Piotr Gurgul
Maciej Paszyński
Anna Paszyńska

HYPERGRAMMAR-BASED PARALLEL MULTI-FRONTAL SOLVER FOR GRIDS WITH POINT SINGULARITIES

Abstract *This paper describes the application of hypergraph grammars to drive a linear computational cost solver for grids with point singularities. Such graph grammar productions are the first mathematical formalisms used to describe solver algorithms, and each indicates the smallest atomic task that can be executed in parallel, which is very useful in the case of parallel execution. In particular, the partial order of execution of graph grammar productions can be found, and the sets of independent graph grammar productions can be localized. They can be scheduled set by set into a shared memory parallel machine. The graph-grammar-based solver has been implemented with NVIDIA CUDA for GPU. Graph grammar productions are accompanied by numerical results for a 2D case. We show that our graph-grammar-based solver with a GPU accelerator is, by order of magnitude, faster than the state-of-the-art MUMPS solver.*

Keywords hypergraph grammar, direct solver, h adaptive finite element method, NVIDIA CUDA GPU

Citation Computer Science 16 (1) 2015: 75–102

1. Introduction

The multi-frontal solver is a state-of-the-art direct solver algorithm for solving systems of linear equations [9, 10]. It is a generalization of the frontal solver algorithm first proposed in [19]. In this paper, we focus on the class of matrices generated by the adaptive finite element method [7, 8]. The finite element method is commonly used to solve many engineering problems [1, 2, 18, 28].

The multi-frontal algorithm constructs an elimination tree based on the analysis of the sparsity pattern of the matrix. The leaves in the elimination tree correspond to frontal matrices associated with particular finite elements. The solver algorithm identifies and eliminates the fully-assembled rows from the frontal matrices. The resulting sub-matrices, called Schur complements, are merged at the parent nodes of the elimination tree, and new fully-assembled nodes are identified and eliminated again. This process is repeated until we reach the root of the elimination tree. The direct solver algorithm can be generalized to the usage of matrix blocks associated with nodes of the computational mesh (called supernodes) rather than particular scalar values [6]. This allows for a reduction of computational cost related to the construction of the elimination tree. Different implementations of the multi-frontal solver algorithm also exist that target specific architectures (see, e.g., [13, 14, 15]). There is also a linearly-computational cost direct solver based on the use of H-matrices [27] with compressed non-diagonal blocks. However, the main limitation of these solvers is that they produce a non-exact solution.

The state-of-the-art multi-frontal solvers determine the way of solving the problem based on the structure of the global matrix. It is possible to improve the performance of the multi-frontal solver algorithm by leveraging the knowledge based on the computational mesh instead of a matrix. Another question is what is the lowest-possible computational cost that can be obtained for two-dimensional computational meshes containing a singularity (adaptations proceed toward a point). The question above should also be generalized onto an arbitrary number of singularities. The example of such a 2D mesh is presented in Figure 1. Namely, it must be decided whether a problem having s singularities instead of one increases the complexity only by a constant factor.

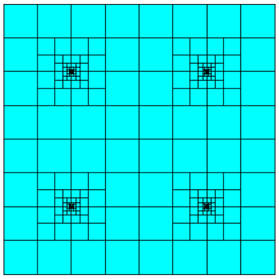

Figure 1. Exemplary two-dimensional mesh.

The last of the questions considered in this work concerns graph grammars and states whether it is possible to express an efficient direct solver algorithm within the graph grammar model.

Most known graph grammars are context-free: only one non-terminal is allowed on the left-hand side of production. The non-terminal can be a label of a node, like in Node Replacement Graph Grammars [26], or a label of an edge, like in Hyperedge Replacement Graph Grammars [16, 17]. This kind of graph grammar does not allow for modeling hp-FEM with uniform refinement. Most non-uniform adaptive FEM codes fulfil the following 1-irregularity rule: a finite element can be broken only once without breaking the adjacent large elements. This is a consequence of limitation for approximation over the common edges of big and small elements. Thus, modelling the non-uniform refinements enforces checking the neighborhood of the edge, which can be modeled only by context-sensitive graph grammars.

The first graph grammar model of mesh transformation was presented in [11]. This approach allows only for modeling uniform refinement. In order to model non-uniform refinements or solver operations, a context-sensitive graph grammar has to be used. One of the context-sensitive graph grammars is the composite graph grammar proposed in [12]. The composite graph grammar models have already been utilized for both modeling of the mesh generation and adaptation process, as well as for modeling of the solver algorithm [20, 21, 22, 23, 25, 30]. However, only the hypergraph grammar model presented in this paper results in logarithmic computational cost of the solver algorithm, as confirmed by numerical experiments. In future works, we also plan to incorporate the reutilization technique discussed in [24]. The alternative for the adaptive finite element method is the isogeometric finite element method [5, 3], where the mesh is uniform and the polynomial order of approximation is increased globally. The isogeometric FEM delivers the higher order global continuity of the solution, but it suffers from a computationally more-expensive direct solver algorithm [4].

2. Problem formulation

Grids with point refinements (as the one presented in Figure 1) can be used for solving numerical problems with point singularities. The examples of such problems are a projection problem for functions having multiple point extrema or singularities, or the solution of the heat transfer problem with several point heat sources (compare Fig. 2), or any other problem where the solution exhibits multiple point gradients.

2.1. Projection problem

In this section, we formulate the projection problem that will be solved using the graph-grammar-driven solver implemented on the GPU. The goal is to find approximation u of given function f, defined over $\Omega \subset \mathcal{R}^2$ to minimize functional with respect to u (1).

$$\|f - u\|_{L^2} \to \min \tag{1}$$

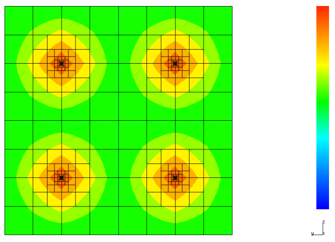

Figure 2. Exemplary two-dimensional heat sources.

This problem is equivalent to the following minimization problem

$$\frac{1}{2}b\left(u,u\right)-l\left(u\right)\rightarrow\min \tag{2}$$

where $b(w,v)$ is a symmetric, bilinear form:

$$b(w,v)=2\int\limits_{\Omega}wv\mathrm{d}\Omega \tag{3}$$

and $l(w)$ is a linear form defined as:

$$l(w)=2\int\limits_{\Omega}wf\mathrm{d}\Omega \tag{4}$$

Let $b(w,v)$ be a bilinear form on $W\times W$ and $l(v)$ be a linear form on $W=H^1\left(\Omega\right)$. Let $J(w)$ be a quadratic functional defined as $\frac{1}{2}b(w,w)-l(w)$. Then finding u that minimizes $J(w)$ over the affine space V is equivalent to finding u that satisfies:

$$b\left(u,v\right)=l\left(v\right)\ \forall_{v\in W}$$

For proof, please see [7].

By applying the above-mentioned fact, we obtain an equivalent problem:

$$b\left(u,v\right)=l\left(v\right)\ \forall_{v\in W} \tag{5}$$

Computational domain Ω is now partitioned into rectangular elements with second order shape functions defined over each finite element vertex, edge, interior. The shape functions $\alpha_i:\mathcal{R}^2\ni(x,y)\mapsto\mathcal{R}$ over master $(0,1)^2$ rectangular element are defined in the following way:

- first order polynomials over four vertices

$$\alpha_1=(1-x)(1-y) \tag{6}$$

$$\alpha_2=(1-x)y \tag{7}$$

$$\alpha_3=xy \tag{8}$$

$$\alpha_4=x(1-y) \tag{9}$$

- second order polynomials over four edges

$$\alpha_{4+1} = (1-x)x(1-y) \tag{10}$$
$$\alpha_{4+2} = (1-x)y(1-y) \tag{11}$$
$$\alpha_{4+3} = (1-x)xy \tag{12}$$
$$\alpha_{4+4} = x(1-y)y \tag{13}$$
$$\tag{14}$$

- second order polynomial over element interior

$$\alpha_9 = x(1-x)y(1-y) \tag{15}$$

Each of the vertex shape functions are equal to one-on-one vertex and vanish on the remaining vertices. Each of the edge-shape functions is a second-order polynomial over one edge, and equal to zero on all other edges. Each of the face-shape functions is a second-order polynomial over one face and zero over all other edges and faces.

