Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

KATEDRA INFORMATYKI

PRACA DOKTORSKA

PIOTR GURGUL

SOLWERY DO SYMULACJI ADAPTACYJNYCH O LINIOWYM KOSZCIE OBLICZENIOWYM

PROMOTOR:
dr hab. Maciej Paszyński
prof. nadzwyczajny AGH

Kraków 2014

AGH
University of Science and Technology in Krakow

Faculty of Computer Science, Electronics and Telecommunications

DEPARTMENT OF COMPUTER SCIENCE

DISSERTATION FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

PIOTR GURGUL

LINEAR COMPUTATIONAL COST SOLVERS FOR ADAPTIVE SIMULATIONS

SUPERVISOR:
Maciej Paszyński, Ph.D.
Associate Professor

Krakow 2014

Acknowledgements

I would like to express my deepest gratitude to Professor Maciej Paszyński for supporting me in my research efforts throughout all my studies. His guidance, inspiration, patience and attitude have provided an invaluable experience that helped me in my personal development and career.
I also wish to thank Professor Robert Schaefer for his very important and accurate comments that resulted in significant improvements.
I appreciate the support I have received from the European Union by means of European Social Fund, PO KL Priority IV: Higher Education and Research, Activity 4.1: Improvement and Development of Didactic Potential of the University and Increasing Number of Students of the Faculties Crucial for the National Economy Based on Knowledge, Subactivity 4.1.1: Improvement of the Didactic Potential of the AGH University of Science and Technology Human Assets, No. UDA – POKL.04.01.01-00-367/08-00. I also appreciate the financial support from Polish National Science Center grants. no 2011/03/N/ST6/01397, 2012/06/M/ST1/00363, 2011/03/B/ST6/01393, NN 519447739, NN 519405737 from which I received contribution. The research concerning linear solvers for three dimensional meshes with point singularities has been supported by Dean's Grant no. 15/11/230/128.

Dedicated to my wonderful parents and wife, supporting me on all steps of my career.

Streszczenie

Niniejsza praca prezentuje nowy formalizm matematyczny, który za pomocą gramatyk hipergrafowych steruje wykonaniem adaptacyjnej Metody Elementów Skończonych dla siatek z osobliwościami punktowymi, o dowolnym poziomie zagęszczenia siatki w stronę osobliwości. Wydajne techniki rozwiązywania tej klasy problemów stanowią niezmiennie obszar zainteresowania środowiska naukowego. Rozprawa ta pokazuje jak wykorzystać szczególną dla problemów z osobliwościami punktowymi strukturę siatki, celem uzyskania liniowej złożoności obliczeniowej w wersji sekwencyjnej oraz logarytmicznej złożoności obliczeniowej w wersji równoległej. Ponadto, zamodelowanie gramatykami grafowymi solwera wielofrontalnego wspomaga wydajne zrównoleglenie, co skutkuje dodatkowym wzrostem wydajności. Modele hipergrafowe zostały w niniejszej pracy stworzone zarówno dla sekwencyjnej jak i równoległej wersji solwera, w dwu i trzech wymiarach. Dla obu wariantów rozwiązania zostały oszacowane teoretyczne koszty wykonania i zużycie pamięci. Obydwa modele teoretyczne zostały zaimplementowane na CPU i GPU. Implementacja na GPU okazała się wyraźnie lepsza od osiągów wiodących solwerów, takich jak MUMPS. Pomiary wydajności zostały zebrane dla problemów dwu- i trójwymiarowych, z jedną lub wieloma osobliwościami punktowymi oraz dla kilku różnych stopni aproksymacji wielomianowej. Rezultaty te zostały zestawione z oszacowaniami teoretycznymi.

Abstract

In this thesis I present a new hypergraph grammar based formalism prescribing execution of the adaptive Finite Element Method solver for grids with point singularities, not restricted by the number of refinement levels. Efficient approaches to handling such problems are of a great interest and importance to the scientific community. This thesis presents how to utilize a special, adapted structure of the computational domain for problems with point singularities in order to achieve linear computational cost in serial and logarithmic computational cost in parallel. Moreover, graph grammar modeling of the multi-frontal direct solver also helps with efficient parallelisation. The hypergraph models have been created for both sequential and parallel versions of the algorithm, in two and three dimensions. The thesis contains theoretical estimates of the computational cost and memory usage for both versions of the newly developed solver. The theoretical models presented in this thesis have been transformed into a working CPU and GPU implementations. The GPU implementation has been proven to outperform current state-of-the-art solvers, like the Multifrontal Massively Parallel sparse direct Solver. The efficiency results have been collected for two and three dimensional problems with various numbers of point singularities and various polynomial approximation levels. The resuls have been compared with the theoretical estimates.

Table of contents

Chapter 1

Introduction

Many challenging and important engineering problems consist in obtaining an approximate solution to a Partial Differential Equation. Solving such an equation is a task that usually requires significant computational power, as the resulting problems often generate millions of unknows. As a consequence, efficient, low complexity algorithms and techniques are at least as important as powerful hardware. Using methods that deliver exponential error decrease rates such as hp adaptive Finite Element Method helps to dramatically reduce the number of unknows. Another important optimization is the speeding up of solving the system of linear equations resulting from the discretization. Some of these problems are already well-described and addressed, but many of them still remain open. This thesis presents how to deliver a direct solver an order of magnitude faster than the existing solutions, which can be applied to a wide range of problems with point singularities. Moreover, this work shows how leveraging emerging architectures such as modern graphic processors can also account for increasing efficiency of scientific computations.

1.1. Motivation

The key motivation of this work is to present the first hypergraph grammar based mathematical formalism driving execution of the adaptive Finite Element Method solver for grids with point singularities, not restricted by the number of refinement levels. The main objective is to deliver a graph grammar based version of a multi-frontal direct solver for grids with point singularities, which allows for exploiting the parallelism hidden within the solver algorithm, and enabling the most efficient parallel implementation, outperforming current state-of-the-art solvers, like MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [2, 35].

Many important engineering problems require high precision solutions that would produce enormous grids if solved using non-adaptive methods. This is why, in this work, I focus only on the adaptive techniques such as the $h-$, $p-$ and $hp-$adaptive Finite Element Method, which perform much better for irregular problems, such as those with singularities. The Finite Element Method, as well as some other well-established industrial and scientific numerical techniques, have the following items as their input:

- The computational mesh that contains finite elements, describing the domain of the problem being solved.
- Basis functions spread over the mesh, used for the approximation of the numerical solution.
- Partial differential equations describing the physical phenomena to be simulated.

The Partial Differential Equation is translated into its weak form which allows transformation the initial problem into a system of linear equations by computing the weak form integrals with basis functions over all finite elements.

A more detailed tutorial to the Finite Element Method in one and two dimensions can be found in Appendices A and B. The resulting system of linear equations is solved by using a direct solver algorithm.

Classical state-of-the-art solvers, like the MUMPS solver [34, 35], receive the global matrix as their input, construct the connectivity graph based on the sparsity of the matrix, and build the elimination tree by running graph partitioning algorithms like nested dissections from the METIS library [66, 67]. Finally, they leverage parallelism by traversing the elimination tree concurrently. In this work, I present an alternative approach that allows for expressing the direct solver algorithm by the means of a graph grammar, based on the hypergraph representation of the computational mesh. The solver algorithm can be expressed in the language of basic indivisable tasks called graph grammar productions. Such notation allows us to analyse the partial order of execution between tasks. This, in turn, helps us to recognize sets of tasks that can be executed in parallel, as well as to deliver efficient parallel implementation, outperforming current state-of-the-art solvers. The graph grammar productions that can enforce a special processing order enable us to take advantage of the hierarchical structure of the mesh. The hierarchical structure of the mesh in the neighborhood of point singularities results from local mesh refinements that are intended to increase the accuracy of the numerical solution. As a rule, these refinements follow local singularities of the numerical solution. The point singularities are very common in numerical simulations. They may result from:

- Local geometrical structure of the mesh (like the internal corner in the L-shape domain model problem - see Appendix B).
- Point sources on the right hand side of the PDE (like point heat sources in the heat transfer problem, or local point antennas in electromagnetic waves simulation problems).
- Non-uniform distribution of material data (point singularities are present when three different material data meet at a given point of the domain).

To summarize, the class of graph grammar driven solvers introduced in this work is suitable for a wide array of computational problems with point singularities. These problems are common in engineering computations, and are of great interest to the scientific community, since they are computationally demanding by their very nature. Therefore, I only consider problems with point singularities, since regular grids can be processed effectively using the existing, well-established methods. Graph grammar formalism is especially important in the case of parallel version of the solver, where graph grammars allow identification of the smallest atomic tasks that can be processed independently. Graph grammar productions enforce the correct execution order of various tasks, since their left hand sides determine what is the necessary and sufficient condition to substitute it with the right hand side of any given production. In particular, the graph grammar based approach allows us to achieve linear computational cost when processing the grid with many singularities in serial, as well as logarithmic computational cost when processing a grid with many singularities in parallel. As a part of this work, a sample CPU and GPU implementations of the hypergraph solver have been developed to challenge the current state-of-the-art solvers and prove the theoretical estimates of the computational cost. The use of emerging architectures, such as GPU for numerical computations is still relatively new to the scientific community. However, it seems a very promising directions as speeding up computations in the traditional manner seems to be reaching its limits. Increasing the number of transistors generates huge problems with excessive heat and, a matter of concern these days, power consumption. Turning to many smaller, more efficient processors can significantly reduce these costs, due to less complex control hardware. In other words, in case of GPUs more transistors can be used directly for computation. This of course comes at the cost of a more restrictive programming model, which in case of GPUs is explicitly parallel.

1.2. Purpose of this book

Section 1.1 explained that there is an open field for formal models prescribing execution of complex algorithms, especially for emerging architectures. The main purpose of this book is to deliver a solution which can successfully utilize this underlying potential and outperform the existing solutions, at least for some classes of problem. The findings presented in this thesis are applicable to problems that contain one or more point singularities triggering a series of mesh refinements toward them. This is because the algorithm presented in this thesis leverages information about the structure of the domain to optimize the order of computations. The graph grammar based solvers presented in this thesis can be easily adapted to accomodate a wider class of two and three dimensional computational problems with point singularities.

1.3. Main thesis

The main thesis of this work may be summarized as follows:

There is a hypergraph grammar based formalism expressing the multi-frontal direct solver algorithm for two and three dimensional computational problems with point singularities in the form of basic tasks called graph grammar productions, allowing for analysis of the order of execution of the tasks resulting in linear computational cost of the sequential execution, as well as allowing for exploitation of the concurrency hidden within the algorithm resulting in logarithmic computational cost of the parallel execution.

1.4. Structure of this book

This work is divided into six chapters and three appendices.
Chapter 2 contains a brief introduction to the Finite Element Method (Section 2.1.1), with two more detailed examples of the FEM problems in Appendix A (one-dimensional case) and Appendix B (two-dimensional case). This is followed by an outline of the existing mesh refinement techniques (Section 2.1.2). In Section 2.2.1, the major focus is on direct solvers - primarily those for sparse matrices such as the frontal and multi-frontal ones (with Schur complements described in detail in Appendix C). Section 2.2.2 explains the relationship between hypergraph grammars and FEM solvers, outlining the existing attempts to model solvers by the means of graph grammar productions. Chapter 2 is concluded by listing open problems (Section 2.3) and major scientific contributions of this thesis (Section 2.4).
Chapter 3 comprises the four major theoretical parts of this thesis. The first is a detailed introduction to the theory of hypergraphs and hypergraph grammar productions (Section 3.1). This is followed by the application of this theory to generating the initial mesh in two dimensions and prescribing a solver, of which the sequential version works in linear time and the parallel version in logarithmic time with respect to the number of unknowns (Section 3.2). Section 3.3 contains theoretical estimates of the exact computational cost and memory usage for the sequential and parallel versions of the proposed solver. Moreover, the linear computational complexity is proven for the sequential solver and the logarithmic complexity for the parallel solver.
Chapter 4 briefly shows how to extend hypergraph grammar reasoning to a three dimensional model problem, as well as how to generalize the theoretical results concerning the linear computational cost and memory usage in three dimensions.
Chapter 5 contains the experimental part of this thesis. Section 5.1 defines a sample numerical problem with point singularities that was solved to measure solver's performance. Section 5.2 describes computations and performance

evaluation for traditional CPU architectures. Section 5.3 defines the GPU environment used to obtain the results and technical details of transformation of the theoretical solver into a working CUDA-enabled program. The chapter is concluded with Section 5.4 which presents the results of the series of experiments conducted to check the efficiency of the parallel GPU solver. In both cases the efficiency is compared with the state-of-the-art MUMPS solver. Chapter 6 contains the evaluation of all the obtained results and outlines the potential for future research.

Chapter 2

Background

This chapter contains an overview of existing literature concerning various versions of the adaptive Finite Element Method, which constitute the motivation for high-efficiency direct solvers. A brief introduction to dense algebra solvers is presented in Section 2.2.1. The state-of-the-art direct solvers for sparse matrices such as the frontal solver and the multi-frontal solver are described in Section 2.2.1. The chapter is concluded with a list of selected open problems existing in the field of direct solvers for sparse matrices, accompanied by a list of research results obtained by the author while adressing some of the open problems.

2.1. Essentials

In this section I present essential background information concerning well-established solutions in the realm of adaptive simulations. This will be used as a reference point for the research described in this thesis. I briefly introduce the idea behind the Finite Element Method and its adaptive versions, comparing it to the Finite Difference Method. I outline the most popular methods of reducing the error rate of the solution such as h, p, r and hp adaptivity, as well as some extensions to the Finite Element Method for better integration with CAD systems.

2.1.1. Finite Element Method

The Finite Element Method (FEM) is a well-established, mainstream numerical technique used for finding approximate solutions to Boundary Value Problems (BVP - see Polyanin [92]). It is well-suited to a wide class of Partial Differential Equations (PDEs). FEM was first introduced in late 1960s by Zienkiewicz [108], based on Galerkin method that is used for discretizing a continuous operator problem. This is done by converting the equation to a weak formulation.

Modern and more sophisticated adaptive versions of FEM were proposed by Demkowicz et al. [23, 27]. There have also been numerous attempts to make Finite Element Analysis (FEA) more compatible with modern CAD systems [21]. Compared to the Finite Difference Method (FDM) [40, 103], FEM is able to handle complex geometries and non-rectangular meshes, which makes mapping to the parent domain easier. FDM is often more suitable for Computational Fluid Dynamic (CFD) problems (with many successful FEM attempts [32, 63]) or in the domain of time, but FEM is usually the only reasonable choice for solid mechanics. In addition, using FEM makes it much easier to incorporate boundary conditions. Conceptually, FEM is based on variational calculus and FDM is based on differential calculus.

The finite element solution process starts with the generation of a mesh describing the geometry of the computational problem. Next, the physical phenomena governing the problem are described using PDEs. In addition

to this differential system, boundary and initial conditions may need to be specified to ensure uniqueness of the solution. The resulting PDE system is discretized into a system of linear equations by using FEM. This algebraic system is inverted using a solver algorithm and a solution with a certain degree of accuracy is obtained. A step-by-step tutorial to the 1D Finite Element Method has been moved to Appendix A. In the 2D case, described in detail in Appendix B, the approach is similar.

A very common example of a 2D FEM problem is the so called *L-shape domain model problem* that was proposed by Babuška [6, 7, 47, 69] to test the convergence of hp-FEM. This example was used in Appendix B.

Isogeometric Finite Element Analysis

Isogeometric analysis is a recently developed concept for integrating FEM into CAD design tools, which are primarily based on Non-Uniform Rational B-Splines (NURBS) [62, 91, 106]. This removes the need for conversion between these two formats, which is a computationally intense task. Such seamless integration between FEM and NURBS is called *NURBS-Enhanced FEM (NEFEM)* [97]. The main difference between the classical and isogeometric FEM is that classical FEM delivers solutions that are C^0 globally (between particular finite elements), whereas the isogeometric FEM delivers globally C^{k-1} regular solutions for B-splines of order k. However, the price to pay is that the computational cost of direct solver algorithm is higher by k^3 for isogeometric FEM than for classical FEM [20]. The isogeometric FEM generates a sparse system of equations that can be solved by multi-frontal direct solver algorithms (see Section 2.2.1).

2.1.2. Adaptive algorithms

Adaptive versions of FEM have a multitude of applications, ranging from weather forecasting (Staniforth et al. [101]), through fluid mechanics (Oden et al. [78]) to MRI scan processing (Gurgul et al. [53]). More specifically, the adaptive FEM is also widely used in geophysical simulations [26, 39, 80, 81, 82, 83, 84, 109] to estimate the electrical or acoustic properties of geophysical formations in the ground. It is customary to record resistivity measurements using logging instruments that move along a borehole. These instruments are equipped with several transmitter tools, whose emitted signals are recorded by the receiver tools that are also located along the logging tool. Thus, logging instruments are intended to estimate properties (electrical or acoustic) of the sub-surface material. The ultimate goal is to identify and characterize hydrocarbon (oil and gas) bearing formations in the ground. The Adaptive FEM algorithms described in this section aim to automatically refine or reorder mesh in order to achieve a solution having a specified accuracy in an optimal fashion. However, optimal strategies for deciding where and how to adjust the mesh to reduce error rate are virtually non-existent. This is why the quality of each refinement is usually evaluated using *a posteriori* methods [23], e.g. by measuring relative error decrease rate for each element. The modern, adaptive algorithms for the Finite Element Method have been developed by the research team of Demkowicz [23, 27, 28, 29]. The specific techniques for refining the solutions are described in the subsections below.

h refinement

h refinement is by far the most popular technique for reducing error rate [11, 15]. Its approach to improving the quality of the solution is to increase the number of elements by dividing the domain into smaller pieces. Some sensitive places require a lot of elements to approximate the solution fairly, whereas for others, even a relatively sparse mesh is acceptable. The key factor in achieving satisfactory results is to find places, which demand a finer domain for acceptable relative error rates. It can be done manually by predicting solution features *a priori* or automatically by refining some elements dynamically and evaluating their error rate. In the latter case, the mesh is recursively subdivided until an acceptable resolution is obtained.

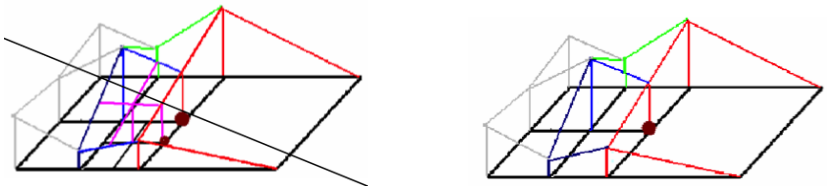
There are two types of h refinement that depend on the strategy.

- Uniform h refinement, where all finite elements over the domain are uniformly and evenly broken into smaller ones.
- Non-uniform h refinement, where only finite elements in certain regions of the domain are broken into smaller ones. These regions can produce higher numerical error than a given threshold.

However, non-uniform h refinement does not mean full freedom in breaking the elements due to so called *1-irregularity rule* described in definition 2.1.1.

Definition 2.1.1. ***1-irregularity rule***. *A finite element can be broken only once without breaking its bigger, neighboring elements. It prevents unbroken element edges from being adjacent to more than two finite elements. When an unbroken edge is adjacent to one large finite element on one side and two smaller finite elements on the other side, the approximation over these two smaller elements is constrained by the approximation of the larger element.*

The 1-irregularity rule helps to prevent overly technically complicated situations with multiple constrained edges (see Figure 2.1). In order to enforce the 1-irregularity rule, several additional refinements on large adjacent elements may be required. As a result, the sequence of refinements presented is Figure 2.2 is incorrect, as the left



(a) Mesh violating 1-irregularity rule (b) Mesh obeying 1-irregularity rule

Figure 2.1: Visualization of the 1-irregularity rule

element ($h = 1$) cannot be the neighbor of the small elements in the bottom part ($h = 2$). If such a refinement is needed, it is necessary to break the left element into four so that the difference of h refinement levels is not greater than one (as in Figure 2.3).

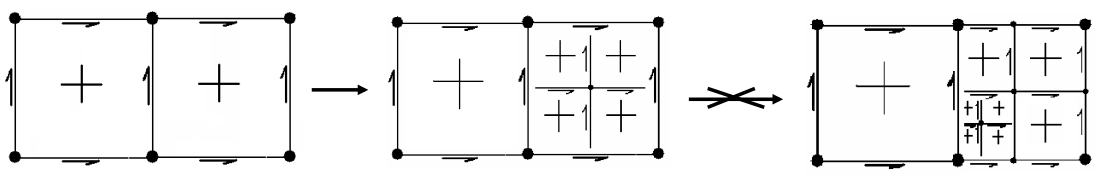


Figure 2.2: Incorrect sequence of mesh refinements

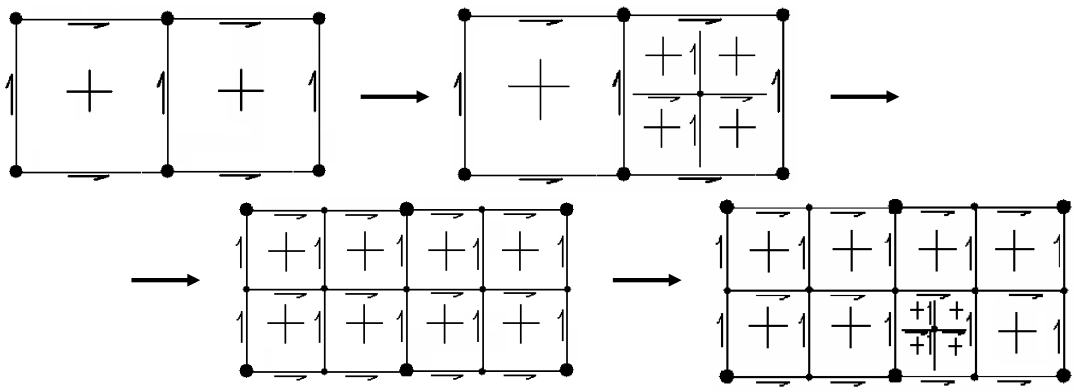


Figure 2.3: Correct sequence of mesh refinements

p refinement

p refinement allows us to locally vary the maximum polynomial degree of the basis functions for each element. This means that the order of the basis functions can be variable across the mesh. p refinement alone is capable of achieving exponential convergence rates, but only for smooth problems. However, even for problems with singularities, p version of FEM can converge twice as fast as the classical, h version [8]. The use of p refinement is most natural on a hierarchical basis, since substantial portions of the stiffness and mass matrices, as well as the load vector will remain unchanged when increasing the polynomial degree of the basis. There are two types of p refinement:

- Uniform p refinement, where the polynomial approximation level is increased for every element in the mesh.
- Non-uniform p refinement, where the polynomial approximation level is increased only for certain elements in the mesh.

In case of p refinements, there is the so-called *minimum rule* to obey.

Definition 2.1.2. ***Minimum rule***. *The polynomial order of approximation on an element edge must be equal to the minimum of corresponding orders of approximation from the element interiors.*

For an exemplary application of the minimum rule see Figure 2.4, which shows that the polynomial approximation level p on the edge between two elements presented is equal to minimum of p on both vertical edges p_{v_1} and p_{v_2}. For more information on p refinement see [8, 46].

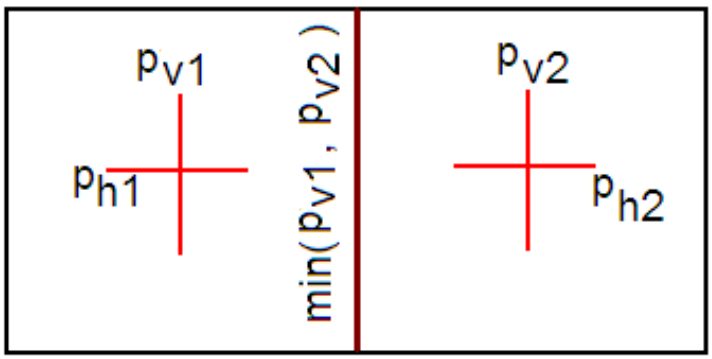


Figure 2.4: Visualization of the minimum rule

hp adaptation

hp adaptation is a combination of automated h and p refinements, which is by far the most powerful and universal technique. The reason for this is that unlike p refinements, it is guaranteed to deliver exponential convergence rates even for the problems with singularities. In case of non-uniform hp adaptation, which is the usual case, it is essential to find a way of localizing regions of the domain to adapt. The original, self-adapted mesh was introduced by Demkowicz [23] in 2006 and consists of the following steps:

1. Generate an initial mesh and call it a *coarse mesh*.
2. Solve the coarse mesh problem by using one of the FEM direct solvers, in order to get the coarse mesh solution u_{hp}.
3. Generate the fine mesh based on the coarse mesh. Having the coarse mesh u_{hp}, generate the fine mesh, where each element is broken into four and the polynomial approximation level of each element is increased by one.
4. Solve the fine mesh problem by using the direct solver in order to get the fine mesh solution $u_{\frac{h}{2}p+1}$.
5. For each finite element decide if the refinement was necessary or not. This is usually done based on the estimation of the relative error decrease rate (see Equation 2.1) between the solution of a given element on a fine mesh and on the coarse mesh.

$$error_{rel} = \|u_{hp} - u_{\frac{h}{2}p+1}\|_{1,\Omega} \tag{2.1}$$

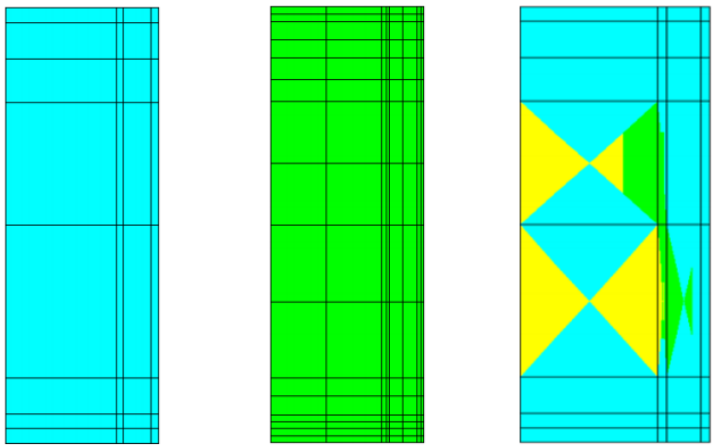
A commonly used norm to compute this difference is $H^1(\Omega)$ Sobolev space norm (Equation 2.2).

$$\|\nu\|_{1,\Omega} = \int_{\Omega} \sqrt{|\nabla\nu|^2 + |\nu|^2} d\mathbf{x} \tag{2.2}$$

If the relative error rate is greater than the arbitrary threshold, the refinement is kept, otherwise it is discarded. A mesh resulting from such an operation is called an *optimal mesh*.

6. If the maximum relative error rate is greater than the required accuracy, go back to step 2 and turn the current mesh into a coarse mesh for step 2. Otherwise, output the solution.

An example of three types of meshes presented in the algorithm above is depicted in Figure 2.5. Colors indicate different polynomial approximation levels.



(a) (b) (c)

Figure 2.5: Coarse mesh (a), fine mesh (b) and optimal mesh (c)

The coarse mesh contains all the elements unrefined and having the initial polynomial approximation level. In the fine mesh all elements have been refined and polynomial approximation level has been increased by one. In the optimal mesh only valuable refinements and polynomial approximation level increases have been kept.

r refinement

Unlike in the previous three types of refinements, in case of r refinement, the number of mesh nodes and elements remains constant, but nodes are relocated to areas where accuracy needs to be increased. r refinement is most useful with transient problems, where elements move to follow an evolving phenomena. In such cases, it would be an unnecessary expense to refine a mesh in one area, as the phenomena may become more computationally intense in a different region. An overview of existing r refinement algorithms is given by McRae [76]. This thesis does not cover this kind of problems.

Adaptation with T-splines

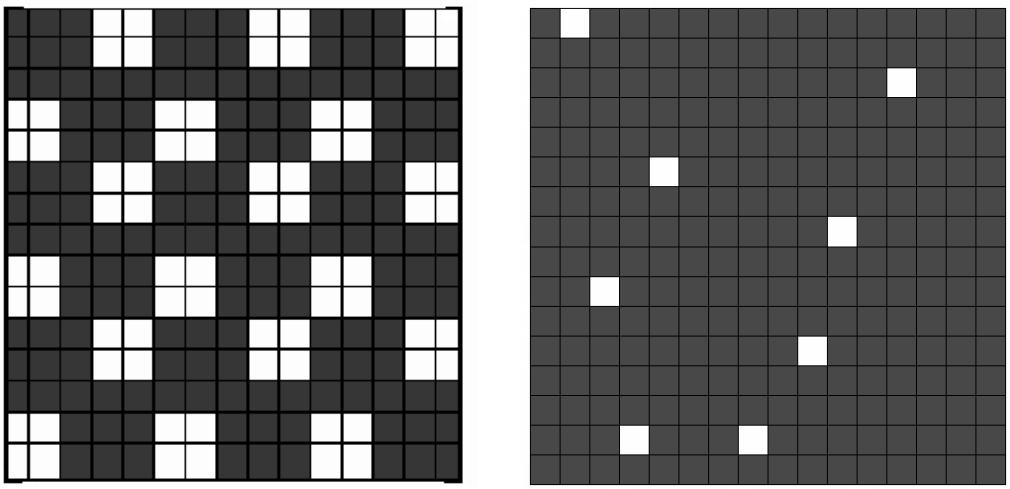
Another important disadvantage of the isogeometric FEM proposed in [21] is the fact that the method supports only uniform p refinements. In other words, there is no way of breaking the elements locally or increasing polynomial orders of approximation. In order to work around this undesirable feature, the T-spline technique has recently been proposed by Bazilevs [12, 13, 37]. It is the mixture of the isogeometric FEM with a non-uniform h refinement technique [95, 96]. Unlike the global refinement, T-splines allow partial rows of control points that terminate in a special control point called a T-junction, which results in local refinement [72].

2.2. State-of-the-art FEM solvers

In this section I present well-established state-of-the-art solvers suitable for the Finite Element Method. This is followed by listing some of the open problems concerning FEM solvers, as well as solutions to them that constitute the scientific value of this thesis. I also discuss their shortcomings which justifies the existence of the solver presented in this thesis.

2.2.1. Direct solvers

Direct solvers are the core, and most computationally expensive part of many engineering analyses performed using the Finite Element Method. First, it must be stated that the class of iterative solvers is generally inapplicable to the FEM problems, since they are numerically unstable [79]. There are iterative solvers that converge for a certain class of variational formulations, but there are no iterative solvers *per se* that converge for every FEM problem (as opposed to direct solvers). Moreover, even the multi-grid iterative solvers [1, 9, 10], more suitable for processing adaptive grids, still utilize a direct solver over the coarser grid. This makes the use of direct solvers absolutely necessary for designing a general purpose FEM solver. This is why, in this section I focus solely on the direct solvers and primarily on the ones optimized for sparse matrices. The visual comparison between a dense and a sparse matrix is depicted in Figure 2.6.



(a) sparse matrix (b) dense matrix

Figure 2.6: An example of a sparse matrix (a) and a dense matrix (b). Non-zero entries are marked in black, zero entries are marked in white.

Dense algebra solvers

As a general rule, matrices resulting from FEM or FDM are rather sparse (see Figure 2.6). For the dense ones there are a number of well-established solutions. Among them, the two most popular algorithms are Gaussian elimination and LU factorization. The ready-to-use packages include LAPACK (by Dongarra and van den Geijn [33]) and its parallel version - PLAPACK (by van de Geijn [105]), as well as cutting-edge libraries which are able to automatically optimize for a given architecture of a sequential or parallel machine such as ELEMENTAL [93] or FLAME by van de Geijn [64]. In recent years, performing numerical computations on emerging architectures has also become increasingly common. Therefore, libraries such as PLASMA (Parallel Linear Algebra Software for Multicore Architectures) and MAGMA (Matrix Algebra on GPU and Multicore Architectures [107]) have been developed for optimized performance on such architectures. Unlike LAPACK, PLASMA uses tiles algorithms that enable fine grained architecture.

For sparse matrices though, the approach is totally different and expressed by frontal [65] and multi-frontal [34, 35] solvers, which are described in detail in the following sections.

2D frontal solver algorithm

In order to convey the idea of a 2D frontal solver algorithms, a simple 2D two finite element example will be used. The domain Ω is described by two elements and fifteen nodes - two interiors, seven edges and six vertices (compare Figure 2.7). In the 2D FEM method, as it is described in Appendix B, we utilize basis functions related to element nodes. In this example, as presented in Figure 2.8, we have linear basis functions related to element vertices, namely to nodes 1, 3, 5, 11, 13 and 15, quadratic basis functions related to element edges, namely to nodes 2, 4, 6, 8, 10, 12 and 14, as well as quadratic basis functions related to element interiors, namely to nodes 7 and 9. In the 2D FEM, as described in Appendix B, we construct the matrix by integrating multiplications of these basis functions or their derivatives over a domain. Thus, matrix rows and columns corresponds to basis functions and matrix entries corresponds to multiplications of pairs of basis functions. Interior basis functions have support over a given element only, edge basis functions have support spread over one or two elements, vertex basis functions also have support spread over one or many elements.

The frontal solver introduced by Irons et al. [65] browses finite elements in a user-determined, arbitrary order. Due to its nature, it is sequential. Nodes (degrees of freedom) are aggregated into so called *frontal matrices*.

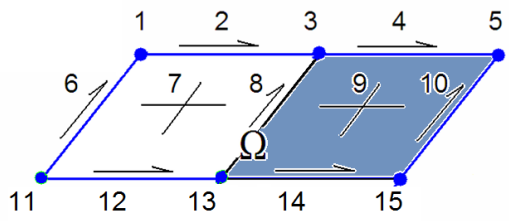


Figure 2.7: Sample computational domain for the frontal solver

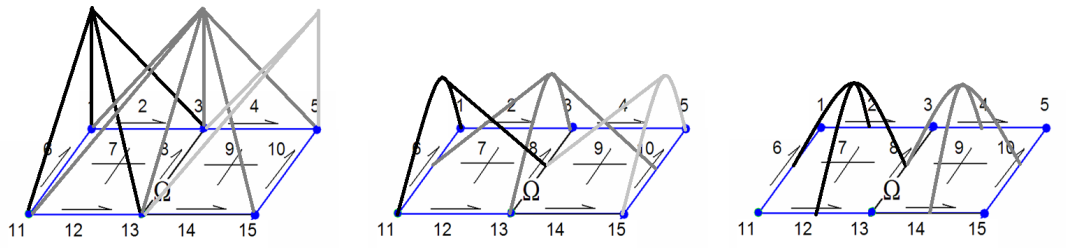


Figure 2.8: Exemplary basis functions spread over element nodes: (a) basis function associated to vertex node 1 (black) vertex node 3 (dark gray) and vertex node 5 (light gray) (b) basis function associated to edge node 6 (black) edge node 8 (dark gray) edge node 10 (light gray) (c) basis function associated to interior node 7 (black) and interior node 9 (dark gray).

Instead of generating just one large finite element method matrix, it generates small matrices, called element frontal matrices. These matrices are obtained by integrating basis functions over a given element. Thus, some entries in the element frontal matrix are fully assembled, and some are not. The row of the frontal matrix is called *fully assembled*, if all of its entries (integrals of products of pairs of basis functions) have been fully computed. This happens if both basis functions have support over a given element only, or the first of the basis functions has support over a given element only (since even the second basis functions also has support over some other element, its product with the first basis function is also zero). The fully assembled rows are self-contained and can be eliminated by the solver algorithm at any time. If a basis function has support defined only over a single element, we say that the node is reduced to the element only, and it has all its contributions already present in the element frontal matrix.

The aim of the frontal solver is to keep the frontal matrix as small as possible. In order to do so, it analyses the connectivity of the nodes and performs partial forward elimination of the fully assembled nodes. Fully assembled nodes have all of their contributions already present in the matrix, so no additional knowledge is necessary to eliminate corresponding rows. Its mechanisms are presented using an example of a two-element mesh. Since the order is arbitrary, we decided to add the right element's nodes to the matrix first. Nodes 4, 5, 9, 10, 14 and 15 are proprietary to the right element and thus, we can call them *fully assembled*. Nodes 3, 8 and 13 are shared with the left element, and hence, will not be fully assembled until the frontal matrix associated with the left element is not merged with the frontal matrix associated with the right element.

The frontal solver browses elements one by one. It starts with generating for the right element, for all the nodes (4, 5, 9, 10, 14, 15, 3, 8, 13). We put the fully assembled nodes (4, 5, 9, 10, 14, 15) in the upper part of the matrix. We perform partial forward elimination, eliminating all fully assembled nodes (4, 5, 9, 10, 14, 15). This is shown in Figure 2.9. The upper triangular part of the frontal matrix has to be stored for future backward substitution. What is left is the reduced frontal matrix associated with nodes (3, 8, 13), which are not yet fully assembled. Such a reduced matrix is called the Schur complement matrix (see Appendix C). The solver now moves to the left

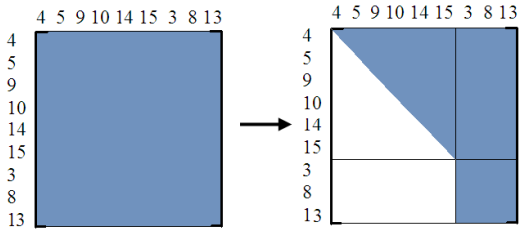


Figure 2.9: Processing of the right element by the frontal solver

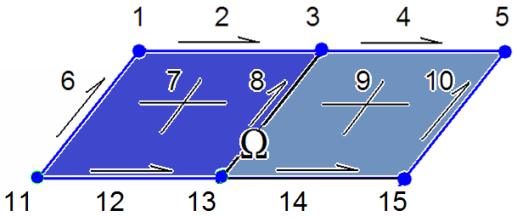


Figure 2.10: Processing of the left element by the frontal solver

element (Figure 2.10) and generates its frontal matrix (3, 8, 13, 1, 2, 6, 7, 11, 12), which is followed by adding the contribution that remained after processing the right element. This is illustrated in Figure 2.11. Now, all nodes are fully assembled in a single matrix, so it is possible to perform full forward elimination. The process is followed by

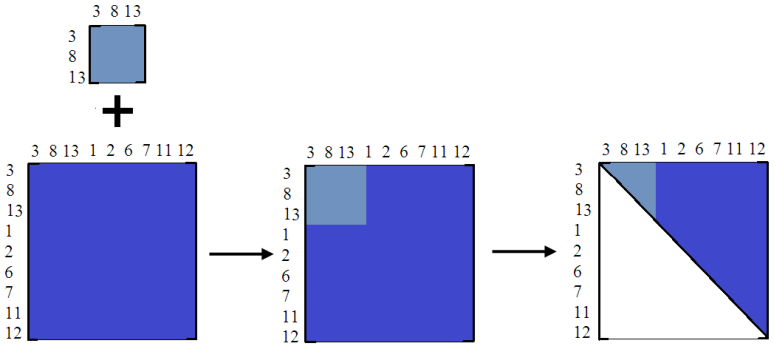


Figure 2.11: The upper triangular form of the frontal matrix after processing the second element

backward substitution, browsing elements in reverse order. The solver takes advantage of the upper triangular form of the frontal matrix and computes the solution at each node, one by one. Such an approach allows us to keep the size of the matrix as small as possible by eliminating unknowns as soon as possible. Unfortunately, as mentioned before, in this case only the matrix operations can be parallelized due to the nature of the algorithm.

2D multi-frontal sequential solver algorithm

The multi-frontal solver introduced by Duff and Reid [34, 35] is the state-of-the-art direct solver algorithm for solving systems of linear equations, which is a generalization of the frontal solver algorithm [65]. However, in case of a multi-frontal solver, connectivity analysis is performed using a so-called *elimination tree*. A computational domain is decomposed into hierarchical subdomains, which account for the elimination tree (Figure 2.12). The construction of the elimination tree for an arbitrary mesh is a complex task itself. It is done by constructing the graph representing the connectivities in the mesh, which is followed by running graph partitioning algorithms such as *nested dissections* from METIS library [66, 67, 68]. Usually, commercial solvers like the MUMPS [34, 35] solver are not aware of the structure of the mesh, and they need to reconstruct the connectivity pattern by analysing

the sparsity pattern of the matrix submitted to the solver. Note that such a matrix is already in its the global form, after the assembly of all element frontal matrices. In the multi-frontal approach, the solver generates a frontal

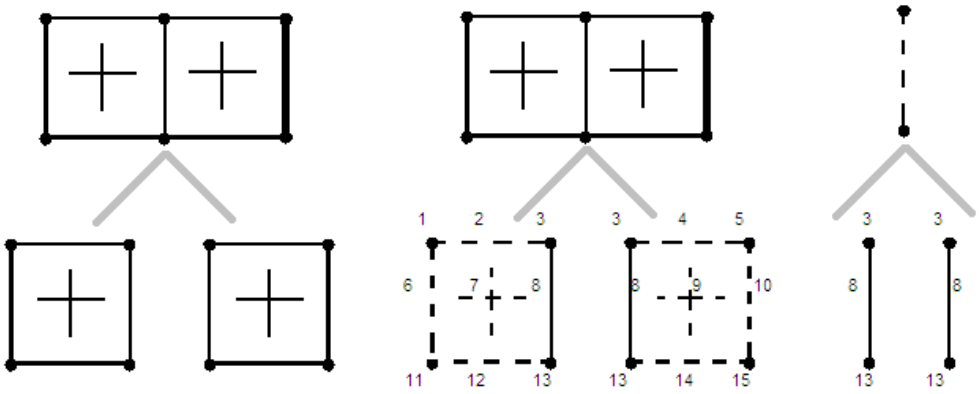


Figure 2.12: Domain decomposed into an elimination tree

matrix for each element of the mesh. This is illustrated in Figures 2.13 and 2.14. It eliminates fully assembled nodes within each frontal matrix, and merges the resulting Schur complement matrices at the parent level of the tree. This is illustrated in Figure 2.15. The key difference with respect to the frontal matrix is that at the parent level the solver works with a smaller matrix, which is a 3×3 matrix obtained from the two Schur complements computed at its son nodes. In other words, the frontal matrix assembles element frontal matrices to a single frontal matrix and eliminates what is possible from the single matrix, while the multi-frontal solver utilizes multiple frontal matrices and thus allows us to reduce the size of the matrices at parent nodes of the tree.

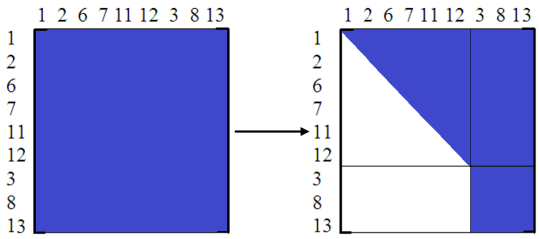


Figure 2.13: Partial forward elimination on the left element

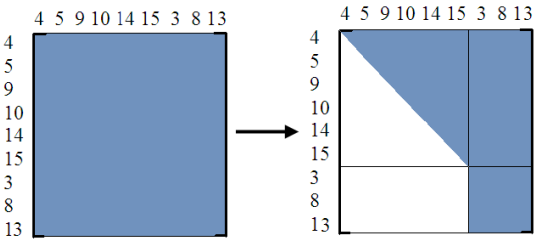


Figure 2.14: Partial forward elimination on the right element

Parallel shared memory implementations

Unlike the frontal solver, the tree structure of the multi-frontal solver makes it very natural to parallelize by executing each branch separately. According to our example, eliminations from Figures 2.13 and 2.14 can be performed simultaneously. Since the memory is shared, there is no additional communication overhead. The cost is determined by length of the longest branch of the tree.

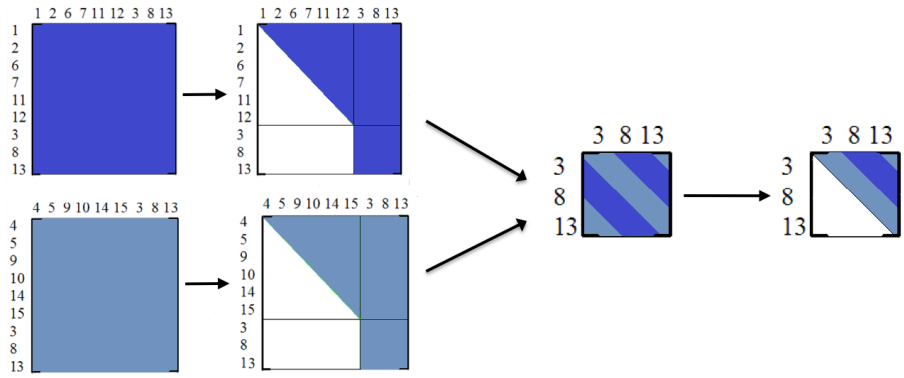


Figure 2.15: Full forward elimination of the interface problem matrix

Parallel distributed memory implementations

In the case of distributed memory, each branch can be executed in parallel, but synchronization in the parent of each branch requires us to send the frontal matrices to another nodes of the distributed parallel machine. In other words, the Schur complement matrix related to nodes 3,8,13 from Figure 2.15 is now stored in distributed manner, and one its contribution needs to be sent from one processor to the other, where it can be merged in the provessor local memory for further processing. Also, the solution obtained in the processor local memory for these top nodes needs to be sent back to the other processor. This scheme of course can be generalized into more levels of the elimination tree.

Extensions of the multi-frontal solver

The multi-frontal 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. In our approach, as we work with computational mesh, the nodes follow naturally from the usage of higher order polynomials over element edges, faces and interiors. In the traditional approach, the nodes must be identified by looking at the sparsity pattern of the global matrix [22]. This allows for a reduction of the computational cost related to the construction of the elimination tree. There are also highly optimized implementations of the multi-frontal solver algorithm for different architectures [48, 49, 50]. Other significant advances in the area of multi-frontal solvers include the design of a hybrid solver, namely the mixture of direct and iterative solvers, where the elimination tree is cut at some level, and the remaining Schur complements are submitted to an iterative solver. There is also a linear computational cost direct solver based on the use of H-matrices [94] having their non-diagonal blocks compressed. However, the main limitation of the H-matrix solver [58, 59, 60] is that it delivers only an approximated solution [73].

Computational costs of direct solvers

A multi-frontal solver is usually faster than a frontal solver. The computational costs for a multi-frontal solver for regular grids with uniform polynomial order of approximation has been recently calculated in [20]. First, we focus on estimating the computational cost for the presented examples. Then, we can think of a more general formula as a function of the degrees of freedom. In order to compute the number of operations implied by different numbering and approaches we define variable a as the number of degrees of freedom that can be eliminated for matrix M and b as the total count of degrees of freedom for matrix M (see Figure 2.16). For a given node, the total number of operations needed for its elimination $C(a,b)$ is equal to:

$$C(a,b) = \sum_{m=b-a+1}^{b} m^2 \tag{2.3}$$

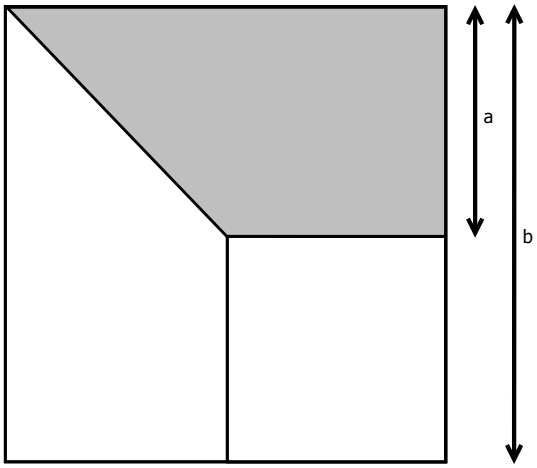


Figure 2.16: Visual explanation of a and b

This is because we have a matrix with size b and there are a rows to be eliminated. The elimination of the first row involves b^2 subtractions, the elimination of the second row involves $(b-1)^2$ subtractions, and so on, up to the last row to be eliminated, which involves $(b-a+1)^2$ subtractions. The exact number of operations involves three $3m^2$ instead of m^2 since for each entry we perform multiplication, division and subtraction. To simplify the transformation we skip the factor of three here.

Rewriting the sum in Equation 2.3 as a difference of two sums, we obtain Equation 2.4.

$$\sum_{m=b-a+1}^{b} m^2 = \sum_{m=1}^{b} m^2 - \sum_{m=1}^{b-a} m^2 \tag{2.4}$$

which can be explicitly written using the following sum of squares:

$$\sum_{m=1}^{b} m^2 = \frac{b(b+1)(2b+1)}{6} \tag{2.5}$$

$$\sum_{m=1}^{b-a} m^2 = \frac{(b-a)(b-a+1)(2(b-a)+1)}{6} \tag{2.6}$$

Finally, we perform the subtraction:

$$\sum_{m=1}^{b} m^2 - \sum_{m=1}^{b-a} m^2 = \frac{a(6b^2-6ab+6b+2a^2-3a+1)}{6} \tag{2.7}$$

Thus:

$$C(a,b) = \frac{a(6b^2-6ab+6b+2a^2-3a+1)}{6} \tag{2.8}$$

Using this formula, we can directly compute the costs for the sample 2D domain presented in the previous sections.

Our summary of the comparison of costs for frontal and multi-frontal solvers is presented in Table 2.1. We can conclude that for such a simple, two finite element mesh the multi-frontal solver is less than twice as fast. Actually, for small 2D grids (such as the layers of elements we process later in this work), the frontal and multi-frontal solver are comparable. However, for large, three dimensional grids, the frontal solver is far more more expensive than the multi-frontal solver, what makes it necessary to use a multi-frontal solver [35].

For a regular, uniform p polynomial order of approximation grids, the detailed estimates of the computational cost has been obtained by Calo et al. [17]. For presentation of the output formulas of this work we assume that p denotes polynomial approximation level, N refers to the number of degrees of freedom and s is the number of the levels in the elimination tree.

Table 2.1: Comparison of computational cost on a sample two element domain for frontal and multi-frontal solver

(a) Sample cost of the frontal solver

Step	a	b	$OPS(a,b)$
1	6	9	271
2	9	9	729
TOTAL			1000

(b) Sample cost of the multi-frontal solver

Step	a	b	$OPS(a,b)$
1	6	9	271
2	6	9	271
3	3	3	27
TOTAL			569

$$FLOPS_{frontal}(p,s) = 2^{2s}p^6 + \sum_{i=1}^{s-1} 2^{2(s-1)}2^{3i}p^3 = O(Np^4) + O(N^{1.5}) \tag{2.9}$$

The computational cost of the multi-frontal solver algorithm is $O(Np^4) + O(N^{1.5})$ for two dimensional problems in regular and uniform grids, p is a global polynomial order of approximation and is assumed to be constant, yet arbitrary.

The cost of eliminating the interior of a single element in a 2D grid is p^6, where p denotes the polynomial approximation level of a given element.

In case of a 2D parallel shared memory multi-frontal solver, with the assumption that the number of cores is infinite, all the additions and subtractions can be performed simultaneously, so the cost can be significantly reduced - see the Equation 2.10 presented in [17].

$$FLOPS_{shared}(p,s) = p^4 + \sum_{i=1}^{s-1} 2^{2i}p^2 = O(Np^4) + O(N) \tag{2.10}$$

When the memory is distributed, it is still possible to perform almost all the computations in parallel. An exception here are row subtractions which have to be performed sequentially on each node. In addition, there is a cost of communication between nodes (see Equation 2.11). The detailed reasoning behind this formula is presented in [17].

$$FLOPS_{distributed}(p,s) = p^6 + \sum_{i=1}^{s-1} 2^{2i}p^4 = O(Np^2) \tag{2.11}$$

Hybrid approaches also exist, where nodes have their own memory, but each node consists of multiple cores that share its memory. In such cases, we can both process and subtract rows in parallel, but we have to send submatrices across the nodes [17, 88].

2.2.2. Graph grammar modeling of the FEM solvers

The first attempt to model FEM by means of graph grammars was presented in 1996 by Flasiński and Schaefer [38]. Graph grammar productions were used to prescribe various transformations of the regular, triangular, two-dimensional, h adaptive meshes. However, using quasi context sensitive grammar turned out to have limitations in terms of adaptive meshes. As an example, application of the *1-irregularity rule* (see Definition 2.1.1) is contextual and cannot be modeled using context-free grammars. As a result, only uniform refinements can be prescribed. Beal and Schephard [14] proposed a topological structure for the finite element mesh, with a hierarchy of vertices,

edges, faces and region. The primary reason for their work was to support mesh generation and data storage. In 1993 Grabska and Hliniak [42, 43, 44] introduced Composite Programmable graphs (CP-graphs) as a mathematical formalism applicable to modeling various design processes. CP-graph grammars describe transformation of the graph representation of the domain object. They are contextual and thus, suitable for a wider range of problems. The CP-graph grammar consists of a set of graph transformations, called productions, which replace a subgraph defined on its left-hand side into a new subgraph defined on its right-hand side. The CP-graphs are very convenient for the modeling the mesh transformations, since they allow for a simple definition of the embedding transformation for the replaced subgraph. This is because the embedding transformation is coded in the production by introducing the so-called free bonds, denoting places where the replaced graph is connected through edges with the remaining parts of the graph. In the CP-graph grammar, the same number of free bonds is assumed on both sides of the production, and the free bonds numbering is utilized to embed the replaced subgraph. CP-graph grammars were used for modeling of the FEM non-uniform adaptive grids for the first time by Paszyński [87, 88, 89]. Not only were grid transformations modeled, but he also completed behavior of the solver. In [85] the composite graph grammar has been used to model two and three dimensional hp FEM. This work, however, presents the use of hypergraphs (for all definitions see Chapter 3) instead of CP-graphs to model mesh generation, transformation and execution of the linear solver. Employing hypergraph grammar formalism instead of composition graph grammars to model mesh transformations can decrease the computational complexity of performed operations, because the number of edges and nodes in the hypergraph representing a computational mesh is much smaller than the number of edges and nodes in the corresponding composite graph. The topological structure of each mesh element is represented here using four hypergraph nodes corresponding to four vertices and five hyperedges corresponding to its edges and interior, whereas in the CP-graph representation of a mesh element 18 graph nodes with 60 node bonds are needed. The most important factor though is the order, in which elements are browsed in case of linear solver. CP-graphs are very good in representing hierarchies, which is exactly what we need for a multi-frontal solver. However, accessing a neighbor element requires finding the lowest common ancestor (LCA), which is not straightforward. In case of the linear solver, we browse elements in the order shown in Figure 3.7, which requires flat access to horizontal neighbors. This can easily be achieved by using hypergraphs that have a flat structure.

2.3. Summary of open problems

We address, among others, the following problems that exist in the field of linear complexity direct solvers for problems with singularities.

1. Graph grammar models for generation of two dimensional meshes with point singularity are level-dependent and they do not allow for generation of an arbitrary number of refinement levels.
2. There is no graph grammar model prescribing execution of the sequential direct solver for two dimensional problems with point singularities resulting in a linear computational cost of the solver algorithm.
3. There is no graph grammar model prescribing execution of the parallel direct solver for two dimensional problems with point singularities resulting in a logarithmic computational cost of the solver algorithm.
4. Graph grammar models for generation of three dimensional meshes with point singularities are level-dependent and do not allow for generation of an arbitrary number of refinement levels.
5. There is no graph grammar model prescribing execution of the sequential direct solver for three dimensional problems with point singularities resulting in a linear computational cost of the solver algorithm.
6. There is no graph grammar model prescribing execution of the parallel direct solver for three dimensional problems with point singularities resulting in a logarithmic computational cost of the solver algorithm.

7. There is no theoretical estimate of the number of operations for the sequential direct solver for two dimensional problems with point singularities.
8. There is no theoretical estimate of the number of operations for the parallel direct solver for two dimensional problems with point singularities.
9. There is no theoretical estimate of the memory usage for the sequential direct solver for two dimensional problems with point singularities.
10. There is no theoretical estimate of the computational cost for the sequential direct solver for three dimensional problems with point singularities.
11. There is no theoretical estimate of the computational cost for the parallel direct solver for three dimensional problems with point singularities.
12. There is no theoretical estimate of the memory usage for the sequential direct solver for three dimensional problems with point singularities.
13. A graph grammar driven solver needs to be implemented to compare and contrast theoretical estimates with the actual performance.
14. The influence of multiple point singularities on the computational cost of the sequential direct solver remains unknown.
15. How singularities other than the point singularities affect the computational cost of the sequential direct solver remains unknown.
16. The influence of multiple point singularities on the computational cost of the parallel direct solver remains unknown.
17. How singularities other than point singularities affect the computational cost of the parallel direct solver remains unknown.

2.4. Major scientific findings

Research leading to this thesis consists of work that addressed the open problems enumerated in Section 2.3. Author's contribution includes, but is not limited to:

1. A hypergraph grammar based model prescribing generation of a two element mesh in 2D refined to an arbitrary level k towards a point singularity.
2. A hypergraph grammar model of the execution of a sequential direct solver for problems with point singularities in 2D that delivers linear computational cost and linear memory usage with respect to the number of unknowns.
3. Analysis of the concurrency of the linear computational cost sequential solver resulting in a logarithmic computational cost algorithm for grids with point singularities in two and three dimensions.
4. A hypergraph grammar model of the execution of a parallel direct solver for problems with point singularities in 2D that delivers logarithmic computational cost with respect to the number of unknowns.
5. A hypergraph grammar based model prescribing the generation of 3D mesh refined to an arbitrary level k towards a point singularity.
6. A hypergraph grammar model of the execution of a parallel solver for problems with multiple point singularities in 3D that delivers logarithmic computational cost with respect to the number of unknowns.

7. A theoretical estimate of the memory usage of the sequential solver in 2D for a two element mesh with a point singularity.
8. A theoretical estimate of the exact computational cost of the sequential solver in 2D for a two element mesh with a point singularity.
9. A theoretical estimate of the exact computational cost of the parallel solver in 2D for a two element mesh with a point singularity.
10. A theoretical estimate of the memory usage of the sequential solver in 3D for a mesh with point singularities.
11. A theoretical estimate of the order of computational cost of the sequential solver in 3D for a mesh with point singularities.
12. A theoretical estimate of the order of computational cost of the parallel solver in 3D for a mesh with point singularities.
13. Implementation of the logarithmic computational cost solver algorithm in parallel for GPU.
14. Numerical experiments confirming the theoretical estimations of the computational cost in two and three dimensions, for grids with one and many point singularities with quadratic and cubic polynomials.

The theoretical component of the results (points 1 - 4 and 7 - 9) in two dimensions is covered in Chapter 3, three dimensional hypergraph models (items 5 - 6 and 10 - 12) are described in Chapter 4, whereas the experimental results and implementation of the solver (items 13 - 14) are described in Chapter 5.

Chapter 3

Graph grammar based models of a solver algorithm in two dimensions

The purpose of this chapter is to present the core work behind this thesis for a two dimensional scenario. Section 3.1 contains a definition of a *hypergraph* and a *hypergraph grammar*. Section 3.2.1 shows how to prescribe mesh generation by using hypergraph grammars. This is followed by a set of productions describing execution of both sequential (Section 3.2.2) and parallel (Section 3.2.3) versions of the solver. The chapter is concluded with the theoretical estimations of computational cost and complexity of a sequential (Section 3.3.1) and parallel version of the solver (Section 3.3.3), as well as with an estimation of memory usage (Section 3.3.2). The findings presented in this chapter have also been partly described in my previous papers [51, 52, 86, 98]. Starting from this chapter, by *graph grammars* we assume *hypergraph grammars* and these two terms will be used interchangeably.

3.1. Basic definitions

Hypergraphs were introduced by Habel and Kreowski [56, 57] in 1987. Hypergraph grammars were their extension for mesh transformations proposed by Ślusarczyk and Paszyńska [100]. This section contains formal definitions of a *hypergraph* and a *hypergraph grammar*, accompanied by a selection of FEM-related comments. The theoretical part of this section is based on work described in [100]. However, this dissertation proposes a hypergraph grammar prescribing the solver algorithm which is a completely new contribution to science.

3.1.1. Hypergraph

A hypergraph is composed of a set of nodes and a set of hyperedges to which sequences of source and target nodes can be assigned. The nodes, as well as the hyperedges can be labeled with labels from the fixed alphabet. To represent the properties of mesh elements, the attributed hypergraphs are used. This means that each node and hyperedge can have some attributes, like for example a polynomial order of approximation. However, since the solvers presented in this work consider only the h version of FEM (with p being fixed, yet arbitrary), labeling is not necessary and will not be used to model the solver.

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

Definition 3.1.1. ***Hypergraph***. *An undirected attributed labeled hypergraph over C and A is a system $G = (V, E, t, l, at)$, 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 labeling 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 for a new hypergraph and for the subhypergraph if a sequence of so called external nodes is specified. The hypergraph replacement 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 remainder of the original hypergraph by fusing its external nodes with the corresponding external nodes in the remainder of original hypergraph. The number of external nodes should be the same in the both hypergraphs.

Definition 3.1.2. ***Hypergraph of type k****. 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* $\|ext\| = k$*.*

3.1.2. Hypergraph grammar

Hypergraph grammar [100] is an extension of Hyperedge Replacement Grammar [56, 57] for modeling mesh transformations and linear computational cost solvers 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 modeled 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 labeled with labels from the fixed alphabet.

Definition 3.1.3. ***Hypergraph production****. 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 3.1.4. ***Subhypergraph****. 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. $\forall e \in E_1 \quad t_1(e) = t_2(e)$*,*
3. $\forall e \in E_1 \quad l_1(e) = l_2(e), \forall v \in V_1 \quad l_1(v) = l_2(v)$*,*
4. $\forall e \in E_1 \quad at_1(e) = at_2(e), \forall v \in V_1 \quad at_1(v) = at_2(v)$*.*

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 3.1.5. ***Derived hypergraph****. 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* $t'_i = v_i$ (v_i *is an* i*-th element of the sequence* ext_R*) if* t_i *is an* i*-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 3.1.6. ***Hypergraph grammar***. *A hypergraph grammar is a system* $G = (V, E, P, X)$, *where:*

– V *is a finite set of labeled nodes,*
– E *is a finite set of labeled 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 definitions above constitute a superset of the formalism used in the following sections.

3.2. Hypergraph grammar models

This section presents the hypergraph grammar for modeling mesh transformations and solvers. The hypergraph grammar productions for generation of grids with point singularities presented in this work are substantially different from the ones introduced by Ślusarczyk and Paszyńska [100]. Namely, they presented a general approach to generating any kind of hp mesh, whereas this section focuses on productions optimized for problems with singularities. Moreover, since the polynomial approximation level p is fixed over the entire domain, unlike Ślusarczyk and Paszyńska, we do not use labels indicating p for each node. 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 v. The hyperedges represent interiors, edges and boundary edges of rectangular finite elements and are labeled I, $F1$, $F2$ and B, corresponding to interior, edges and boundary edges, respectively. Figure 3.1 presents a sample mesh and its hypergraph representation. It contains six vertices, six boundary edges B, one vertical interface edge $F1$ and two interior nodes I.

3.2.1. Hypergraph grammar model for generation of a mesh with point singularities

This section introduces graph grammar productions for generation of a mesh with point singularities. For simplicity, the sample mesh is restricted to initial two finite element mesh and one point singularity in the center of the bottom boundary. However, this is only a subset of graph grammar productions that can be used for generation of grids with many initial mesh elements and multiple point singularities.

We start with the graph grammar productions that can be used for both sequential and parallel generation of the initial mesh with a point singularity located at the center of the bottom of the mesh. We start with executing production P_{init} (Figure 3.2) that transforms the initial state S into the initial mesh. Next, we proceed with refinements of the left and right elements, by executing the production in Figure 3.3. It is not allowed to break the interface edge between those elements. This is due to the *1-irregularity rule* already described in Definition 2.1.1. Before breaking the interface edge we need to ensure that both of the adjacent elements are at the same level of refinement.

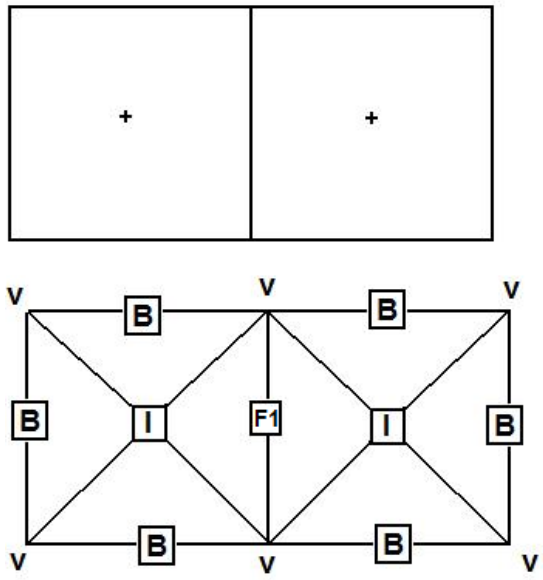


Figure 3.1: Hypergraph representation of a two element mesh

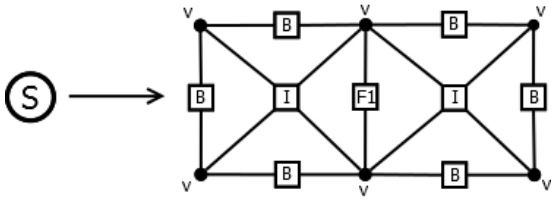
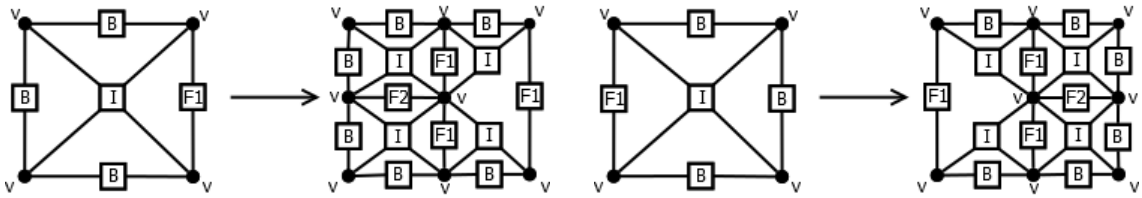


Figure 3.2: Initial production P_{init}



(a) (b)

Figure 3.3: Productions breaking the initial mesh: $P_{initleft}$ (a), $P_{initright}$ (b).

After applying $P_{initleft}$ and $P_{initright}$, we comply with the *1-irregularity rule* and are ready to apply production $P_{irregularity}$ presented in Figure 3.4. This was a special case for breaking the initial mesh. The following refinements need two more productions that can be chained until the desired level of h refinement is obtained. In

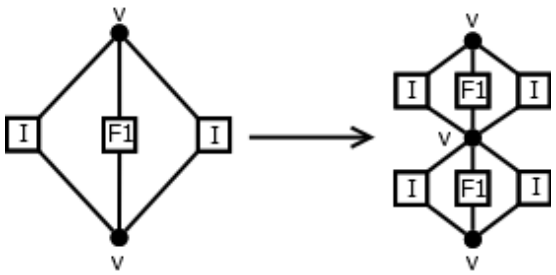


Figure 3.4: Production $P_{irregularity}$ enforcing the *1-irregularity rule* after the first refinement

order to proceed with refinements towards the central singularity, we apply production $P_{breakinterior}$ (Figure 3.5) breaking the interiors of the two elements adjacent to the point singularity (due to symmetry, we can now use a single production for breaking interior I of both left and right innermost elements), and $P_{enforceregularity}$ (Figure 3.6) breaking the common edge between them.

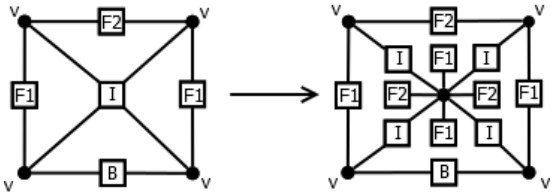


Figure 3.5: Production $P_{breakinterior}$ for breaking an interior into four even, smaller interior nodes

Figure 3.6: Production $P_{enforceregularity}$ enforcing the *1-irregularity rule*

3.2.2. Sequential solver prescribed by the hypergraph grammars

The solver is designed to process the mesh from the two bottom elements surrounding the singularity, level by level, up to the level of initial elements. Assuming we apply 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 an output mesh as in Figure 3.7. Yellow arrows indicate the order of browsing, which consists of three phases. Phases 1 and 3 are the special cases for the innermost and outermost layers of elements and will be executed exactly once for each problem. Phase 2 describes the processing of any number of intermediate layers and it is executed zero or more times depending on the h refinement level. Each node also has

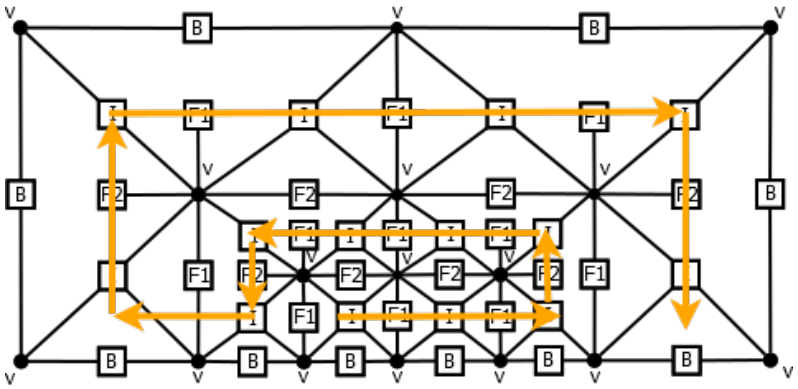


Figure 3.7: Browsing order of the linear solver

a label describing its present status in the matrix. To improve visual presentation, the color-coded labels have been used in the following manner:

Status=Unprocessed White background. Node in this state has not yet been reached by the solver algorithm and none of the contributions coming from this node are present in the matrix.

Status=Assembled Yellow background. Node in this state has been, at least partly, processed by the solver algorithm and has some (or all) of its contributions already present in the matrix.

Status=Eliminated Green background. All contributions coming from this node are already present in the matrix and partial elimination has already been executed.

Status=Distributed Red background. This node contributes to two frontal matrices from two different layers.

Phase 1

In Phase 1 we process the innermost layer that has been marked in blue in Figure 3.8. This layer consists of just two elements that are nearest the singularity. In case further adaptation is needed, they will be subjected to refinement resulting in an additional layer.

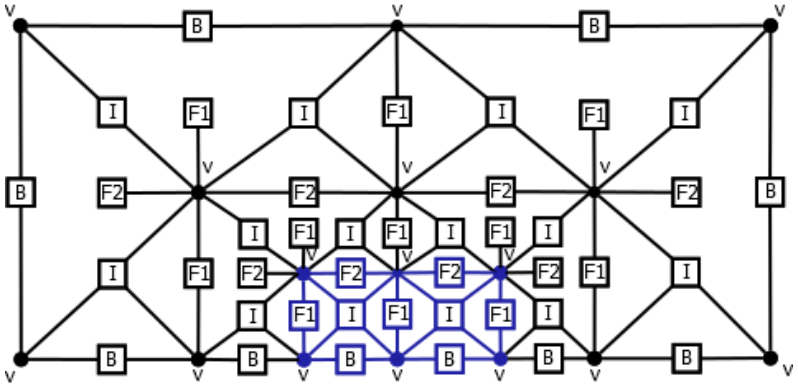


Figure 3.8: Phase 1 of the solver algorithm

We begin with generating 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 3.9. We mark in yellow the hypergraph nodes already generated in the matrix. The system of equations for the element has the following structure:

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

where α_I is the shape function related to the interior node, b is the left hand side of the equation called stiffness matrix and $l(v)$ is right hand sand of the equation called load vector (see Appendix A). The element frontal matrix is also graphically illustrated on the right panel of Figures 3.9 - 3.21. 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 the productions $P_{addboundary}$ presented in Figure 3.10. The system of equations for the element after this operation looks like

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

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

Next, we add the contributions related to the interactions of the left interface edge node with the interior and boundary nodes to the element frontal matrix. This is done by executing production $P_{addF1layer}$ presented in Figure 3.11. The system of equations for the element after this operation looks like

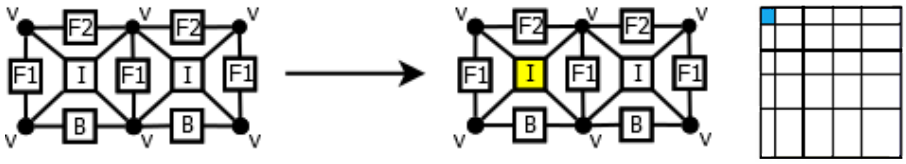


Figure 3.9: Production P_{addint} adding the interior node to the element matrix

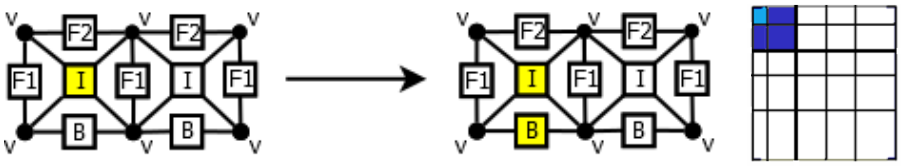


Figure 3.10: Production $P_{addboundary}$ adding the boundary node to the element matrix

$$\begin{pmatrix} b(\alpha_I,\alpha_I) & b(\alpha_I,\alpha_B) & b(\alpha_I,\alpha_{F1}) \\ b(\alpha_B,\alpha_I) & b(\alpha_B,\alpha_B) & b(\alpha_B,\alpha_{F1}) \\ b(\alpha_{F1},\alpha_I) & b(\alpha_{F1},\alpha_B) & b(\alpha_{F1},\alpha_{F1}) \end{pmatrix} = \begin{pmatrix} l(\alpha_I) \\ l(\alpha_B) \\ l(\alpha_{F1}) \end{pmatrix}$$

where α_{F1} corresponds to the shape functions associated with the vertical interface edge.

Next, the same operation is done for the upper interface edge node, as it is illustrated in Figure 3.12 by production $P_{addF2layer}$, as well as for the vertices, as illustrated in Figure 3.13 by production $P_{addvertices}$. Each time the frontal matrix is augmented with new entries, as graphically illustrated on the right panels.

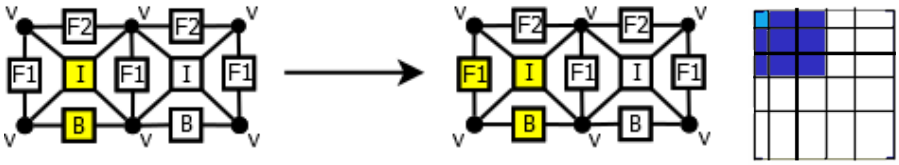


Figure 3.11: Production $P_{addF1layer}$ adding the left interface edge to the element matrix

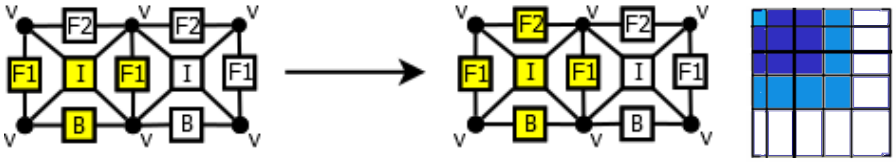


Figure 3.12: Production $P_{addF2layer}$ adding the upper interface edge to the element matrix

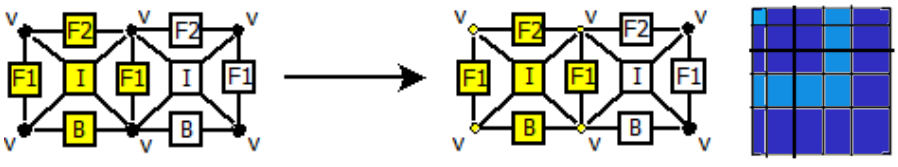


Figure 3.13: Production $P_{addvertices}$ adding vertices to the element matrix

The resulting state of the element matrix is depicted in Figure 3.13. 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 of their contributions already present in the matrix. It is worth mentioning that interior nodes are always fully assembled, since their basis functions vanish on all edges and have support on just one element. Edge basis functions, unless they are boundary edges, are always shared by two elements. Vertex node basis functions can be shared by from one to four distinct elements.

We start with eliminating the interior's contribution from the matrix. This is done by executing production $P_{elimint}$ presented in Figure 3.14. We mark eliminated nodes in a dark color. The only other fully assembled node

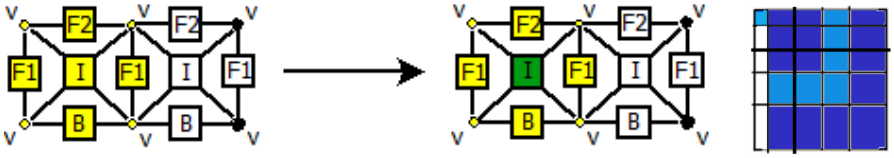


Figure 3.14: Production $P_{elimint}$ eliminating the interior's contribution from the matrix

is the boundary node B. Thus, we proceed with its elimination by executing production $P_{elimboundary}$ presented in Figure 3.15.

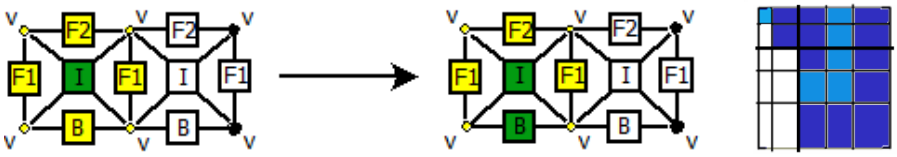


Figure 3.15: Production $P_{elimboundary}$ eliminating the boundary edge from the element matrix

At this time we are done with the first element. No more nodes can be eliminated without further knowledge about neighboring elements. We start processing the right neighbor of the current element and begin adding its nodes' contributions. Again, first we add contributions coming from the interior node of the second element. This is done by executing production $P_{addint2}$, see Figure 3.16. Then we add contributions coming from the boundary

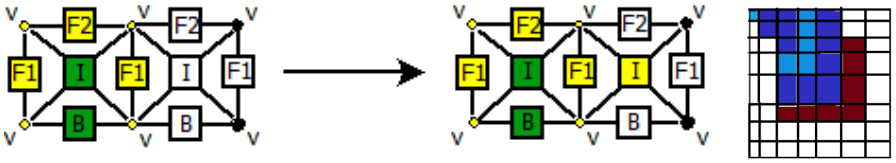


Figure 3.16: Production $P_{addint2}$ adding the interior of the second element to the matrix

edge, by running the production $P_{addboundary2}$ (Figure 3.17). And the upper and right interface edges (productions

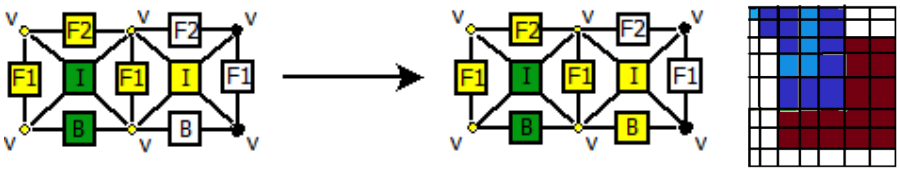


Figure 3.17: Production $P_{addboundary2}$ adding the boundary edge to the matrix

$P_{addF1layer2}$, $P_{addF2layer2}$, see Figure 3.18). Once we add all vertices (Figure 3.18), we can proceed with the

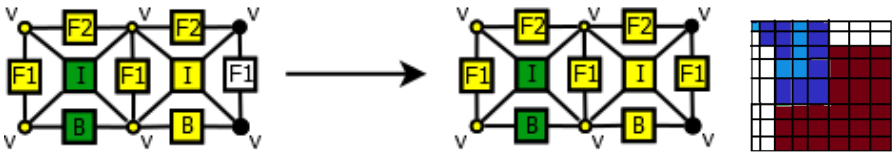


Figure 3.18: Productions $P_{addF1layer2}$, $P_{addF2layer2}$ adding the layer nodes to the matrix

elimination of the fully assembled nodes for the second element.

Again, we start with eliminating contribution coming from the interior node (production $P_{elimint2}$, see Figure 3.19). Similarly to the first element, we can eliminate also the boundary edge (production $P_{elimboundary2}$, Figure

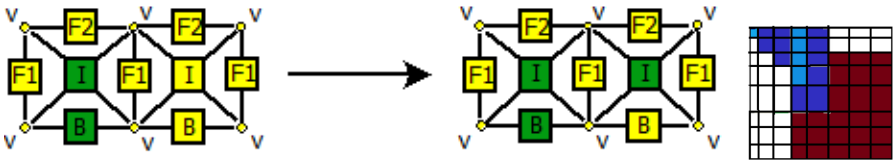


Figure 3.19: Production $P_{elimint2}$ eliminating the interior of the second element

3.20). After adding the second element, we now have all the contributions for the interface edge and vertex between elements, so both can now be eliminated (production $P_{elimcommon}$, Figure 3.21).

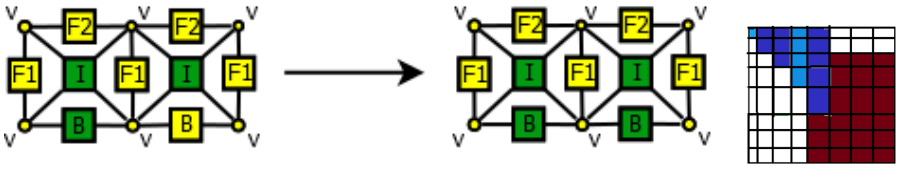


Figure 3.20: Production $P_{elimboundary2}$ eliminating the boundary edge in the element matrix

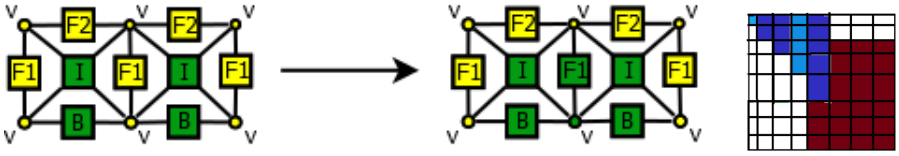


Figure 3.21: Production $P_{elimcommon}$ eliminating the common edge

In summary, the execution of the solver algorithm in Phase 1 can by expressed by the following sequence of hypergraph grammar productions: $P_{addint} \mapsto P_{addboundary} \mapsto P_{addF1layer} \mapsto P_{addF2layer} \mapsto P_{addvertices} \mapsto P_{elimint} \mapsto P_{enforceregularity} \mapsto P_{elimboundary} \mapsto P_{addint2} \mapsto P_{addboundary2} \mapsto P_{addF1layer2} \mapsto P_{add21layer2} \mapsto P_{elimint2} \mapsto P_{elimboundary2} \mapsto P_{elimcommon}$.

Phase 2

Phase 2 concerns the layer marked in purple in Figure 3.22. From this point we do not plot the matrices associated with elements, since they can be deduced from the hypergraph. Similarly to previous steps, we begin by adding contributions coming from neighboring element's interior (Figure 3.23). This time though, the outer edge spans over the two elements due to the *1-irregularity rule*. Moreover, due to the plentitude of productions in this phase we switch to numerical names, starting from P_{17}, which is the seventeenth of the productions.

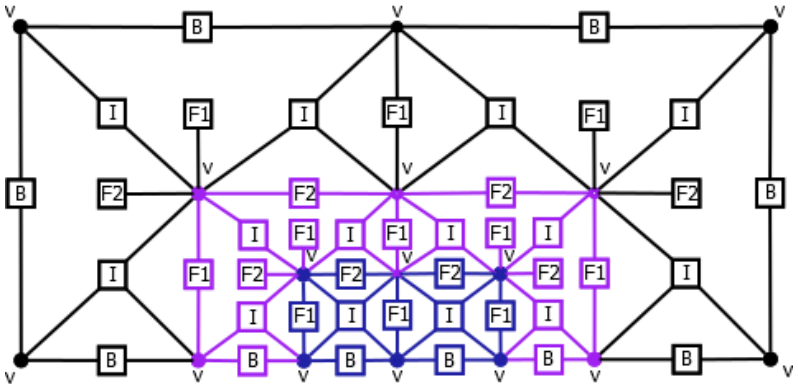


Figure 3.22: Phase 2 of the solver algorithm

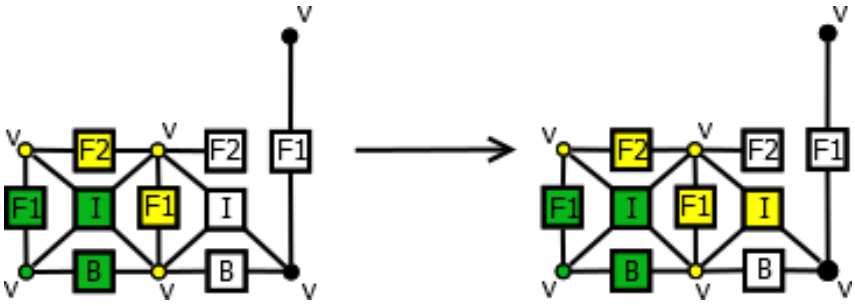


Figure 3.23: Production P_{17}

Next, we add contributions coming from the boundary edge (Figure 3.24) and the upper interface edge (Figure 3.25).

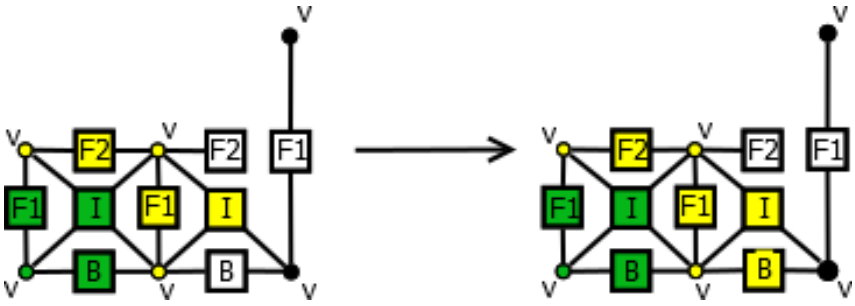


Figure 3.24: Production P_{18}

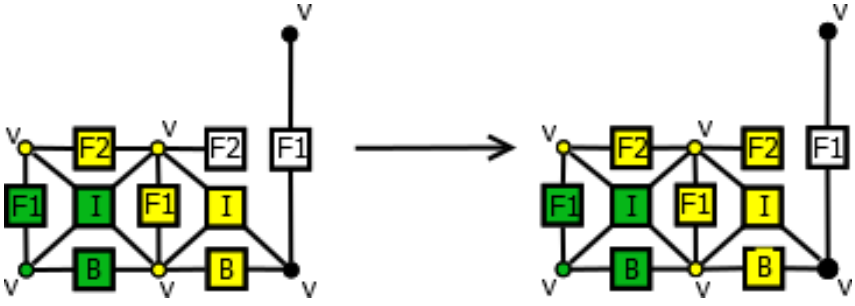


Figure 3.25: Production P_{19}

Finally, we add contributions from the outer, (*large*) interface edge (Figure 3.26) and its vertices (Figure 3.27).

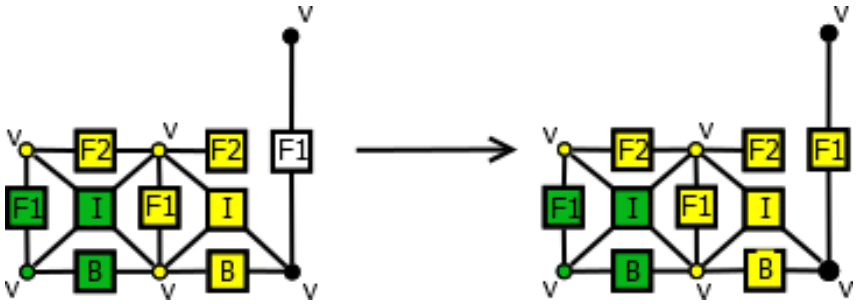


Figure 3.26: Production P_{20}

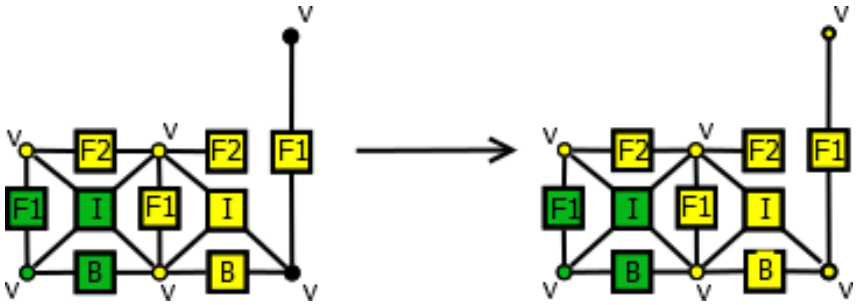


Figure 3.27: Production P_{21}

Having finished adding contributions from this element, we can proceed with partial elimination. The fully assembled nodes are: the interior I, boundary edge B, left interface $F1$ and interface vertices on the left. Again, we start with the elimination of the interior node in the matrix (Figure 3.28) and then the boundary edge node (Figure 3.29).

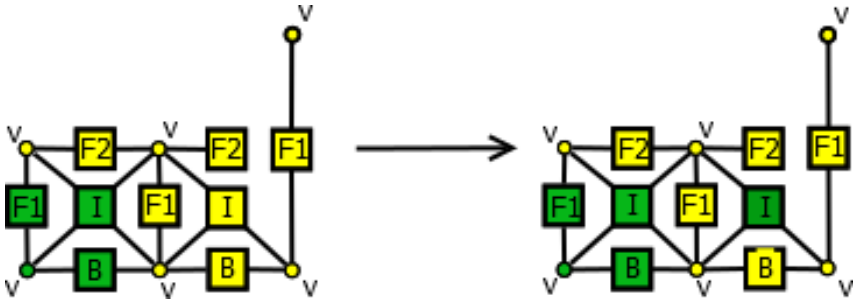


Figure 3.28: Production P_{22}

At this time, we can also eliminate the interface edge and vertex between those elements 3.30. Since further eliminations are not possible, we proceed with adding contributions from the next neighboring element. Again, we start with adding the interior of a new element (Figure 3.31), followed by interface edge $F1$ (Figure 3.32) and a *large* interface edge $F2$ (Figure 3.33).

We also add the vertex terminating the *big* edge $F2$ (Figure 3.34). Similarly to the previous steps, we can now start eliminating nodes. As always, we can eliminate the interior I (Figure 3.35). We have also collected all the information required to eliminate edge $F2$ (Figure 3.36). Unfortunately, this was all we could do, since in order to eliminate the remaining nodes we need additional contributions coming from the left neighbor. In Figures 3.37 - 3.38 we add the missing contributions - interior I and vertical edge $F1$. Again, we are now ready to eliminate the

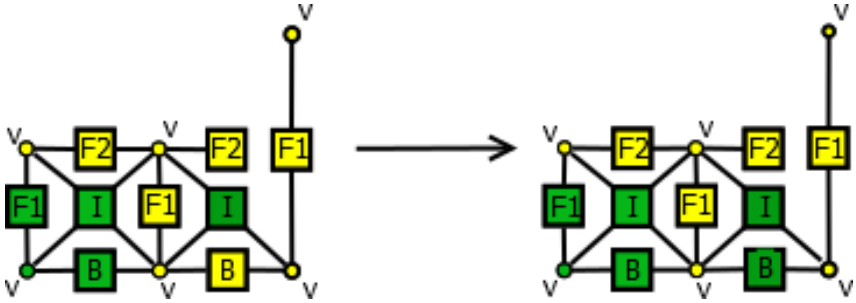


Figure 3.29: Production P_{23}

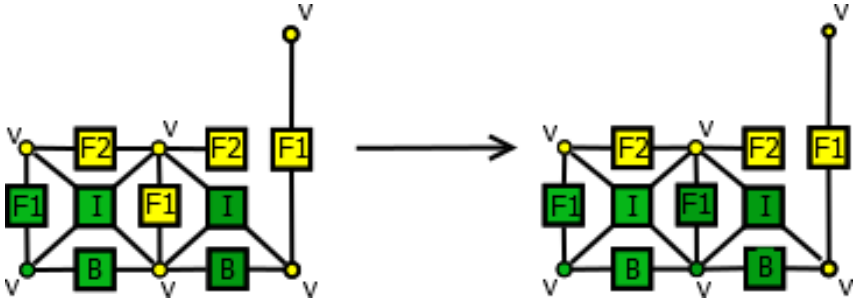


Figure 3.30: Production P_{24}

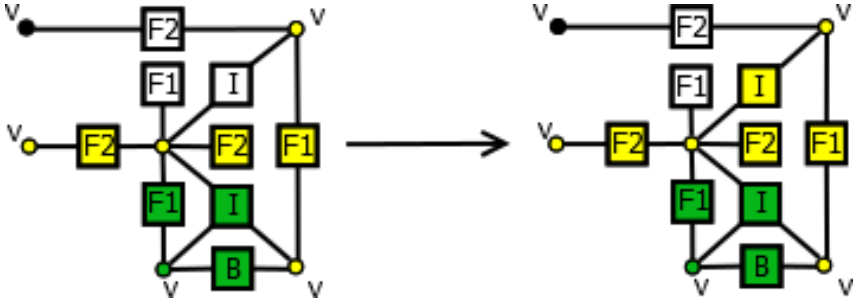


Figure 3.31: Production P_{25}

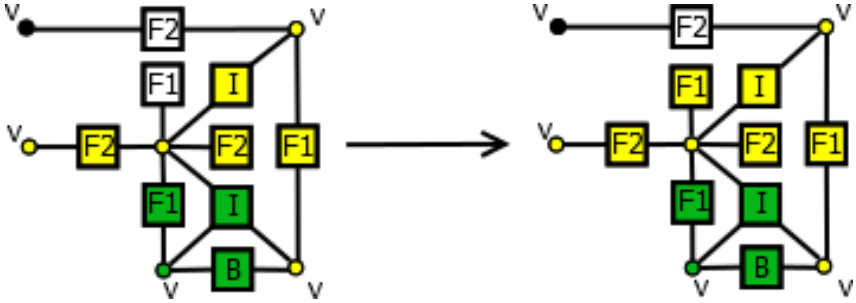


Figure 3.32: Production P_{26}

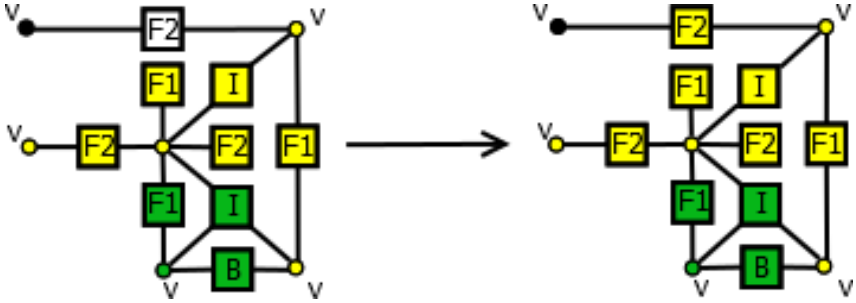


Figure 3.33: Production P_{27}

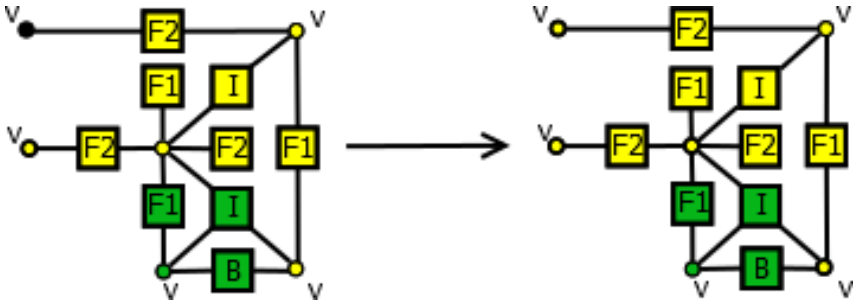


Figure 3.34: Production P_{28}

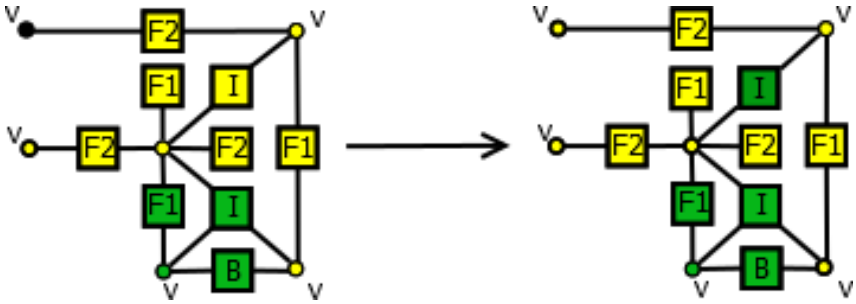


Figure 3.35: Production P_{29}

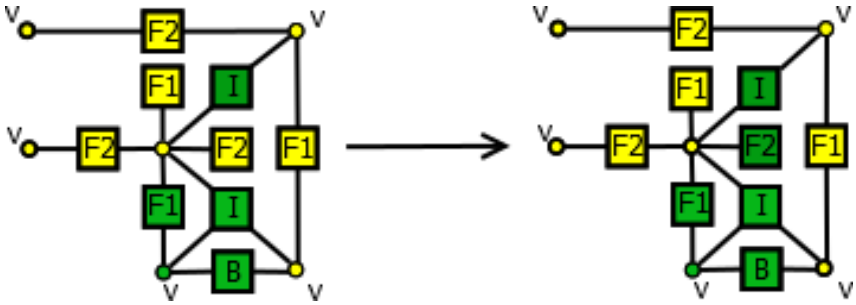


Figure 3.36: Production P_{30}

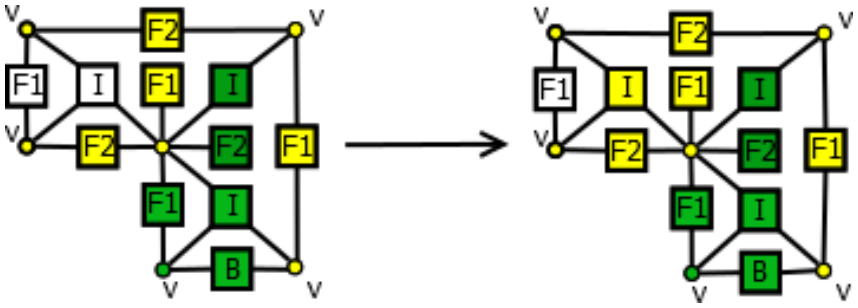


Figure 3.37: Production P_{31}

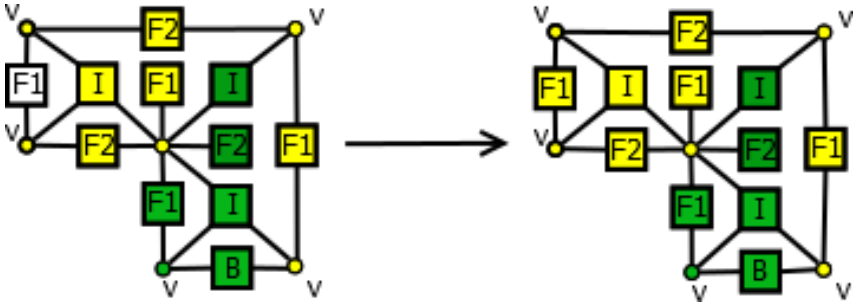


Figure 3.38: Production P_{32}

contribution coming from the interior (Figure 3.39). We can also eliminate the interface edge $F1$ (Figure 3.40). On

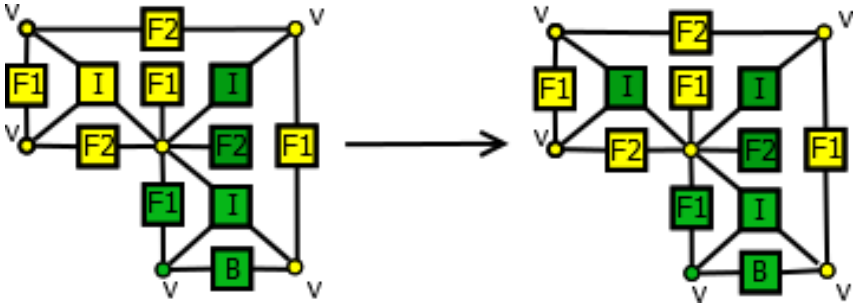


Figure 3.39: Production P_{33}

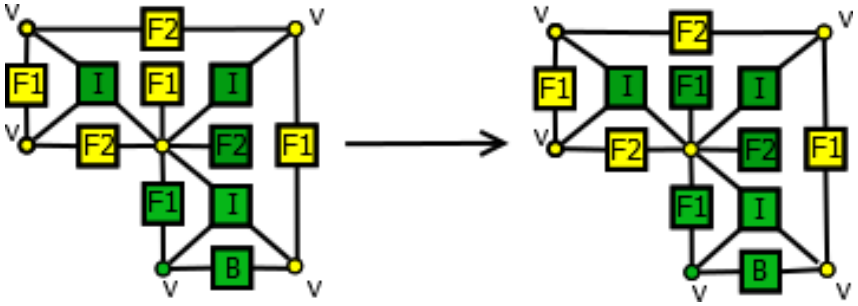


Figure 3.40: Production P_{34}

top of that, since the bottom neighbors are already in the matrix, we can also perform productions in Figure 3.41. In Figures 3.42 - 3.45 we add contributions coming from the left neighbor. Next, following our usual procedure, we start with eliminating interior's contribution (Figure 3.46). The next step is to eliminate interface edge $F1$ and vertex between these elements (Figure 3.47). We proceed with elimination of the edge $F2$ in Figure 3.48. At

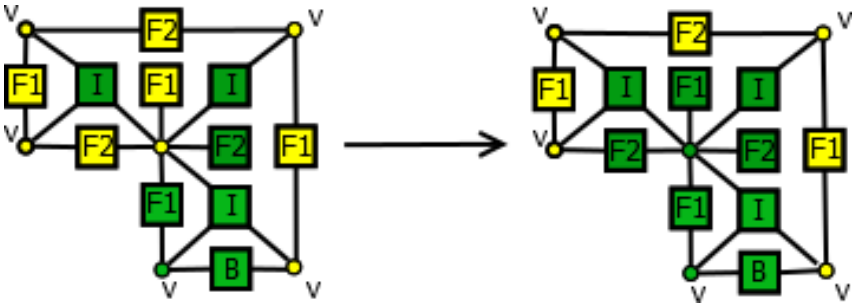


Figure 3.41: Production P_{35}

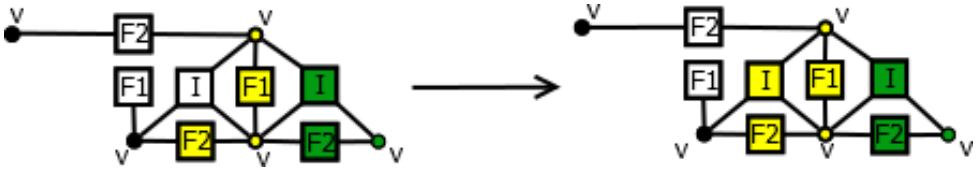


Figure 3.42: Production P_{36}

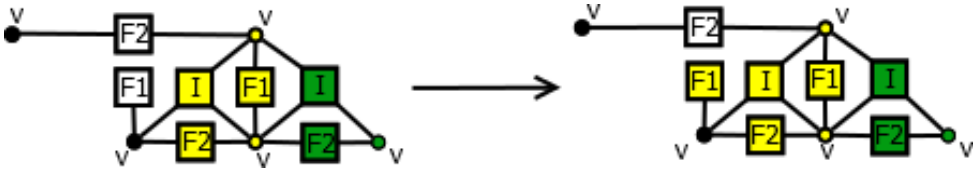


Figure 3.43: Production P_{37}

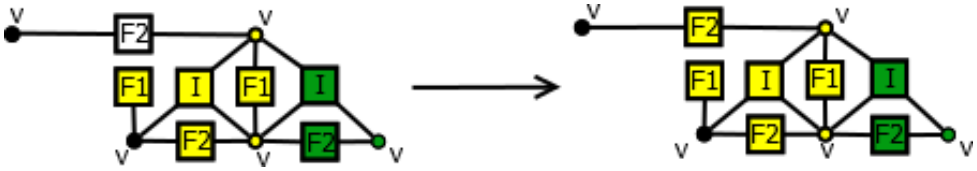


Figure 3.44: Production P_{38}

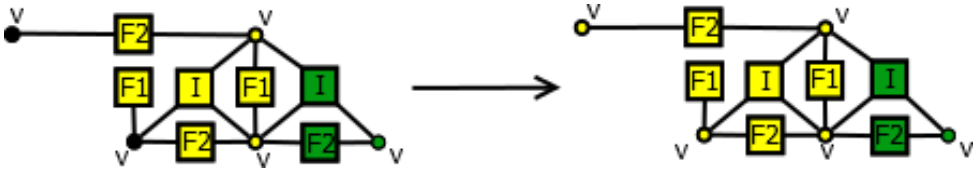


Figure 3.45: Production P_{39}

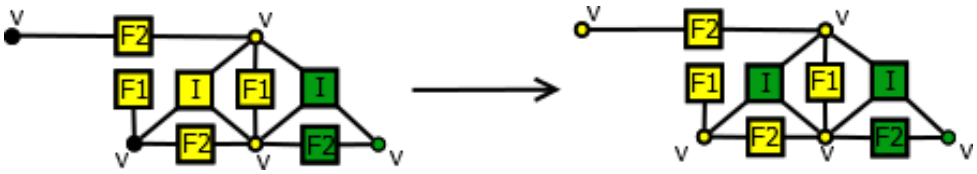


Figure 3.46: Production P_{40}

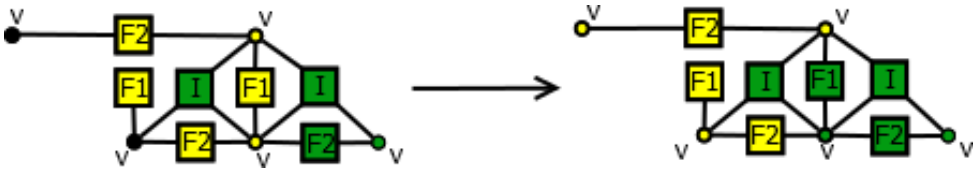


Figure 3.47: Production P_{41}

this point, we need to add contributions from another element from the layer, which is shown in Figures 3.49 - 3.53. Again, we eliminate the interior node (Figure 3.54), followed by elimination of the interface edge $F1$ (Figure 3.55). At this point, we cannot eliminate anything else and we continue following the order from Figure 3.22.

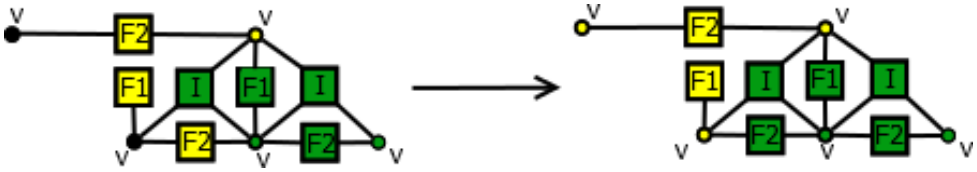


Figure 3.48: Production P_{42}

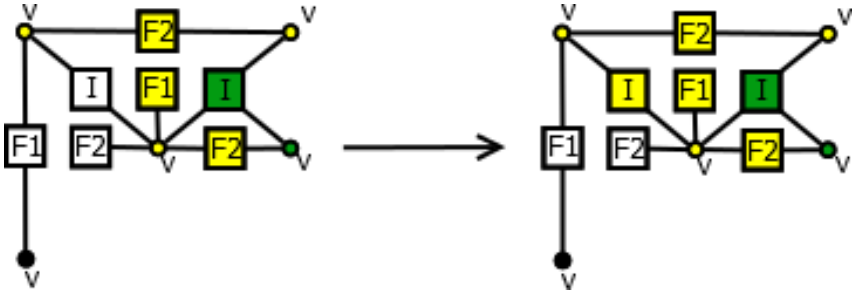


Figure 3.49: Production P_{43}

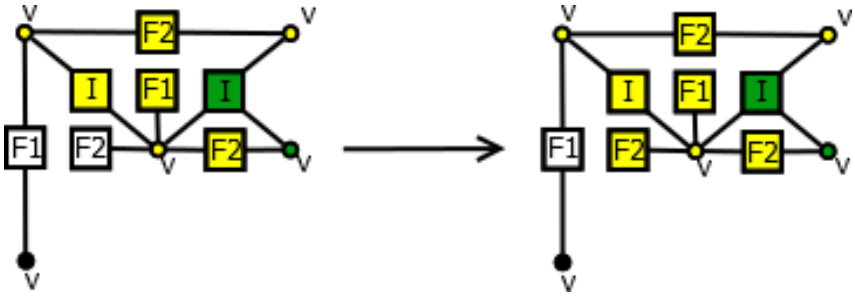


Figure 3.50: Production P_{44}

Productions in Figures 3.56 - 3.57 add nodes of the next element, whereas productions in Figures 3.58 - 3.63 eliminate the remaining nodes so that only the outer edges and vertices of the layer remain.

To sum up, the execution of the solver algorithm in Phase 2 can by expressed by the following sequence of hypergraph grammar productions: $P_{17} \mapsto P_{18} \mapsto P_{19} \mapsto P_{20} \mapsto P_{21} \mapsto P_{22} \mapsto P_{23} \mapsto P_{24} \mapsto P_{25} \mapsto P_{26} \mapsto P_{27} \mapsto P_{28} \mapsto P_{29} \mapsto P_{30} \mapsto P_{31} \mapsto P_{32} \mapsto P_{33} \mapsto P_{34} \mapsto P_{35} \mapsto P_{36} \mapsto P_{37} \mapsto P_{38} \mapsto P_{39} \mapsto P_{40} \mapsto P_{41} \mapsto P_{42} \mapsto P_{43} \mapsto P_{44} \mapsto P_{45} \mapsto P_{46} \mapsto P_{47} \mapsto P_{48} \mapsto P_{49} \mapsto P_{50} \mapsto P_{51} \mapsto P_{52} \mapsto P_{53} \mapsto P_{54} \mapsto P_{55} \mapsto P_{56} \mapsto P_{57}$.

Phase 2 occurs exactly k times, where $k \geq 0$ denotes the count of refinement levels. This means concatenating the chain of productions k times: $(P_{17} \mapsto P_{18} \mapsto ... \mapsto P_{56} \mapsto P_{57})^k$.

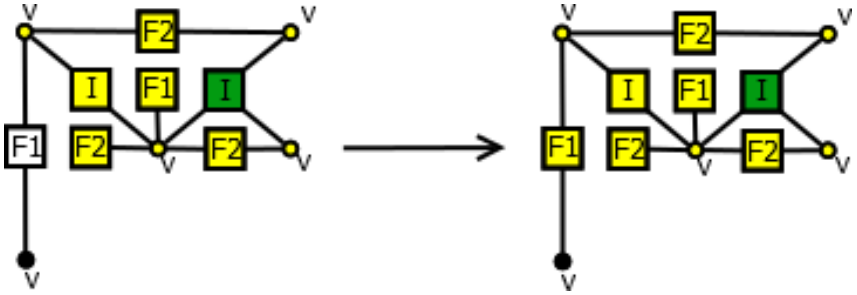


Figure 3.51: Production P_{45}

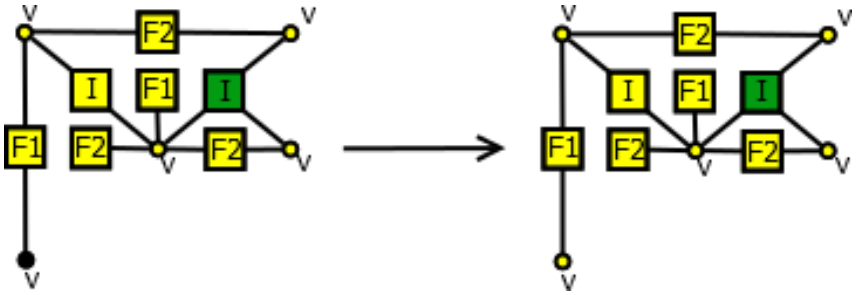


Figure 3.52: Production P_{46}

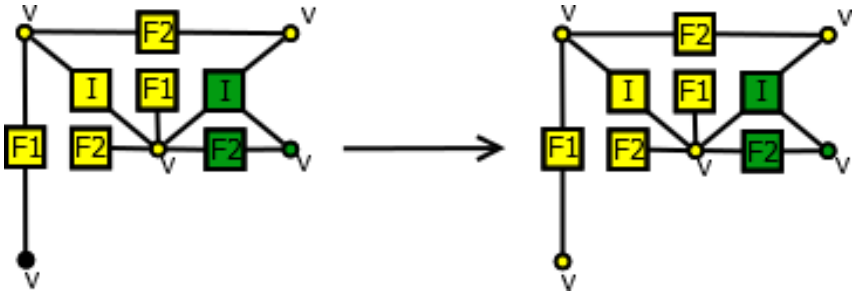


Figure 3.53: Production P_{47}

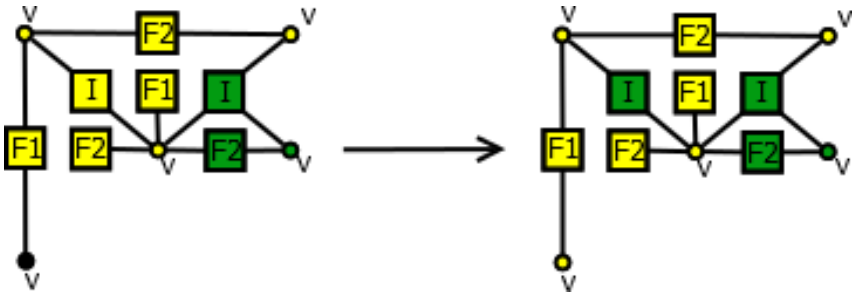


Figure 3.54: Production P_{48}

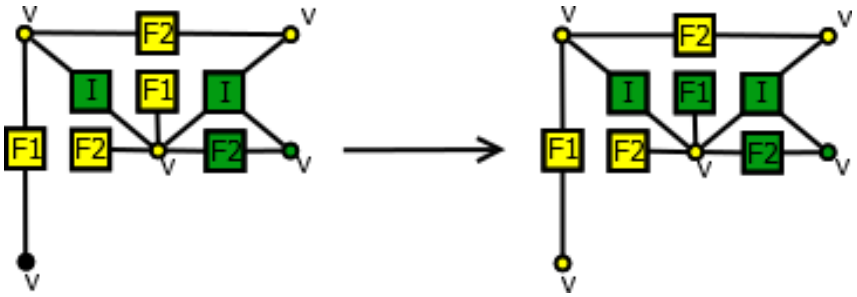


Figure 3.55: Production P_{49}

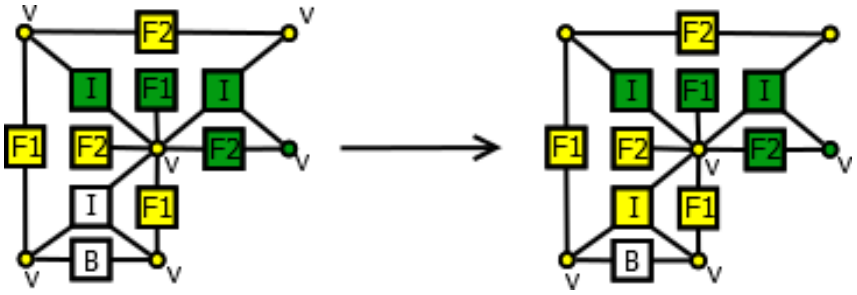


Figure 3.56: Production P_{50}

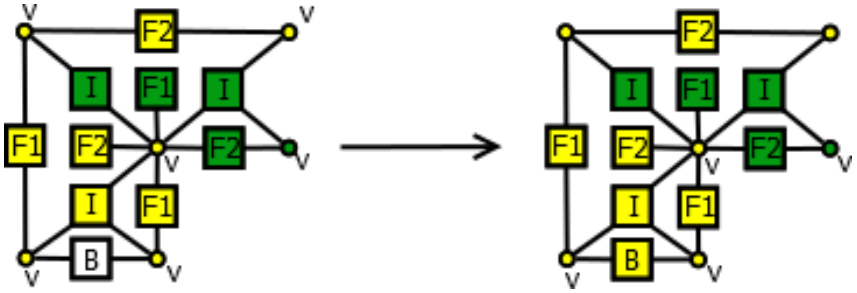


Figure 3.57: Production P_{51}

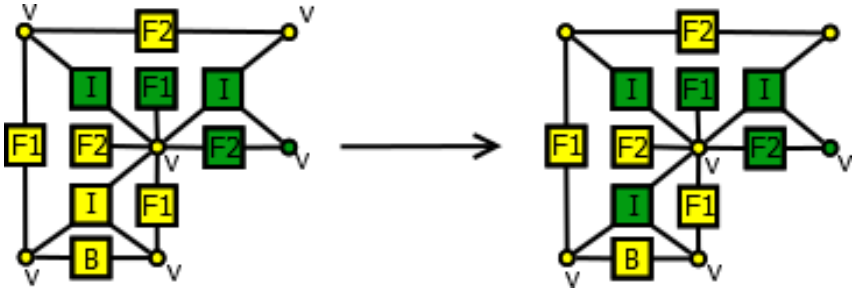


Figure 3.58: Production P_{52}

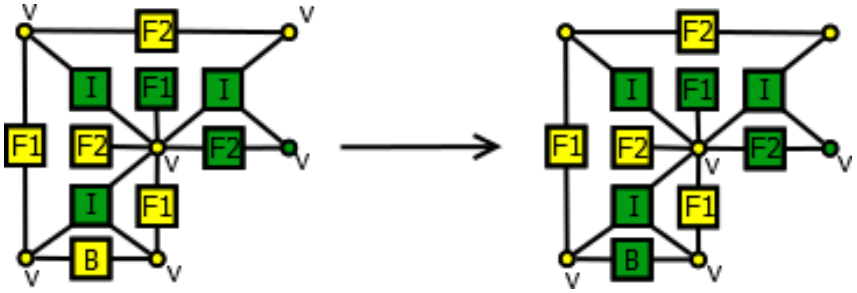


Figure 3.59: Production P_{53}

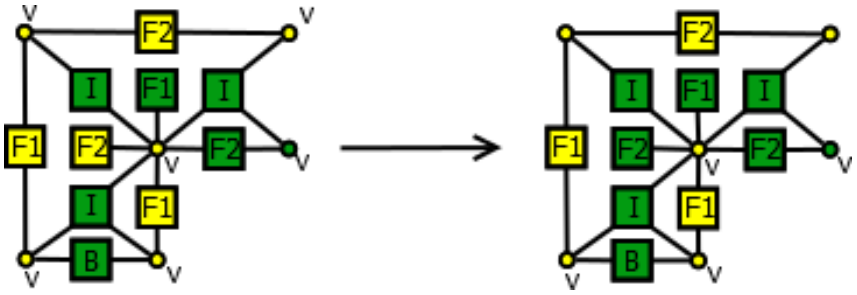


Figure 3.60: Production P_{54}

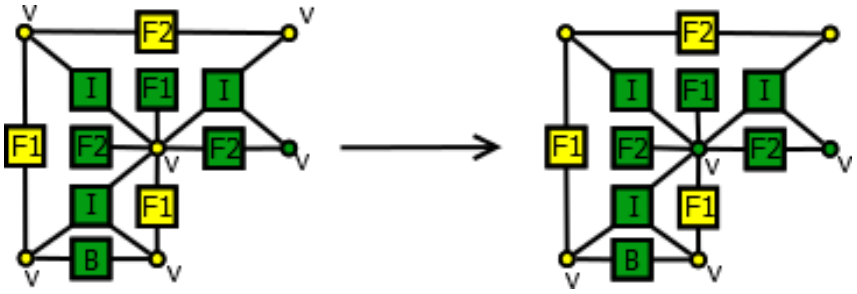


Figure 3.61: Production P_{55}

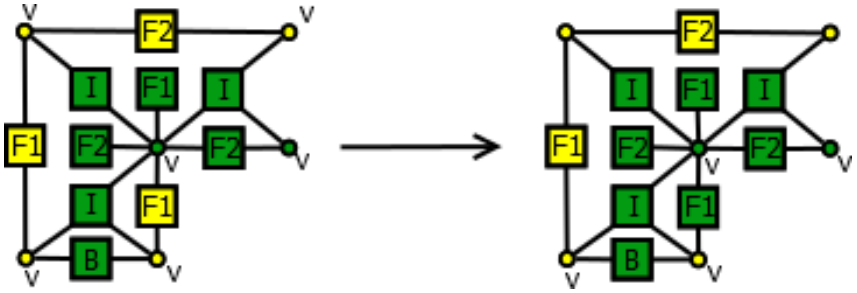


Figure 3.62: Production P_{56}

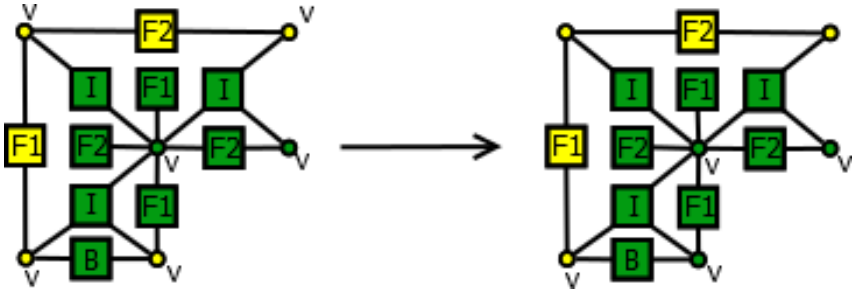


Figure 3.63: Production P_{57}

Phase 3

After Phase 2 is over, we process the outermost layer in Phase 3 in a similar manner (Figure 3.64). This occurs only once and is analogical to Phase 1. Since there is no added value coming from the productions at this stage, their presentation will be omitted. We follow the order shown in Figure 3.7 until we reach the very last element.

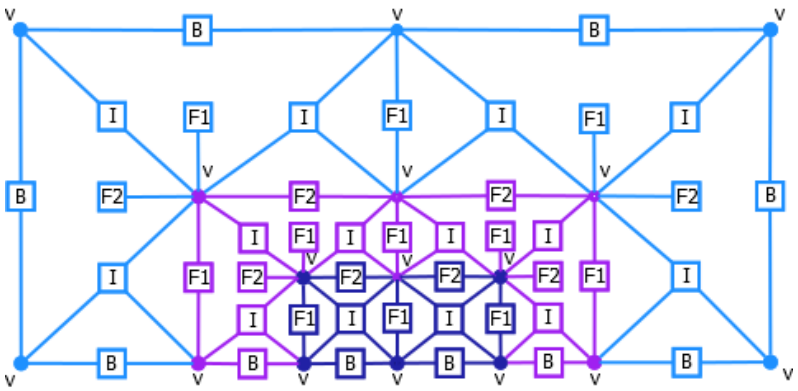


Figure 3.64: Phase 3 of the solver algorithm

Then, we can proceed with a backward substitution that provides us with a solution with a given accuracy. It is up to the adaptation algorithm whether a solution for a given mesh is already satisfactory. Since usually it is not possible to estimate error rate *a priori*, we compare the relative error decrease rate on each element. If it is more than a fixed (but arbitrary) threshold, we refine this element (and possibly, increase p). In the case of a given mesh, if we decide to refine elements, we already know, due to the point singularity, which elements need refinement.

3.2.3. Parallel solver prescribed by the hypergraph grammars

In this section we mark the partial order of the productions and decide, which of them can be executed in parallel. This analysis will allow us to switch from a frontal to a multi-frontal solver by assigning a single frontal matrix to each layer of the mesh. These layers can be all processed in parallel. The resulting Schur complements (see Appendix C) from each layer, associated with the interfaces between layers, will be merged using the binary tree elimination pattern.

Mesh generation

When compared with the remaining parts of the algorithm, mesh generation based on graph grammars is computationally inexpensive 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 trigger some additional graph grammar productions, $P_{separatetop}$ presented in Figure 3.65 as many times as there are layers of the mesh, except for the last layer, where we trigger $P_{separatebottom}$, presented in Figure 3.66. These graph grammar productions separate particular layers of the mesh, so we can process them independently, without overlap of nodes.

Processing of each layer

Having partitioned the computational mesh 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 instead of two elements. 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

$$\begin{aligned} &P_{addint} \mapsto P_{addboundary} \mapsto P_{addF1layer} \mapsto \\ &P_{addF2layer} \mapsto P_{addvertices} \mapsto P_{elimint} \mapsto \\ &P_{elimboundary} \mapsto ... \end{aligned} \tag{3.1}$$

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

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

Processing of the interfaces

Unlike the sequential algorithm, for the parallel algorithm we keep all the interface nodes uneliminated until all the layers are processed. Having eliminated the interiors of all layers, we have several frontal matrices associated with all the layers, one frontal matrix per layer. The frontal matrices contain the nodes from both interfaces, located on the top and on the bottom side of each layer. We execute graph grammar productions $P_{mergetop}$ merging the two top layers into one layer, additionally merging the two frontal matrices into one matrix, in such a way that the

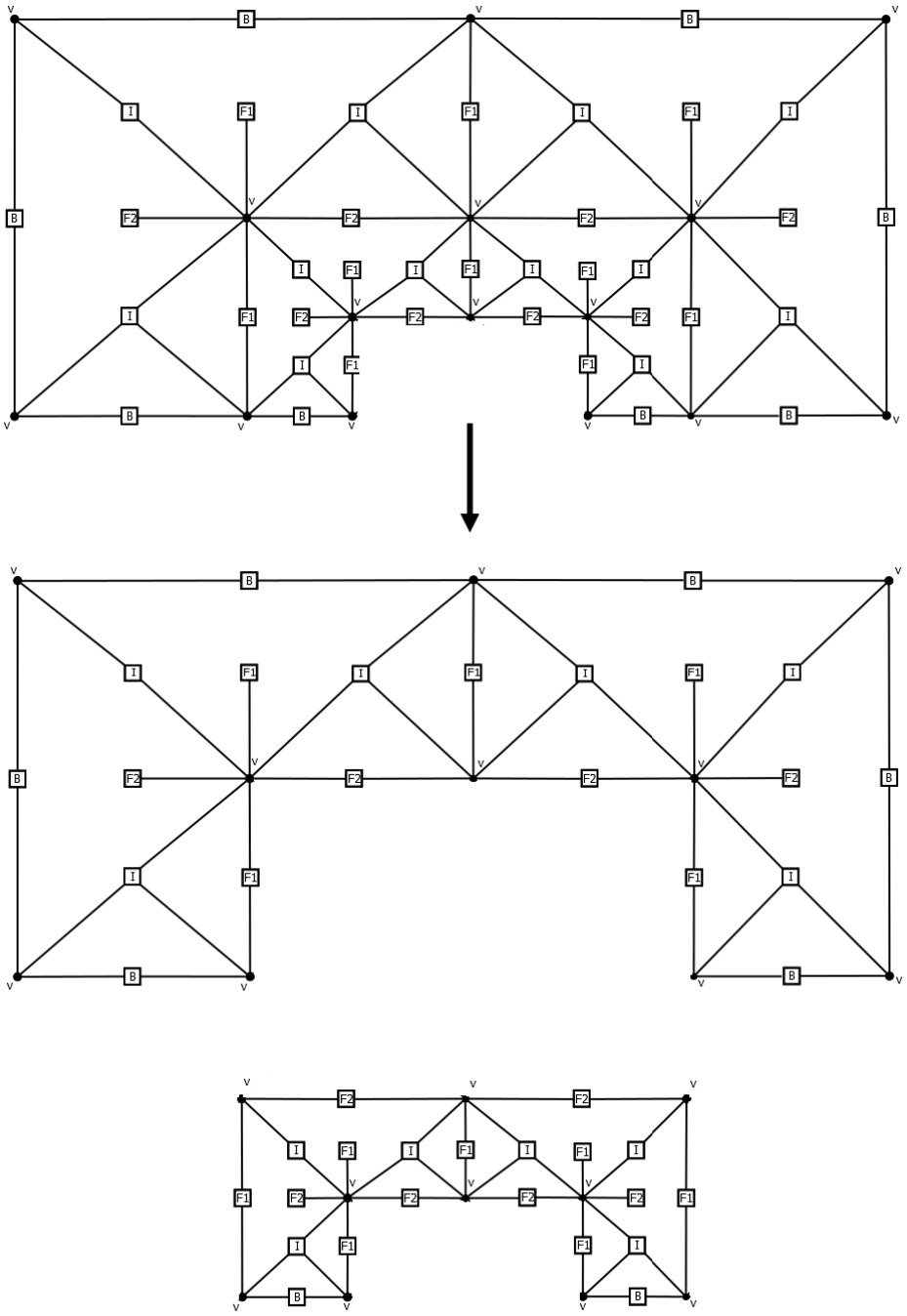


Figure 3.65: Production $P_{separatetop}$ for separation of the top layer

rows of the merged matrix associated with the common interface are now fully assembled. This is shown in Figure 3.67, with fully assembled nodes highlighted in 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 3.68. The fully assembled nodes are eliminated, which is denoted by changing their red color into green. The same scenario applies to the two bottom layers, which is denoted by productions $P_{mergebottom}$ and $P_{elimbottom}$ presented in Figures 3.69 and 3.70. Finally, we merge the patches of two layers, obtained from the previous steps, into a single path. This is expressed by production $P_{mergetop}$ presented in Figure 3.71. Additionally, the two frontal matrices associated with the two patches are merged into one frontal matrix, with the rows associated with nodes on the

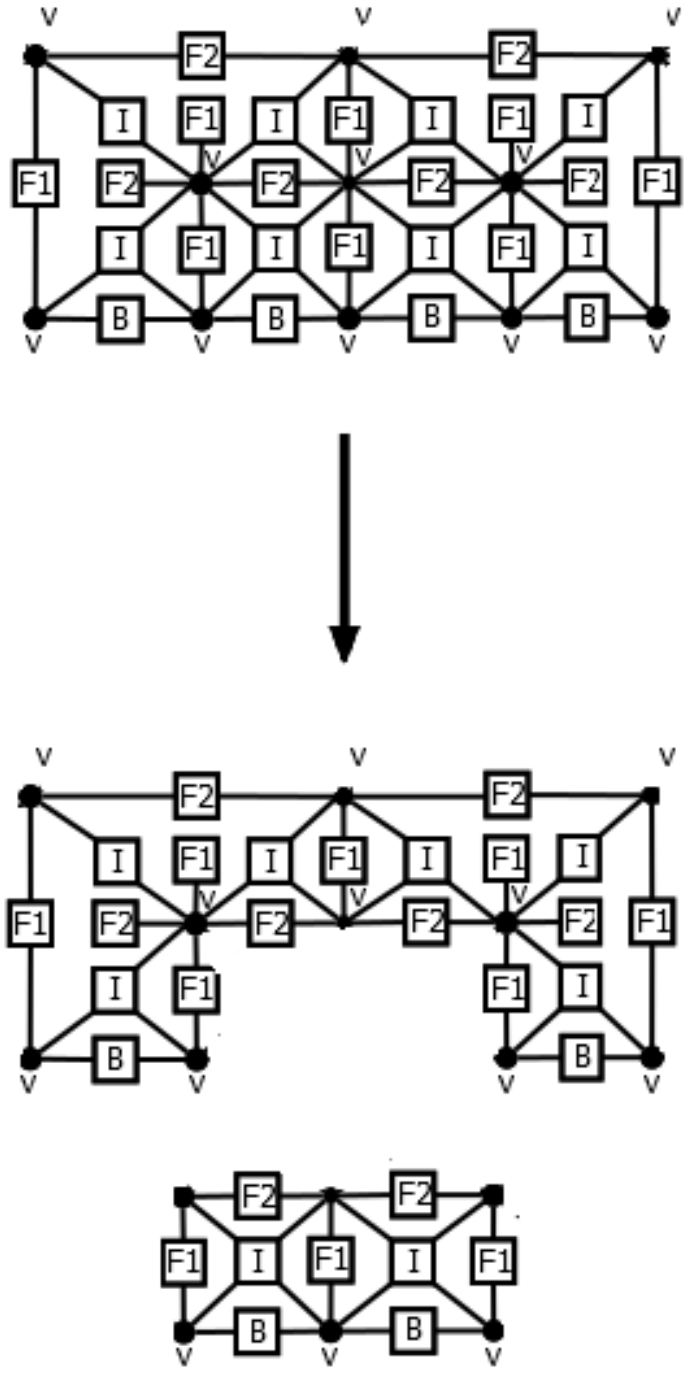


Figure 3.66: Production $P_{separatebottom}$ for separation of the bottom layer

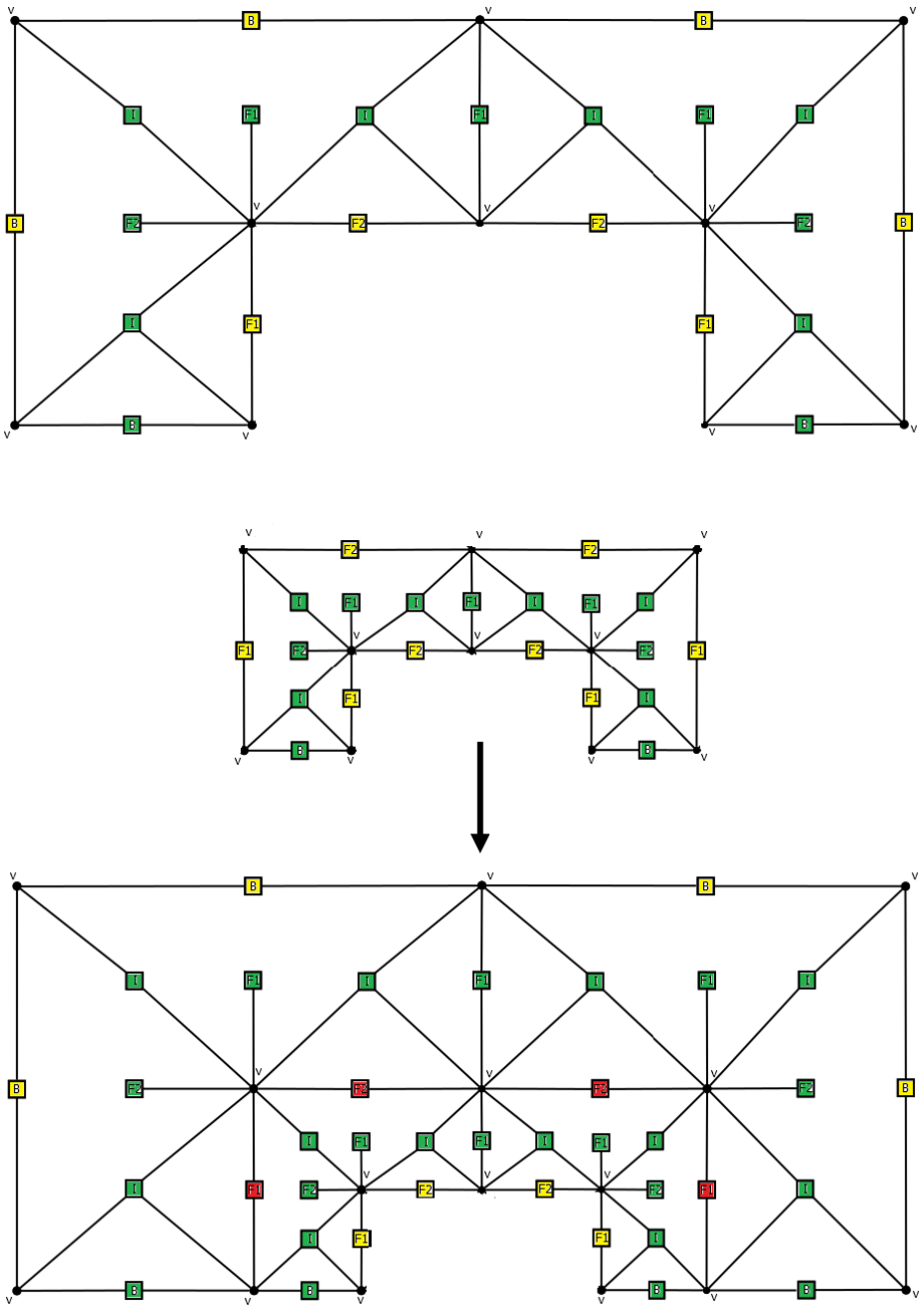


Figure 3.67: Production $P_{mergetop}$ for merging of the two top layers with interface matrices

common interface fully assembled. These nodes are denoted in Figure 3.71 by red. At this point we can solve the top problem, since all the nodes are fully assembled, including the top boundary nodes. This is expressed by production $P_{solvetop}$ presented in Figure 3.72. Having solved the top problem, we process with analogous backward substitutions, which can be expressed simply by identical graph grammar productions, but executed in the 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. This technique is illustrated in Chapter 5 with a few examples in two and three dimensions.

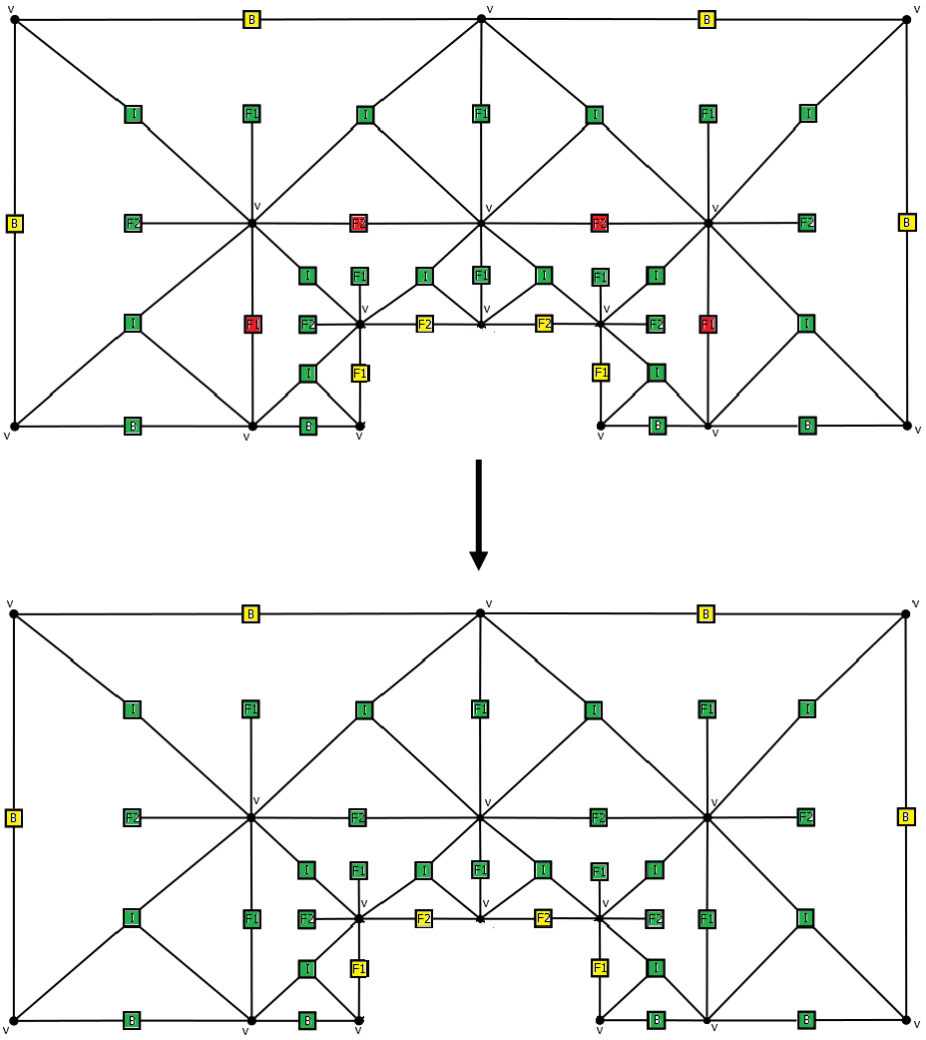


Figure 3.68: Production $P_{elimtop}$ for elimination of the common layer over the two already merged top layers

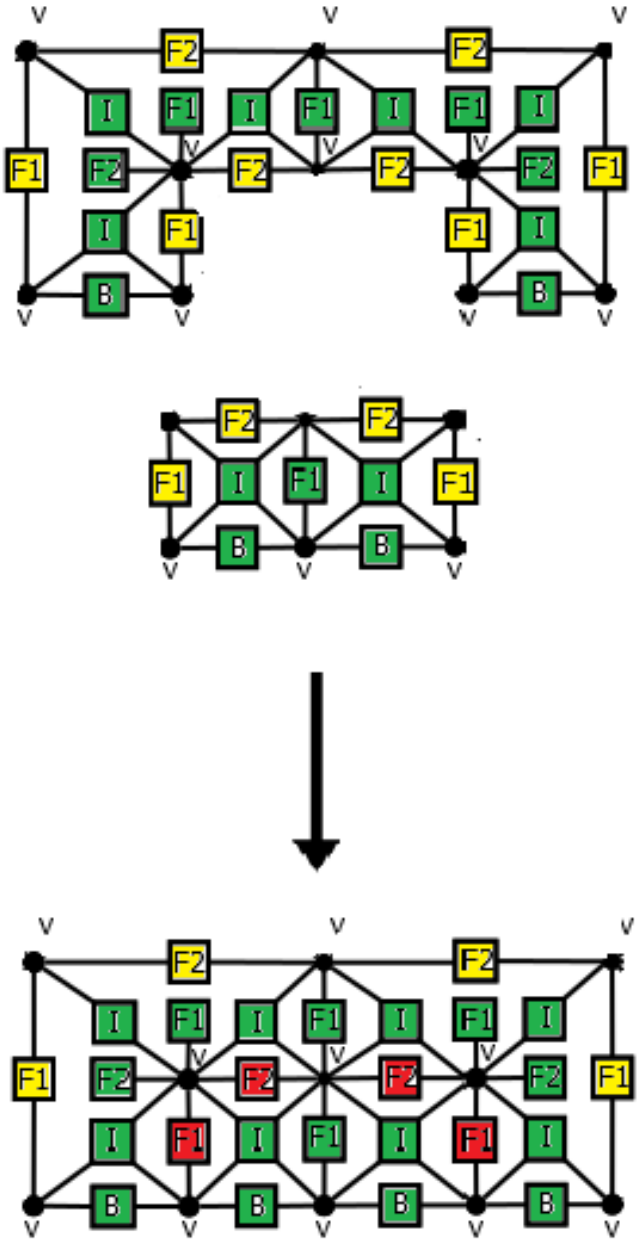


Figure 3.69: Production $P_{mergebottom}$ for merging of the two bottom layers with interface matrices

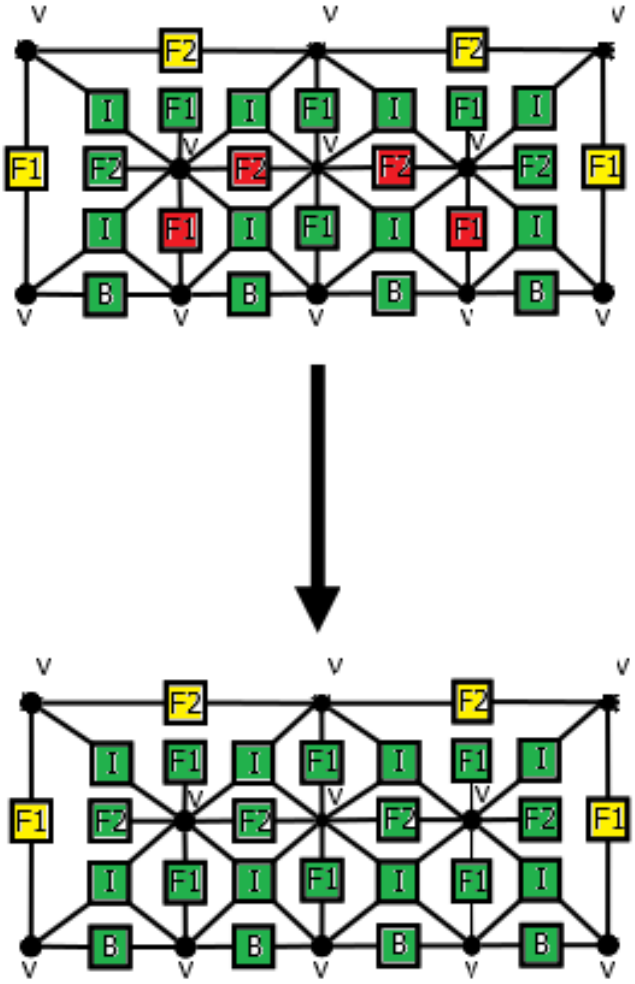


Figure 3.70: Production $P_{elimbottom}$ for elimination of the common layer over the two already merged bottom layers

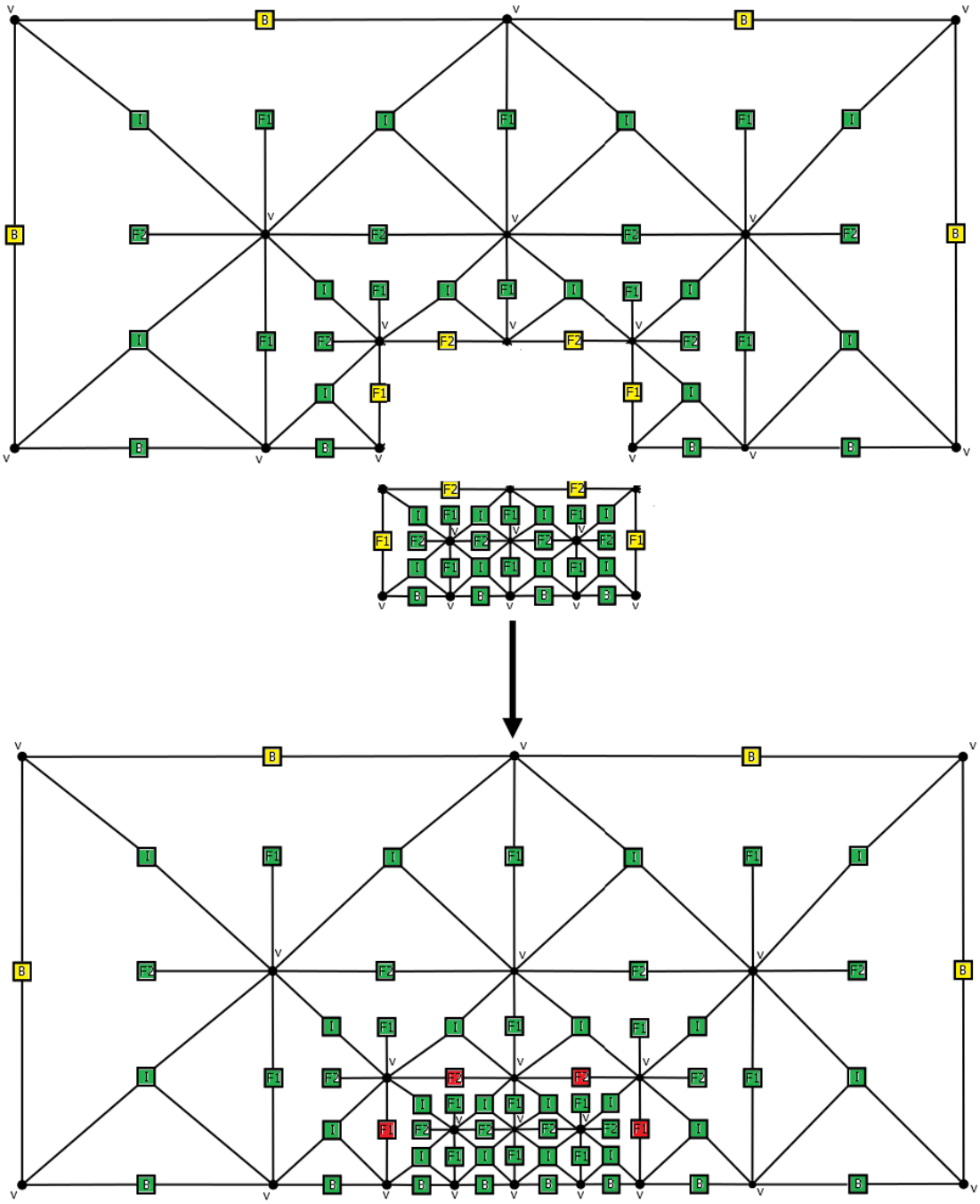


Figure 3.71: Production $P_{mergetop}$ for merging of the bottom and top layers, resulting in a top problem

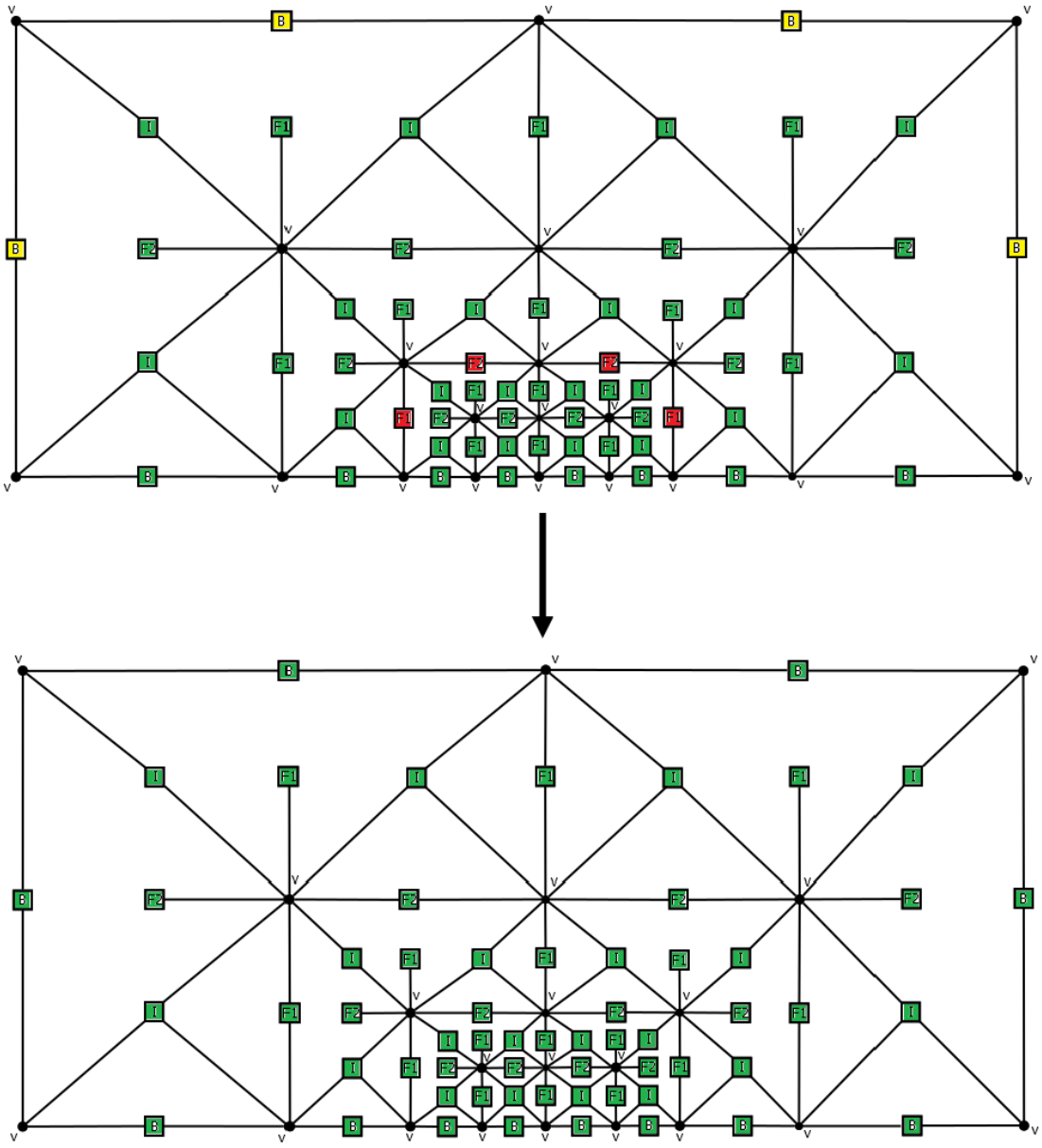


Figure 3.72: Production $P_{solvetop}$ for solving the top problem

3.3. Estimation of the computational cost and memory usage

This section aims to estimate computational costs and memory usages of the hypergraph grammar driven solvers presented in the previous sections. This means proving linear computational cost for the sequential solver and logarithmic computational cost for the parallel solver. The proofs are conducted by counting the number of operations incurred by the proposed algorithms and expressing it as a function of the number of the degrees of freedom (unknowns). This is done only for the case of the two element mesh with one point singularity used previously for derivation of the graph grammars. However, it is easy to show that different element configurations with a single point singularity differ only by a constant factor in terms of the computational cost and memory usage. The crucial observation is that in the case of a point singularity, each additional level of refinement adds a constant number of new elements. This observation, however, is not necessarily true for other types of singularities, such as edge singularities. Moreover, the linear cost still holds true for multiple (arbitrary, yet constant) numbers of point singularities, since we can divide the problem into s subproblems, one singularity each. This reasoning is, in fact, applied in Chapter 5. Rough estimates of the computational cost and memory usage for the three dimensional case have been placed in Sections 4.4 and 4.5. Free software called Maxima[1], for the manipulation of symbolic and numerical expressions, was used to evaluate the polynomials in the proofs.

3.3.1. Estimation of the computational cost of the hypergraph grammar based sequential solver

In this section I estimate the total number of operations for the graph grammar based sequential solver browsing the computational mesh in level-by-level order, aggregating the element frontal matrices into a single frontal matrix and eliminating the fully assembled degrees of freedom as quickly as possible. This means the order exactly as prescribed by the graph grammars in Section 3.2.2.
In order to prove that the computational complexity is $O(N)$, where N denotes the total number of unknowns, first, in Lemma 3.3.1, we express the exact computational cost as a function of polynomial approximation level p and refinement level k.

Lemma 3.3.1. *The number of operations incurred by solving the adaptive problem with a point singularity for a two element mesh using a hypergraph grammar driven solver is equal to:*

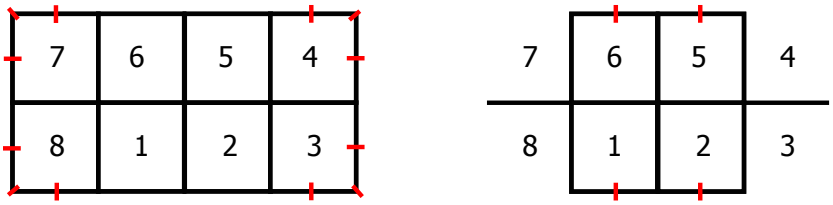
$$T(p,k) = \frac{16p^6 + 96p^5 + 264p^4 + 864p^3 + 533p^2 + 93p}{6} + k\frac{12p^6 + 72p^5 + 198p^4 + 1558p^3 - 441p^2 - 277p - 24}{6}$$

where k stands for the number of h refinement levels and p is the global polynomial order of approximation. Parameter p is assumed to be uniform, yet arbitrary, over the entire mesh.

Proof. First, we refer back to Figure 2.16 and Equation 2.8 which indicate the cost of partial elimination of an element matrix in a given state. With Equation 2.8 it is possible to evaluate the cost of each step of the elimination process. Counting a and b is even easier, since they correspond to colors (states) used in hypergraph productions. Using this nomenclature, a can be defined as count of nodes marked in green in the RHS of the production, whereas b is the count of nodes marked in yellow in the LHS of the production.
The proof can be split into two independent steps. First, we compute the constant part associated with the elimination of the initial, unrefined mesh ($k = 0$) and then the count of operations incurred by the each increase of k by one. This is sufficient, since each layer produced as a result of incrementing k is identical and contains six new elements.

[1] http://maxima.sourceforge.net

Step 1. $k = 0$. We begin by computing the cost of eliminating the fully assembled nodes. This process is called *static condensation*. To eliminate entries produced by these nodes we do not need to know about any other degrees of freedom. Static condensation usually occurs just after attaching an element to the given structure (as in Production $P_{elimint}$). In case of a hypergraphs from Section 3.2, static condensation is illustrated by productions in Figures 3.14, 3.15, 3.20, 3.21. As such entries are self-contained and can be processed independently, we can count their contributions to the total number of operations independently. For each element 1, 2, 5, 6 (see Figure 3.73a) the interior, two edges and a vertex are fully assembled and can be eliminated immediately (or at any time). For each element 3, 4, 7, 8 (see Figure 3.73b) interior and only one edge are fully assembled. The number of



(a) SC for elements 3, 4, 7, 8 (b) SC for elements 1, 2, 5, 6

Figure 3.73: Static condensation for $k = 0$ two element mesh

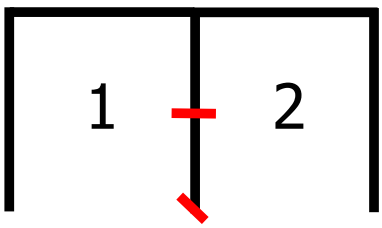
operations incurred by each of two cases above has been summarized in Table 3.1. Column $TOTAL$ is the sum of contributions for each of the elements in the Table 3.1 (which means multiplying the value in the first row by four and adding it to the value in the second row, also multiplied by four). The next step is to count the operations

Table 3.1: Operations incurred by static condensation for $k = 0$

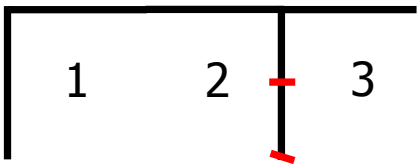
Elements	a for arbitrary p	b for arbitrary p	$C(a,b)$
3, 4, 7, 8	$1+2(p-1)+(p-1)^2$	$4+4(p-1)+(p-1)^2$	$\frac{2p^6+12p^5+33p^4+36p^3+13p^2}{6}$
1, 2, 4, 6	$p-1+(p-1)^2$	$4(p-1)+(p-1)^2$	$\frac{2p^6+12p^5+33p^4+2p^3-32p^2-13p}{6}$
TOTAL			$\frac{16p^6+96p^5+264p^4+136p^3-19p^2-13p}{6}$

incurred by the interface elimination, still for the $k = 0$ two element grid. This time, we need to obey the proper order for the elimination, starting from what is operationally called *Interface 1-2* and reflected by the Production $P_{elimcommon}$. In general, during this step we follow the chain of productions from Phase 2 described earlier in this chapter. This order is also concisely summarized by Figure 3.74. The results are presented in Table 3.2. Since in this step we do not eliminate contributions coming from interior nodes (all of them were eliminated in the previous step), p in the results appears only in its third power. Now we have finished the first step of computations and received the constant part for $k = 0$, which can be expressed as $T(p, 0)$ in Equation 3.2.

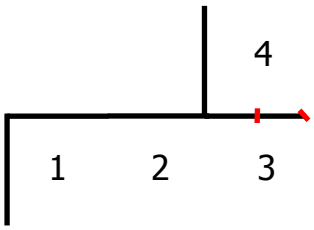
$$\begin{aligned} T(p,0) &= \frac{728p^3+552p^2+106p+16p^6+96p^5+264p^4+136p^3-19p^2-13p}{6} \\ &= \frac{16p^6+96p^5+264p^4+864p^3+533p^2+93p}{6} \end{aligned} \tag{3.2}$$



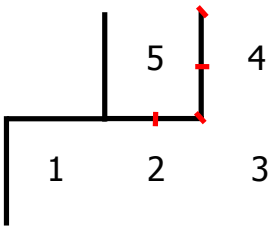
(a) Interface 1-2



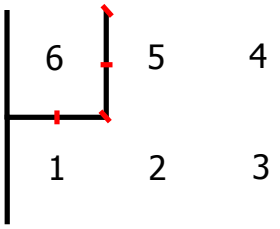
(b) Interface 2-3



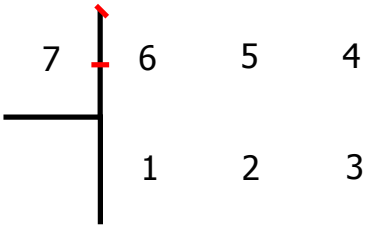
(c) Interface 3-4



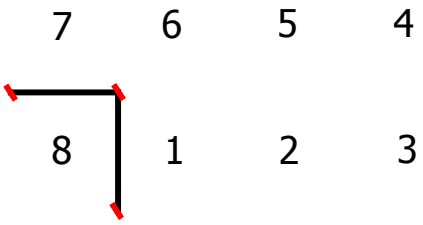
(d) Interface 4-5



(e) Interface 5-6



(f) Interface 6-7



(g) Interface 7-8

Figure 3.74: Interface elimination order, $k = 0$

Table 3.2: Computational cost incurred by element elimination for $k = 0$

Interface	a for arbitrary p	b for arbitrary p	$C(a,b)$
1-2	$1+p-1$	$6+5(p-1)$	$\frac{122p^3+81p^2+13p}{6}$
2-3	$1+p-1$	$6+5(p-1)$	$\frac{122p^3+81p^2+13p}{6}$
3-4	$1+p-1$	$6+5(p-1)$	$\frac{122p^3+81p^2+13p}{6}$
4-5	$2+2(p-1)$	$6+5(p-1)$	$\frac{196p^3+144p^2+26p}{6}$
5-6	$2+2(p-1)$	$5+4(p-1)$	$\frac{112p^3+108p^2+26p}{6}$
6-7	$1+(p-1)$	$4+3(p-1)$	$\frac{38p^3+45p^2+13p}{6}$
7-8	$3+2(p-1)$	0	$\frac{16p^3+12p^2+2p}{6}$
TOTAL			$\frac{728p^3+552p^2+106p}{6}$

Step 2. The second step of the proof is to compute the linear increase of the operation count depending on k. Increasing k by one adds one more layer of elements as presented in Figure 3.75. Again, for simplicity, we introduce temporary naming for the consecutive interfaces (Interface 1-2 to Interface 7-8) that will be processed. Let $k \geq 1$.

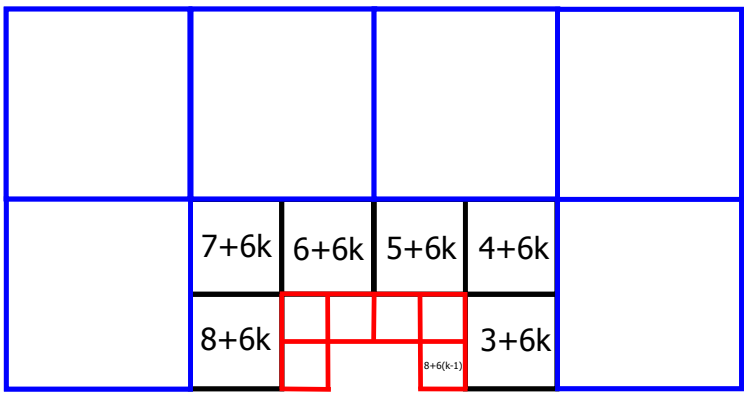
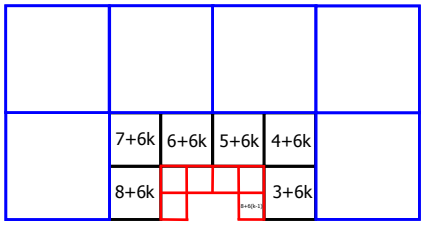
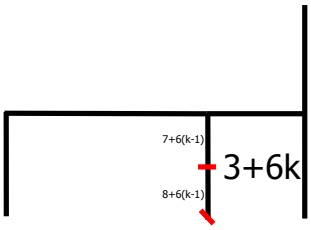


Figure 3.75: Ordering for arbitrary k

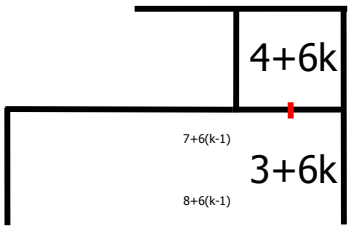
As in Step 1, we begin with the contribution of static condensation. In the case of elements $3+6k$ and $8+6k$ we can eliminate for each one edge and its interior independently. For the remaining elements the elimination of their interiors is the only possibility at this point. The results has been summarized in Table 3.3. Again, $TOTAL$ indicates first row multiplied by two plus the second row multiplied by four. In terms of eliminating interfaces, as in Step 1, we have to follow the order, but the direction (left to right or right to left) is not relevant to the result, so we may assume that we are proceeding from right to left. For a summary of the results, see Table 3.4. Having all



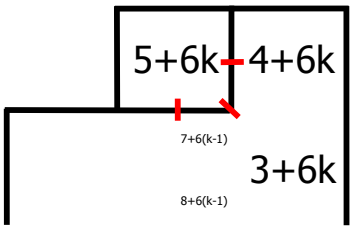
(a) Ordering within a layer



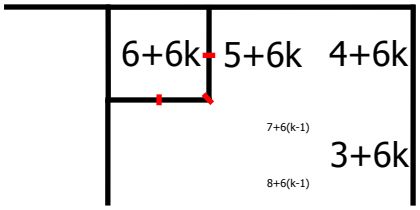
(b) Interface 1-2



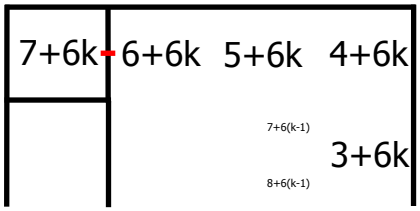
(c) Interface 2-3



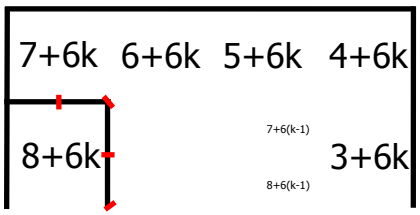
(d) Interface 3-4



(e) Interface 4-5



(f) Interface 5-6



(g) Interface 7-8

Figure 3.76: Interface elimination order, arbitrary k

Table 3.3: Computational cost incurred by static condensation for arbitrary k

Elements	a for arbitrary p	b for arbitrary p	$C(a,b)$
3, 8	$(p-1)+(p-1)^2$	$4+4(p-1)+(p-1)^2$	$\frac{2p^6+12p^5+33p^4-2p^3-32p^2-13p}{6}$
4, 5, 6, 7	$(p-1)^2$	$4+4(p-1)+(p-1)^2$	$\frac{2p^6+12p^5+33p^4-76p^3+p^2+22p+6}{6}$
TOTAL			$\frac{12p^6+72p^5+198p^4-308p^3-60p^2+62p+24}{6}$

Table 3.4: Computational cost incurred by interface elimination for arbitrary k

Interface	a for arbitrary p	b for arbitrary p	$C(a,b)$
1-2	$1+(p-1)$	$7+6(p-1)$	$\frac{182p^3+99p^2+13p}{6}$
2-3	$(p-1)$	$7+7(p-1)$	$\frac{254p^3-177p^2-71p-6}{6}$
3-4	$2(p-1)$	$7+7(p-1)$	$\frac{436p^3-228p^2-178p-30}{6}$
4-5	$1+2(p-1)$	$7+7(p-1)$	$\frac{436p^3-78p^2-58p-6}{6}$
5-6	$(p-1)$	$7+7(p-1)$	$\frac{254p^3-177p^2-71p-6}{6}$
6-7	$2+2(p-1)$	$7+6(p-1)$	$\frac{304p^3+180p^2+26p}{6}$
TOTAL			$\frac{1866p^3-381p^2-339p-48}{6}$

the terms, it is now possible to assemble the final formula:

$$T(p,k) = T(k,0) + k\frac{12p^6+72p^5+198p^4+1558p^3-441p^2-277p-24}{6} \tag{3.3}$$

Thus:

$$\begin{aligned} T(p,k) = {} & \frac{16p^6+96p^5+264p^4+864p^3+533p^2+93p}{6} \\ & + k\frac{12p^6+72p^5+198p^4+1558p^3-441p^2-277p-24}{6} \end{aligned} \tag{3.4}$$

which completes the proof. □

Having Lemma 3.3.1 proven and using the formula 3.4 it is possible to count the exact number of operations for the linear solver for different numbers of levels and various polynomial orders of approximation (see Table 3.5).

Table 3.5: Theoretical cost estimates for two element mesh for the sequential linear solver

p/*k*	2	3	4	5	$T(p,k)$
2	27384	40638	53892	67146	876 + 13254k
3	93452	137634	181816	225998	5088 + 44182k
4	259226	378987	498748	618509	19704 + 119761k
5	622804	905357	1187910	1470463	57698 + 282553k

Now, we can simplify the reasoning above into more rough and general estimations of the computational complexity.

Lemma 3.3.2. *The computational complexity of the hypergraph grammar driven solver for problems with point singularities is equal to $O(N)$, where N is the number of unknowns.*

Proof. The proof relies on Lemma 3.3.1 and transforms $T(p,k)$ into a function of N. In order to achieve this goal, we first need to determine the relationship between the number of elements N_e and the number of redinement levels k. The initial mesh for $k=0$ has eight elements. Each new refinement level adds a layer of six new elements. Thus:

$$N_e(k) = 8 + 6k \tag{3.5}$$

Since each element contains 4 degrees of freedom associated with its vertex, $4(p-1)$ degrees of freedom associated with its edges and $(p-1)^2$ degrees of freedom associated with its interior, we multiply number of elements and number of degrees of freedom per element.

$$N(N_e,p) = (8+6k)((p-1)^2 + 4(p-1) + 4) = O(p^2 + kp^2) \tag{3.6}$$

Substituting this result in $T(p,k)$ we receive:

$$T(p,N) = O(Np^4) \tag{3.7}$$

Since p can be treated as a constant and in fact, due to numerical side-effects is very rarely greater than nine, we receive:

$$T(N) = O(N) \tag{3.8}$$

which proves the linear complexity. □

3.3.2. Memory usage of the hypergraph grammar based sequential solver

The order of the memory usage of the hypergraph solver can be well-approximated by computing the count of non-zero entries in the matrix. Since we do not store zero values, the real space usage is the function of non-zero entries (NZ). NZ at any given stage can be expressed as a function of previously defined a and b, as defined by the following formula.

$$NZ(a,b) = \frac{a(a+1)}{2} + a(b-a) \tag{3.9}$$

We will compute $NZ(k,p)$ with k and p being defined the same way as in the previous proof.

Lemma 3.3.3. *The order of memory usage for a two element mesh with a point singularity using the hypergraph grammar driven solver is equal to:*

$$MEM(N) = O(N)$$

where N is the total number of the degrees of freedom.

Proof. Similarly to the previous section, the proof consists of two steps. One for the constant part ($k=0$) and one for estimating the number of non-zero entries introduced by each additional layer.

Step 1. Compute the constant part of the cost ($k=0$). As in Section 3.3.1, we start with computing the part that does not depend on k. Contribution to the non-zero entries coming from the static condensation is presented in Table 3.6. The contributions coming from the interface elimination are summed up in Table 3.7. The logic is exactly the same as for the tables in Section 3.3.1.

Table 3.6: Count of non-zero entries incurred by static condensation for $k = 0$

Elements	a for arbitrary p	b for arbitrary p	NZ for arbitrary p
3, 4, 7, 8	$1 + 2(p-1) + (p-1)^2$	$4 + 4(p-1) + (p-1)^2$	$\frac{1}{2}(p^4 + 4p^3 + 3p^2)$
1, 2, 5, 6	$p - 1 + (p-1)^2$	$4 + 4(p-1) + (p-1)^2$	$\frac{1}{2}(p^4 + 4p^3 - 9p^2 + 2p + 1)$
TOTAL			$6p^4 + 24p^3 - 30p^2 + 8p + 4$

Table 3.7: Non-zero entries incurred by interface elimination for $k = 0$

Interface	a for arbitrary p	b for arbitrary p	NZ for arbitrary p
1-2	$1 + p - 1$	$6 + 5(p-1)$	$\frac{1}{2}(3p(3p+1))$
2-3	$1 + p - 1$	$6 + 5(p-1)$	$\frac{1}{2}(3p(3p+1))$
3-4	$1 + p - 1$	$6 + 5(p-1)$	$\frac{1}{2}(3p(3p+1))$
4-5	$2 + 2(p-1)$	$6 + 5(p-1)$	$p(8p+3)$
5-6	$2 + 2(p-1)$	$5 + 4(p-1)$	$p(6p+3)$
6-7	$1 + (p-1)$	$4 + 3(p-1)$	$\frac{1}{2}(p(5p+3))$
7-8	$3 + 2(p-1)$	0	$(p+1)(2p+1)$
TOTAL			$32p^2 + 15p + 1$

Table 3.8: Count of non-zero entries incurred by static condensation for arbitrary k

Elements	a for arbitrary p	b for arbitrary p	NZ for arbitrary p
1,6	$(p-1) + (p-1)^2$	$4 + 4(p-1) + (p-1)^2$	$\frac{1}{2}(p^4 + 4p^3 - 2p^2 - 3p)$
2,3,4,5	$(p-1)^2$	$4 + 4(p-1) + (p-1)^2$	$\frac{1}{2}(p^4 + 4p^3 - 9p^2 + 2p + 2)$
TOTAL			$3p^4 + 12p^3 - 20p^2 + p + 4$

Table 3.9: Non-zero entries incurred by interface elimination for arbitrary k

Interface	a for arbitrary p	b for arbitrary p	NZ for arbitrary p
1-2	$1 + (p-1)$	$7 + 6(p-1)$	$\frac{1}{2}p(11p + 3p)$
2-3	$(p-1)$	$7 + 7(p-1)$	$\frac{1}{2}(13p^2 - 11p - 2)$
3-4	$1 + 2(p-1)$	$7 + 7(p-1)$	$12p^2 - 4p - 1$
4-5	$1 + 2(p-1)$	$7 + 7(p-1)$	$12p^2 - 4p - 1$
5-6	$(p-1)$	$7 + 7(p-1)$	$\frac{1}{2}(13p^2 - 11p - 2)$
6-7	$2 + 2(p-1)$	$7 + 6(p-1)$	$10p^2 + 3p$
TOTAL			$105p^2 - 29p - 8$

Step 2. Count the NZ increase with k. This is the analogous step to Step 2 in the proof of linear computational cost. Also the interface numbering corresponds to that from Figure 3.76. The results are presented in Table 3.8. The corresponding results for the interface elimination step are presented in Table 3.9.

When we assemble the above results into a single equation we receive:

$$\begin{aligned} MEM(k,p) = NZ(k,p) &= \frac{1}{2}(12p^4 + 48p^3 + 11p^2 + 35p + 4) + k\frac{1}{2}(105p^2 - 29p - 8) \\ &= O(p^4 + kp^2) \end{aligned} \tag{3.10}$$

We also already know that:

$$N_e(k) = 8 + 6k \tag{3.11}$$

And the relationship between N, N_e and p is as below:

$$N(N_e, p) = (8 + 6k)((p-1)^2 + 4(p-1) + 4) = O(p^2 + kp^2) \tag{3.12}$$

Since p^2 does not depend on N and kp^2 was proven to be $O(N)$ in Lemma 3.3.1, we obtain:

$$MEM(N) = O(N) \tag{3.13}$$

which completes the proof. □

3.3.3. Estimation of the computational cost of the hypergraph grammar based parallel solver

As discussed in Section 3.2.3, in the case of a parallel solver each layer can be processed concurrently. The aim of this section is to prove Lemma 3.3.4.

Lemma 3.3.4. *The computational complexity of the hypergraph grammar driven solver is logarithmic with the respect to number of unknowns N on shared memory parallel machines.*

Proof. We assume t_{L_i} is the computational cost associated with processing layer i. However, the size of each layer is fixed, so this cost is also constant provided that P is fixed. The bottom layer L_1 consists of two elements and the cost of its elimination is always equal to $C_1(p)$.

$$t_{L_1} = C_1(p) \tag{3.14}$$

In order to determine $C_1(p)$ we have to sum up the first two rows from Table 3.1 with the first two rows from Table 3.2.

$$\begin{aligned} C_1(p) &= \frac{122p^3 + 81p^2 + 13p}{3} + \frac{2p^6 + 12p^5 + 33p^4}{3} + \frac{36p^3 + 13p^2}{3} \\ &= \frac{2p^6 + 12p^5 + 33p^4 + 158p^3 + 94p^2 + 13p}{3} \end{aligned} \tag{3.15}$$

Similarly, for each of the layers $L_2, ..., L_{k-1}$ the cost is also equal to some $C_2(p)$ (see Equation 3.16). We determine this in the similar manner as $C_1(p)$. In fact, its upper boundary value is expressed by the coefficient standing by k in Equation 3.3, which is constant with respect to N. Its actual value is even smaller since in the case of parallel solver we do not eliminate contributions coming from top and bottom nodes of each layer at this time.

$$t_{L_i} = C_2(p) \quad where \quad 1 < i < k \tag{3.16}$$

The top layer L_k has the same number of elements as layers $L_2, ..., L_{k-1}$, but the different cost, since its outer nodes lie on the boundary (see Equation 3.17). Nevertheless, it is also constant with respect to N.

$$t_{L_k} = C_3(p) \tag{3.17}$$

Provided that each layer is processed in parallel, the total cost of this step is equal to the maximum cost of processing of a single layer (as in Equation 3.18).

$$t_{layers} = max(C_1(p), C_2(p), C_3(p)) \tag{3.18}$$

As mentioned before, we assume p to be constant over the entire grid and t_{layers} is independent of k (and, as a result, of N). The next step is to merge each layer following the binary tree order. We denote $t_{interface}$ as the cost of eliminating (and merging) degrees of freedom lying on the interface between any two layers. Since there are $k+2$ layers, the total cost of this part is:

$$t_{merger} = t_{interface} log_2(k+2) \tag{3.19}$$

This is due to the fact that we process the neighboring layers pairwise, producing the binary tree of height $log_2(k+2)$. $t_{interface}$ is constant for each two layers and a given p. It can be computed based on $P_{mergetop}$ (see Figure 3.67) leveraging the previously used formula for $C(a,b)$.

$$a_{mt} = 5 + 4(p-1) \tag{3.20}$$

$$b_{mt} = 15 + 12(p-1) \tag{3.21}$$

and hence

$$C(a_{mt}, b_{mt}) = \frac{2432p^3 + 2064p^2 + 580p + 54}{6} = O(p^3) \tag{3.22}$$

Substituting these values in Equation 3.19 we receive:

$$T(N) = max(C1(p), C2(p), C3(p)) + p^3 log_2(\frac{N}{p^2}) \tag{3.23}$$

Assuming that p is constant over the entire domain we receive the final formula:

$$T(N) = O(1) + O(logN) = O(logN) \tag{3.24}$$

which completes the proof. □

Chapter 4

Graph grammar based model of a solver algorithm in three dimensions

The goal of this chapter is to present the extension of the hypergraph grammar model to the process of generation and adaptation of three dimensional grids, followed by a description of hypergraph grammar productions driving the execution of the multi-frontal solver algorithm in a special order, which allows for faster processing. The reasoning is similar to the previous chapter, but due to a very large number of possible productions, we introduce only most important ones.

4.1. Hypergraph grammar model for generation and adaptation of a three dimensional mesh with point singularities

The process of generation of the three dimensional computational mesh with hexahedral elements starts with execution of the P_{init} production, presented in Figure 4.1. It generates a hypergraph representing a single three-dimensional finite element. In the case of uniform mesh adaptations, we can prepare a sequence of graph grammar productions replacing the single element by a uniform cluster of elements. The model production $P_{initbreak}$ presented in Figure 4.2 generates a uniform mesh of $2 \times 2 \times 2$ elements. In order to get non-uniform mesh refinements, we need to enforce the *1-irregularity rule* as described in Definition 2.1.1. The rule does not allow for breaking a single element for the second time without first breaking any adjacent elements. In order to enforce the *1-irregularity rule*, we must break element interiors first, as is expressed by production $P_{breakint}$ presented in Figure 4.3. The example execution of the production over the eight finite element mesh is presented in Figure 4.4. In the case of faces, a face can be broken only if two adjacent interiors have already been broken, or one adjacent interior has been broken and the face is located on the boundary of the mesh. This second case is illustrated in production $P_{breakface}$ presented in Figure 4.5. Finally, edges can be broken only if all adjacent faces have already been broken, or the edge is located on the boundary (and in such a scenario we need to check fewer of adjacent faces). This is illustrated in Figure 4.6 by production $P_{breakedge}$.

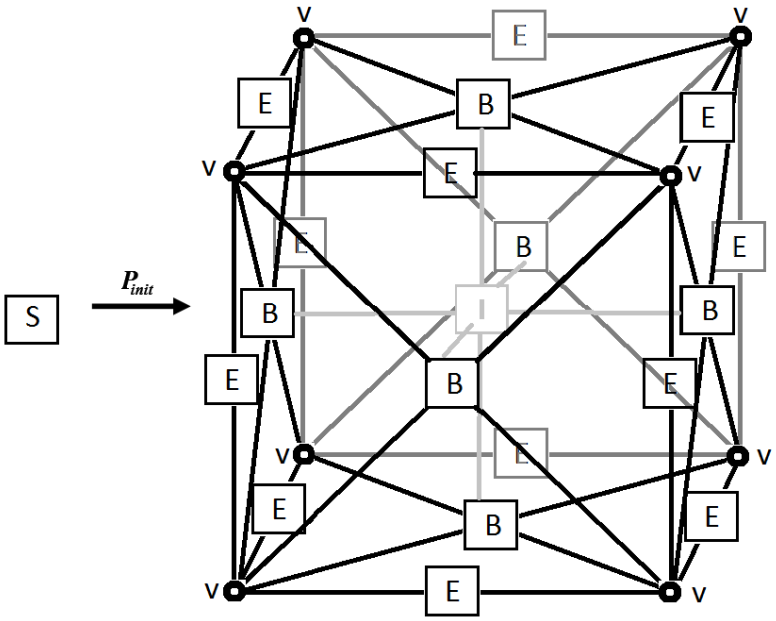


Figure 4.1: The initial production generating a single cubic element

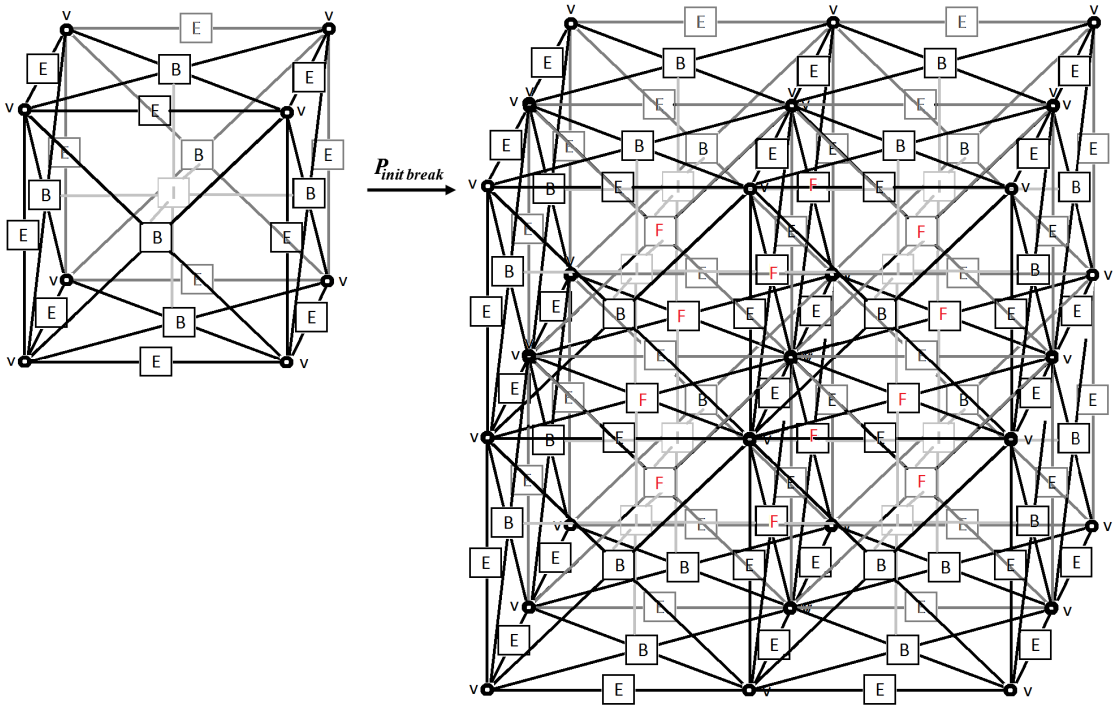


Figure 4.2: The production breaking a single element into eight elements

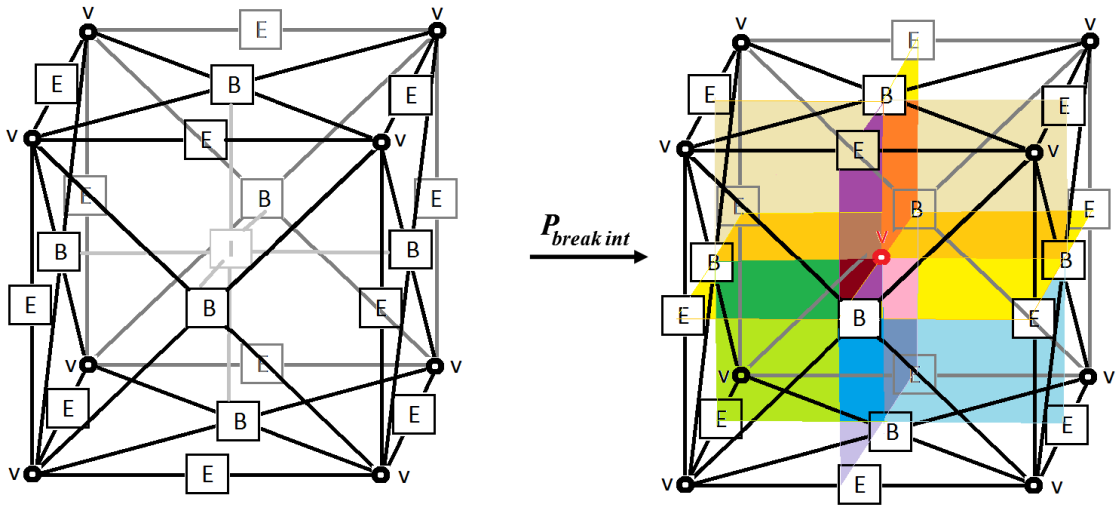


Figure 4.3: The production breaking an interior of a single element

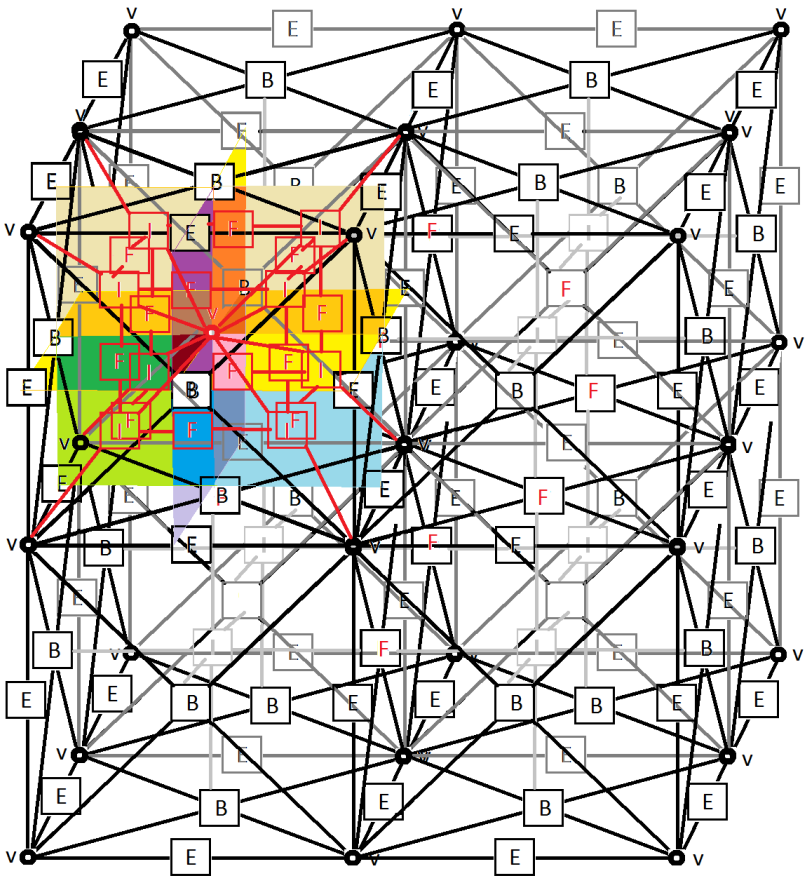


Figure 4.4: The eight element mesh after breaking the interior of the front element

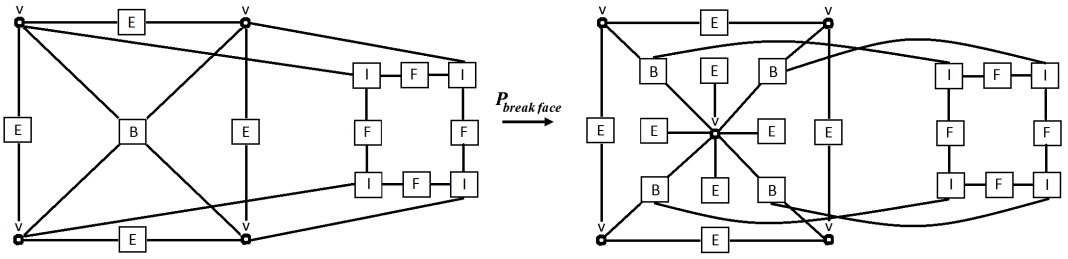


Figure 4.5: The production for breaking a face

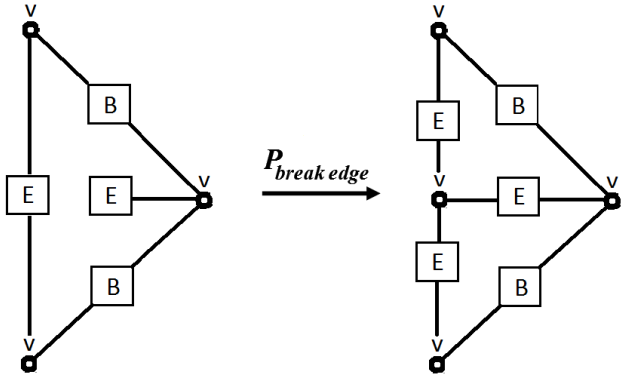


Figure 4.6: The production for breaking an edge

4.2. Multi-frontal solver algorithm prescribed by the hypergraph grammars

The first graph grammar productions are responsible for generation of the frontal element matrices. This is done by productions $P_{agregint}$ responsible for generation of the matrix entries associated with interior nodes, $P_{agregboundary}$ and $P_{agregface}$ responsible for generation of matrix entries associated with boundary and interior faces, $P_{agregedge}$ and $P_{agregvertex}$ responsible for generation of matrix entries associated with element edges and vertices. These graph grammar productions are illustrated in Figure 4.7.

Having assembled the frontal matrix, it is possible now to start with elimination of the fully assembled nodes. We can eliminate the interior node, which is denoted by production $P_{elimint}$ presented in Figure 4.8, we can also eliminate boundary faces, which is denoted by production $P_{elimface}$ as well as boundary edges and vertices, compare productions $P_{elimedge}$ and $P_{elimvertex}$ in Figure 4.8.

Having adjacent elements with frontal matrices and eliminated interior and boundary nodes, we can now merge the frontal matrices into one matrix and eliminate fully assembled nodes from the common face. It is expressed by productions $P_{mergeeliminate}$ illustrated in Figure 4.9. This procedure of merging frontal matrices and eliminating fully assembled nodes located on common faces is repeated until all the nodes in the mesh are eliminated.

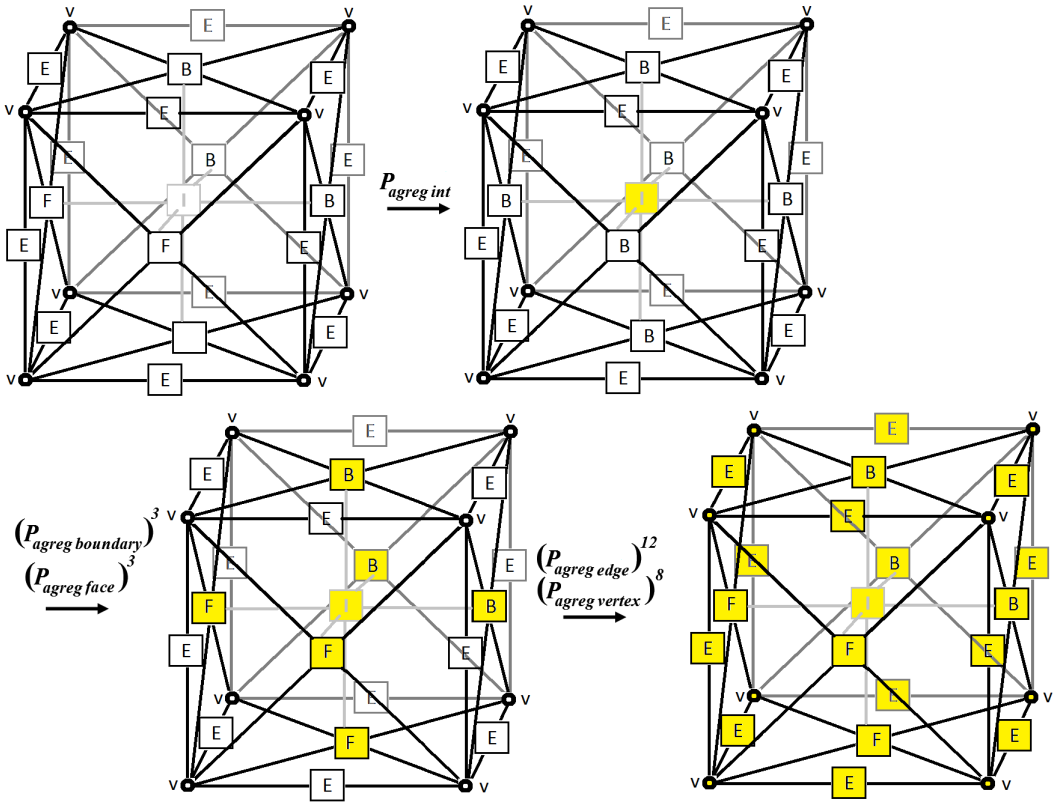


Figure 4.7: The productions for assembly of an element frontal matrix

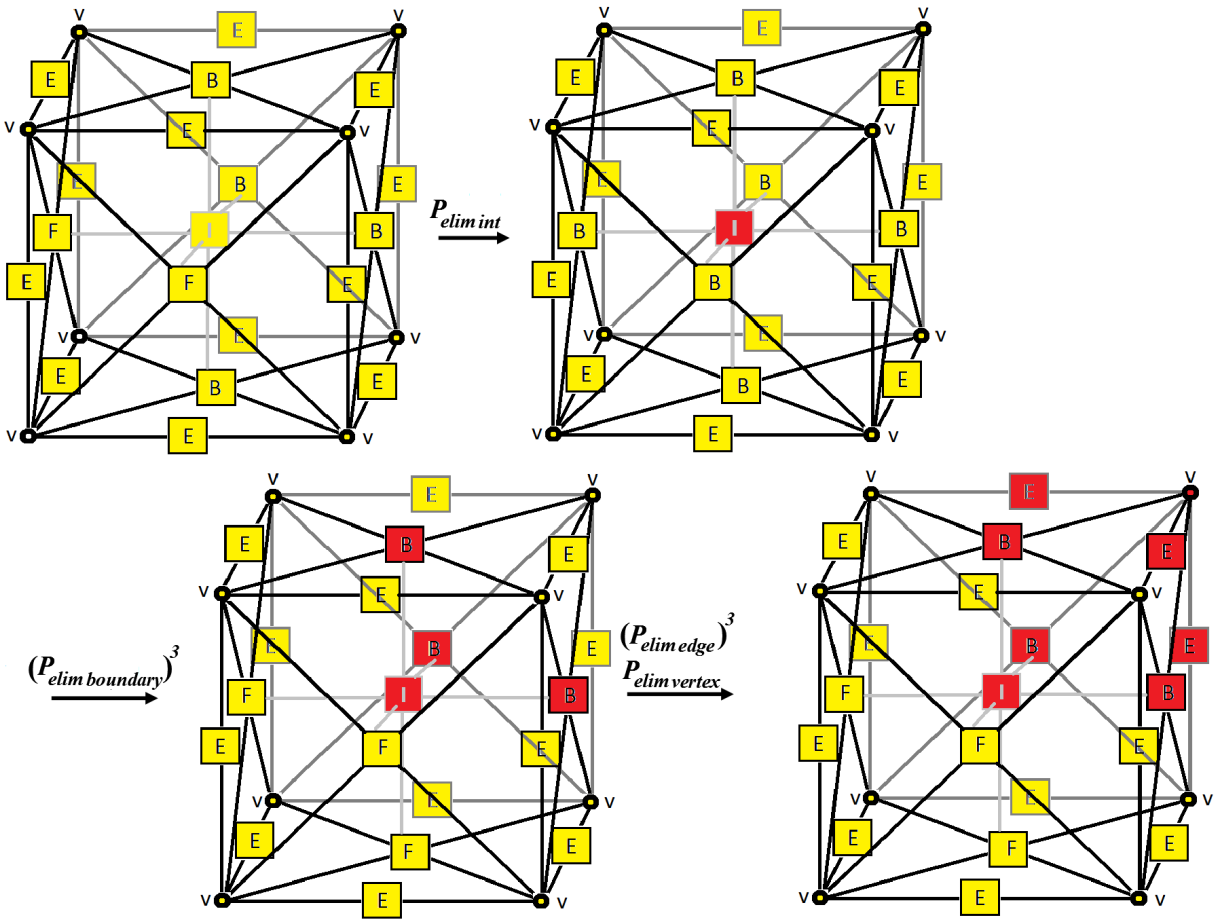


Figure 4.8: The production for elimination of an interior and boundary nodes

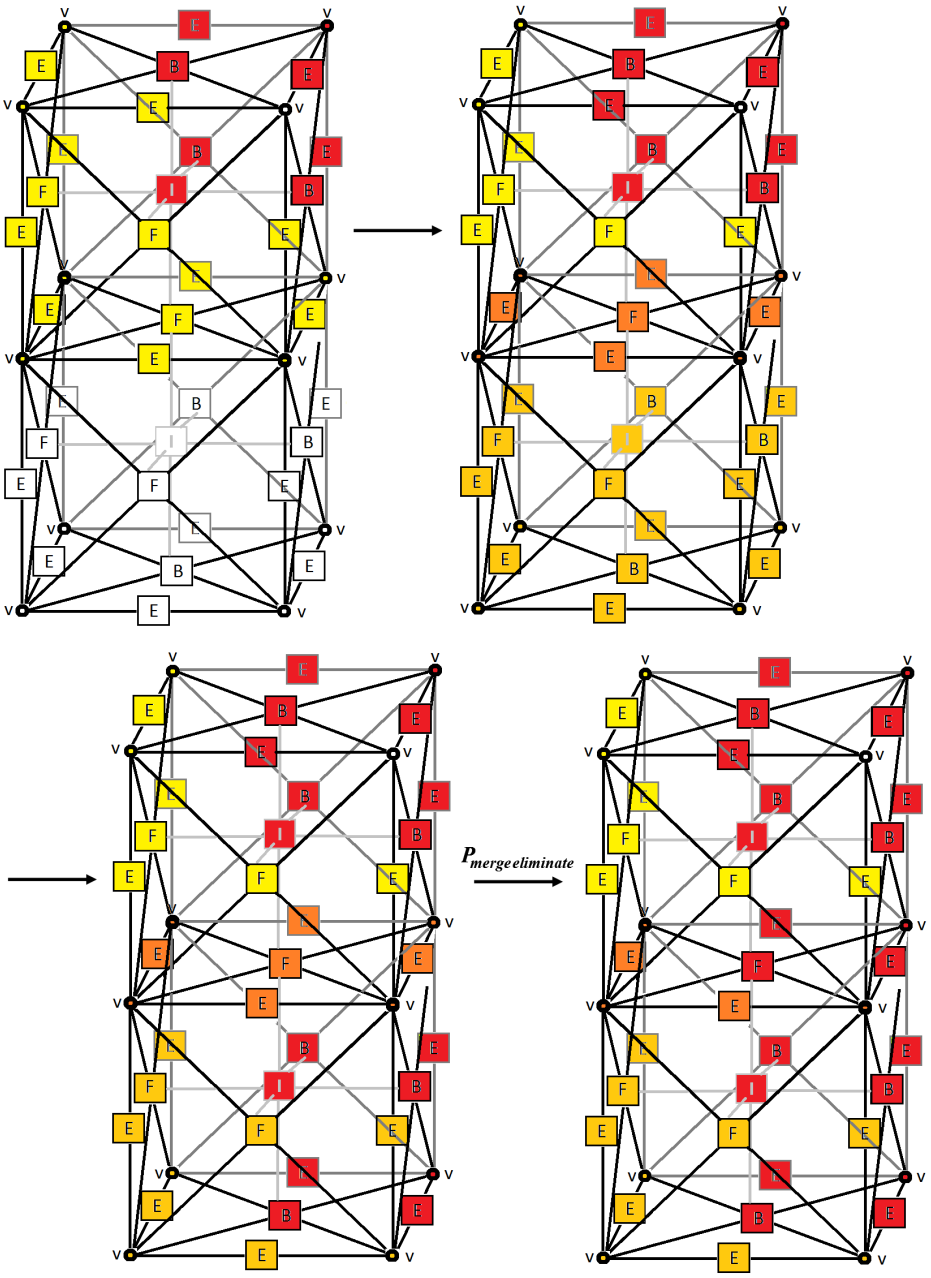


Figure 4.9: The productions for merging two frontal matrices and elimination of fully assembled nodes from a common face

4.3. Linear computational cost solver for three dimensional meshes with point singularities

In the case of a mesh with point singularities, as the one presented in Figure 4.10, we can take advantage of the multi-level structure of the computational grid in the following way. We start with the grid presented on the right panel in Figure 4.2. The first step is to execute the multi-frontal solver algorithm for all the elements, except the elements located closest to the point singularity.

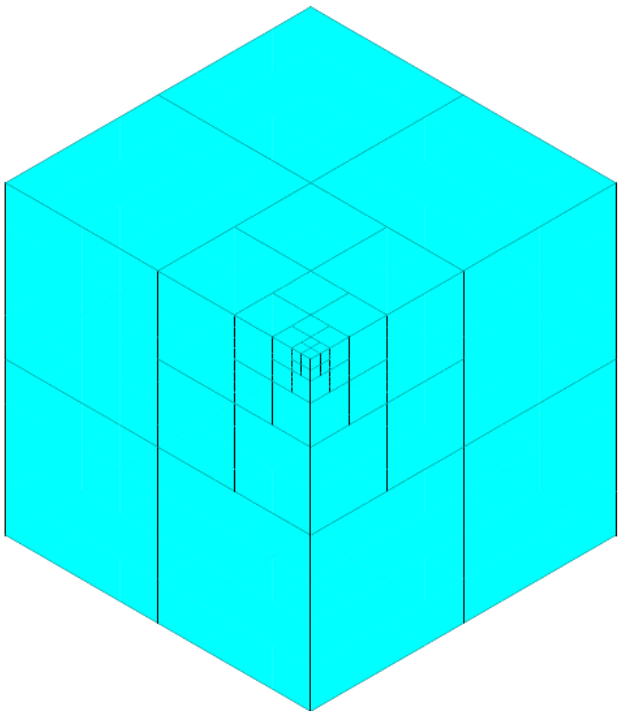


Figure 4.10: Three dimensional mesh with a single point singularity

In order to create element frontal matrices for all elements on the top level we execute the following chain of productions: $P_{agreginit} \mapsto P_{agregboundary} \mapsto P_{agregface} \mapsto P_{agregedge} \mapsto P_{agregvertex}$. The next step is to eliminate entries that resulted from the previously aggregated element contributions, which can be achieved by executing the following chain of productions: $P_{elimboundary} \mapsto P_{elimedge} \mapsto P_{elimvertex}$.

Finally, we merge frontal matrices, by executing productions $P_{mergeeliminate}$ as many times as necessary to end up with the interface of the top level with respect to the next level, as presented in Figure 4.11. We store the Schur complement matrix associated with the interface.

At this point, we can aggregate the frontal matrix associated with the element nearest to the point singularity, by executing productions $P_{agreginit} \mapsto P_{agregboundary} \mapsto P_{agregface} \mapsto P_{agregedge} \mapsto P_{agregvertex}$.

Next, we can merge the element frontal matrix with the Schur complement matrix, by executing production $P_{mergewithSchur}$. This results in a fully assembled matrix and we can solve the problem close the singularity.

If a more accurate solution is required, we can break the element neighboring the singularity by executing the production $P_{breaksingularity}$ presented in Figure 4.12, preserving the Schur complement adjacent to the broken element.

At this point, we can continue to solve the problem over the newly refined elements by executing the following chain of productions $P_{agreginit} \mapsto P_{agregboundary} \mapsto P_{agregface} \mapsto P_{agregedge} \mapsto P_{agregvertex}$ generating the element frontal matrices, followed by productions $P_{elimboundary} \mapsto P_{elimedge} \mapsto P_{elimvertex}$,

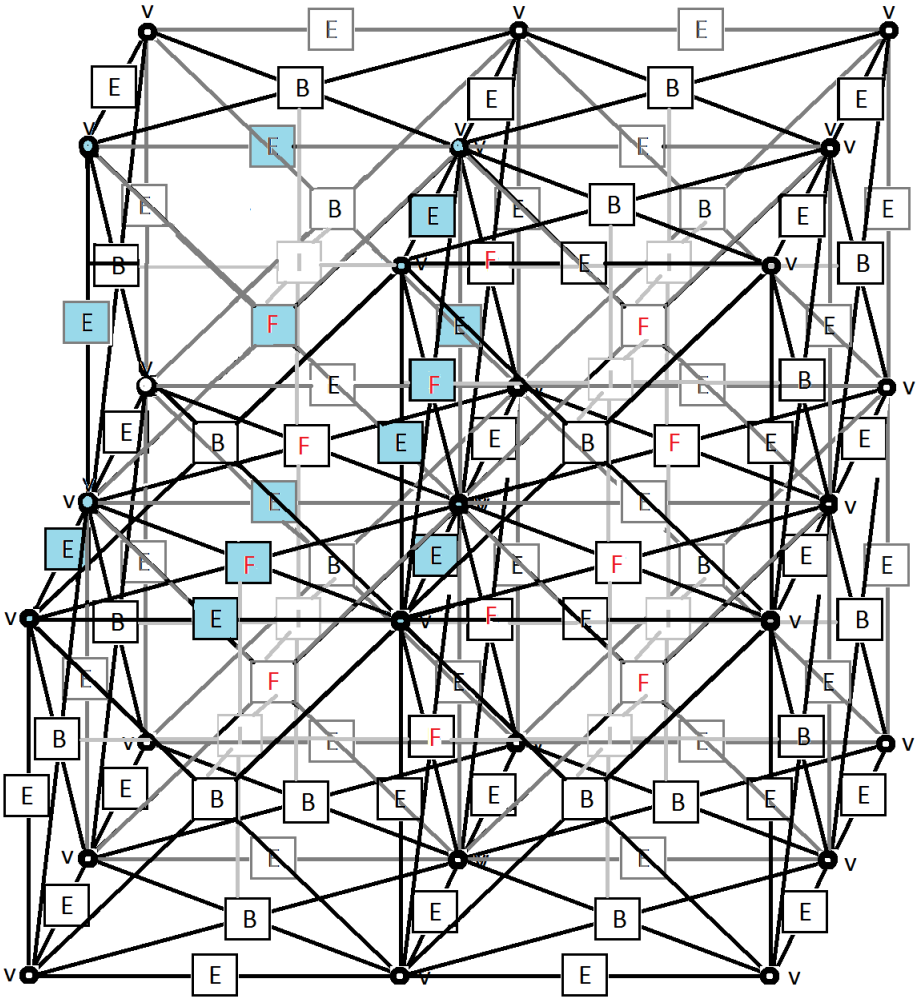


Figure 4.11: The interface between layers

together with $P_{mergewithSchur}$ reutilizing the Schur complement matrix for the interface with other part of the mesh.

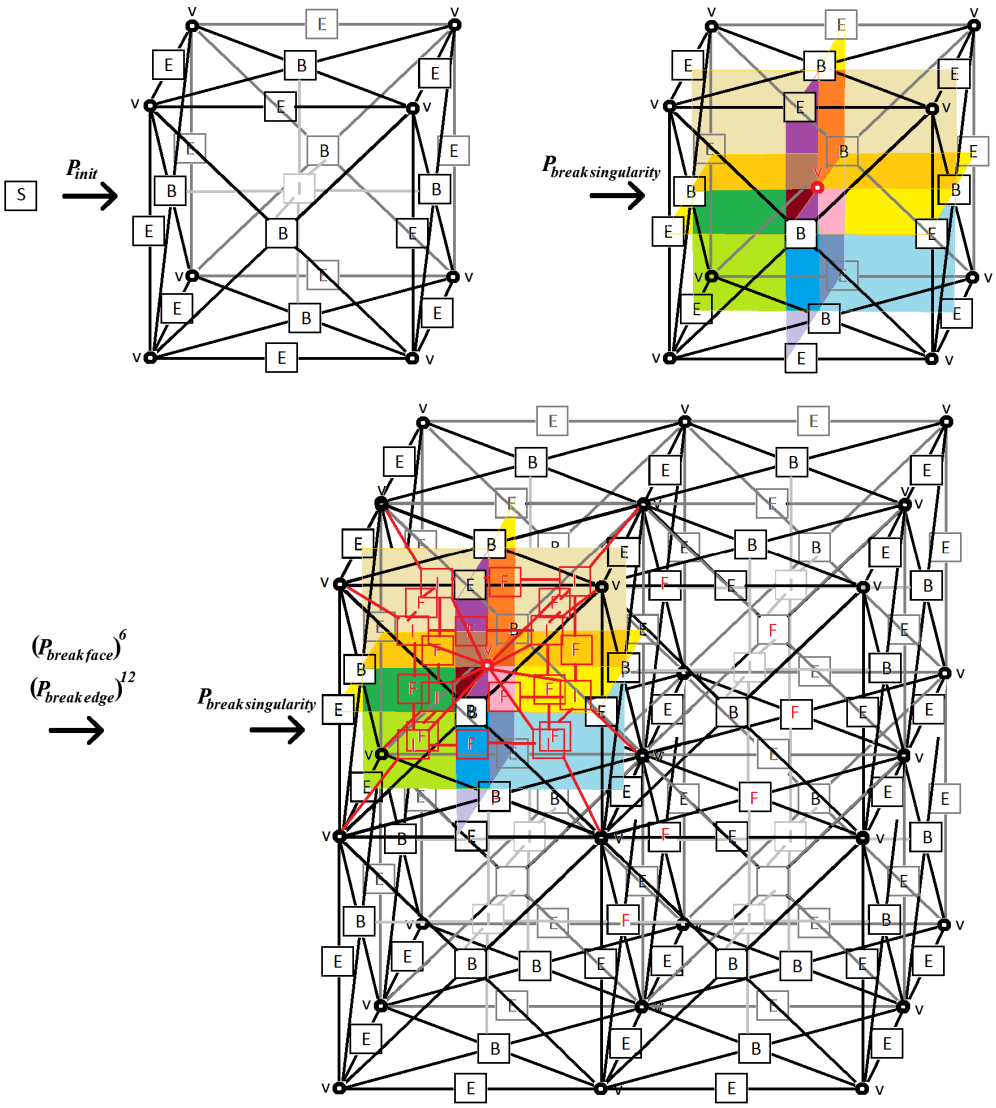


Figure 4.12: The hypergraph grammar derivation of a grid with point singularity

4.4. Computational complexity of sequential hypergraph grammar based solver for three dimensional meshes with point singularities

As an estimation of the exact computational cost for the three dimensional solver would be a very strenuous task, we restrict ourselves to a rough approximation of the computational complexity.

Lemma 4.4.1. *Computational complexity of the sequential solver with respect to the number of degrees of freedom N and polynomial order of approximation p for a three dimensional grid with point singularity is equal to $T(p,N) = O(Np^6)$.*

Proof. A three dimensional element has number of degrees of freedom over an element edge of the order of $O(p)$, the number of degrees of freedom over an element face of the order of $O(p^2)$ and the number of degrees of freedom over an element interior of the order of $O(p^3)$. The computational complexity of elimination of the interior-related degrees of freedom is of the order of $O((p+p^2+p^3)^2p^3) = O(p^9)$. The computational complexity of the static condensation is of the order of $O(N_e p^9)$, where N_e denotes the number of elements. The remaining faces and

edges are eliminated level by level (layer by layer), and the computational complexity of elimination of a single level is of the order of $O((p^2+p)^3) = O(p^6)$. The number of elements N_e is of the order of $O(N_e) = O(\frac{N}{p^3})$, and the number of levels k is of the order of $O(k) = O(\frac{N}{p^3})$ thus the total computational complexity is of the order of $O(N_e p^9 + kp^6) = O(Np^6 + Np^3) = O(Np^6)$ which completes the proof. □

4.5. Memory usage of hypergraph grammar based solver for three dimensional meshes with point singularities

Memory usage in case of the hypergraph grammar driven solver for the three dimensional meshes with point singularities remains linear with respect to the number of the degrees of freedom N. The order of memory usage is roughly estimated in the lemma below.

Lemma 4.5.1. *Memory usage of the solver with respect to the number of degrees of freedom N and polynomial order of approximation p for the three dimensional grid with point singularity is of the order of $MEM(N,p) = O(Np^3)$.*

Proof. A three dimensional element contains $O(p)$ degrees of freedom over element edges, $O(p^2)$ degrees of freedom over element faces and $O(p^3)$ degrees of freedom over element interiors. The memory usage complexity of storing an element frontal matrix is of the order of $O((p+p^2+p^3)^2) = O(p^6)$. The total memory usage complexity of storing element frontal matrices is of the order of $O(N_e p^6)$. The matrices at higher levels contain contributions from faces and edges and the memory usage complexity of storing such a matrix is of the order of $O((p^2+p)^2) = O(p^4)$. The number of elements N_e is of the order of $O(N_e) = O(\frac{N}{p^3})$, and the number of levels k is of the order of $O(k) = O(\frac{N}{p^3})$, thus the total memory usage complexity is of the order of $O(N_e p^6 + kp^3) = O(Np^6 + Np^3) = O(Np^3)$. □

4.6. Computational complexity of the parallel hypergraph grammar based solver for three dimensional meshes with point singularities

This section roughly estimates the theoretical computational complexity of the fastest version of the solvers presented in this thesis - parallel hypegraph grammar solver for three dimensional meshes with point singularities.

Lemma 4.6.1. *Computational complexity of the parallel solver with respect to the number of degrees of freedom N and polynomial order of approximation p for three dimensional grid with point singularity is of the order of $O(p^6 log(\frac{N}{p^3}))$.*

Proof. The three dimensional elements contains $O(p)$ degrees of freedom over element edges, $O(p^2)$ degrees of freedom over element faces and $O(p^3)$ degrees of freedom over element interiors. The computational complexity of elimination of interior degrees of freedom is of the order of $O(p^3(p+p^2+p^3)^2) = O(p^9)$. The remaining faces and edges are eliminated level by level (layer by layer), and the computational complexity of their elimination over a single level is of the order of $O((p^2+p)^3) = O(p^6)$. In parallel we construct an elimination tree with $O(log(k))$ levels, where k is the previously defined number of the refinement levels. In particular, the number of levels k is of the order of $O(k) = O(\frac{N}{p^3})$ and thus $O(log(k)) = O(log(\frac{N}{p^3}))$. This is why the computational complexity of the parallel solver is $O(p^9 + log(p^6 \frac{N}{p^3})) = O(p^6 log(\frac{N}{p^3}))$. This completes the proof. □

Chapter 5

Numerical results

This chapter presents numerical results for both sequential and parallel versions of the hypergraph grammar based solver. The sequential version has been benchmarked on a traditional CPU , whereas the parallel version has been implemented and tested on shared memory Graphics Processing Unit. The numerical results deliver the solution to an exemplary heat transfer problem described in Section 5.1.3. The solution has been computed by means of the h adaptive Finite Element Method (see Section 2.1.2). Since this book presents rather general mathematical formalism for prescribing solvers, the GPU solver described in this thesis is just one of the possible implementations. For performance reasons, graph grammar productions have not been mapped exactly as presented in Chapter 3. The left hand sides of the productions have been minimized and various purely technical implementation details have also been applied to facilitate better specifics of GPU architecture.

5.1. Problem statement

The grids with refinements towards a point, such as the ones presented in Figures 5.1 - 5.2 can be used for solving the numerical problems with point singularities. The examples of such problems are projection problems for functions having multiple point extrema or singularities, or the solution of the heat transfer problems with different material data varying by several orders of magnitude, or heat transfer problems with several point heat sources (compare Figure 5.5), or any other problem where the solution exhibits multiple point gradients.

5.1.1. Projection problem

In this section, we formulate the projection problem (for a detailed description see [18, 23, 24, 25, 53, 55, 99]) that will be solved using the hypergraph grammar driven solver implemented on a GPU. The goal is to find the approximation u to the given function f, defined over $\Omega \subset \mathbb{R}^2$, which satisfies Equation 5.1.

$$\|f - u\|_{L^2} \to \min \tag{5.1}$$

By applying the definition of the L_2 norm we receive:

$$\sqrt{\int_{\Omega} (f-u)^2 \mathrm{d}\mathbf{x}} \to \min \tag{5.2}$$

Minimizing the above formula is equivalent to:

$$\int_{\Omega} (f-u)^2 \, \mathrm{d}\mathbf{x} \to \min \tag{5.3}$$

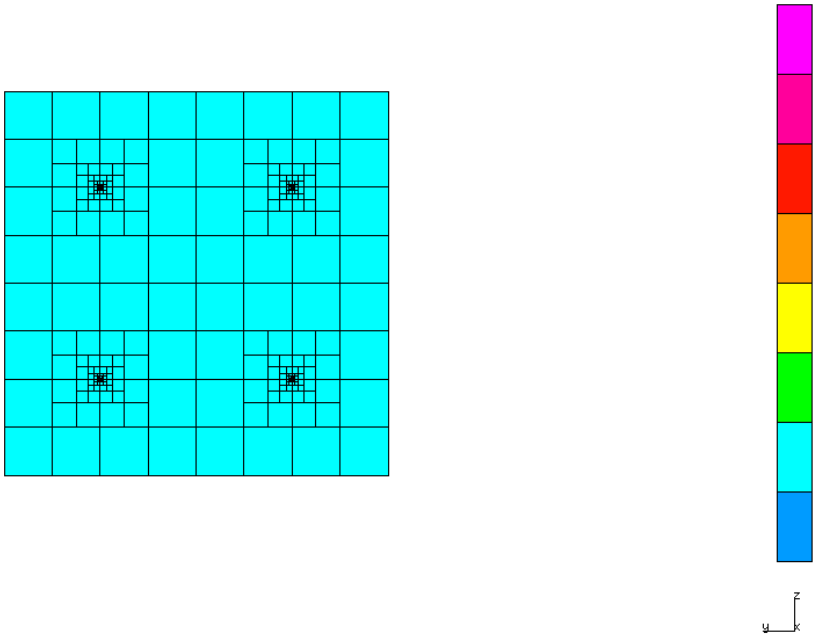


Figure 5.1: Exemplary two dimensional mesh

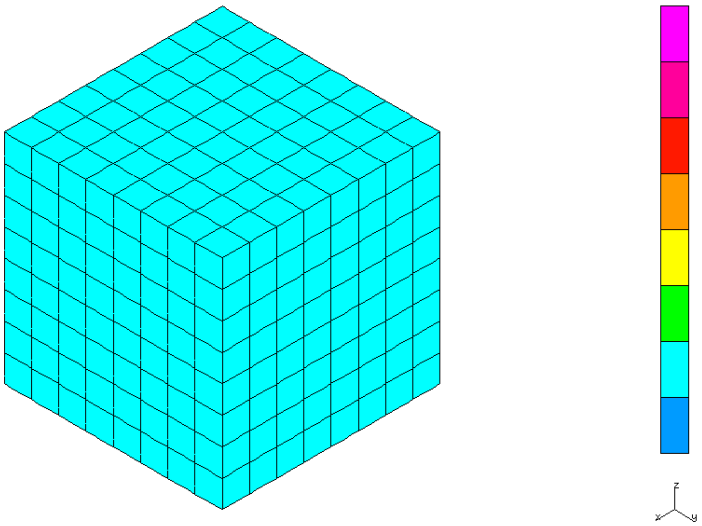


Figure 5.2: Exemplary three dimensional mesh

Hence:

$$\int_{\Omega} f^2 \mathrm{d}\mathbf{x} - 2 \int_{\Omega} f u \mathrm{d}\mathbf{x} + \int_{\Omega} u^2 \mathrm{d}\mathbf{x} \rightarrow \min \tag{5.4}$$

Since

$$\int_{\Omega} f^2 \mathrm{d}\mathbf{x} = \mathrm{const} \tag{5.5}$$

the problem can be reduced to:

$$-2 \int_{\Omega} f u \mathrm{d}\mathbf{x} + \int_{\Omega} u^2 \mathrm{d}\mathbf{x} \rightarrow \min \tag{5.6}$$

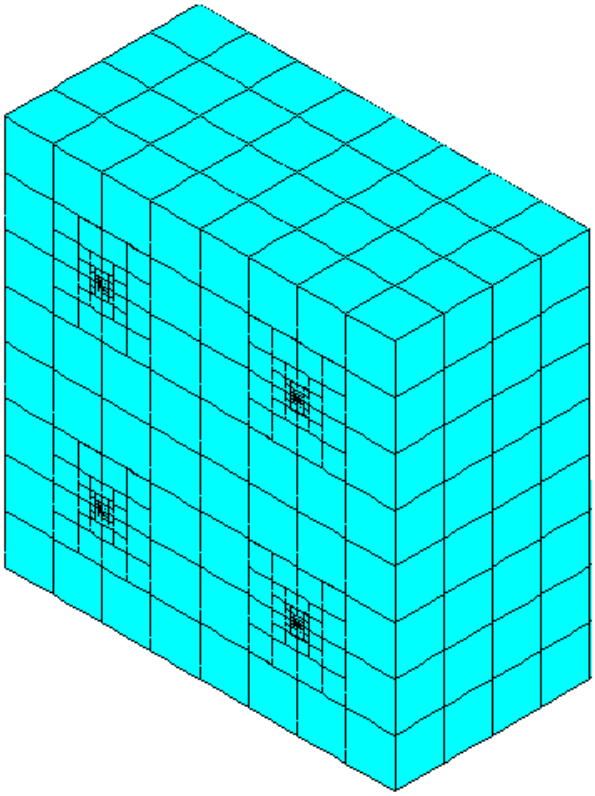


Figure 5.3: 3D mesh with four point singularities

Let $b(w,v)$ be a symmetric, bilinear form:

$$b(w,v) = 2\int_{\Omega} wv\mathrm{d}\mathbf{x} \tag{5.7}$$

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

$$l(w) = 2\int_{\Omega} wf\mathrm{d}\mathbf{x} \tag{5.8}$$

This allows us to rewrite the problem in the following way:

$$\frac{1}{2}b\left(u,u\right) - l\left(u\right) \rightarrow \min \tag{5.9}$$

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

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

For the detailed proof, please refer to [23].

By applying the above fact we obtain an equivalent problem, as in Equation 5.10:

$$b\left(u,v\right) = l\left(v\right) \qquad \forall_{v\in V} \tag{5.10}$$

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

– first order polynomials over four vertices

$$\alpha_1(x,y) = (1-x)(1-y)$$
$$\alpha_2(x,y) = (1-x)y$$
$$\alpha_3(x,y) = xy$$
$$\alpha_4(x,y) = x(1-y)$$

– second order polynomials over four edges

$$\alpha_{4+1}(x,y) = (1-x)x(1-y)$$
$$\alpha_{4+2}(x,y) = (1-x)y(1-y)$$
$$\alpha_{4+3}(x,y) = (1-x)xy$$
$$\alpha_{4+4}(x,y) = x(1-y)y$$

– second order polynomial over element interior

$$\alpha_9(x,y) = x(1-x)y(1-y)$$

Each of the vertex shape functions is equal to one on one vertex and vanishes on the remaining vertices. Each of the edge shape functions is second order polynomial over one edge, and equal to zero on all other edges. Each of the face shape functions is second order polynomial over one face and zero over all other edges and faces. The interior shape function is second order polynomial inside an element and is equal to zero over all the faces. The shape functions over an arbitrary rectangular element are obtained by affine transformation of the master element basis functions. For more details please refer to [23, 27].

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

$$u = \sum_{i=1}^{n} u_i \alpha_i. \tag{5.11}$$

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

$$\sum_{i=1}^{n} u_i b\left(\alpha_i, \alpha_j\right) = l\left(\alpha_j\right) \qquad \forall_{\alpha_j \in V} \tag{5.12}$$

The equation is considered element by element, 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:

$$\begin{pmatrix} b(\alpha_9,\alpha_9) & ... & b(\alpha_9,\alpha_1) \\ ... & ... & ... \\ b(\alpha_1,\alpha_9) & ... & b(\alpha_1,\alpha_1) \end{pmatrix} = \begin{pmatrix} l(\alpha_9) \\ ... \\ l(\alpha_1) \end{pmatrix}$$

where rows and columns are ordered thus interior, followed by edges, followed by vertex shape functions.

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

$$\Omega = (-1,1) \times (0,1) \ni (x,y) \mapsto f(x,y) \tag{5.13}$$

$$= min(tan(\frac{\pi}{2}(1-|x|))tan(\frac{\pi}{2}(1-y)), M)$$

M denotes the strength of the singularities. 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 the function f with increasing accuracy.

5.1.2. Heat transfer problem with heterogeneous materials

In this section we present another sample model problem, where we solve a heat transfer problem over a domain with heterogeneous materials ranging over several orders of magnitude. In such a case, each time three different materials meet, or two materials meet at a corner, they generate a local point singularity. For a visualisation, see Figure 5.4.

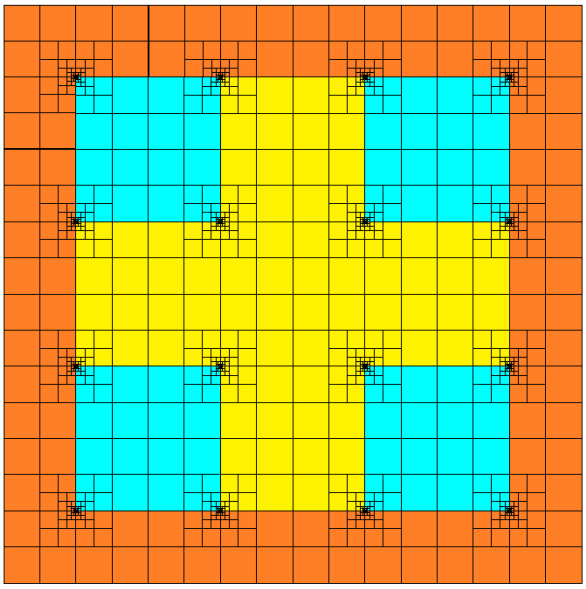


Figure 5.4: Sample two dimensional mesh with different material data, refined towards point singularities

The strong form of the problem is defined by Equation 5.14.

$$\text{Find } u = u(x,y) \in H^1(\Omega) \text{ such that} \tag{5.14}$$

$$\nabla(K\nabla u) = 0 \tag{5.15}$$

where $\Omega = (0,1)^2$, with the following boundary conditions:

$$u(\cdot,0) = u(\cdot,1) = 0 \tag{5.16}$$

$$\frac{\partial u}{\partial x}(0,\cdot) = \frac{\partial u}{\partial x}(1,\cdot) = 1 \tag{5.17}$$

The $K(\cdot,\cdot)$ is the material data function given by:

$$K(x,y) = 10^3 \quad \text{for } x < \frac{3}{8} \text{ or } y < \frac{1}{8} \text{ or } x > \frac{7}{8} \text{ or } y > \frac{7}{8} \tag{5.18}$$

$$K(x,y) = 1 \quad \text{for } \frac{3}{8} < x < \frac{5}{8} \text{ and } \frac{1}{8} < y < \frac{7}{8} \text{ or } \frac{1}{8} < x < \frac{7}{8} \text{ and } \frac{3}{8} < y < \frac{5}{8} \tag{5.19}$$

$$K(x,y) = 10^{-4} \quad \text{elsewhere} \tag{5.20}$$

The variational formulation is obtained by taking the L^2 inner product with functions $v \in H^1(\Omega)$ and integrating by parts.

Find

$$u \in V = \{u \in H^1(\Omega) : tru(\cdot, 0) = tru(\cdot, 1) = 0\} \tag{5.21}$$

such that

$$b(u, v) = l(v), \forall v \in V \tag{5.22}$$

$$b(u, v) = \int_\Omega K \nabla u \cdot \nabla v dV \tag{5.23}$$

$$l(v) = -\int_{\Gamma_{0\times(0,1)}} v dV + \int_{\Gamma_{1\times(0,1)}} v dV \tag{5.24}$$

where tru denotes trace of function u and:

$$\Gamma_{0\times(0,1)} = \{(x, y) : x = 0, y \in (0, 1)\}$$
$$\Gamma_{1\times(0,1)} = \{(x, y) : x = 1, y \in (0, 1)\}$$

5.1.3. Heat transfer problem with point sources

In this section we consider a model of a three dimensional heat transfer problem with Dirichlet and Neumann boundary conditions, namely:

$$\text{For a given } f \text{ find } u = u(x, y, z) \in H^1(\Omega) \text{ such that} \tag{5.25}$$

$$\Delta u = f \tag{5.26}$$

where $\Omega = (0, 1)^3$, with the following boundary conditions:

$$u(\cdot, \cdot, 0) = 0 \tag{5.27}$$

$$\frac{\partial u}{\partial x}(0, \cdot, \cdot) = \frac{\partial u}{\partial x}(1, \cdot, \cdot) = \frac{\partial u}{\partial y}(\cdot, 0, \cdot) = \frac{\partial u}{\partial y}(\cdot, 1, \cdot) = \frac{\partial u}{\partial y}(\cdot, \cdot, 1) = 0 \tag{5.28}$$

The right hand side defines the point-wise heat sources, e.g.

$$\begin{aligned} \Omega = (0, 1)^3 \ni (x, y, z) \mapsto f(x, y, z) &= min_x\{tan(2\pi x), M\} \\ &*min_y\{tan(2\pi y), M\} \\ &*min_z\{tan(2\pi z), M\} \end{aligned} \tag{5.29}$$

M stands for a constant describing strength of singularity.

The weak variational formulation is obtained by taking the L^2 inner product with functions $v \in H^1(\Omega)$ and integrating by parts.

Find

$$u \in V = \{u \in H^1(\Omega) : u(\cdot, \cdot, 0) = u(\cdot, \cdot, 1) = 0\} \tag{5.30}$$

such that

$$b(u, v) = l(v), \forall v \in V \tag{5.31}$$

$$b(u, v) = \int_\Omega \nabla u \cdot \nabla v dV \tag{5.32}$$

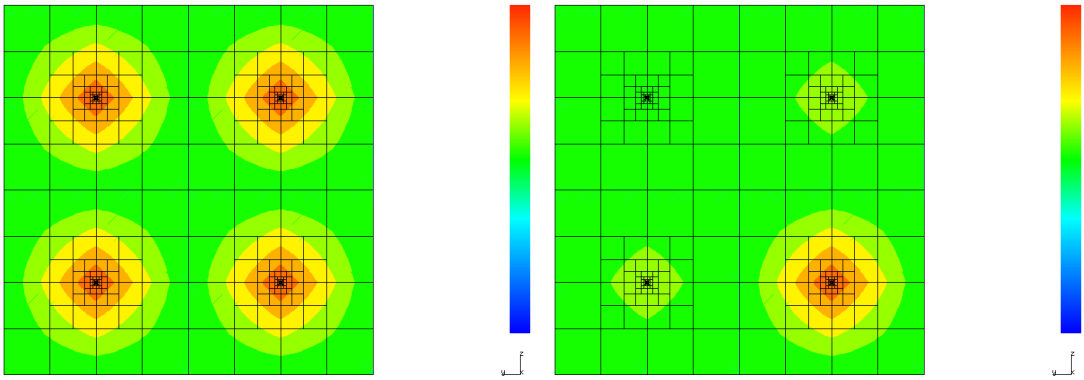


Figure 5.5: Solutions to heat transfer problems with several point heat sources with the same or different intensities.

$$l(v) = -\int_{\Omega} f v dV \tag{5.33}$$

The exemplary solution on a cross-section of the domain is presented in Figure 5.5.

Although this study is restricted to a Laplace equation, the results presented here are still valid when applied to other PDEs including linear elasticity, Stokes and Maxwell's equations, or a number of multi-physics problems, where the forcing in the equation is point-wise in nature. In these cases, the problem implies the necessity for local point refinements.

5.2. CPU implementation

This section delivers efficiency measurements of the solver algorithm for the simple two- and three-dimensional problems with one point singularity. A simple heat transfer problem with artificially enforced singularity at the bottom-center of the mesh has been used to measure the behavior of the solver. The goal of these experiments was to compare our graph grammar based approach with classical state-of-the-art MUMPS solver [2, 3, 4] performing the construction of the elimination tree with a nested dissection algorithm from the METIS [68] library based on recursive partitions of the connectivity graph obtained from the global matrix sparsity pattern. The numerical experiments for 2D are summarized in Figures 5.6 - 5.9. The performance of the solver for the 3D problem is depicted in Figure 5.10. The horizontal axis denotes the number of degrees of freedom when we increase the number of refinement levels, the vertical axis denotes the execution time. We have enforced the level-by-level ordering in the frontal solver algorithm developed by Irons [65] as well as executing the MUMPS solver [2, 3, 4] with a nested dissection algorithm. The numerical results show that both our algorithm and MUMPS algorithm deliver a linear computational cost in sequential case. The oscillations can be attributed to the fact that one singularity problem is very small, and therefore activities of the system deamons and other third-party phenomena can influence the results. The results obtained can be considered good, since the MUMPS solver is highly optimized code, and we have shown that our hand-made code can provide the same scalability due to our graph grammar based approach. In order to improve the scalability, we need to go for parallel shared memory GPU implementation. This result became strong encouragement for developing the parallel shared memory graph grammar based implementation of our algorithm, where we can take advantage of the graph grammar based parallel implementation, where the concurrency of the algorithm results from analysis of the dependency on basic undivisible tasks, defined as graph grammar productions.

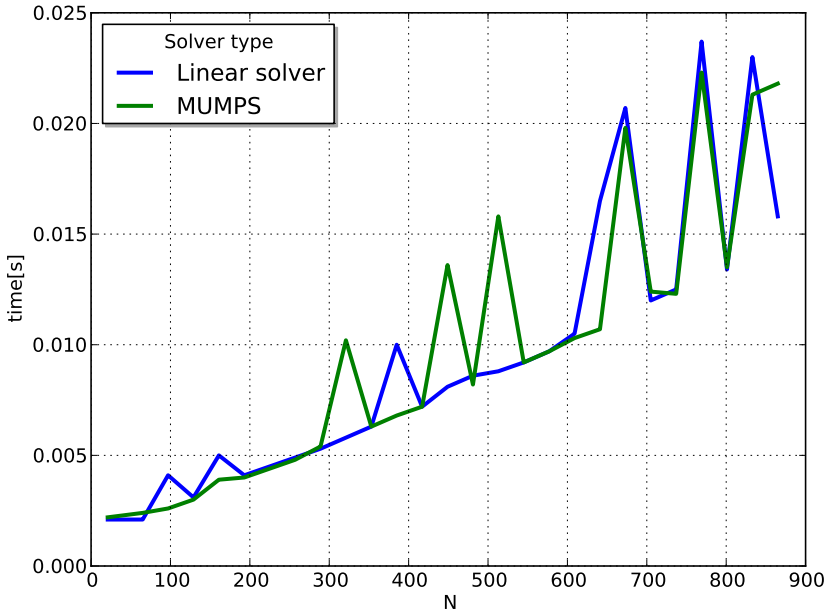


Figure 5.6: The linear computational cost of the solver algorithm for second order polynomials

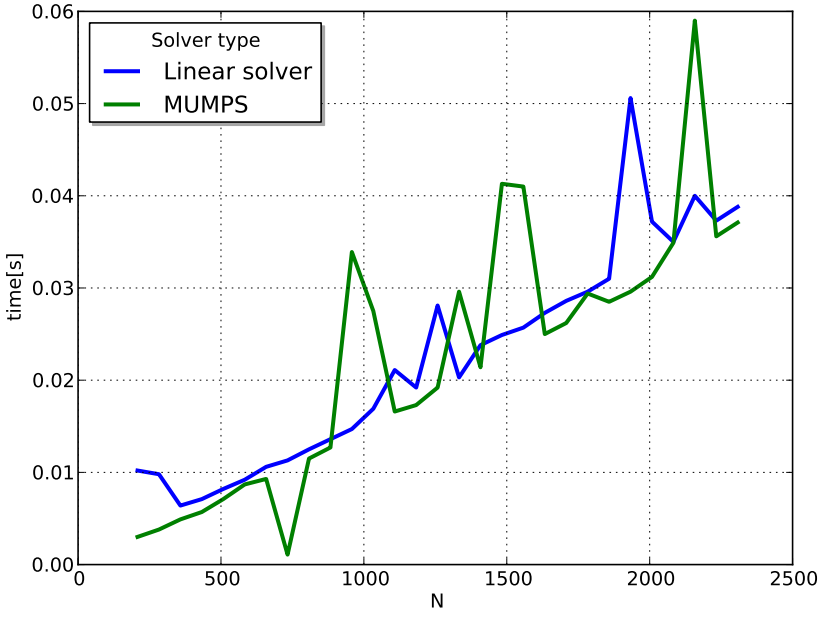


Figure 5.7: The linear computational cost of the solver algorithm for third order polynomials

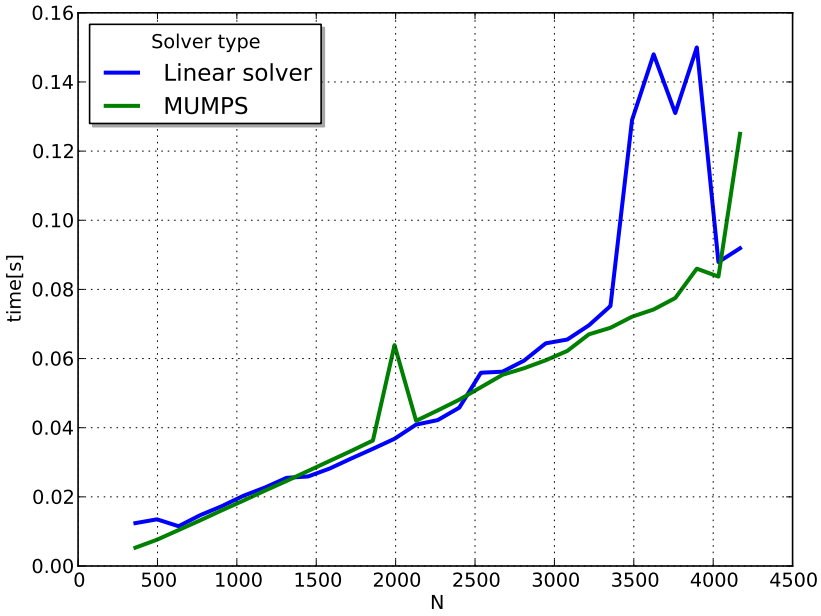


Figure 5.8: The linear computational cost of the solver algorithm for fourth order polynomials

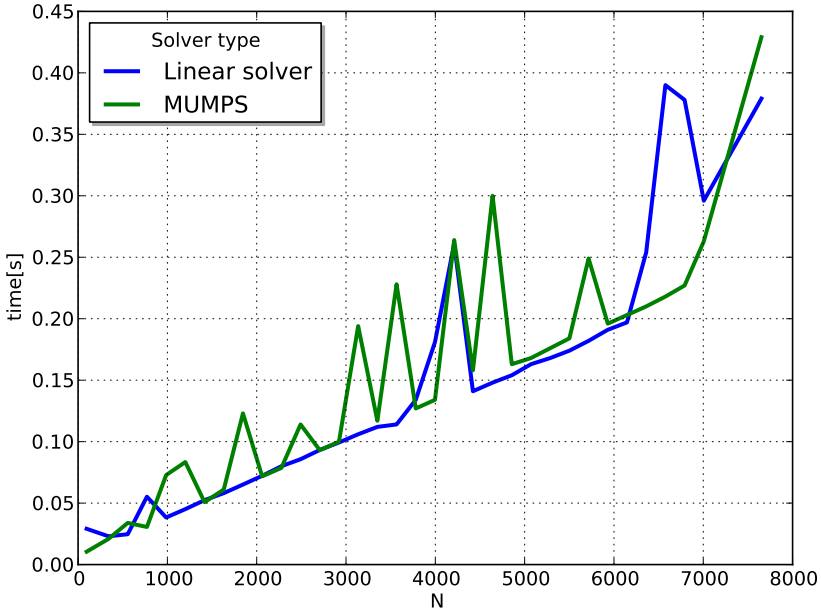


Figure 5.9: The linear computational cost of the solver algorithm for fifth order polynomials

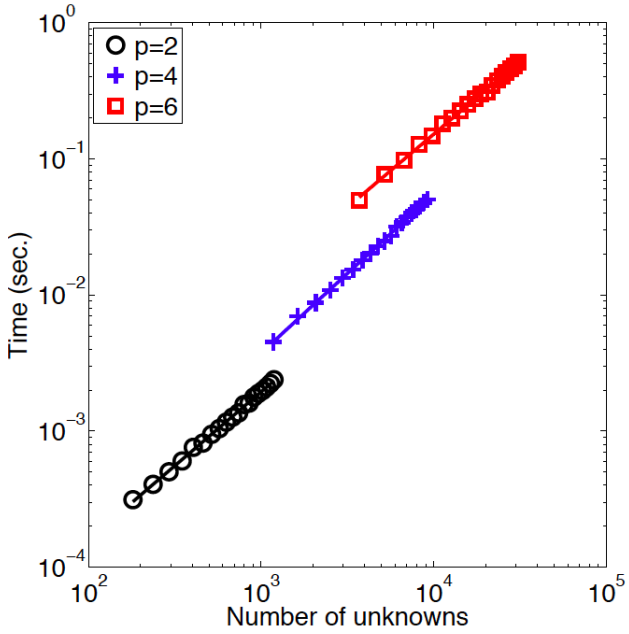


Figure 5.10: The linear computational cost of the solver algorithm for 3D mesh with point singularity

5.3. GPU implementation

The graph grammar based solver has been implemented on GPU with NVIDIA CUDA inspired by the B-spline [74] based solver developed by Kuźnik [71]. The graph grammar productions have been partitioned into sets of independent tasks, and scheduled set by set, concurrently, into nodes of the GPU. The algorithm is the following:

1. Execute in serial (on CPU) hypergraph productions constructing a mesh with point singularity refined to an arbitrary level k. $P_{init} \mapsto P_{breakinitleft} \mapsto P_{breakinitleft} \mapsto P_{regularity} \mapsto [P_{breakinterior} \mapsto P_{enforceregularity}]^k$, where k corresponds to the number of generated layers.
2. Execute in serial the graph grammar production partitioning the mesh into layers: $P_{separatetop} \mapsto P_{separatetop} \mapsto P_{separatebottom}$.
3. Execute concurrently 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.
4. 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 one after another.
5. Execute the graph grammar production responsible for merging and solving the top problem $P_{mergetop}$ and $P_{solvetop}$.

NVIDIA CUDA environment

GPUs constitute a very promising architecture for scientific computations as they allow us to process data much faster than ordinary CPUs. Instead of faster clocks, more sophisticated processors and adding more work per clock cycle, the GPU approach is to put together many smaller, simpler processors. Calculations are performed in parallel by hundreds of threads allowing us to achieve logarithmic scaling of the solution. GPUs also have a large memory

bandwidth and a substantial amount of processing cores. CUDA (Compute Unified Device Architecture [77]) is a language for programming both CPUs and NVIDIA GPUs using the same CUDA program written in C (with some extensions for expressing the the parallelism). CUDA assumes that the GPU is a coprocessor for the CPU. The CPU is called a *host*, whereas the GPU is called a *device*. The host plays the superior role orchestrating computations and initiating memory copying. Parallel portions of an application are executed on the device as kernels, which look like serial programs. However, the GPU will run each kernel on many threads. A typical CUDA program includes the following steps:

1. CPU allocates memory on GPU using *cudaMalloc* instruction.
2. CPU copies data from CPU to GPU using *cudaMemcpy* instruction.
3. CPU launches kernels on GPU.
4. GPU processes the data in parallel.
5. CPU copies results back to CPU from GPU (again using *cudaMemcpy* instruction).

Understandably, a successful CUDA program should have a high computation to communication ratio. In the case of the parallel solver presented in this thesis, this holds true, since processing element matrix for a single layer significantly more expensive than distributing and merging the solution. CUDA is available so far on GeForce, ION, Quadro and Tesla chips. Tesla is dedicated to scientific computations and has been enthusiastically welcomed [45] due to its reasonable price, high performance and low energy consumption. CUDA programming is now backed by established commercial software like Mathematica (via *CUDALink*) or MATLAB (via *Parallel Computing Toolbox*). Using Tesla is also becoming increasingly popular among FEM researchers - see [70, 75]. Recently, Tesla-based supercomputers were prominent on the TOP500[1] list. What is even more interesting is that all of the top 10 supercomputers on the Green500[2] list are Tesla-based.

Modern GPUs consist of several multiprocessors (usually 8 to 12), each containing many cores (8 for GTX 260, 32 for Tesla C2070). Moreover, there are four kinds of memory: global, shared, constant, and texture. Unlike traditional CPUs, which are optimized for latency, GPUs are primarily optimized for high throughput, which makes more sense in graphic computations. Global memory is over 1 GB and can be accessed from every multiprocessor at the cost of high latency. Shared memory (up to 48 KB per multiprocessor) can be accessed by all threads running on one multiprocessor. This is a very limited amount, but accessed with low latency and high throughput. In this work, only global and shared memory are used. Using constant and texture memory could improve the presented results even more, but then, the presented implementation would be tightly bound to a specific hardware solution.

5.4. Numerical experiments

In order to prove the theoretical discoveries in this paper, a series of experiments have been conducted to compare and contrast efficiency of the hypergraph driven parallel GPU solver with the well-established MUMPS solver (see Section 2.2.1 or [2, 3, 4]) over 2D and 3D grids with point singularities. The numerical tests were performed on GeForce GTX 260 graphic card with 24 multiprocessors, each of them equipped with 8 cores. This means the total count of cores was equal to 192. The global memory of the graphic card was 896 MB. Polynomial approximation level p was fixed over the entire domain. The results presented below in a form of graphs were obtained for $p = 2$ and $p = 3$.

Performance evaluation

We compare 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 Figure 5.11). Such a structure of the grid may result

[1] TOP500 List ranks and details the top 500 most powerful computer systems in the world - http://www.top500.org
[2] Green500 List lists the most energy-efficient supercomputers that made the TOP500 List

either from a projection problem or from a heat transfer problem, again, either with point sources or with different material data. The different problems only influence the values at the matrix entries but they do not change the number of operations or the computational cost of the solver.

It can be easily observed that the GPU solver is very predictable in terms of computational costs, which logarithmically increases with the number of degrees of freedom. In the case of MUMPS, it is full of sharp rises and falls, but, in general, rises in a linear way. The results are presented in eight graphs (Figures 5.11 - 5.20).

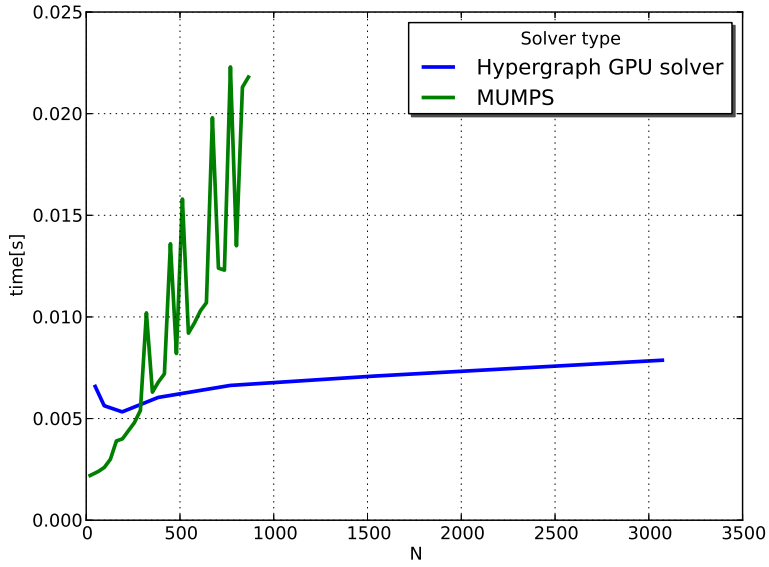


Figure 5.11: 2D mesh with a point singularity, $p = 2$

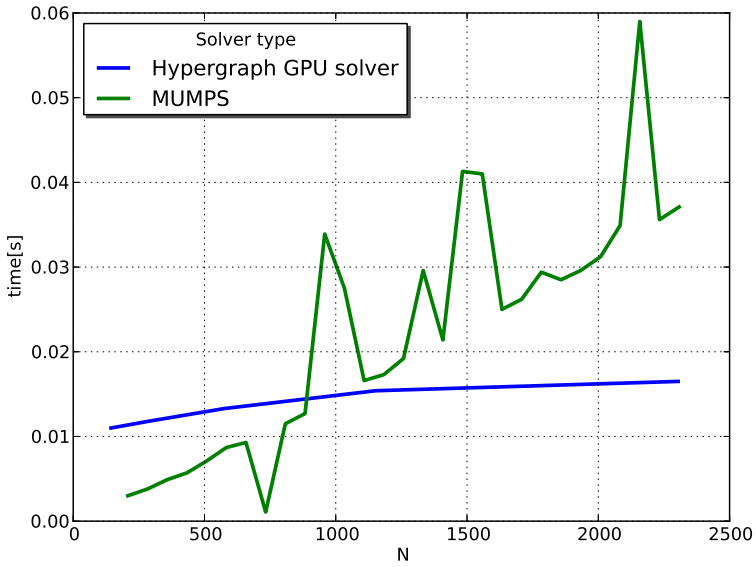


Figure 5.12: 2D mesh with a point singularity, $p = 3$

For $p = 3$ this difference remains visible in favor of the GPU solver (compare Figure 5.13).

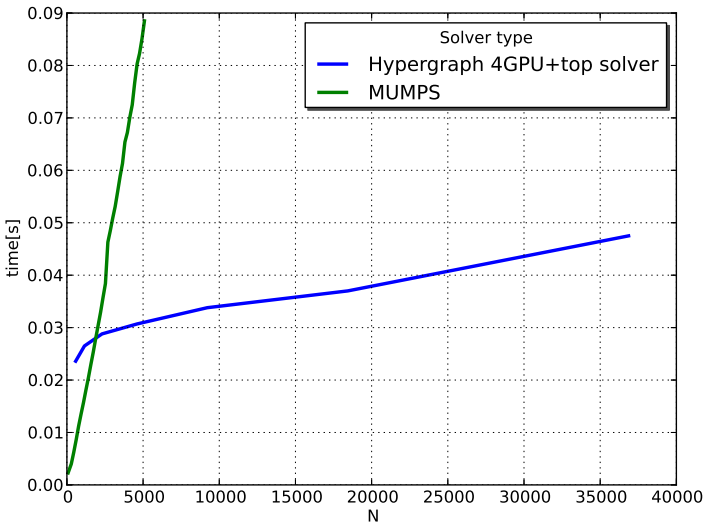


Figure 5.13: 2D mesh with 2×2 point singularities, $p = 2$

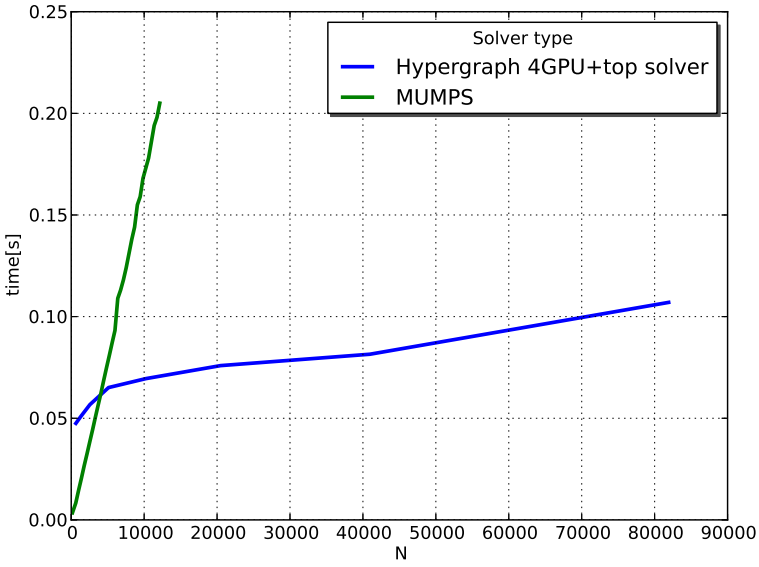


Figure 5.14: 2D mesh with 2×2 point singularities, $p = 3$

With the increase in the number of singularities this difference becomes even more obvious. The computational cost of MUMPS still seems linear, but rising at a much steeper angle. When there is only 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 *4GPU+top* for 2×2 singularites or *16GPU+top* for 4×4 singularities and *64GPU+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 our GPU based solver is presented in Figures 5.13 - 5.18 for 2×2 and 4×4 and 8×8 singularities, respectively.

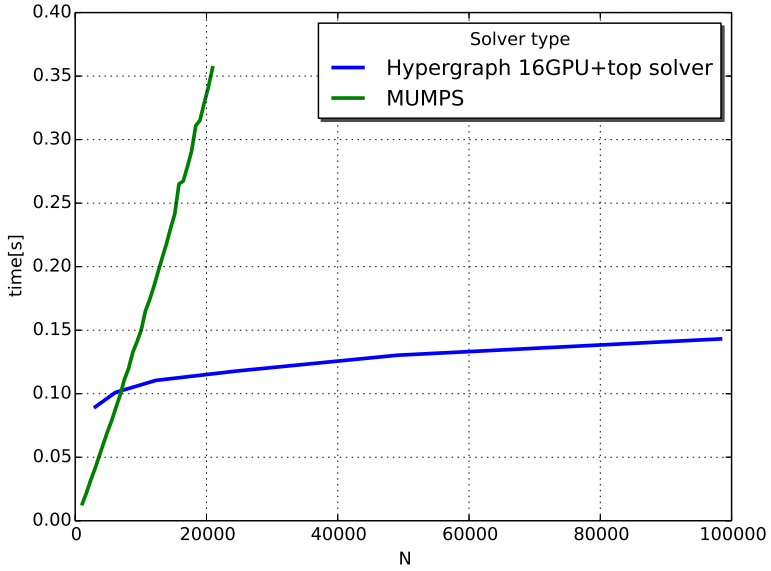


Figure 5.15: 2D mesh with 4×4 point singularities, $p = 2$

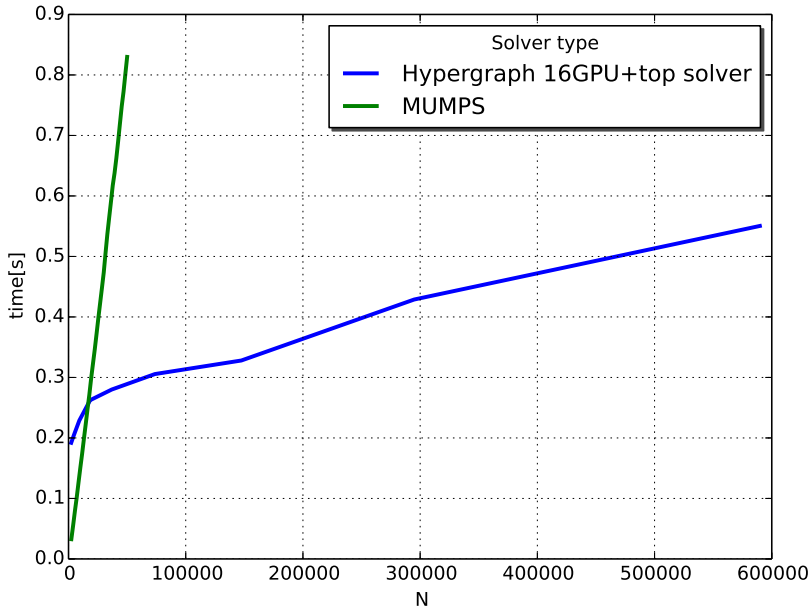


Figure 5.16: 2D mesh with 4×4 point singularities, $p = 3$

We have performed some experiments for three dimensional point singularities, for both one singularity cases as well as for $2 \times 2 \times 2$ singularities. We present these results in Figures 5.19 - 5.20. The results for a 3D problem also confirm logarithmic scalability of our hypergraph grammar based solver.

Convergence rate

It is also worth measuring what was the loss of convergence rate associated with using h FEM instead of hp FEM. The results presented in Figure 5.21 indicate that the h adaptive FEM still converges at a reasonable rate, however, the convergence is no longer exponential. In turn, the linear cost of the hp-adaptive version of the

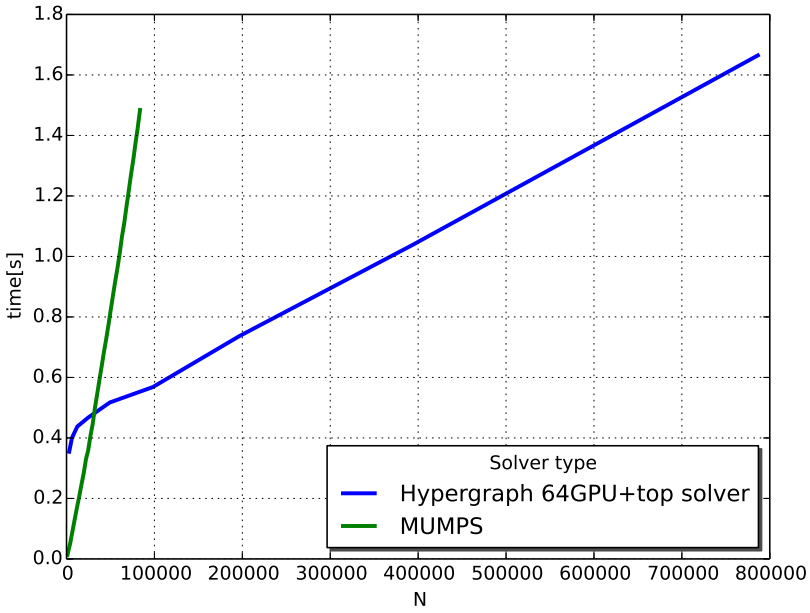


Figure 5.17: 2D mesh with 8×8 point singularities, $p = 2$

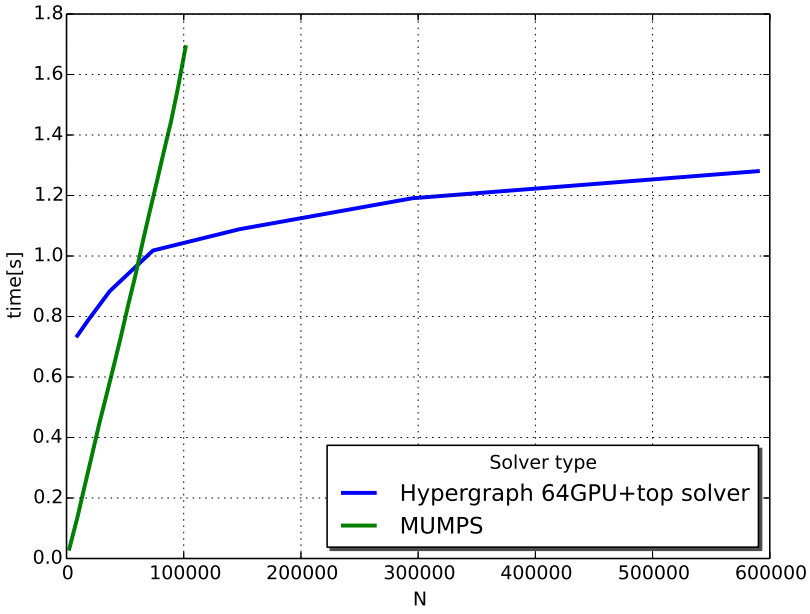


Figure 5.18: 2D mesh with 8×8 point singularities, $p = 3$

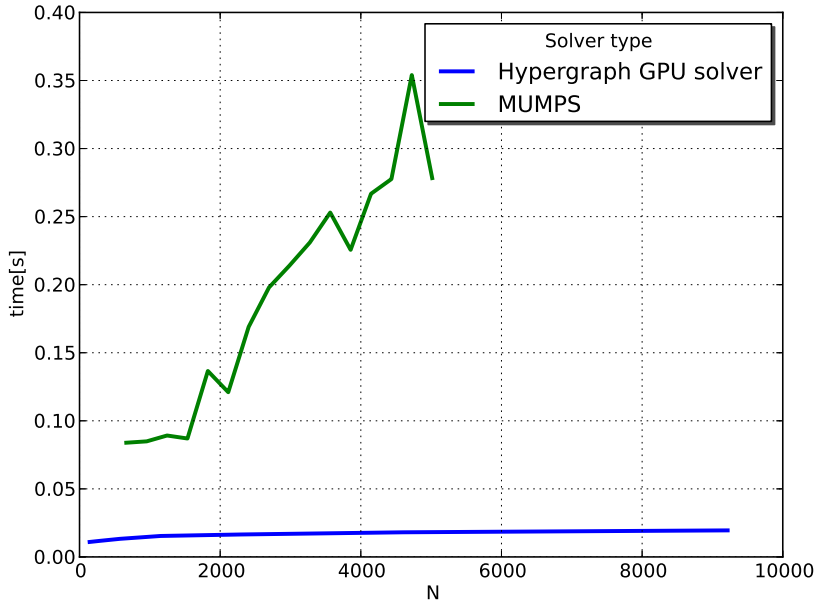


Figure 5.19: 3D mesh with a point singularity, $p = 2$

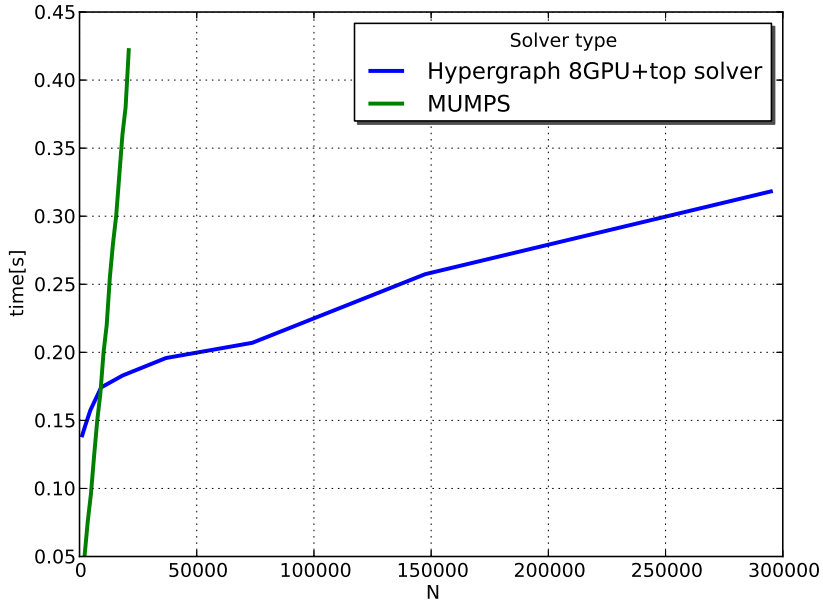


Figure 5.20: 3D mesh with $2 \times 2 \times 2$ point singularities, $p = 2$

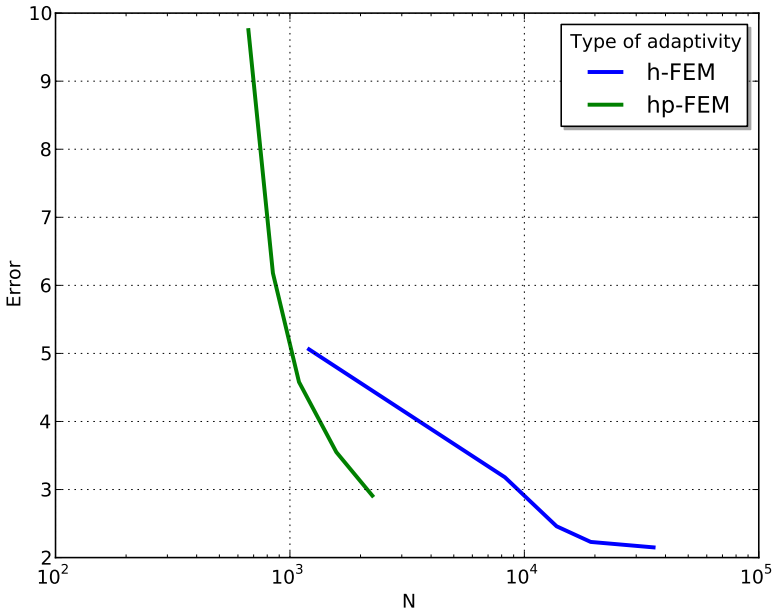


Figure 5.21: Convergence rate comparison for a sample 3D problem

graph grammar solver has not been theoretically proven, but the preliminary experiments for a presented problem suggest that it is also close to linear. The subject of convergence of the adaptive FEM methods has been profoundly researched by Babuška [6, 7] and Demkowicz et al. [23, 27, 29].

Chapter 6

Conclusions and future research

This chapter contains a summary of the research described in this thesis. This is followed by an evaluation of the scientific contribution made by this work. The most important directions for the future work have also been investigated and outlined.

6.1. Summary and significance of the obtained results

This section summarises all the results obtained and presented in this thesis and compares them with the open problems listed in Section 2.3. The main and most significant achievement of this work is creation of the graph grammar models applicable to a wide class of the adaptive algorithms, namely grids with one or multiple point singularities. The formal methodology used to achieve this goal was hypergraph grammar formalism. The hypergraph grammar productions introduced in this work represent all the steps of the muti-frontal direct solver algorithm applied for fast numerical solutions of the grids resulting from adaptive finite element method computations. All the graph grammar productions presented in this work can also be applied to grids with multiple point singularities, however, for the sake of simplicity, we restricted ourselves here to the subset of hypergraph grammar productions expressing the grids with single point singularities. Nevertheless, in the numerical sections we analyzed meshes with one or many point singularities, for two and three dimensional grids. The sequential version of the hypergraph grammar driven solver has been implemented on CPU and its efficiency has been compared with the state-of-the-art MUMPS solver. The graph grammar formalism presented in this work allowed for analysis of the concurrency of the algorithm and for localizing sets of graph grammar productions that can be executed concurrently. The analysis allowed us to obtain the graph grammar model suitable for parallel shared memory implementation. The formalism was implemented on GPU thanks to NVIDIA CUDA technology.
The author's contribution specifically includes the following items:

Hypergraph grammar model for generation of the two element mesh. Section 3.2.1 presented hypergraph grammars for generating a non-uniform mesh refined towards a point singularity. These productions are not designed for parallel execution, but are computationally inexpensive and can be executed in serial without any undesirable loss of efficiency.

Hypergraph grammar models of the sequential direct solver in 2D. Section 3.2.2 contains a set of hypergraph productions driving execution of the sequential direct solver for two dimensional problems with one or multiple point singularities. Its linear computational cost has been proven in Section 3.3.1. Its memory usage has been estimated in Section 3.3.2. The solver algorithm is also easily extendable to problems with multiple point singularities, which was the case in Chapter 5.

Hypergraph grammar model of the parallel direct solver in 2D. Analysis of concurrency of the sequential solver algorithm led to discoveries presented in Section 3.2.3 and a hypergraph model for the parallel solver for 2D problems with point singularities delivering logarithmic computational cost (which has been estimated in Section 3.3.3).

Hypergraph grammar model prescribing generation of a mesh in 3D. Section 4.1 contains a set of hypergraph grammar productions for generating an arbitrarily refined three dimensional mesh.

Hypergraph grammar model prescribing direct solver execution in 3D. Section 4.2 contains a summary of hypergraphs grammar productions driving execution of the solver in three dimensions.

Numerical experiments and efficiency comparison. Chapter 5 considers sample 2D and 3D problems that are exercised primarily to measure performance of the GPU implementation of the hypergraph grammar driven solver for one and many point singularities. Its execution time is compared with both MUMPS and theoretical cost estimations.

Theoretical estimations of the computational cost and memory usage. The computational cost of the proposed graph grammar based algorithms has been analysed in Section 3.3. Proof of linear complexity of the sequential solver in two dimensions was based on exact cost estimations. Proof of logarithmic complexity of the parallel solver was also based on exact estimations of computational cost on shared memory architectures. Theoretical estimation of the memory usage for the two dimensional sequential solver has been performed in Section 3.3.2.

6.2. Potential for future research

There remain several interesting unanswered questions in this field. They will definitely be of interest to me in the next stages of my research. First, it needs to be estimated what is the exact computational cost in the case of hp adaptive FEM with one or multiple point singularities. Proof of the linear cost of h adaptive FEM can be used to determine the upper limit of the cost of hp adaptive FEM, whereas it is far from being exact.

It is important to decide, whether problems with single or multiple edge singularities can also be solved in linear time using a similar approach to the one described in this thesis. In other words it is an open question if the same linear computational cost can be obtained for graph grammar based solvers dedicated to edge singularities, and if not, what will be the computational cost of such cases. Moreover, we can investigate if the proposed approach can be generalized for grids with mixed point and edge singularities.

The findings presented here can also be applied to three dimensional grids. This concerns the theoretical estimates, the graph grammar models, as well as other types of singularities that can be identified in three dimensional cases, like face singularities and the mixtures of various singularities.

Another area for future research is to continue my agent-based FEM research started in [53, 54, 55, 99]. This means leveraging the graph-grammar formalism presented in this thesis to orchestrate high-level [5, 31] parallel FEM codes with a certain level of autonomy [16, 19, 90, 104]. The role of the graph grammars is to increase locality and robustness of the computation in a heterogeneous environment. Each host could be equipped with a certain amount of productions that would be the only necessary context.

The last planned field of future research would be to implement hypergraph grammar formalism on different types of traditional and emerging architectures. Comparison of their performance would help us to determine where the benefits from using graph grammars are most substantial.

Appendices

Appendix A

Exemplary 1D problem solved with the Finite Element Method

This Appendix presents a step-by-step solution to a rudimentary, one dimensional Finite Element Method problem. It will be solved over the two element domain in Figure A.1 to outline all relevant phases of this method. The domain is defined by a segment $[0, 1]$, which, in our case, determines also the element size ($h = 0.5$). The polynomial approximation level is constant and set to 1 ($p = 1$).

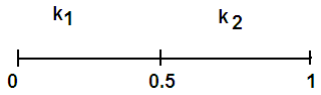


Figure A.1: Domain of the exemplary 1D FEM problem

We begin with the initial, strong formulation of the heat transfer (Equation A.1).

We are looking for $[0, 1] \ni x \to u(x) \in \mathbb{R}$ such:

$$-\frac{d}{dx}(k(x)\frac{du}{dx}) = 0 \tag{A.1}$$

where $[0, 1] \ni x \to k(x) \in \mathbb{R}$ is a given material data function, in our case defined as

$$k(x) = \begin{cases} k_1 & \text{if } x \in [0, 0.5) \\ k_2 & \text{if } x \in [0.5, 1] \end{cases} \tag{A.2}$$

On both ends of the domain there are certain boundary conditions (BC). There are three possible types:

- Dirichlet BC, where $u(x_l) = u_l$ is given,
- Cauchy BC, where boundary conditional is determined by the following equation $\alpha\frac{du(x_l)}{dx} + \beta u(x_l) = \gamma$,
- Neumann BC, which is a special case of Cauchy BC where $\frac{du(x_l)}{dx}$ is given.

We assume Dirichlet BC in $x = 0$ to be the following:

$$u(0) = u_w \tag{A.3}$$

where u_w is a given temperature value outside of the domain on the left-hand-side Cauchy Boundary in $x = 1$ is expressed by Equation A.4.

$$k(1)\frac{du(1)}{dx} + hu(1) = hu_z \tag{A.4}$$

where $k(1) = k_1$ is the value of k material data function at point 1, h is a given constant, u_z is a given temperature value outside of the domain on the right-hand-side. To obtain variational (weak) formulation of the problem, we multiply both sides of the strong formulation by a test function v and integrate left side by parts.

$$-\int_0^1 \frac{d}{dx}(k\frac{du}{dx})vdx = 0 \tag{A.5}$$

$$\int_0^1 k\frac{du}{dx}\frac{dv}{dx}dx - k(1)\frac{du(1)}{dx}v(1) + k(0)\frac{du(0)}{dx}v(0) = 0 \tag{A.6}$$

Finally we incorporate boundary conditions, and the fact that $v(0) = 0$

$$\int_0^1 k\frac{du}{dx}\frac{dv}{dx}dx - (hu_z - hu(1))v(1) = 0 \tag{A.7}$$

Now, the problem statement is converted to the formulation below.
Find function u satisfying the following equation:

$$\int_0^1 k\frac{du}{dx}\frac{dv}{dx}dx + hu(1)v(1) = hu_z v(1) \qquad \forall_{v:v(0)=0} \tag{A.8}$$

To satisfy Dirichlet BC, also $u(0) = u_w$. In order to make this equation solvable by FEM, we need to extend the boundary condition so that $u(0) = 0$.

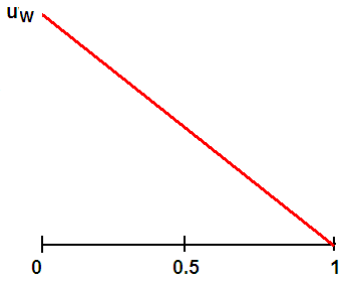


Figure A.2: Graph of $\tilde{u}$

We also define $\tilde{u}$ as a function of x.

$$\tilde{u}(x) = u_w(1 - x) \tag{A.9}$$

We also substitute u in Equation A.10.

$$u = \tilde{u} + \hat{u} \tag{A.10}$$

Hence

$$\int_0^1 k\frac{d(\tilde{u} + \hat{u})}{dx}\frac{dv}{dx}dx + h(\tilde{u} + \hat{u})(1)v(1) = hu_z v(1) \qquad \forall_{v:v(0)=0} \tag{A.11}$$

$$\int_0^1 k\frac{d\tilde{u}}{dx}\frac{dv}{dx}dx + \int_0^1 k\frac{d\hat{u}}{dx}\frac{dv}{dx}dx + h\tilde{u}(1)v(1) + h\hat{u}(1)v(1) = hu_z v(1) \qquad \forall_{v:v(0)=0} \tag{A.12}$$

The formula can now be rewritten in the following way:

$$\int_0^1 k\frac{d\hat{u}}{dx}\frac{dv}{dx}dx + h\hat{u}(1)v(1) = hu_z v(1) - \int_0^1 k\frac{d\tilde{u}}{dx}\frac{dv}{dx}dx - h\tilde{u}(1)v(1) \qquad \forall_{v:v(0)=0} \tag{A.13}$$

Now, the problem can be expressed as:

Find u such that $b(u, v) = l(v), \quad \forall_{v:v(0)=0}$ where

$$b(u, v) = \int_0^1 k\frac{du}{dx}\frac{dv}{dx}dx + hu(1)v(1) \qquad b : V \times V \to \mathbb{R} \tag{A.14}$$

$$l(v) = hu_z v(1) - \int_0^1 k\frac{d\tilde{u}}{dx}\frac{dv}{dx}dx - h\tilde{u}(1)v(1) \qquad l : V \to \mathbb{R} \tag{A.15}$$

and $u(0) = 0$. We finally construct the finite element subspace $V_h \subset V$. For the sake of simplicity it was assumed that $p = 1$ and hence, we construct only vertex basis function. Function e_1 is defined over the first vertex (Figure A.3), e_2 over the second (Figure A.4) and e_3 over the third (Figure A.5). Starting from $p = 2$, basis functions have their supports over a single element only. This is one of the reasons that the FEM-assembled matrices are sparse.

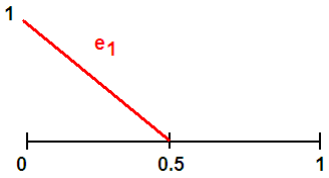


Figure A.3: Basis function e_1

$$e_1(x) = \begin{cases} 1 - 2x & \text{if } x \in (0, 0.5) \\ 0 & \text{if } x \in (0.5, 1) \end{cases} \qquad \frac{de_1(x)}{dx} = \begin{cases} -2 & \text{if } x \in (0, 0.5) \\ 0 & \text{if } x \in (0.5, 1) \end{cases} \tag{A.16}$$

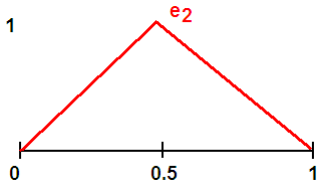


Figure A.4: Basis function e_2

$$e_2(x) = \begin{cases} 2x & \text{if } x \in (0, 0.5) \\ 2 - 2x & \text{if } x \in (0.5, 1) \end{cases} \qquad \frac{de_2(x)}{dx} = \begin{cases} 2 & \text{if } x \in (0, 0.5) \\ -2 & \text{if } x \in (0.5, 1) \end{cases} \tag{A.17}$$

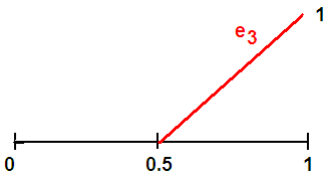


Figure A.5: Basis function e_3

$$e_3(x) = \begin{cases} 0 & \text{if } x \in (0, 0.5) \\ 2x - 1 & \text{if } x \in (0.5, 1) \end{cases} \qquad \frac{de_3(x)}{dx} = \begin{cases} 0 & \text{if } x \in (0, 0.5) \\ 2 & \text{if } x \in (0.5, 1) \end{cases} \tag{A.18}$$

The original problem can be expressed as follows:

$$\text{Find such } u \in V \text{ that}$$
$$b(u,v) = l(v) \qquad \forall v \in V$$

We turn it into an approximated problem that can be solved in the Finite Element space.
It is stated as follows:

$$\text{Find } u_h \in V_h \text{ such that}$$
$$b(u_h, v_h) = l(v_h) \qquad \forall v_h \in V_h$$

$$u \approx u_h = \sum_{i=1}^{3} u_i e_i \tag{A.19}$$

$$b(\sum_{i=1}^{3} u_i e_i, v_j) = l(v_j) \tag{A.20}$$

$$\sum_{i=1}^{3} u_i b(e_i, e_j) = l(e_j) \tag{A.21}$$

$$\begin{bmatrix} b(e_1,e_1) & b(e_1,e_2) & b(e_1,e_3) \\ b(e_2,e_1) & b(e_2,e_2) & b(e_2,e_3) \\ b(e_3,e_1) & b(e_3,e_2) & b(e_3,e_3) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} l(e_1) \\ l(e_2) \\ l(e_3) \end{bmatrix} \tag{A.22}$$

But, $u(0) = 0$ and Dirichlet BC can be enforced right in the Matrix A.23.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & b(e_2,e_2) & b(e_2,e_3) \\ 0 & b(e_3,e_2) & b(e_3,e_3) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ l(e_2) \\ l(e_3) \end{bmatrix} \tag{A.23}$$

As soon as we compute both u_2 and u_3, we obtain the approximate solution $\hat{u}(x)$, which is defined by the following formula:

$$\hat{u}(x) = \begin{cases} u_2 e_2(x) & \text{if } x \in (0, 0.5) \\ u_2 e_2(x) + u_3 e_3(x) & \text{if } x \in (0.5, 1) \end{cases}$$

And finally we receive $u(x)$ from space V_h:

$$u(x) = \tilde{u}(x) + \hat{u}(x) = u_w(1-x) + \begin{cases} u_2 e_2(x) & \text{if } x \in (0, 0.5) \\ u_2 e_2(x) + u_3 e_3(x) & \text{if } x \in (0.5, 1) \end{cases}$$

Appendix B

Exemplary 2D problem solved with the Finite Element Method

This Appendix focuses on the L-shape domain problem, which is complicated enough to present an end-to-end 2D FEM approach. The L-shape domain problem is a model academic problem formulated by Babuška ([6], [7]) in 1986 to investigate the convergence of p and hp adaptive FEM algorithms. The problem consists in solving a partial differential equation over the L-shaped domain presented in Figure B.1. The L-shaped domain Ω is defined as $[-1,1] \times [-1,1] \setminus [-1,0) \times [-1,0)$. There are two types of boundary conditions applied - Dirichlet BC $\partial\Omega_D$ marked in blue and Neumann BC $\partial\Omega_N$ marked in red.

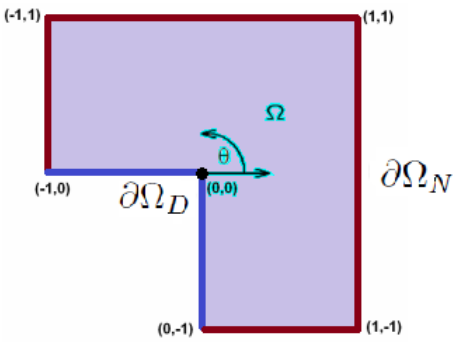


Figure B.1: L-shape domain

To illustrate the point singularity in this problem, the Euclidean norm of the gradient of the solution to the L-shape domain problem has been shown in Figure B.2. The gradient goes to infinity at the central point which drives mesh refinements towards this point when the error is measured in H^1 norm. In this example though, the problem will be solved only on the initial mesh, without any further mesh refinements.

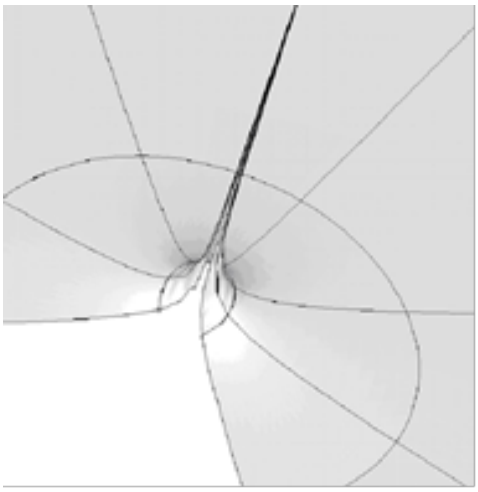


Figure B.2: Euclidean norm of the gradient of the solution to the L-shape domain problem

B.1. Strong formulation

We seek the temperature scalar field defined by Equation B.1.

$$\mathbb{R}^2 \ni (x_1, x_2) \rightarrow u(x_1, x_2) \in \mathbb{R} \tag{B.1}$$

Value $u(x_1, x_2)$ indicates temperature at the point (x_1, x_2). The strong formulation in Equation B.2 reflects the heat transfer problem.

$$\sum_{i=1}^{2} \frac{\partial^2 u}{\partial x_i^2} = 0 \quad over \quad \Omega \tag{B.2}$$

This can be rewritten as the so-called Laplace Equation B.3.

$$\triangle u = 0 \tag{B.3}$$

over the L-shape domain Ω.
The zero Dirichlet boundary condition:

$$u = 0 \quad on \quad \partial\Omega_D \tag{B.4}$$

is imposed on the internal part of the boundary $\partial\Omega_D$.
The Neumann boundary condition is defined as follows:

$$\frac{\partial u}{\partial n} = \nabla u \cdot n = g \quad on \quad \partial\Omega_N \tag{B.5}$$

is imposed on the external part of the boundary $\partial\Omega_N$. The temperature gradient in the direction normal to the boundary is defined in the radial system of coordinates with the origin point 0 (Equation B.6).

$$g(r, \theta) = r^{\frac{2}{3}} sin\frac{2}{3}(\theta + \frac{\pi}{2}) \tag{B.6}$$

The additional constraints are expressed by Equation B.7.

$$\frac{\partial u}{\partial n} = \nabla u \cdot n = (\frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}) \cdot (n_1, n_2) = \frac{\partial u}{\partial x_1} n_1 + \frac{\partial u}{\partial x_2} n_2 = g \tag{B.7}$$

where $\nabla u = (\frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2})$ is a gradient of u and (n_1, n_2) are the components of the normal vector.

B.2. Weak formulation

Weak (variational) formulation can be obtained by computing L_2 scalar product with test functions v.

$$\int_\Omega \triangle u v dx = 0 \quad \forall v \in V \tag{B.8}$$

integrating by parts

$$\int_\Omega \nabla u \cdot \nabla v dx - \int_{\partial\Omega} \nabla u \cdot n v dS = 0 \quad \forall v \in V \tag{B.9}$$

and incorporating boundary conditions

$$\int_\Omega \nabla u \cdot \nabla v dx - \int_{\partial\Omega_N} g v dS = 0 \quad \forall v \in V \tag{B.10}$$

We end up with the following week formulation: Find $u \in V$ such that

$$b(u, v) = l(v) \quad \forall v \in V \tag{B.11}$$

$$b(u, v) = \int_\Omega \nabla u \cdot \nabla v dx \tag{B.12}$$

$$l(v) = \int_{\partial\Omega_N} g v dS \tag{B.13}$$

The space of test function V is defined by Equation B.14.

$$V = \left\{ v \in H^1(\Omega) : trv = 0 \quad on \quad \partial\Omega_D \right\} \tag{B.14}$$

where trv means trace of function v.

B.3. Discretization into Finite Elements

As stated before, the problem will be solved using 2D Finite Element Method, but for simplicity, p is assumed to be constant over the entire domain and equal to one. This means absence of edge and interior basis functions . Finite Element Method implies partitioning Ω into the space of finite elements. In this example, we divide it into three square elements E_1, E_2 and E_3, as shown in Figure B.3.

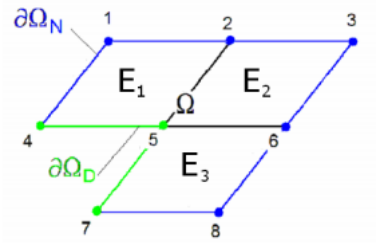


Figure B.3: Division of the domain into three finite elements

For $p = 1$ there are eight basis functions spanning over these elements (compare Figure B.4).

Having these, it is possible to generate a system of linear equations for the new, discretized problem.

$$u \approx u_h = \sum_{i=1}^{N} a_i e_i$$

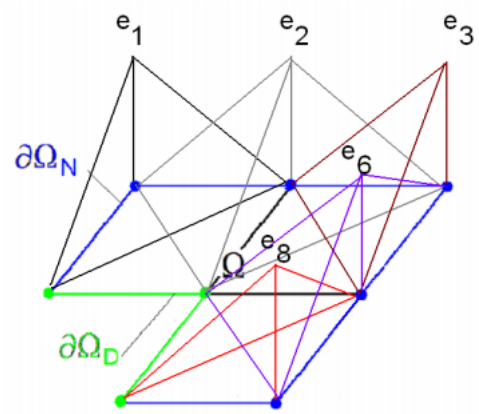


Figure B.4: Basis functions spanning over the domain

$$\sum_{i=1}^{N} a_i b(e_i, e_j) = l(e_j) \quad j = 1, ..., N$$

$$b(e_i, e_j) = \int_\Omega \nabla e_i \cdot \nabla e_j dx = \int_\Omega \sum_{k=1}^{2} \frac{\partial e_i}{\partial x_k} \frac{\partial e_j}{\partial x_k} dx$$

$$l(e_j) = \int_{\partial\Omega_N} e_j g dS$$

The Dirichlet BC is enforced by simply setting rows and columns related to nodes 4, 5 and 7 to zero.

In Figure B.5 we introduce four vertex shape functions over each element Φ_1, Φ_2, Φ_3 and Φ_4 ($p = 1$).

Each basis functions is a sum of one or two shape functions Φ_i. As an example - basis function e_2 consists of shape functions Φ_3 over the element E_1 and the fourth basis function over the element E_2. This is presented in Figure B.6.

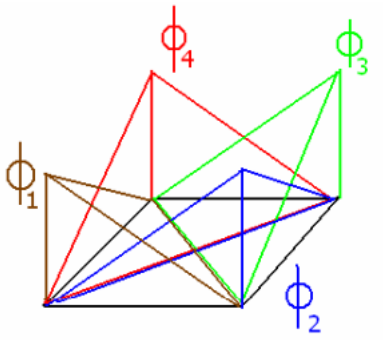


Figure B.5: Vertex basis functions span over the single element

The efficient computation of an arbitrary basis function Φ_i requires some additional, purely technical, improvements. The first of them is to introduce the so called master element $\hat{E} = [0, 1] \times [0, 1]$ and to define a set of template shape functions $\hat{\Phi}_i$ over this element. The template functions are defined in Equations B.15 - B.18. Master element is presented in Figure B.7.

$$\hat{\Phi}_1(\xi_1, \xi_2) = \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2) = (1 - \xi_1)(1 - \xi_2) \tag{B.15}$$

$$\hat{\Phi}_2(\xi_1, \xi_2) = \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2) = \xi_1(1 - \xi_2) \tag{B.16}$$

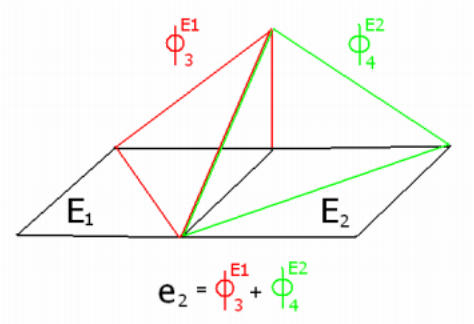


Figure B.6: Vertex basis functions span over two elements

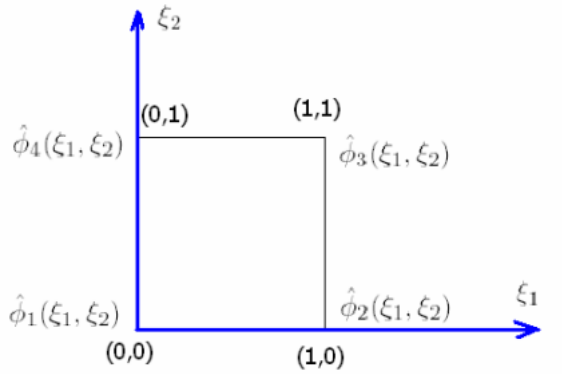


Figure B.7: Image definition of a master element

$$\hat{\Phi}_3(\xi_1,\xi_2) = \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2) = \xi_1\xi_2 \tag{B.17}$$

$$\hat{\Phi}_4(\xi_1,\xi_2) = \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2) = (1-\xi_1)\xi_2 \tag{B.18}$$

Each rectangular element E can be defined based on its location (b_1, b_2), length and height (a_1, a_2). In our case, the elements E_1, E_2 and E_3 defining the L-shape domain are defined in the following way.

$$E_1 : (b_1, b_2) = (-1, 0); (a_1, a_2) = (1, 1) \tag{B.19}$$

$$E_2 : (b_1, b_2) = (0, 0); (a_1, a_2) = (1, 1) \tag{B.20}$$

$$E_3 : (b_1, b_2) = (0, -1); (a_1, a_2) = (1, 1) \tag{B.21}$$

We define the mapping from master element $\hat{E}$ into an arbitrary element E (Equation B.22).

$$\hat{E} \ni (\xi_1, \xi_2) \rightarrow x_{E_k}(\xi_1, \xi_2) = (b_1 + a_1\xi_1, b_2 + a_2\xi_2) = (x_1, x_2) \in E_k \tag{B.22}$$

For the purposes described further we also need reverse mapping for an arbitrary element E into master element $\hat{E}$ (Equation B.23).

$$E_k \ni (x_1, x_2) \rightarrow x^{-1}_{E_k}(x_1, x_2) = (\frac{x_1 - b_1}{a_1}, \frac{x_2 - b_2}{a_2}) = (\xi_1, \xi_2) \in \hat{E} \tag{B.23}$$

In other words

$$x_1 = b_1 + a_1\xi_1 \qquad\qquad x_2 = b_2 + a_2\xi_2 \tag{B.24}$$

and

$$\xi_1 = \frac{x_1 - b_1}{a_1} \qquad\qquad \xi_2 = \frac{x_2 - b_2}{a_2} \tag{B.25}$$

Now, we can prescribe formulas for arbitrary shape functions $\Phi_i^k, i = 1, 2, 3, 4$ defined over element E_k by using the map x_E^{-1}.

$$\begin{aligned}\Phi_1^k(x_1, x_2) &= \hat{\Phi}_1(x_{E_k}^{-1}(x_1, x_2)) \\ &= \hat{\Phi}_1(\frac{x_1 - b_1}{a_1}, \frac{x_2 - b_2}{a_2}) \\ &= \hat{\chi}_1(\frac{x_1 - b_1}{a_1})\hat{\chi}_1(\frac{x_2 - b_2}{a_2}) \\ &= (1 - \frac{x_1 - b_1}{a_1})(1 - \frac{x_2 - b_2}{a_2})\end{aligned} \tag{B.26}$$

$$\begin{aligned}\Phi_2^k(x_1, x_2) &= \hat{\Phi}_2(x_{E_k}^{-1}(x_1, x_2)) \\ &= \hat{\Phi}_2(\frac{x_1 - b_1}{a_1}, \frac{x_2 - b_2}{a_2}) \\ &= \hat{\chi}_2(\frac{x_1 - b_1}{a_1})\hat{\chi}_1(\frac{x_2 - b_2}{a_2}) \\ &= \frac{x_1 - b_1}{a_1}(1 - \frac{x_2 - b_2}{a_2})\end{aligned} \tag{B.27}$$

$$\begin{aligned}\Phi_3^k(x_1, x_2) &= \hat{\Phi}_3(x_{E_k}^{-1}(x_1, x_2)) \\ &= \hat{\Phi}_3(\frac{x_1 - b_1}{a_1}, \frac{x_2 - b_2}{a_2}) \\ &= \hat{\chi}_2(\frac{x_1 - b_1}{a_1})\hat{\chi}_2(\frac{x_2 - b_2}{a_2}) \\ &= \frac{x_1 - b_1}{a_1}\frac{x_2 - b_2}{a_2}\end{aligned} \tag{B.28}$$

$$\begin{aligned}\Phi_4^k(x_1, x_2) &= \hat{\Phi}_4(x_{E_k}^{-1}(x_1, x_2)) \\ &= \hat{\Phi}_4(\frac{x_1 - b_1}{a_1}, \frac{x_2 - b_2}{a_2}) \\ &= \hat{\chi}_1(\frac{x_1 - b_1}{a_1})\hat{\chi}_2(\frac{x_2 - b_2}{a_2}) \\ &= (1 - \frac{x_1 - b_1}{a_1})\frac{x_2 - b_2}{a_2}\end{aligned} \tag{B.29}$$

We can compute integrals of the functions above element-wise, since integral of the sum is the sum of integrals.

$$\begin{aligned}b(\Phi_i^k, \Phi_j^k) = &\int_{E_k} \frac{\partial \Phi_i^k}{\partial x_1}(x_1, x_2)\frac{\partial \Phi_j^k}{\partial x_1}(x_1, x_2)dx_1dx_2 \\ &+ \int_{E_k} \frac{\partial \Phi_i^k}{\partial x_2}(x_1, x_2)\frac{\partial \Phi_j^k}{\partial x_2}(x_1, x_2)dx_1dx_2\end{aligned} \tag{B.30}$$

For $p = 1$ computing the integrals is an easy task as it is sufficient to take the value in the center of the element and the area of the element (a_1a_2). This transformation is presented in Equation B.31.

$$\begin{aligned}b(\Phi_i^k, \Phi_j^k) \approx &[\frac{\partial \Phi_i^k}{\partial x_1}(b_1 + \frac{a_1}{2}, b_2 + \frac{a_2}{2})\frac{\partial \Phi_j^k}{\partial x_1}(b_1 + \frac{a_1}{2}, b_2 + \frac{a_2}{2})a_1a_2 \\ &+ [\frac{\partial \Phi_i^k}{\partial x_2}(b_1 + \frac{a_1}{2}, b_2 + \frac{a_2}{2})\frac{\partial \Phi_j^k}{\partial x_2}(b_1 + \frac{a_1}{2}, b_2 + \frac{a_2}{2})a_1a_2\end{aligned} \tag{B.31}$$

B.3.1. Finite Element Method Algorithm

At this point we can formulate a Finite Element Method algorithm. The algorithm constructs a global system of linear equations to be send to some solver algorithm. The system is build by summing up the integrals of basis functions from particular finite elements.

```
1   B(1:8,1:8)=0 (creation of the global matrix)
2   L(1:8) (creation of the right hand side)
3   Loop with respect to elements E_k, k=1,2,3
4     Loop with respect to functions Φ_i^k, i=1,2,3,4
5     i_1 = row of the matrix related to Φ_i^k
6     L(i_1) += l(Φ_i^k)
7       Loop with respect to functions Φ_j^k, j=1,2,3,4
8       j_1 = row of the matrix related to Φ_j^k
9       B(i_1,j_1) += b(Φ_i^k, Φ_j^k)
10      end loop
11    end loop
12  end loop
13  B(4,1:8)=0 (enforcing Dirichlet b.c. at node 4)
14  B(5,1:8)=0 (enforcing Dirichlet b.c. at node 5)
15  B(7,1:8)=0 (enforcing Dirichlet b.c. at node 7)
16  L(4)=0 (enforcing Dirichlet b.c. at node 4)
17  L(5)=0 (enforcing Dirichlet b.c. at node 5)
18  L(7)=0 (enforcing Dirichlet b.c. at node 7)
19  B(4,4)=1 (1 on diagonal at row 4)
20  B(5,5)=1 (1 on diagonal at row 5)
21  B(7,7)=1 (1 on diagonal at row 7)
```

The sum of integrals from line 9 is assembled into a proper row and column of the global matrix $\mathbf{B}(i_1, j_1)$. This scheme is presented in Equation B.32.

$$B(i_1,j_1) += \int_{E_k} \frac{\partial \Phi_i^k}{\partial x_1}(x_1,x_2)\frac{\partial \Phi_j^k}{\partial x_1}(x_1,x_2)dx_1dx_2 + \int_{E_k} \frac{\partial \Phi_i^k}{\partial x_2}(x_1,x_2)\frac{\partial \Phi_j^k}{\partial x_2}(x_1,x_2)dx_1dx_2 \tag{B.32}$$

The open problem that remains, is how we can translate i and j into the global row i_1 and i_2. To answer this question we observe that each row and column of the global matrix $\mathbf{B}$ is related to one coefficient a_i of one basis function e_i. See Figure B.8 where the shape functions Φ_i^k over elements $k = 1, 2, 3$, are marked in red and the corresponding basis functions e_i are marked in black as row/column numbers in the matrix.

The sum of integrals from line 6 is related to the integration over the boundary (Equation B.33).

$$l(\Phi_i^k) = \int_{E_k \cap \partial\Omega_N} g(x_1,x_2)\Phi_i^k(x_1,x_2)dx_1dx_2 \tag{B.33}$$

We need to determine whether the edges of a given element E_k are located on the Neumann boundary $\partial\Omega_N$. If the answer is positive, then we need to add the additional integral over the edge to the right hand side of the equation.

$$\int_{edge} g(x_1,x_2)\Phi_i^k(x_1,x_2)dx_1dx_2 = g(x_1^*,x_2^*)\Phi_i^k(x_1^*,x_2^*)|edge| \tag{B.34}$$

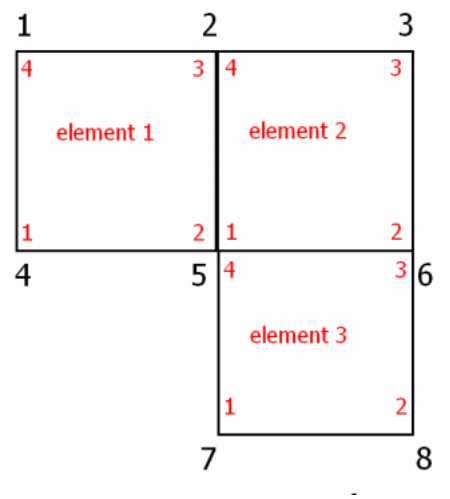


Figure B.8: Order of the integrals

where
(x_1^*, x_2^*) is the central point of the edge,
$g(x_1^*, x_2^*)$ is the g function value in this point,
$\Phi_i^k(x_1^*, x_2^*)$ is the shape function value in this point,
$|edge|$ is the length of the edge.

At this point, we invoke the procedure solving the equation $Ba = L$, and we get the solution defined as the linear combination of the basis functions:

$$u \approx u_h = \sum_{i=1}^{8} a_i e_i \tag{B.35}$$

The solution to the L-shape domain model problem has been presented in Figure B.9.

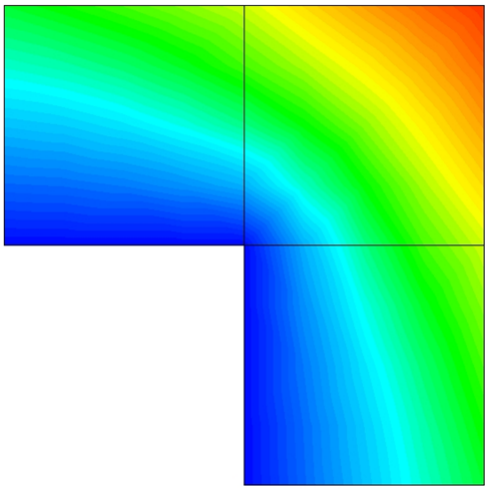


Figure B.9: Solution to the L-shape domain model problem

Some solver algorithms, like the frontal or multi-frontal solvers considered in this work do not construct the global system of linear equations. They rather collect all the element frontal matrices and elements' load vectors. In such the case, lines 3 - 11 from the FEM algorithm are replaced by the following algorithm:

```
3   Loop with respect to elements E_k, k=1,2,3
4     Loop with respect to functions Φ_i^k, i=1,2,3,4
5     L^k(i) += l(Φ_i^k)
6       Loop with respect to functions Φ_j^k, j=1,2,3,4
7       B^k(i,j) += b(Φ_i^k, Φ_j^k)
8       end loop
9     end loop
10  end loop
```

Such an algorithm submits to the solver a sequence of element frontal matrices $\mathbf{B}^k$ and right-hand sides $\mathbf{L}^k$. Nevertheless, it is also necessary to enforce the Dirichlet boundary conditions, which is performed on the element matrices level.

In order to increase the accuracy of the numerical solution, it is also possible to add additional basis functions located at element edges and interiors. In the general case considered in this work, we introduce four shape functions over the four vertices of the two dimensional rectangular element $\{(\xi_1,\xi_2) : \xi_i \in [0,1], i = 1,2\}$:

$$\begin{aligned}
\hat{\Phi}_1(\xi_1,\xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2) \\
\hat{\Phi}_2(\xi_1,\xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2) \\
\hat{\Phi}_3(\xi_1,\xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2) \\
\hat{\Phi}_4(\xi_1,\xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)
\end{aligned} \tag{B.36}$$

$p_i - 1$ shape functions over each of the four edges of the element

$$\begin{aligned}
\hat{\Phi}_{5,j}(\xi_1,\xi_2) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2) \quad j = 1,...,p_1-1 \\
\hat{\Phi}_{6,j}(\xi_1,\xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad j = 1,...,p_2-1 \\
\hat{\Phi}_{7,j}(\xi_1,\xi_2) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_2(\xi_2) \quad j = 1,...,p_3-1 \\
\hat{\Phi}_{8,j}(\xi_1,\xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad j = 1,...,p_4-1
\end{aligned} \tag{B.37}$$

where p_i is the polynomial order of approximation utilized over the i-th edge, and $(p_h - 1) \times (p_v - 1)$ shape functions over an element interior

$$\hat{\Phi}_{9,ij}(\xi_1,\xi_2) = \hat{\chi}_{2+i}(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad i = 1,...,p_h-1, \quad j = 1,...,p_v-1 \tag{B.38}$$

where (p_h, p_v) are the horizontal and vertical polynomial orders of approximation utilized over an element interior, and

$$\begin{aligned}
\hat{\chi}_1(\xi) &= 1-\xi \\
\hat{\chi}_2(\xi) &= \xi \\
\hat{\chi}_3(\xi) &= \xi(1-\xi) \\
\hat{\chi}_l(\xi) &= (1-\xi)\xi(2\xi-1)^{l-3}, l = 4,...,p+1
\end{aligned} \tag{B.39}$$

where p is the polynomial order of approximation over an edge.

In particular, in the case of higher order shape functions, the elimination of an edge e_i implies elimination of all $p_i - 1$ shape functions related to the edge. Also, the elimination of an interior implies elimination of all $(p_h - 1)(p_v - 1)$ shape functions related to the edge.

Appendix C

Schur complements and partial Gaussian eliminations

This Appendix outlines the theoretical foundations of the hypergraph solvers presented in my thesis, which are based on the Schur complement method [41]. Schur complements were named after Issai Schur who used them to prove Schur's Lemma [36]. For more information on the theory of Schur complements, please refer to [30, 61, 102].

C.1. Schur complement

Let M be an $n \times n$ matrix written as a 2×2 block matrix, and b the right hand side vector.

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \qquad b = \begin{pmatrix} c \\ d \end{pmatrix} \tag{C.1}$$

where A is a $p \times p$ matrix and D is a $q \times q$ matrix, with $n = p + q$ (so, B is a $p \times q$ matrix and C is a $q \times p$ matrix). We can try to solve the following linear system:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix} \tag{C.2}$$

hence

$$\begin{aligned} Ax + By &= c \\ Cx + Dy &= d \end{aligned} \tag{C.3}$$

If we assume that D is invertible, we can solve for y by doing the following:

$$y = D^{-1}(d - Cx) \tag{C.4}$$

By substituting this result for y in Equation C.3 we receive:

$$Ax + B(D^{-1}(d - Cx)) = c \tag{C.5}$$

and we transform it into

$$(A - BD^{-1}C)x = c - BD^{-1}d \tag{C.6}$$

Provided that the matrix $(A - BD^{-1}C)$ is invertible, we are able to obtain the solution to our linear system:

$$x = (A - BD^{-1}C)^{-1}(c - BD^{-1}d)$$
$$y = D^{-1}(d - C(A - BD^{-1}C)^{-1}(c - BD^{-1}d))$$

Definition C.1.1. ***Schur complement****. The matrix $A - BD^{-1}C$ is called the* Schur complement *of D in M.*

If A is invertible, we can use the Schur complement of A, which is $D - CA^{-1}B$ to obtain the following factorization of M:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & D - CA^{-1}B \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix} \tag{C.7}$$

Lemma C.1.1. ***Schur complement lemma****. Let S be symmetrical matrix partitioned into blocks.*

$$S = \begin{pmatrix} A & B \\ B^T & C \end{pmatrix} \tag{C.8}$$

where both A and C are symmetric and square. Assume that C is positive definite. Then the following properties are equivalent:

- *S is positive semi-definite.*
- *The Schur complement of C in S, defined as the matrix $A - BC^{-1}B^T$, is positive semi-definite.*

C.2. LU factorization scheme

The idea behind the LU factorization scheme is to find matrices L and U such that:

$$A = LU \tag{C.9}$$

where
A is a matrix to solve
L is a lower triangular matrix
U is an upper triangular matrix

The LU factorization can be implemented by the Gaussian elimination algorithm. Namely, to compute the upper triangular part of the matrix A we need to execute the Gaussian elimination algorithm:

```
for irow = 1 ... sizeof(M)
  for jrow =irow+1 ... sizeof(M)
    b(jrow) = b(jrow) - b(irow)/M(jrow,jrow)
    for col = irow ... sizeof(M)
      M(jrow,col) = M(jrow,col) - M(irow,col)/M(jrow,jrow)
```

We do not consider the pivoting, since all the problems considered in this work are positive definite, and thus the pivoting is unnecessary.

C.3. Relation of Schur complement to partial forward elimination

The Schur complement operation presented in Section C.1 can, in fact, be computed by performing a partial Gaussian elimination. Namely, we only need to trick the Gaussian elimination algorithm so it stops after eliminating all the rows of block A. This results in a much simpler and efficient algorithm, since we do not compute the expensive inverse A^{-1} at any stage of the algorithm.

```
for irow = 1 ... sizeof(A)
  for jrow =irow+1 ... sizeof(M)
    b(jrow) = b(jrow) - b(irow)/M(jrow,jrow)
    for col = irow ... sizeof(M)
      M(jrow,col) = M(jrow,col) - M(irow,col)/M(jrow,jrow)
```

The Schur complement computed by executing the above algorithm is equivalent to the one introduced in Section C.1.

List of figures

List of tables

Abbreviations

BC Boundary Condition

BVP Boundary Value Problems

CFD Computer Fluid Dynamics

CP-graph Composite Programmable Graph

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

FEA Finite Element Analysis

GPU Graphic Processing Unit

IGA Isogeometric analysis

LHS Left-Hand-Side of the equation

MUMPS MUltifrontal Massively Parallel sparse direct Solver

NURBS Non-Uniform Rational B-Splines

PDE Partial Differential Equation

RHS Right-Hand-Side of the equation

Index

Bibliography

[1] M. Adams and J. W. Demmel. Parallel multigrid solver for 3D unstructured finite element problems. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[2] P. Amestoy, I. Duff, and J. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 200(184):501–520, 2000.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal of Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Computer Methods in Applied Mechanics and Engineering*, 32(2):136–156, 2001.

[5] G. C. Archer. *Object-Oriented Finite Element Analysis*. PhD thesis, University of California at Berkeley, 1996.

[6] I. Babuška. The rate of convergence for the finite element method. *SIAM Journal on Numerical Analysis*, 8(2):304–315, 1971.

[7] I. Babuška. *Accuracy estimates and adaptive refinements in finite element computations*. John Wiley & Sons, 1986.

[8] I. Babuška, B. Szabo, and I. Katz. The p-Version of the Finite Element Method. *SIAM Journal on Numerical Analysis*, 18(3):515–545, 1981.

[9] K. Banaś. *Scalability Analysis for a Multigrid Linear Equations Solver*, volume 4967. Springer Berlin Heidelberg, 2008.

[10] K. Banaś and J. Płażek. Parallel iterative solvers for the finite element method. *CAMES*, 20(1), 1997.

[11] G. Bao, G. Hu, and D. Liu. An h-adaptive finite element solver for the calculations of the electronic structures. *Journal of Computational Physics*, 231(14):4967–4979, 2012.

[12] Y. Bazilevs, V. Calo, J. Cottrell, J. Evans, T. Hughes, S. Lipton, M. Scott, and T. Sederberg. Isogeometric analysis: Toward unification of computer aided design and finite element analysis. *Trends in Engineering Computational Technology*, 4:1–16, 2008.

[13] Y. Bazilevs, V. Calo, J. Cottrell, J. Evans, T. Hughes, S. Lipton, M. Scott, and T. Sederberg. Isogeometric analysis using T-splines. *Computer Methods in Applied Mechanics and Engineering*, 199(5-8):229–263, 2010.

[14] M. W. Beal and M. S. Shephard. A General Topology-Based Mesh Data Structure. *Int. J. Numer. Meth. Engng*, (40):1573–1596, 1997.

[15] T. Belytschko and M. Tabbar. H-Adaptive finite element methods for dynamic problems, with emphasis on localization. *International Journal for Numerical Methods in Engineering*, 36(24):4245–4625, 1993.

[16] C. Biniaris. A Three-Dimensional Object-Oriented Distributed Finite Element Solver Based on Mobile Agent Technology. *Electromagnetics*, 24(1):25–37, 2004.

[17] V. Calo, O. Collier, D. Pardo, and M. Paszyński. Computational complexity and memory usage for multi-frontal direct solvers used in p finite element analysis. *Procedia Computer Science*, 4:1854–1861, 2011.

[18] W. Cao and L. Demkowicz. Optimal error estimate of a projection based interpolation for the p-version approximation in three dimensions. *Computers & Mathematics with Applications*, 50(3):359–366, 2005.

[19] D. Chibisov, V. Ganzha, and C. Zenger. Object oriented Finite element calculations using Maple. *Selcuk Journal of Applied Mathematics*, 4(1):58–86, 2003.

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

[21] J. Cottrell, T. Hughes, and Y. Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. Wiley, 2009.

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

[23] L. Demkowicz. *Computing With Hp-adaptive Finite Elements. Vol. 1: One and Two Dimensional Elliptic and Maxwell Problems*. Chapman & Hall CRC, Texas, 2006.

[24] L. Demkowicz. Polynomial exact sequences and projection-based interpolation with application to Maxwell equations. In *Mixed finite elements, compatibility conditions, and applications*, pages 101–158. Springer, 2008.

[25] L. Demkowicz and A. Buffa. H1, H(curl) and H(div)-conforming projection-based interpolation in three dimensions: Quasi-optimal p-interpolation estimates. *Computer Methods in Applied Mechanics and Engineering*, 194(2):267–296, 2005.

[26] L. Demkowicz, P. Gatto, J. Kurtz, M. Paszyński, W. Rachowicz, E. Bleszyński, M. Bleszyński, M. Hamilton, C. Champlin, and D. Pardo. Modeling of bone conduction of sound in the human head using hp-finite elements: Code design and verification. *Computer Methods in Applied Mechanics and Engineering*, 200(21):1757–1773, 2011.

[27] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszyński, W. Rachowicz, and A. Zdunek. *Computing With Hp-adaptive Finite Elements. Vol. 2: Frontiers: Three Dimensional Elliptic and Maxwell Problems with Applications*. Chapman & Hall CRC, Texas, 2006.

[28] L. Demkowicz, J. T. Oden, W. Rachowicz, and O. Hardy. Toward a universal hp adaptive finite element strategy, part 1. Constrained approximation and data structure. *Computer Methods in Applied Mechanics and Engineering*, 77(1):79–112, 1989.

[29] L. Demkowicz, W. Rachowicz, and P. Devloo. A fully automatic hp-adaptivity. *Journal of Scientific Computing*, 17(1-4):117–142, 2002.

[30] J. Demmel. *Applied Numerical Linear Algebra*. SIAM Publications, 1997.

[31] P. R. B. Devloo. An object oriented environment for scientific programming. *Computational Methods Applied Mechanics and Engineering*, (150):133–153, 1997.

[32] J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. 2003.

[33] J. J. Dongarra and R. A. van de Geijn. Lapack working note 37: Two dimensional basic linear algebra communication subprograms. 1991.

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

[35] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *International Journal of Numerical Methods in Engineering*, (5):633–641, 1984.

[36] D. Dummit and R. Foote. *Abstract Algebra*. Wiley, 2003.

[37] M. R. Dörfel, B. Jüttler, and B. Simeon. Adaptive isogeometric analysis by local h-refinement with T-splines. *Computer Methods in Applied Mechanics and Engineering*, 199(5–8):264 – 275, 2010.

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

[39] J. Gawąd, M. Paszyński, P. Matuszyk, and L. Madej. Cellular automata coupled with hp-adaptive finite element method applied to simulation of austenite-ferrite phase transformation with a moving interface. *Steel Research International*, 79:579–586, 2008.

[40] C. Gerald and P. Wheatley. *Applied Numerical Analysis*. Addison Wesley Longman Inc., 1997.

[41] P. Gill, W. Murray, M. Saunders, and M. Wright. *A Schur-complement method for sparse quadratic programming*. 1987.

[42] E. Grabska. Theoretical Concepts of Graphical Modeling. Part One: Realization of CP-Graphs. *Machine Graphics and Vision*, 2(1):3–38, 1993.

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

[44] E. Grabska and G. Hliniak. Structural Aspects of CP-Graph Languages, Schedae Informaticae. *Machine Graphics and Vision*, (5):81–100, 1993.

[45] G. D. Guerrero, R. M. Wallace, J. L. Vázquez-Poletti, J. M. Cecilia, J. M. García, D. Mozos, and H. