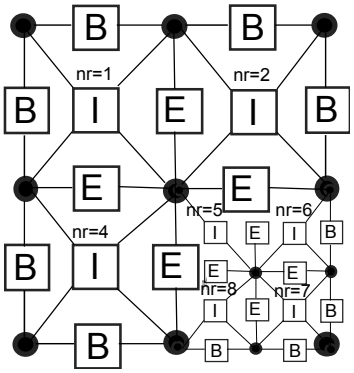


Fig. 18. The hypergraph for exemplary mesh from figure 17.

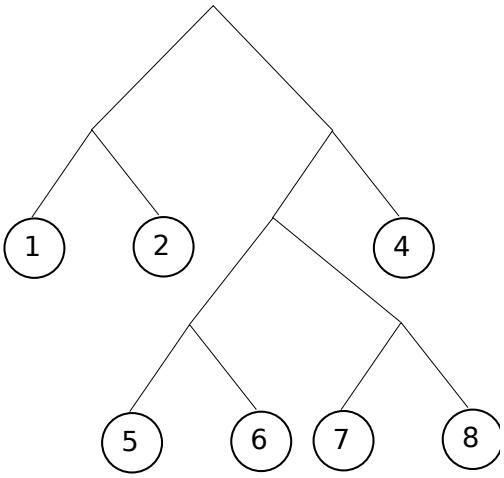


Fig. 19. *Element partition tree constructed for mesh from figure 17.*

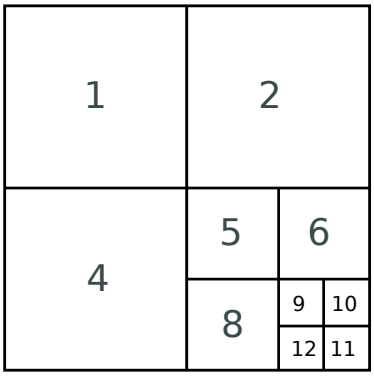


Fig. 20. *The mesh from figure 11 after third refinement towards the point.*

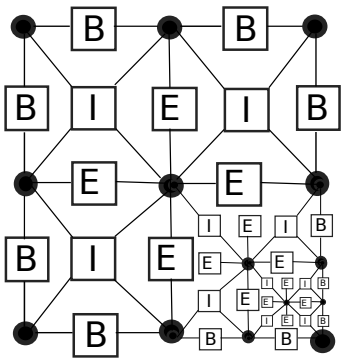


Fig. 21. *The hypergraph for exemplary mesh from figure 20.*

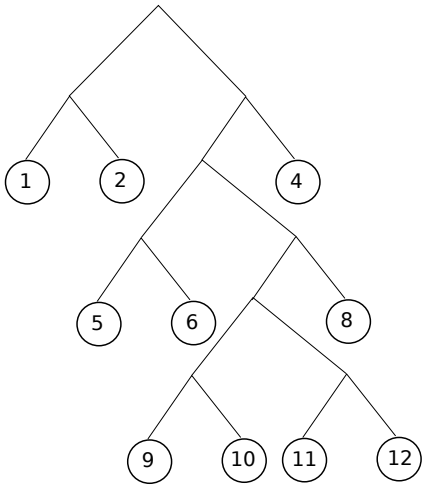


Fig. 22. *Element partition tree constructed for mesh from figure 20.*

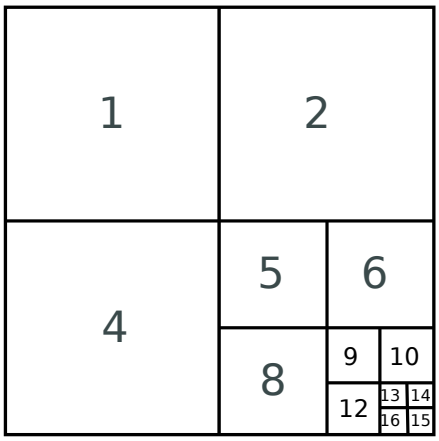


Fig. 23. *The mesh from figure 11 after fourth refinement towards the point.*

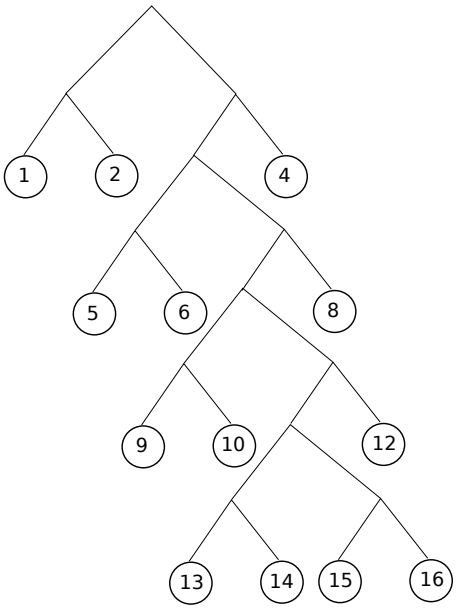


Fig. 24. *Element partition tree constructed for mesh from figure 23.*

Figures 25-41 presents step by step the mesh and the corresponding element partition tree for edge singularity. For clarity of the figures values of attribute *nr* are omitted in some hypergraphs. Figures 25-28 are similar as for the case of one point singularity. Figure 28 represents element partition tree after applying the production for adding new nodes into the element partition tree. Because the attribute *h* for two nodes is equal to 1, the production for rebuilding the tree must be performed in order to make this two nodes sons of the same node. The tree from figure 28 after application of production for rebuilding the tree is presented in figure 29. Similar situation is presented in figure 34.

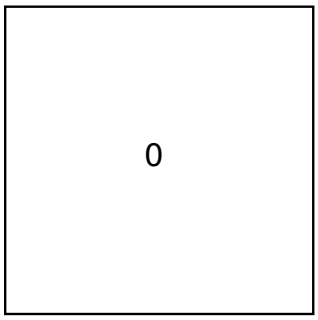


Fig. 25. *An exemplary initial mesh.*

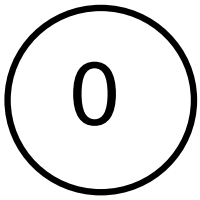


Fig. 26. *Element partition tree constructed for mesh from figure 25.*

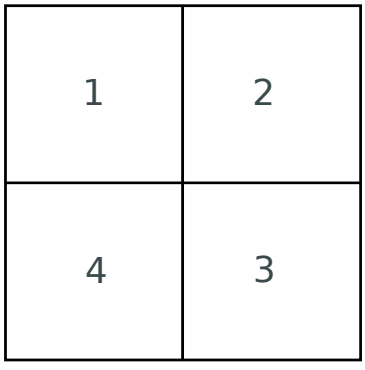


Fig. 27. *The mesh from figure 25 after first refinement.*

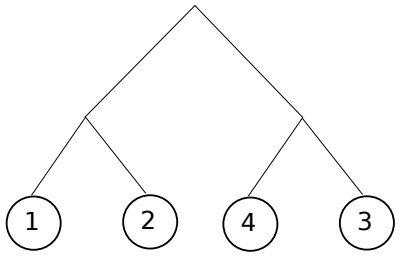


Fig. 28. *Element partition tree constructed for mesh from figure 27.*

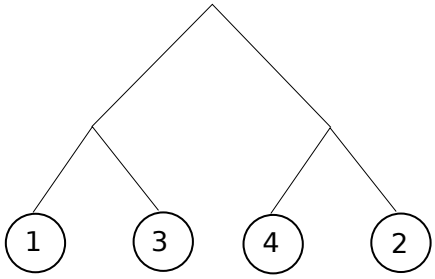


Fig. 29. *Element partition tree from figure 28 after applying of production of rebuilding the tree from figure 10.*

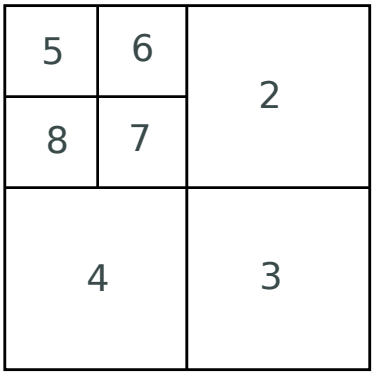


Fig. 30. *The mesh from figure 25 after second refinement towards the edge.*

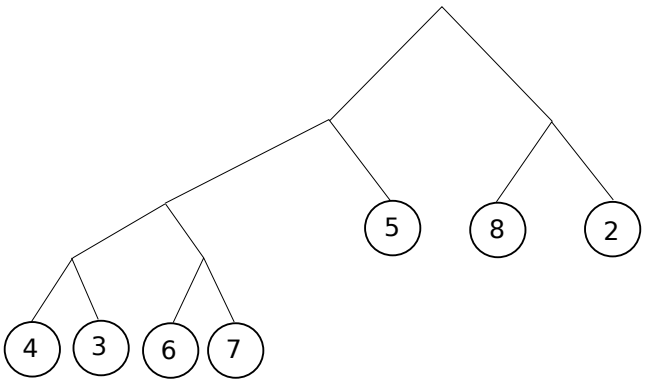


Fig. 31. *Element partition tree constructed for mesh from figure 30.*

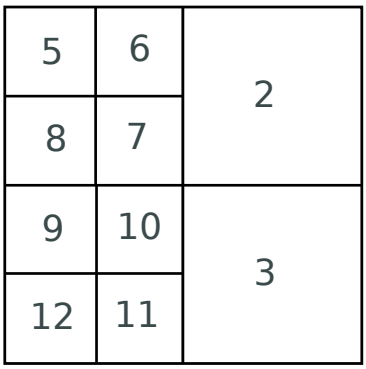


Fig. 32. *The mesh from figure 25 after third refinement towards the edge.*

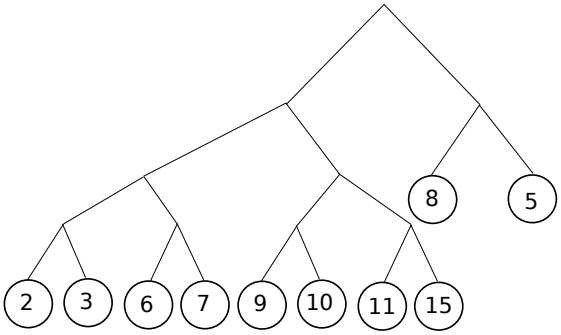


Fig. 33. *Element partition tree constructed for mesh from figure 32.*

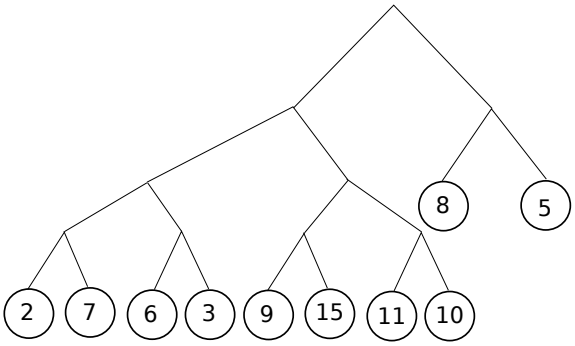


Fig. 34. *Element partition tree from figure 33 after application of the production of rebuilding the tree (twice).*

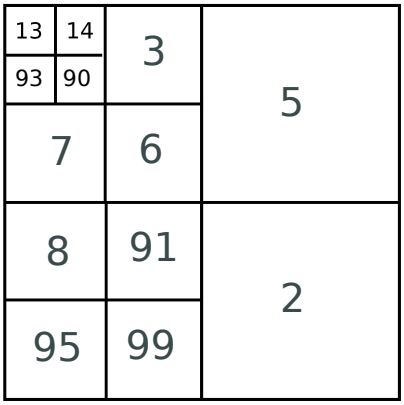


Fig. 35. *The mesh from figure 11 after fourth refinement towards the edge.*

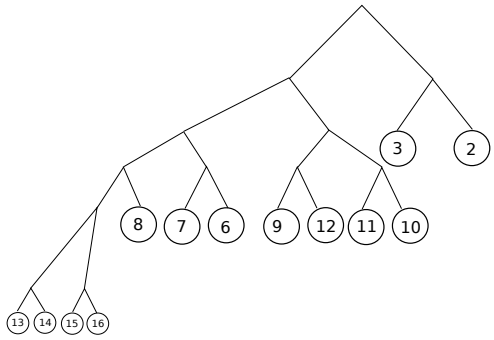


Fig. 36. *Element partition tree constructed for mesh from figure 35.*

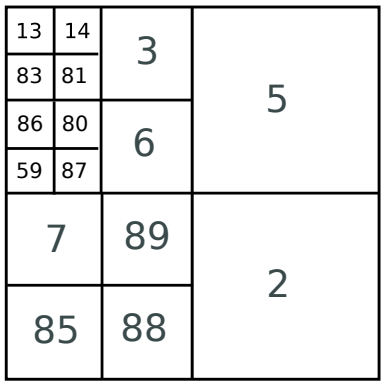


Fig. 37. *The mesh from figure 25 after fifth refinement towards the edge.*

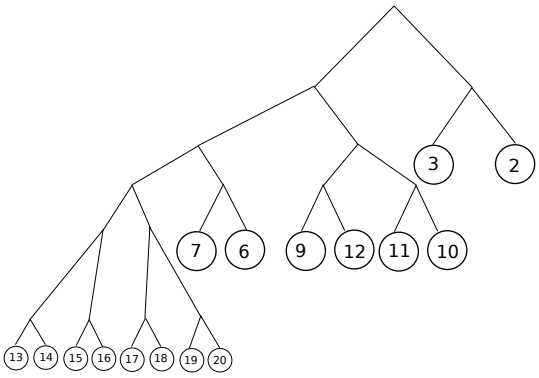


Fig. 38. *Element partition tree constructed for mesh from figure 37.*

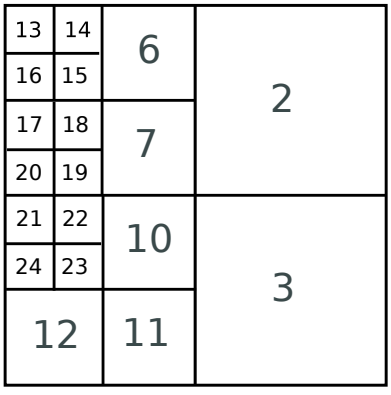


Fig. 39. *The mesh from figure 25 after sixth refinement towards the edge.*

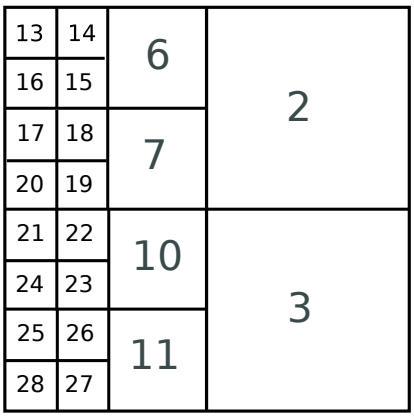


Fig. 40. *The mesh from figure 25 after seventh refinement towards the edge.*

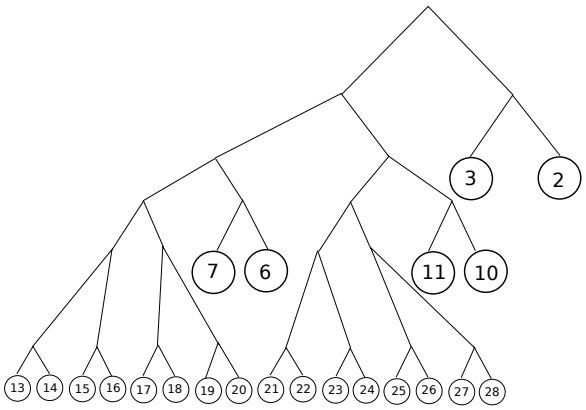


Fig. 41. *Element partition tree constructed for mesh from figure 40.*

4. TOOL FOR IMPLEMENTING GRAPH AND TREE GRAMMAR PRODUCTIONS

Many different tools for graph rewriting are available from both commercial and scientific area. The most popular are PROGRES (PROgrammed Graph REwriting Systems) or GROOVE (Graphs for Object-Oriented Verification). The motivation for work on GraphTool - a new tool for graphs - was to propose an unified environment that offers the possibility not only to create a structure of graphs with semantic layers, but to generate new graphs in an automatic manner as well.

The application is implemented to support modeling by means of graph. The functional areas that are addressed by the tool cover support for building different types of graphs and rewriting them using transformation rules. GraphTool offers graph micromodeling using composite graphs (Ryszka et al., 2015) and graphs with layers (Palacz et al., 2015).

Next available version of GraphTool focuses on the support for attributed hypergraphs. The user can model such objects by means of hyperedges and vertexes. A graph transformation rule called a production can be created for these graphs as well. To leverage the process of defining embedding transformation for the production attributes and template attributes are used.

Further improvement implemented in the GraphTool covers the process of graph derivation. It has been enhanced by the possibility of automatic generating of an element partition tree. Using predefined hyperedges labels and operations on attributes during the graph rewriting process the corresponding graph is modeled presenting the main graph transformations.

Regarding implementation details GraphTool is an application developed using Java SDK 7.0 environment. It is implemented as a plugin for Eclipse. However, it can be shipped as a stand-alone application (Eclipse Rich Client Application) for Windows or Linux platform as well. The tool uses open source libraries for Eclipse such as Graphical Editing Framework (GEF 2016) for visualization layer. For modeling an elimination tree was used a Zest library (ZEST 2016) that is the Eclipse Visualization Toolkit - a set of visualization components built for Eclipse. Its main advantage is predefined layout package supporting tree or radial layout algorithm. The Java Universal Network/Graph Framework (JUNG, 2016) defines the base for modeling the structure graph elements in the application. Although the library offers support for hypergraphs, some extensions were needed in order to follow designed graph transformations (e.g.. support for template attributes.)

5. NUMERICAL RESULTS

We have sent the element partition tree through the format required by the GALOIS multi-frontal solver, and executed the multi-thread GALOIS sovler on the trees, and compared with parallel MUMPS solver working with ordering provided by nested-dissection algorithm implemented in METIS library (METIS). We focus on two dimensional grid with edge singularity, as presented in figure 40, and solve 2D Laplace equations with finite element method. But the equation solved is not impportant here, since scalability of the solver will be identical for any scalar valued 2D elliptic PDE. In our numerical experiments we increase the mesh size by performing additional refinements.

In this section we present the detailed comparison of the execution time, efficiency and speedup for the implementation of our graph grammar solver (Paszyńska et al., 2015) with GALOIS scheduler (Pingali et al., 2011), for two dimensional grids with edge singularities.

MUMPS solver has been compiled with gfortran-4.8.0 and linked to metis-4.0.3, atlas-3.10.1, LAPACK-3.4.2, ScaLAPACK-2.0.2.

We also compare in figure 46 the number of FLOPS of the ordering resulting from our graph grammar approach, and the one provided by

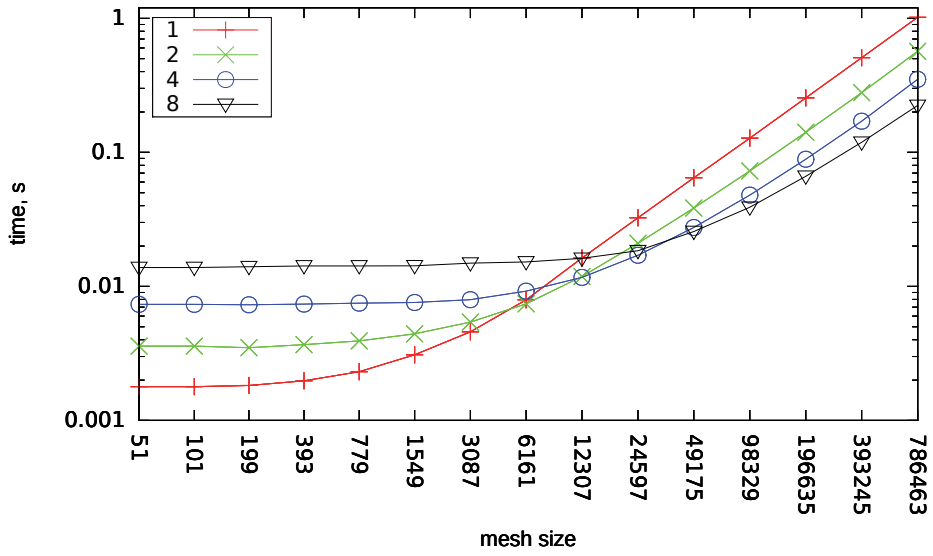


Fig. 42. *Execution time of GALOIS solver for edge singularity for 1,2,4,8 cores*

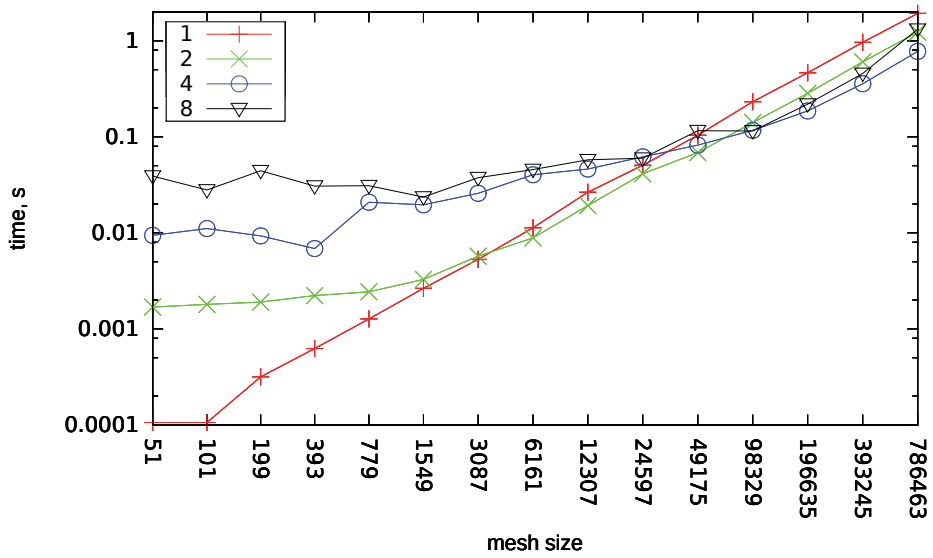


Fig. 43. *Execution time of MUMPS solver for edge singularity for 1,2,4,8 cores*

From the comparison presented in figures 42-45 for the edge singularity we can read that our sequential GALOIS solver has faster execution time than sequential MUMPS solver. MUMPS solver utilizes Cholesky factorization (the problem is symmetric and positive definite) and our solver utilizes LU factorization, so our implementation of factorization routines is actually two times slower than in MUMPS case. Our GALOIS solver is a pure C code and it has been compiled with gcc-4.8.0. The MUMPS solver. The number of operations performed by our solver with the trees generated by graph grammar is lower than the number of operations performed by the MUMPS solver with nested-dissections orderings.

Our GALOIS based solver has also better parallel efficiency and speedup than MUMPS solver, for grid with edge singularity. Tests have been performed on a single node of ATARI linux cluster with 8 cores Intel(R) Xeon(R) CPU, with 2.4 GHz,

total 16 GB RAM. However MUMPS solver uses MPI as the communication mechanism, and the GALOIS is using its own mechanism for multi-thread communication.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a graph grammar for generation of the element partition trees for

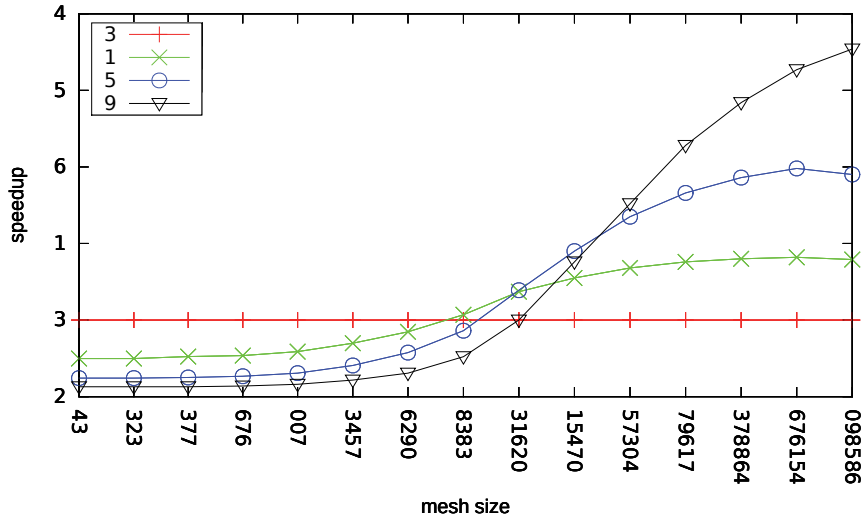


Fig. 44. *Speedup of GALOIS solver for edge singularity for 1,2,4,8 cores*

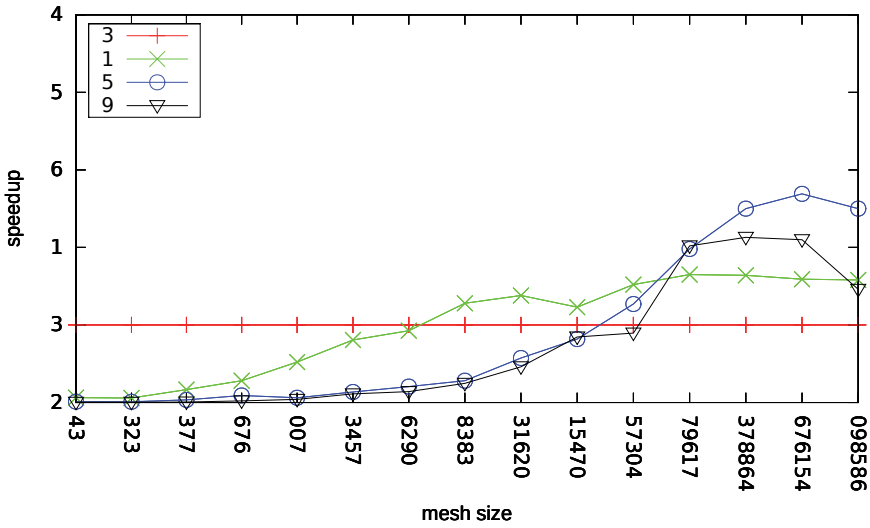


Fig. 45. *Speedup of MUMPS solver for edge singularity for 1,2,4,8 cores*

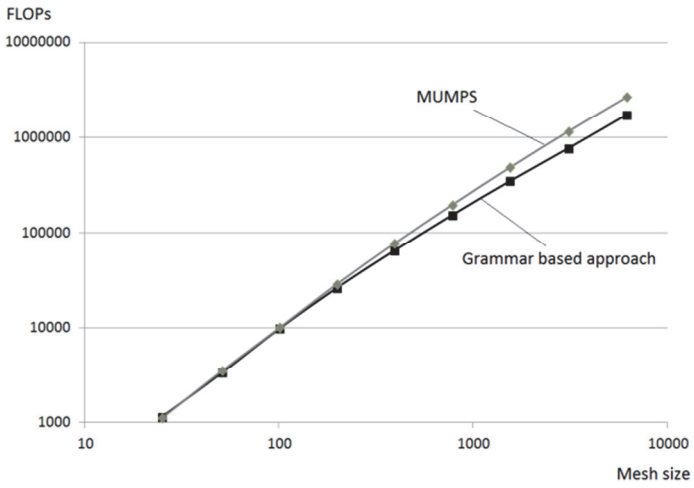


Fig. 46. *Comparison of nimber of FLOPS*

adaptive grids, at the same time as the mesh adaptation. The element partition tree controls the execution of the multi-frontal solver algorithm. The obtained element partition trees have been interfaced with GALOIS multi-frontal solver. The scalability of the resulting multi-thread multi-frontal direct solver was tested by a sequence of numerical experiments. We show that GALOIS based solver with provided element partition trees outperforms the MUMPS solver on the example of edge singularity. This is because we have a better element partition tree resulting from graph grammar analysis as well as better scheduler for shared memory parallel machine.

ACKNOWLEDGEMENTS

The work presented in this paper has been supported by National Science Centre, Poland grant no. DEC-2012/07/B/ST6/01229.

REFERENCES

Demkowicz, L., 2006, Computing with hp-Adaptive Finite Elements, vol. I. One and Two Dimensional Elliptic and Maxwell Problems, Chapman and Hall / CRC Press, New York.

Duff, I. S., Reid, J. K., 1983, The multifrontal solution of indefinite sparse symmetric linear, *ACM Trans. Math. Softw*, 9(3), 302-325.

Duff, I. S., Reid, J. K., 1984, The multifrontal solution of unsymmetric sets of linear equations, *Journal on Scientific and Statistical Computing* 5, 633-641.

GEF, 2016, Graphical editing framework for eclipse, available online at: http://www.eclipse.org/gef, accessed: 23.10.2016.

Geng, P., Oden, T. J. van de Geijn, R. A., 2006, A parallel multifrontal algorithm and its implementation, *Computer Methods in Applied Mechanics and Engineering*, 149, 289-301.

JUNG, 2016, Java universal network/graph framework, available online at: http://jung.sourceforge.net/, accessed: 23.10.2016.

Liu, J., 1990, The role of element partition trees in sparse factorization, *SIAM Journal of Matrix Analysis Applications*, 11(1), 134-172.

METIS, Metis - graph partitioning and fill-reducing matrix ordering, available online at: http://glaros.dtc.umn.edu/gkhome/views/metis, accessed: 23.10.2016.

MUMPS, Multi-frontal massively parallel sparse direct solver, available online at: http://mumps.enseeiht.fr/, accessed: 23.10.2016.

Palacz,W., Ryszka, I., Grabska, E., 2015, A graph grammar tool for generating computational grid structures, *Proceedings of the Artificial intelligence and soft computing Conference*, eds, Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L., Zurada, J., ICAISC 2015, Zakopane, Poland, 436-447.

Paszyńska, A., Paszyński, M., Jopek, K., Woźniak, M., Goik, D., Gurgul, P., AbouEisha, P., Moshkov, M., Calo, V. M., Lenharth, A., Nguyen, D., Pingali, K., 2015, Quasi-optimal elemination trees for 2d grids with singularities, *Scientific Programming*, article ID 303024, 1-18.

Paszyński, M., 2016, *Fast Solvers for Mesh Based Computations*, Taylor & Francis, CRC Press., New York

Paszyński, M. Schaefer, R., 2010, Graph grammar driven parallel partial differential equation solver, *Concurrency & Computations,Practise & Experience*, 22(9), 1063-1097.

Pilat, A., 2004, Femlab software applied to active magnetic bearing analysis, *International Journal of Applied Mathematics and Computer Science*, 14(4), 497-501.

Pingali, K., Nguyen, D., Kulkarni, K., Burtscher, M., Hassaan, M., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Mendez-Lojo, M., Prountzos, D. Sui, X., 2011, The tao of parallelism in algorithms, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming language design and implementation*, eds Mary Hall, David Padua, 12-25.

Ryszka, I., Paszyńska, A., Grabska, E., Sieniek, M. Paszyński, M., 2015,. Graph transformation systems for modeling three dimensional finite element method. part ii, *Fundamenta Informaticae*, 2, 173-203.

Ślusarczyk, G. Paszyńska, A., 2013, Hypergraph grammars in hp-adaptive finite element method, *Procedia Computer Science*, 18, 1545-1554.

Strug, B., Paszyńska, A., Paszyński, M. Grabska, E., 2013,. Hypergraph grammars in hp-adaptive finite element method, International *Journal of Applied Mathematics and Computer Science*, 23(4), 839-853.

ZEST, 2016, The eclipse visualization toolkit, Available online at: http://www.eclipse.org/gef/zest accessed: 23.10.2016

ZASTOSOWANIE SYSTEMU GRAMATYK GRAFOWYCH DO GENERACJI QUASI-OPTYMANYCH DRZEW PODZIAŁÓW SIATKI W DWÓCH WYMIARACH

Streszczenie

W artykule tym prezentujemy gramatykę grafową do modelowania algorytmów *h* adaptacyjnej metody elementów skończonych oraz solwera wielo-frontalnego. Solwer wielofronatlny używany jest do rozwiązania układu równań liniowych stworzonych przez metodę elementów skończonych. Solwer ten kontrolowany jest przez tak zwany porządek eliminacji. Jakość porządku eliminacji wpływa na efektywność solwera wielofrontalnego. W naszym podejściu siatka metody elementów skończonych reprezentowana jest przez hipergraf oraz związane z nim drzewo podziałów siatki. Operacje na elementach skończonych takie jak generacja siatki oraz *h* adaptacja modelowane są przez produkcję gramatyki grafowej. Dodatkowo, gramatyka grafowa posiada powiązane produkcje do generacji drzewa podziałów siatki. Drzewo podziałów siatki z kolei transformowane jest w porządek eliminacji, który kontroluje wykonanie algorytmu solwera. Pokazujemy że porządek eliminacji uzyskany na podstawie naszego drzewa podziałów siatki daje lepszą wydajność algorytmu solwera równoległego w porównaniu z klasycznym porządkiem nested-dissections dostępnym w solwerze MUMPS, dla klas siatek adaptowalnych do lokalnych osobliwości.

Received: June 21, 2016
Received in a revised form: September 18, 2016
Accepted: October 20, 2016