We utilize a linear combination of second-order polynomials for numerical solutions, spread over finite elements vertices, edges, and interiors:

$$u = \sum_{i=1}^{n} u_i e_i. \tag{16}$$

The basis functions e_j of space V belong to the space, and the form b is linear with respect to first argument, then we can finally rewrite the problem as:

$$\sum_{i=1}^{n} u_i b\left(\alpha_i, \alpha_j\right) = l\left(\alpha_j\right) \forall_{\alpha_j \in W} \tag{17}$$

Equation (17) is considered element wise, and we generate element frontal matrices (one for each element) to be interfaced with the multi-frontal solver. Namely, for each rectangular element, we utilize a set of graph grammar productions contributing to the element matrix. The element matrix in this model projection problem looks like this:

$$\begin{pmatrix} b(\alpha_8,\alpha_8) & ... & b(\alpha_8,\alpha_1) \\ ... & ... & ... \\ b(\alpha_1,\alpha_8) & ... & b(\alpha_1,\alpha_1) \end{pmatrix} = \begin{pmatrix} l(\alpha_8) \\ ... \\ l(\alpha_1) \end{pmatrix}$$

where rows and columns are ordered with interior, followed by edges, followed by vertices shape functions.

In the numerical simulations, we have selected function f so it has a gradient going to infinity at the bottom center of the rectangular domain Ω.

$$\begin{aligned} \Omega = [-1,1] \times [0,1] \ni (x,y) \mapsto f(x,y) = \\ tan(\frac{\pi}{2}(1-|x|))tan(\frac{\pi}{2}(1-y)) \end{aligned} \tag{18}$$

The graph grammar generates a sequence of two-dimensional, increasingly-refined grids with rectangular finite elements with basis functions spread over finite element vertices, edges, and interiors, approximating function f with increasing accuracy.

The numerical examples concern a simple projection problem, but they can be easily generalized into heat transfer, linear elasticity, Maxwell equations, Stokes problem etc., provided the numerical problems generate local-point singularities.

3. Hypergraph grammar

This paper presents the hypergraph grammar([29]), which is an extension of Hyperedge Replacement Grammar ([16, 17]) for modelling mesh transformation and linear computational cost solver for grids with point singularities. The rectangular elements of a mesh as well as the whole mesh are described by means of hypergraphs. The mesh transformations are modelled by hypergraph grammar productions. Each hypergraph is composed of a set of nodes and a set of hyperedges with sequences of source and target nodes assigned to them. The nodes as well as the hyperedges are labelled with labels from a fixed alphabet. To represent the properties of mesh elements, the attributed hypergraphs are used; i.e., each node and hyperedge can have some attributes like, for example, the polynomial order of approximation.

Let C be a fixed alphabet of labels for nodes and hyperedges. Let A be a set of hypergraph attributes.

Definition 1. An undirected attributed labelled hypergraph over C and A is a system $\mathrm{G} = (V, E, t, l, at)$, where:

1. V is a finite set of nodes,
2. E is a finite set of hyperedges,
3. $t : E \mapsto V^*$ is a mapping assigning sequences of target nodes to hyperedges of E,
4. $l : V \cup E \mapsto C$ is a node and hyperedge labelling function,
5. $at : V \cup E \mapsto 2^A$ is a node and hyperedge attributing function.

The hypergraphs are created from simpler hypergraphs by replacing their subhypergraphs by new hypergraphs. This operation is possible if, for new hypergraph and the subhypergraph, a sequence of so-called external nodes is specified. The hypergraph replacing is defined as follows: the subhypergraph is removed from the original hypergraph, and the new hypergraph is embedded into the original hypergraph. The new hypergraph is glued to the reminder of the original hypergraph by fusing its external nodes with the corresponding external nodes in the reminder of the original hypergraph. The number of external nodes should be the same in both hypergraphs.

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

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

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

A production p can be applied to a hypergraph H if H contains a subhypergraph isomorphic with L.

Definition 4. Let $G_1 = (V_1, E_1, t_1, l_1, at_1)$ and $G_2 = (V_2, E_2, t_2, l_2, at_2)$ be two hypergraphs. G_1 is a subhypergraph of G_2 if:

1. $V_1 \subset V_2, E_1 \subset E_2$,
2. $\forall e \ \in E_1 t_1(e) = t_2(e)$,
3. $\forall e \in E_1 l_1(e) = l_2(e), \forall v \in V_1 \quad l_1(v) = l_2(v)$,
4. $\forall e \in E_1 at_1(e) = at_2(e), \forall v \in V_1 \quad at_1(v) = at_2(v)$.

The application of a production $p = (L, R)$ to a hypergraph H consists of replacing a subhypergraph of H isomorphic with L by a hypergraph R and replacing nodes of the removed subhypergraph isomorphic with external nodes of L by the corresponding external nodes of R.

Definition 5. Let P be a fixed set of hypergraph productions. Let H and H' be two hypergraphs.

H' is directly derived from H ($H \Rightarrow H'$) if there exists $p = (L, R) \in P$ such that:

- h is a subhypergraph of H isomorphic with L,
- Let ext_h be a sequence of nodes of h composed of nodes isomorphic with nodes of the sequence ext_L.

  The replacement of $h = (V_h, E_h, t_h, l_h, at_h)$ in $H = (V_H, E_H, t_H, l_H, at_H)$ by $R = (V_R, E_R, t_R, l_R, at_R)$ yields the hypergraph $G = (V_G, E_G, t_G, l_G, at_G)$, where:
  - $V_G = V_H - V_h \cup V_R$,
  - $E_G = E_H - E_h \cup E_R$,
  - $\forall e \in E_R t_G(e) = t_R(e)$,
  - $\forall e \in E_H - E_h$ with $t_H(e) = t_1, ..., t_n, t_G(e) = t'_1, ..., t'_n$, where each $t'_i = t_i$ if t_i does not belong to the sequence ext_h or $t'_i = v_j$ (v_j is an j-th element of the sequence ext_R) if t_i is an j-th element of the sequence ext_h,
  - $\forall e \ \in E_H - E_h \ l_G(e) = l_H(e)$, $at_G(e) = at_H(e)$, $\forall e \ \in E_R \ l_G(e) = l_R(e)$, $at_G(e) = at_R(e)$,
  - $\forall v \in V_H - V_h \ l_G(v) = l_H(v)$, $at_G(v) = at_H(v)$, $\forall v \in V_R \ l_G(v) = l_R(v)$, $at_G(v) = at_R(v)$.
- H' is isomorphic with the result of replacing h in H by R.

Let $A_T = V \cup E$ be a set of nodes and hyperedges of H, where H denotes a family of hypergraphs over C and A.

Definition 6. A hypergraph grammar is a system $G = (V, E, P, X)$, where:

- V is a finite set of labelled nodes,
- E is a finite set of labelled hyperedges,
- P is a finite set of hypergraph productions of the form $p = (L, R)$, where L and R are hypergraphs of the same type composed of nodes of V and hyperedges of E,
- X is an initial hypergraph called axiom of G.

The paper presents the hypergraph grammar for modeling mesh transformation and solver. The rectangular elements of a mesh as well as the whole mesh are described by means of hypergraphs. The hypergraph nodes represent mesh nodes and are labeled by v. The hyperedges represent interiors, edges, and boundary edges of rectangular finite elements, and are labelled by I, F and B, corresponding to interior, edge, and boundary edges, respectively. Figure 3 presents an exemplary mesh and its hypergraph representation.

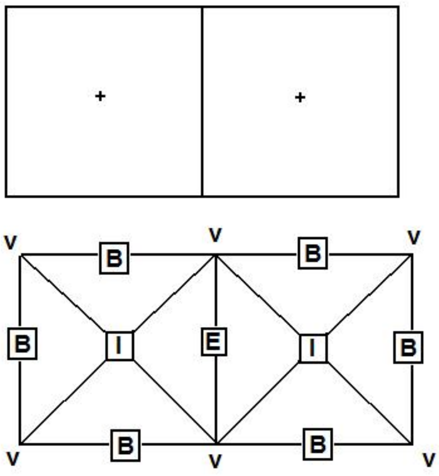

Figure 3. Exemplary mesh and graph.

4. Mesh generation

We start with the graph grammar productions that can be used for both sequential and parallel generation of the initial mesh with point singularity located at the center of the bottom of the mesh. We start with executing the production in Figure 4 that transforms the initial state into the initial mesh.

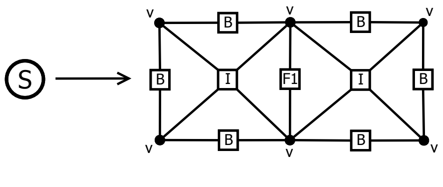

Figure 4. P_{init}

Next, we proceed with refinements of the left and right elements by executing the production in Figure 5. It needs to be explained why it is not allowable to break the interface edge between these element. This is due to the so-called *1-irregularity rule*, which states that the neighboring elements cannot vary in their refinement levels by more than one. Before breaking their interface edge, we need to ensure that both of the adjacent elements are at the same refinement level.

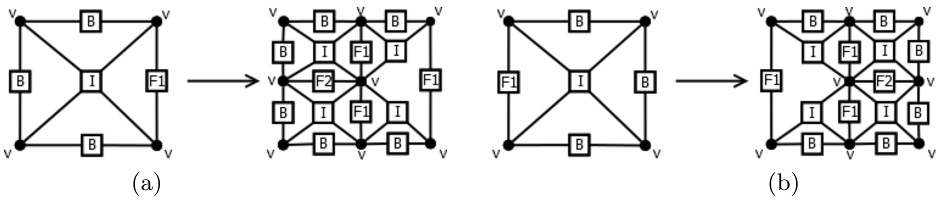

Figure 5. Productions breaking the initial mesh: $P_{initleft}$ (a), $P_{initright}$ (b).

After breaking the two adjacent elements, we can break the central edge shared between them by executing the graph grammar production presented in Figure 6.

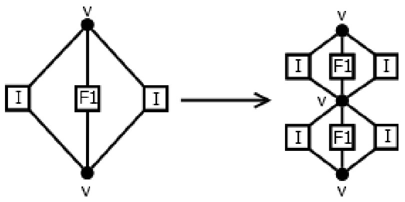

Figure 6. The graph grammar production breaking the edge $P_{irregularity}$.

In order to proceed with further refinements towards the central singularity, we can use the following productions: $P_{breakinterior}$ breaking the interiors of the two elements adjacent to the point singularity (due to symmetry, now we can use a single production for breaking interior I of both left and right innermost elements), and $P_{enforceregularity}$ breaking the common edge between them (Fig. 7 and 8). These productions can lead to a mesh refined to the arbitrary level.

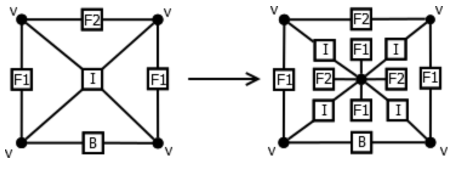

Figure 7. $P_{breakinterior}$

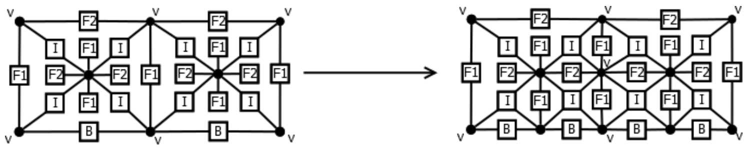

Figure 8. $P_{enforceregularity}$

5. Graph grammars for the sequential solver

Assuming we applied the following chain of productions: $P_{init} \mapsto P_{breakinitleft} \mapsto P_{breakinitleft} \mapsto P_{regularity} \mapsto P_{breakinterior} \mapsto P_{breakinterior} \mapsto P_{enforceregularity}$ we receive output mesh as in Figure 9.

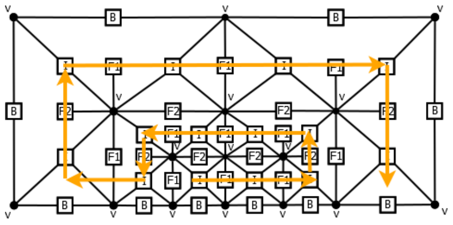

Figure 9. Browsing order of the linear solver.

The solver processes the mesh from the two bottom elements surrounding the singularity, level by level, up to the level of initial elements. To convey the idea of the graph-grammar solver, we present its behavior for the first two elements being processed. In this section, we present the complete process for building up and eliminating the element matrix for the first element browsed. This means the left innermost element of the adapted grid.

We begin with generating a contribution to the element frontal matrix coming from the interior node of the processed elements. This is done by production P_{addint} presented in Figure 10.

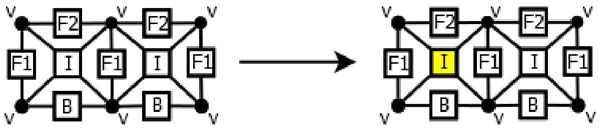

Figure 10. Production P_{addint} adding interior node to the element matrix.

There, we mark the hypergraph nodes already generated in the matrix. The system of equations for the element looks like

$$\left(\begin{array}{c} b(\alpha_8, \alpha_8) \end{array} \right) = \left(\begin{array}{c} l(\alpha_8) \end{array} \right)$$

where α_8 is the shape function related to the interior node. The element frontal matrix is also graphically illustrated on the right panel of Figures 10–22. In the next step, we add the contributions to the element frontal matrix related to interactions of the boundary node with the interior node. This is done by executing productions $P_{addboundary}$ presented in Figures 11. The system of equations for the element after this operation looks like

$$\left(\begin{array}{cc} b(\alpha_8, \alpha_8) & b(\alpha_8, \alpha_5) \\ b(\alpha_5, \alpha_8) & b(\alpha_5, \alpha_5) \end{array} \right) = \left(\begin{array}{c} l(\alpha_8) \\ l(\alpha_5) \end{array} \right)$$

where α_5 corresponds to the shape functions associated with bottom edge.

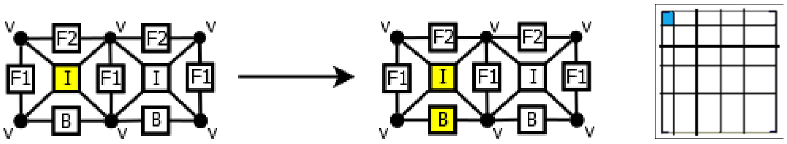

Figure 11. Production $P_{addboundary}$ adding the boundary node to the element matrix.

Next, we add the contributions to the element frontal matrix related to the interactions of the left interface edge node with the interior and boundary nodes. This is done by executing productions $P_{addF1layer}$ presented in Figure 12.

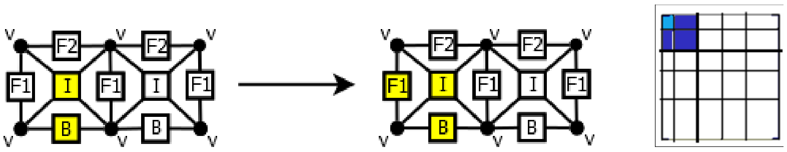

Figure 12. Production $P_{addF1layer}$ adding the left interface edge to the element matrix.

The system of equations for the element after this operation looks like

$$\begin{pmatrix} b(\alpha_8,\alpha_8) & b(\alpha_8,\alpha_7) & b(\alpha_8,\alpha_5) \\ b(\alpha_7,\alpha_8) & b(\alpha_7,\alpha_7) & b(\alpha_7,\alpha_5) \\ b(\alpha_5,\alpha_8) & b(\alpha_5,\alpha_7) & b(\alpha_5,\alpha_5) \end{pmatrix} = \begin{pmatrix} l(\alpha_8) \\ l(\alpha_7) \\ l(\alpha_5) \end{pmatrix}$$

where α_7 corresponds to the shape functions associated with left interface edge.

Next, the same operation is done for the upper interface edge node, as illustrated in Figure 13 by production $P_{addF2layer}$, as well as for the vertices, as illustrated in Figure 14 by production $P_{addvertices}$. Each time, the frontal matrix is augmented with new entries, as graphically illustrated in the right panels.

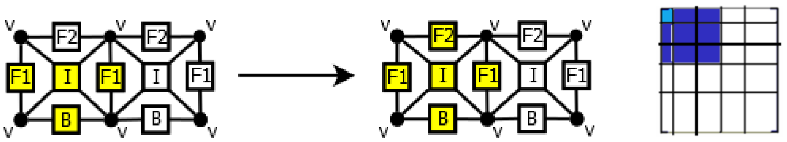

Figure 13. Production $P_{addF2layer}$ adding upper interface edge to the element matrix.

The resulting state of the element matrix is depicted in Figure 14. At this time, we completed the addition of the first element's nodes to the matrix. Now, we can proceed with the elimination of the fully-assembled nodes. Fully-assembled nodes are the ones that have all contributions already present in the matrix. It is worth mentioning that interior nodes are always fully assembled, since they vanish on all edges and have support on just one element. Edges are always shared by two elements (unless they are boundary edges). Vertex nodes can be shared by between one and four distinct elements.

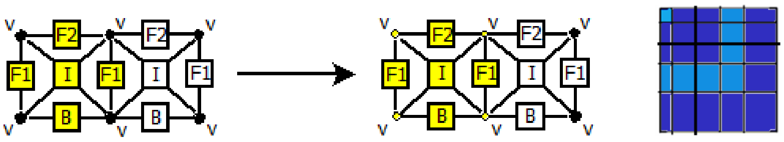

Figure 14. Production $P_{addvertices}$ adding vertices to the element matrix.

We start with eliminating the interior's contribution from the matrix. This is done by executing production $P_{elimint}$ presented in Figure 15.

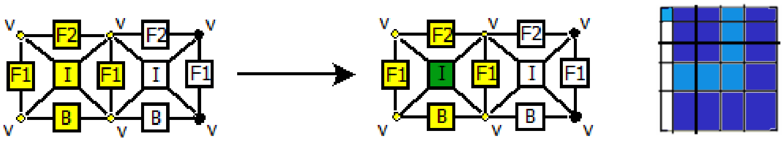

Figure 15. Production $P_{elimint}$ eliminating interior's contribution from the matrix.

We mark the eliminated nodes in a dark color. The only other fully-assembled node is the boundary node B. Thus, we proceed with its elimination by executing production $P_{elimboundary}$ presented in Figure 16.

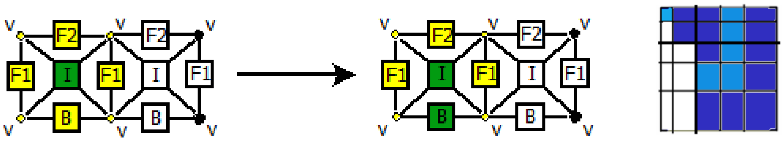

Figure 16. Production $P_{elimboundary}$ eliminating boundary edge from the element matrix.

At this time, we are done with the first element. No more nodes can be eliminated without further knowledge about the neighboring elements. We move onto the right element and start adding the contributions of its nodes. Again, first we add contribution coming from the interior node of the second element. This is done by executing production $P_{addint2}$, see Figure 17.

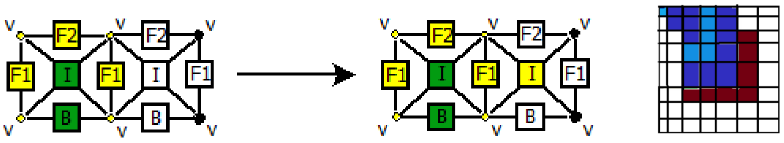

Figure 17. Production $P_{addint2}$ adding interior of the second element to the matrix.

Then, we add the contribution coming from the boundary edge by running production $P_{addboundary2}$ (Fig. 18).

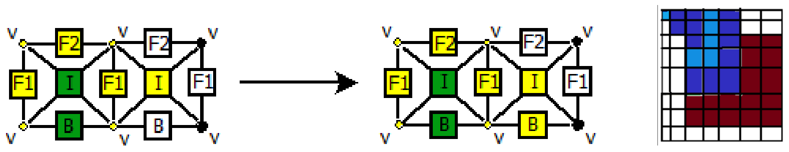

Figure 18. Production $P_{addboundary2}$ adding boundary edge to the matrix.

And the upper and right interface edges (productions $P_{addF1layer2}$, $P_{addF2layer2}$, see Fig. 19).

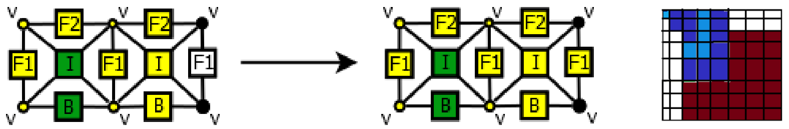

Figure 19. Productions $P_{addF1layer2}$, $P_{addF2layer2}$ adding layer nodes to the matrix.

Once we add all vertices (Fig. 19), we can proceed with the elimination of the fully-assembled nodes for the second element.

Again, we start with eliminating the contribution coming from the interior node (production $P_{elimint2}$, see Fig. 20).

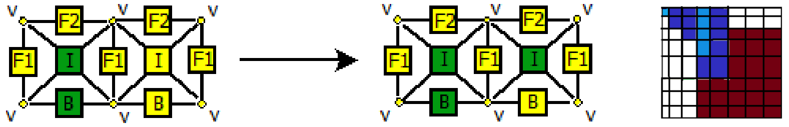

Figure 20. Production $P_{elimint2}$ eliminating interior of the second element.

Similar to the first element, we can also eliminate boundary edge (production $P_{elimboundary2}$, Fig. 21).

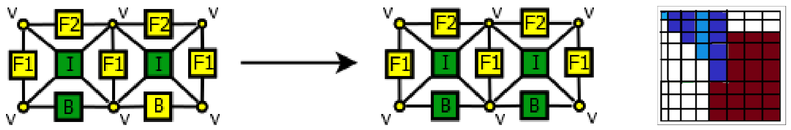

Figure 21. Production $P_{elimboundary2}$ eliminating boundary edge in the element matrix.

After adding the second element, we also have all contributions for the interface edge and vertex between elements, so now we can eliminate both (production $P_{elimcommon}$, Fig. 22).

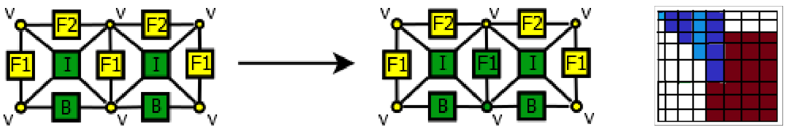

Figure 22. Production $P_{elimcommon}$ eliminating the common edge.

We continue accordingly with the next productions following the order shown in Figure 9 until we reach the very last element. Then, we can proceed with backward substitution that provides us with the solution with the given accuracy.

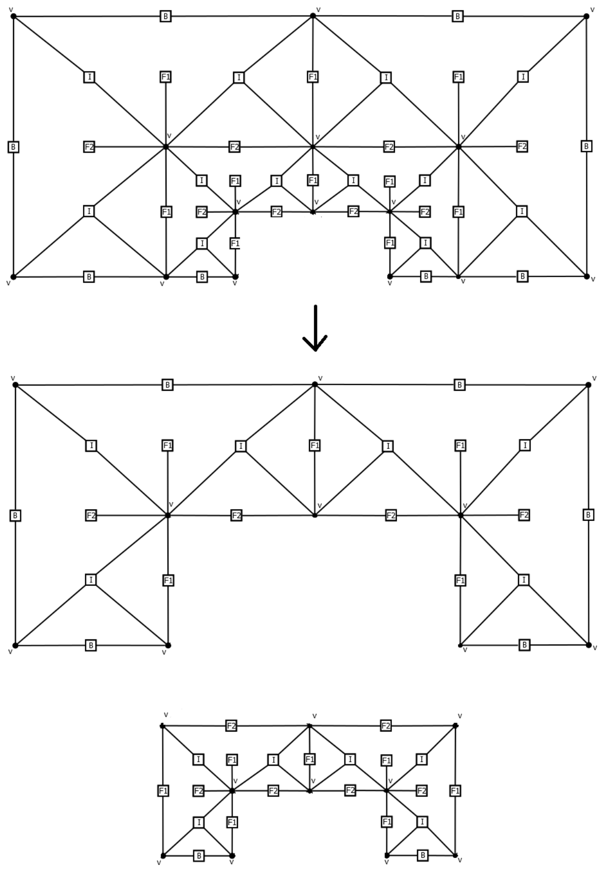

Figure 23. Production $P_{separatetop}$ for separation of the top layer.

6. Graph grammar for the parallel solver

6.1. Concurrency analysis

In this section, we mark a partial order of the productions and decide how many of them can be executed in parallel.

6.1.1. Mesh generation

Compared with the remaining parts of the algorithm, mesh generation based on graph grammars is cheap, and there is no point in parallelizing this part. Besides, most of the productions must be executed in a strictly-enforced order, since most steps depend on the mesh being in a certain state.

However, to enable parallel processing, once the mesh is generated, we fire some additional graph grammar productions, $P_{separatetop}$ presented in Figure 23 as many times as there are layers of the mesh, except for the last layer, where we fire $P_{separatebottom}$, presented in Figure 24. These graph grammar productions separate particular layers of the mesh so we can process them independently, without the overlapping of nodes.

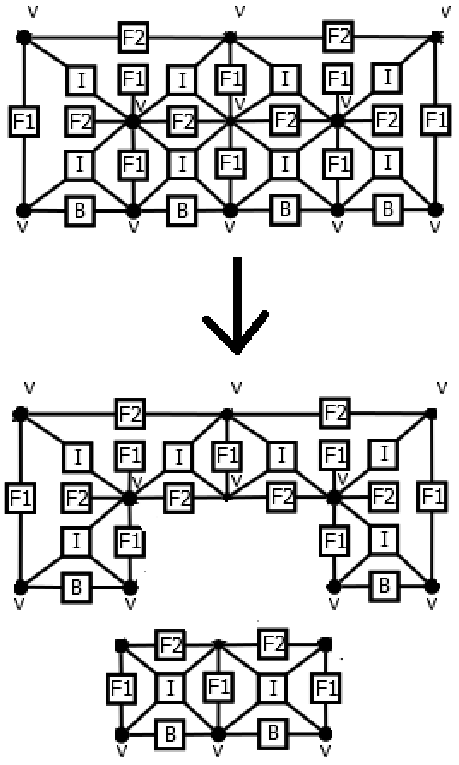

Figure 24. Production $P_{separatebottom}$ for separation of the bottom layer.

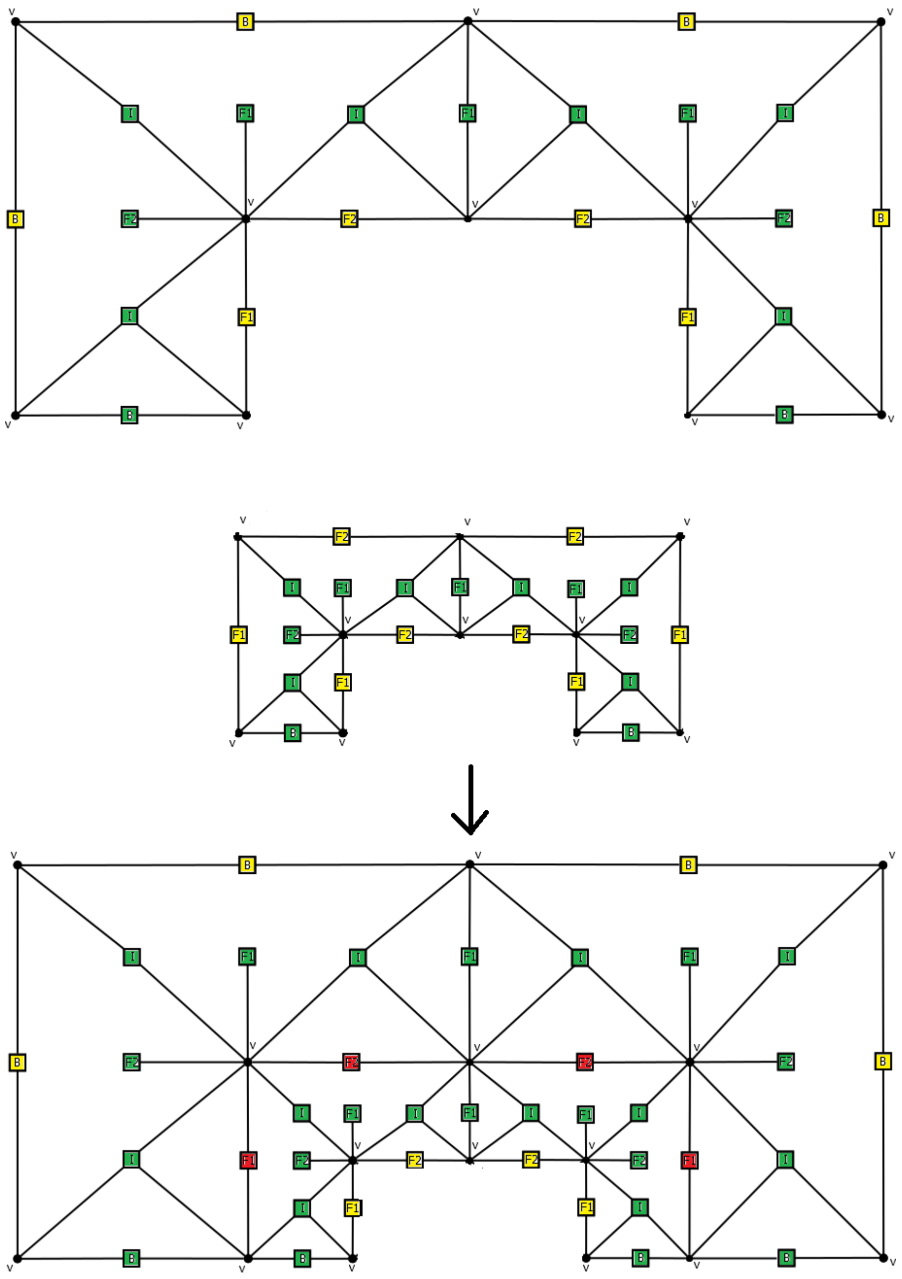

Figure 25. Production $P_{mergetop}$ for merging of the two top layers with interface matrices.

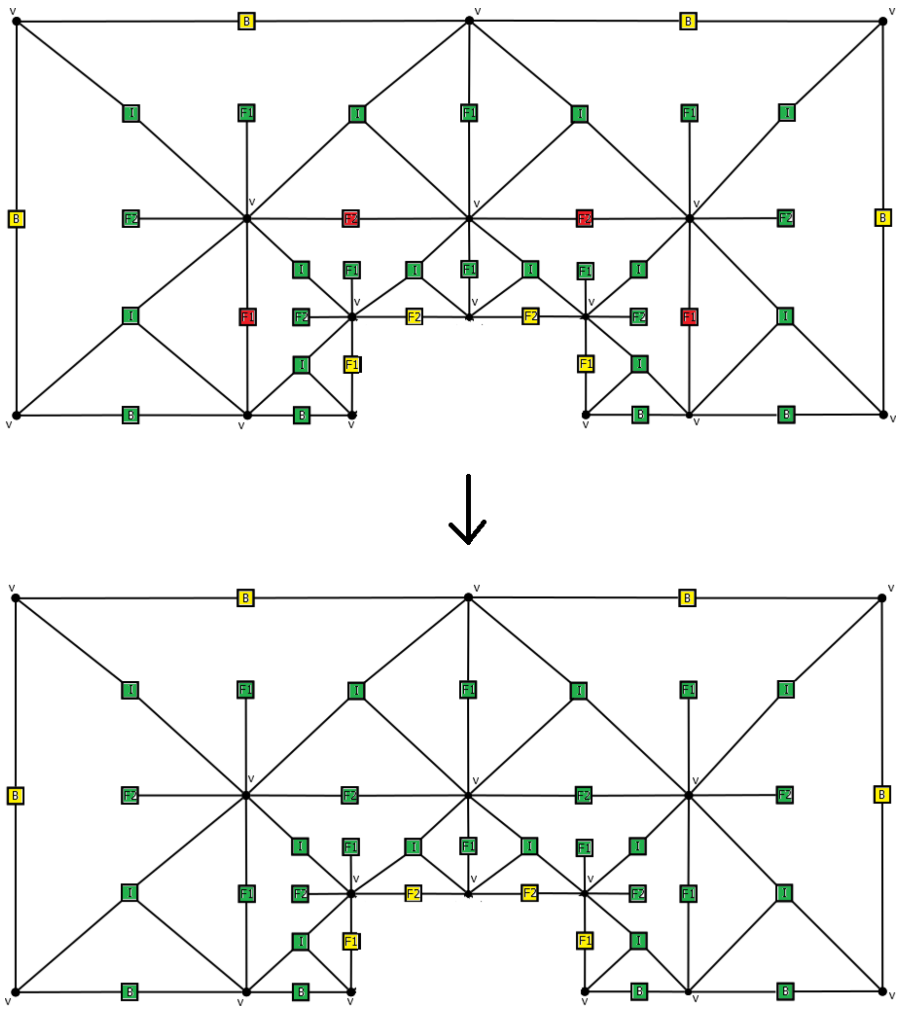

Figure 26. Production $P_{elimtop}$ for elimination of the common layer over the two already marged top layers.

6.1.2. Processing of each layer

Having the computational mesh partitioned into layers, we can process each layer fully in parallel, independently from the other layers.

The considerations presented for the bottom layer with two elements can be generalized into an arbitrary layer. The general idea of the graph grammar for the arbitrary layer is the same; however, the number of productions is two times larger, since there are four elements instead of two. Each layer can be processed independently at the same time. In other words, the graph grammar productions responsible for aggregation and merging over one layer can be executed in parallel

$$\begin{aligned} &P_{addint} \mapsto P_{addboundary} \mapsto P_{addF1layer} \mapsto \\ &P_{addF2layer} \mapsto P_{addvertices} \mapsto P_{elimint} \mapsto \\ &\qquad\qquad P_{elimboundary} \mapsto \dots \end{aligned} \tag{19}$$

where **...** denotes the additional productions for layers consisting with four elements instead of two for each layer.

Moreover, for point singularities located inside the elements, all of the considerations presented in this paper remain the same; however, the bottom layer consists of four elements and upper layer consists of eight elements.

6.1.3. Processing of the interfaces

Unlike the sequential algorithm, we keep all of the interface nodes for the parallel algorithm uneliminated until all of the layers are processed.

Having the interiors of all layers eliminated, we have several frontal matrices associated with all of the layers, one frontal matrix per layer. The frontal matrices contain the nodes from both interfaces, located on the top and bottom sides of each layer.

We execute graph grammar productions $P_{mergetop}$ merging the top two layers into one layer, additionally merging the two frontal matrices into one matrix, in such a way that the rows of the merged matrix associated with the common interface are now fully assembled. This is denoted in Figure 25, with fully-assembled nodes denoted by the color red.

Having the single frontal matrix with fully-assembled nodes related to the common interface allows us to eliminate these nodes by executing the graph grammar production $P_{elimtop}$ presented in Figure 26. The fully-assembled nodes are eliminated, which is denoted by changing their color from red to green.

The same scenario applies to the two bottom layers, which is denoted by productions $P_{mergebottom}$ and $P_{elimbottom}$ presented in Figures 27 and 28.

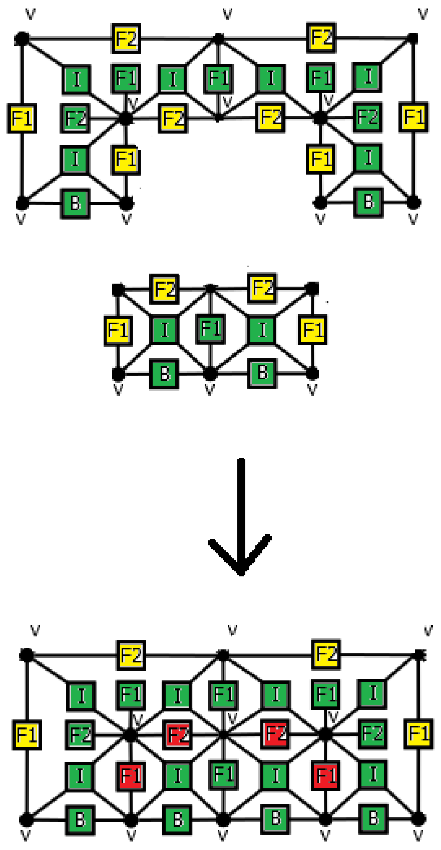

Figure 27. Production $P_{mergebottom}$ for merging of the two bottom layers with interface matrices.

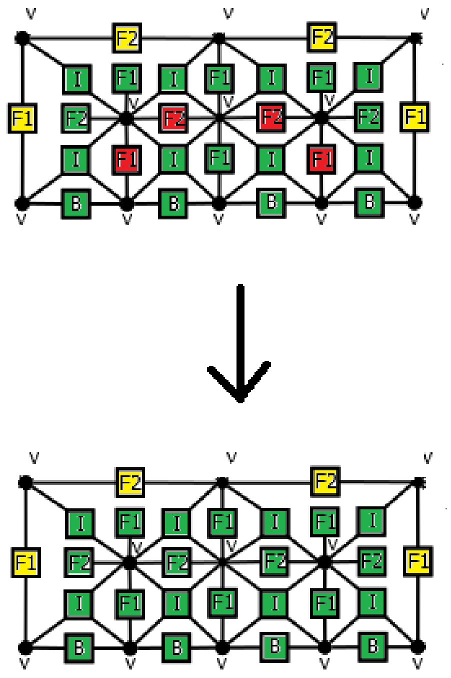

Figure 28. Production $P_{elimbottom}$ for elimination of the common layer over the two already marged bottom layers.

Finally, we merge the patches of two layers (obtained in the previous steps) into a single path. This is expressed by production $P_{mergetop}$ presented in Figure 29. Additionally, the two frontal matrices associated with the two patches are merged into one frontal matrix, with the rows associated with nodes on the common interface fully assembled. These nodes are denoted in Figure 29 by the color red. At this point, we can solve the top problem since all of the nodes are fully assembled, including the top boundary nodes. This is expressed by production $P_{solvetop}$ presented in Figure 30.

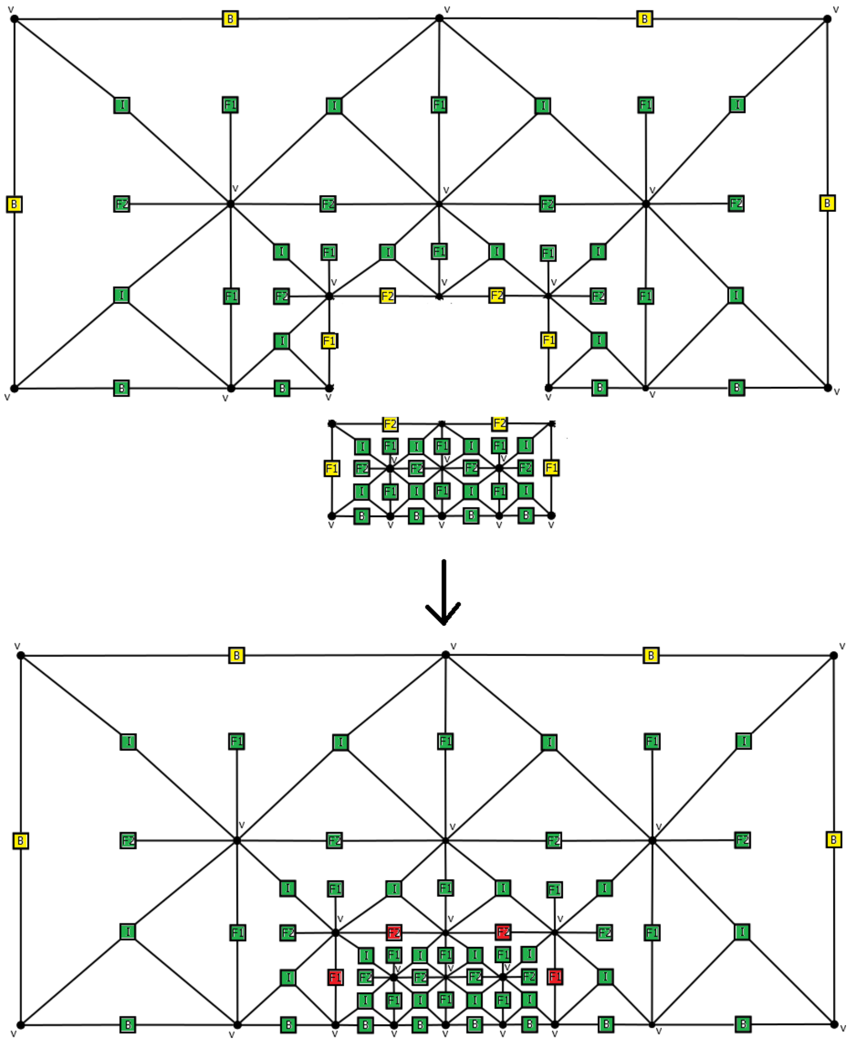

Figure 29. Production $P_{mergetop}$ for merging of the bottom and top layers, resulting in a top problem.

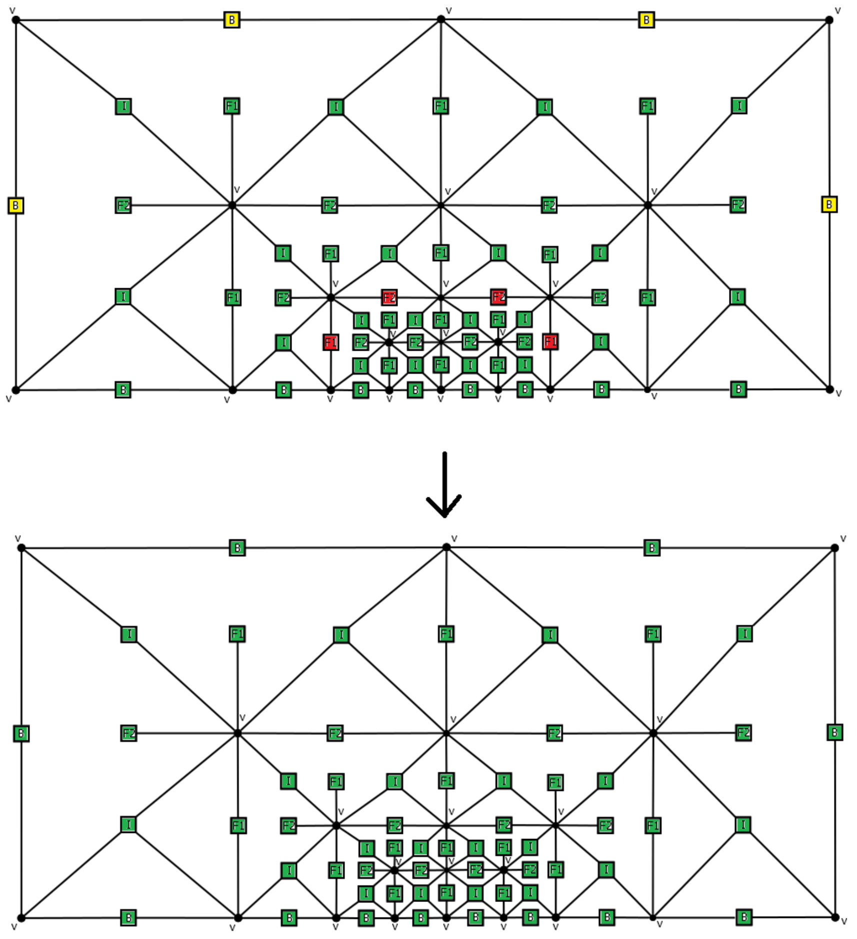

Figure 30. Production $P_{solvetop}$ for solving the top problem.

Having the top problem solved, we process with analogous backward substitutions, which basically can be expressed by identical graph grammar productions, but executed in reverse order.

For more than four layers, the procedure is identical; we only need to generate additional productions for merging patches of four, eight, or more layers.

We will illustrate this technique in the *Numerical results* section with the examples in two dimensions.

7. Solver algorithm for GPU

The graph-grammar-based solver has been implemented on a GPU with NVIDIA CUDA. The graph grammar productions have been partitioned into sets of independent tasks, and scheduled set by set concurrently, into the nodes of the GPU. The algorithm is the following:

- Execute in serial the graph grammar production constructing the mesh with point singularity $P_{init} \mapsto P_{breakinitleft} \mapsto P_{breakinitleft} \mapsto P_{regularity} \mapsto [P_{breakinterior} \mapsto P_{enforceregularity}]^n$ where n corresponds to the number of generated layers.
- Execute in serial the graph grammar production partitioning the mesh into layers $P_{separatetop} \mapsto P_{separatetop} \mapsto P_{separatebottom}$.
- Execute in concurrent the graph grammar productions responsible for generation and aggregation of interior nodes from particular layers $P_{addint} \mapsto P_{addboundary} \mapsto P_{addF1layer} \mapsto P_{addvertices} \mapsto P_{elimint} \mapsto P_{elimboundary} \mapsto$ ... where **...** corresponds to additional productions for upper layers.
- Execute graph grammar productions for merging and elimination of the interface nodes from top layers $P_{mergetop} \mapsto P_{elimtop}$ in concurrent with graph grammar productions for merging and elimination of the interface nodes from bottom layers $P_{mergebottom} \mapsto P_{elimbottom}$. For more than four layers, we obtain the binary tree structure here, processed level by level in concurrent.
- Execute the graph grammar production responsible for merging and solving the top problem $P_{mergetop}$ and $P_{solvetop}$.

NVIDIA GPU CUDA (Compute Unified Device Architecture) is a perfect architecture for these kind of problems, as it allows us to process data much more efficiently than with ordinary CPUs. Calculations are performed in parallel by hundreds of threads, allowing us to achieve logarithmic scaling of our solution.

To quickly outline modern GPU architecture – it consists of several multiprocessors (usually 8–12), each containing many cores (8 for GTX 260, 32 for Tesla C2070). Moreover, there are 4 kinds of memory: global, shared, constant, and texture. For this article, we are only interested in the first two types of memory. We could use constant and texture to speed computation, although the algorithm would then be tied to CUDA forever. Global memory can have up to 1.5GB of space and can be accessed from every multiprocessor, but its latency is considerably high, while its shared memory (up to 48KB per multiprocessor) can be accessed by all threads running on one multiprocessor. This memory is really small, but compared to global memory, it has much lower latency and much higher throughput. Our implementation tries to use global memory as efficiently as possible (avoiding scattered access) and makes heavy use of shared memory to speed up computation.

8. Numerical experiments

In order to prove theoretical discoveries in this paper, a series of experiments have been conducted to compare and contrast the efficiency of the graph-grammar driven parallel GPU solver and a well-known, sequential MUMPS solver over 2D grids with singularities. The numerical tests were performed on a GeForce GTX 260 graphic card with 24 multiprocessors, each equipped with 8 cores. The total number of cores equals 192. The global memory on graphic card was 896 MB. The polynomial approximation level p is set to two. This means we employ h-adaptive Finite Element Method [7, 8]

We present the results in eight graphs (Fig. 31–34).

We compare the performance of the traditional MUMPS solver with the graph-grammar driven linear GPU solver starting from a 2D grid with a single point singularity (see Fig. 31). It can be easily observed that the GPU solver is very predictable in terms of computational costs, which increase logarithmically with the number of degrees of freedom. In the case of MUMPS, it is full of sharp rises and drops, but generally rises in a linear way.

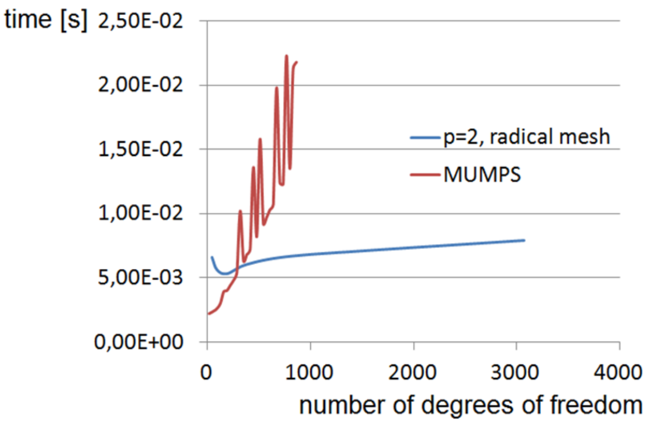

Figure 31. 2D mesh with point singularity.

For $p = 3$ this difference remains visible in favor of the GPU solver (compare Fig. 32).

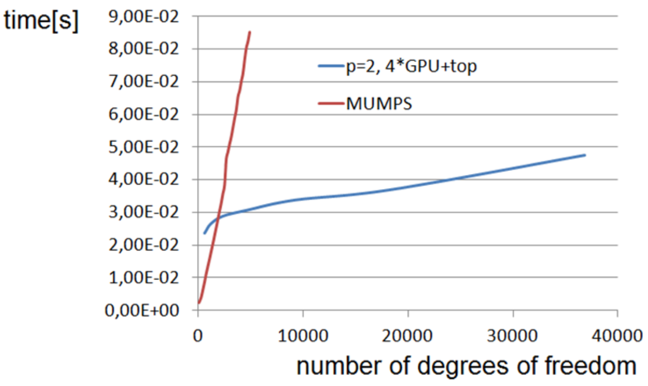

Figure 32. 2D mesh with 2×2 point singularities.

With the increase of the number of singularities, this difference becomes even more obvious. MUMPS computational cost still seems linear, but it rises at much steeper angle. In the case of one GPU available, we can process each of the singularities one by one and then submit the remaining Schur complements for the top problem to the MUMPS solver in order to solve the 2×2 regular grid, with element matrices replaced by Schur complements coming from elimination of the point singularities. This scenario is called "4*GPU+top" for 2×2 singulariites or "16*GPU+top" for 4×4 singularities and "64*GPU+top" for 8×8 singularities. We compare this approach to submitting the entire problem to the MUMPS solver and solving it once (denoted by "MUMPS" solver).

The comparison of the MUMPS solver with GPU based solver is presented in Figures 32–34 for 2×2, 4×4 and 8×8 singularities, respectively.

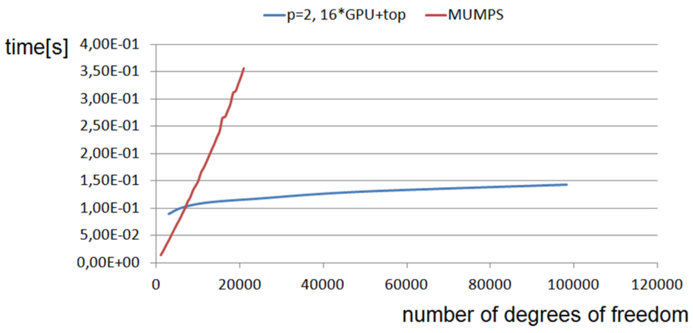

Figure 33. 2D mesh with 4×4 point singularities.

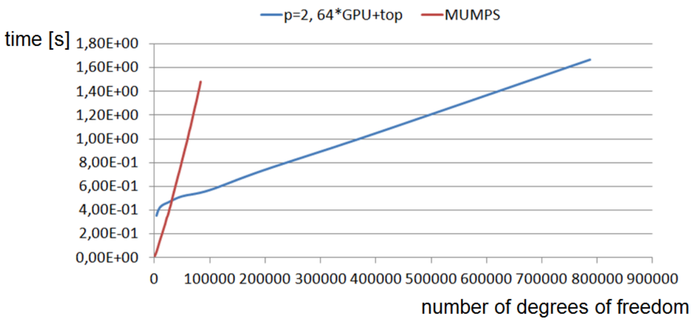

Figure 34. 2D mesh with 8×8 point singularities.

We also present a convergence of our *h* adaptive method for increasing the number of layers, and comparison with the *hp* adaptive case, where we also change the polynomial order of approximation, but the computational cost is only asymptotically linear (linear with respect to the maximum order), see Figure 35.

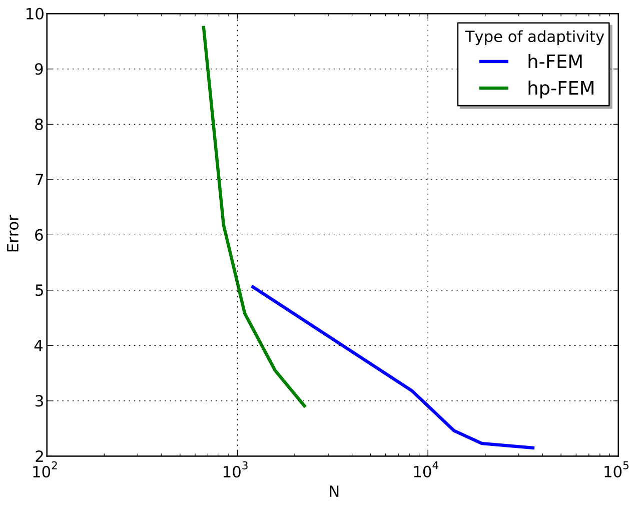

Figure 35. Comparison of *h* and *hp* adaptivity for the heat transfer problem with point singularities.

9. Conclusion

In this paper, we presented the hypergraph grammar based model of multi-frontal solver algorithm for computational grids with point singularities. The expressing of the solver algorithm in terms of hypergraph grammar allows for the identification of sets of graph grammar productions that can be concurrently executed. The graph-grammar-based solver algorithm was implemented in NVIDIA CUDA for two-dimensional problems with point singularities. The numerical results showed that our graph-grammar-based solver with GPU accelerator outperform the sequential CPU MUMPS solver by the order of magnitude.

Acknowledgements

The work presented in this paper is supported by Polish National Science Center grant no. DEC-2012/07/B/ST6/01229 and DEC-2011/03/B/ST6/01393.

References

[1] Albers B., Savidis S., Tasan E., von Estorff O., Gehlken M.: BEM and FEM results of displacements in a poroelastic column. *Journal of Applied Mathematics and Computer Science*, vol. 22(4), pp. 883–896, 2012.

[2] Barboteu M., Bartosz K., Kalita P.: An analytical and numerical approach to a bilateral contact problem with nonmonotone friction. *Journal of Applied Mathematics and Computer Science*, vol. 23(2), pp. 263–276, 2013.

[3] Bazilevs Y., da Veiga L. B., Cottrell J. A., Hughes T. J. R., Sangalli G.: Isogeometric analysis: Approximation, stability and error estimates for h-refined meshes. *Mathematical Methods and Models in Applied Sciences*, vol. 16, pp. 1031–1090, 2006.

[4] Colier N., Pardo D., Dalcin L., Calo V. M.: *The cost of continuity: A study of the performance of isogeometric finite elements using direct solvers. Computer Methods in Applied Mechanics and Engineering*, vol. 213, pp. 353–361, 2012.

[5] Cottrel J. A., Hughes T. J. R., Bazilevs J.: Isogeometric Analysis. Toward Integration of CAD and FEA. Wiley, 2009.

[6] David A., Hager W.: Dynamic supernodes in sparse cholesky update / downdate and triangular solves. *ACM Transactions on Mathematical Software*, vol. 35(4), pp. 1–23, 2009.

[7] Demkowicz L.: *Computing with hp-Adaptive Finite Elements*, *Vol. I. One and Two Dimensional Elliptic and Maxwell Problems.* Chapman and Hall/Crc Applied Mathematics and Nonlinear Science, 2006.

[8] Demkowicz L., Kurtz J., Pardo D., Paszyński M., Rachowicz W., Zdunek A.: Computing with hp-Adaptive Finite Elements, Vol. II. Frontiers: Three Dimensional Elliptic and Maxwell Problems with Applications. Chapman and Hall/Crc Applied Mathematics and Nonlinear Science, 2007.

[9] Duff I. S., Reid J. K.: The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, vol. 9, pp. 302–325, 1983.

[10] Duff I. S., Reid J. K.: The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, vol. 5, pp. 633–641, 1984.

[11] Flasiński M., Schaefer R.: Quasi context sensitive graph grammars as a formal model of FE mesh generation. *Computer-Assisted Mechanics and Engineering Science*, vol. 3, pp. 191–203, 1996.

[12] Grabska E.: Theoretical Concepts of Graphical Modeling. Part Two: CP-Graph Grammars and Languages. *Machine Graphics and Vision*, vol. 2, pp. 149–178, 1993.

[13] Gupta A.: Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, vol. 28, pp. 301–324, 2002.

[14] Gupta A., Gustavson F. G., Toledo J.: The design, implementation, and evaluation of a symmetric banded linear solver for distributed-memory parallel computers. *ACM Transactions on Mathematical Software*, vol. 24(1), pp. 74–101, 1998.

[15] Gupta A., Karypis V., Kumar V.: Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, vol. 8(5), pp. 502–520, 1997.

[16] Habel A., Kreowski H. J.: May we introduce to you: Hyperedge replacement. *Lecture Notes in Computer Science*, vol. 291, pp. 5–26, 1987.

[17] Habel A., Kreowski H. J.: Some structural aspects of hypergraph languages generated by hyperedge replacement. *Lecture Notes in Computer Science*, vol. 247, pp. 207–219, 1987.

[18] Hild P.: *A sign preserving mixed finite element approximation for contact problems. Journal of Applied Mathematics and Computer Science*, vol. 21(3), pp. 487–498, 2011.

[19] Irons B.: A frontal solution program for finite-element analysis. *International Journal of Numerical Methods in Engineering*, vol. 2, pp. 5–32, 1970.

[20] Paszyńska A., Grabska E., Paszyński M.: A Graph Grammar Model of the hp Adaptive Three Dimensional Finite Element Method. Part I. *Fundamenta Informaticae*, vol. 114(2), pp. 149–182, 2012.

[21] Paszynska A., Grabska E., Paszynski M.: A Graph Grammar Model of the hp Adaptive Three Dimensional Finite Element Method. Part II. *Fundamenta Informaticae*, vol. 114(2), pp. 183–201, 2012.

[22] Paszyńska A., Paszyński M., Grabska E.: *Graph Transformations for Modeling hp-Adaptive Finite Element Method with Mixed Triangular and Rectangular Elements. Lecture Notes in Computer Science*, vol. 5545, pp. 875–884, 2009.

[23] Paszyński M.: On the Parallelization of Self-Adaptive hp-Finite Element Methods Part I. Composite Programmable Graph GrammarModel. *Fundamenta Informaticae*, vol. 93(4), pp. 411–434, 2009.

[24] Paszyński M., Pardo D., Calo V.: A direct solver with reutilization of LU factorizations for h-adaptive finite element grids with point singularities. *Computers & Mathematics with Applications*, vol. 65(8), pp. 1140–1151, 2013.

[25] Paszyński M., Schaefer R.: *Graph grammar-driven parallel partial differential equation solver. Concurrency and Computation: Practice and Experience*, vol. 22(9), pp. 1063–1097, 2010.

[26] Rozenberg G.: Handbook of graph grammars and computing by graph transformation, vol I: Foundations. *World Scientific*, vol. 1997.

[27] Schmitz P., Ying L.: A fast direct solver for elliptic problems on general meshes in 2d. *Journal of Computational Physics*, vol. 231, pp. 1314–1338, 2012.

[28] Sieniek M., Gurgul P., Magiera K., Skotniczny P.: Agent-oriented image processing with the hp-adaptive projection-based interpolation operator. *Journal of Computational Science*, vol. 4, pp. 1844–1853, 2011.

[29] Ślusarczyk G., Paszyńska A.: Hypergraph grammars in hp-adaptive finite element method. *Procedia Computer Science*, pp. 1545–1554.

[30] Szymczak A., Paszyńska A., Gurgul P., Paszyński M., Calo V.: Graph Grammar Based Direct Solver for hp-adaptive Finite Element Method with Point Singularities. *Procedia Computer Science*, vol. 18, pp. 1594–1603, 2013.

Affiliations

Piotr Gurgul
AGH University of Science and Technology, Krakow, Poland

Maciej Paszyński
AGH University of Science and Technology, Krakow, Poland

Anna Paszyńska
Jagiellonian University, Krakow, Poland

Received: 26.09.2014
Revised: 02.11.2014
Accepted: 04.11.2014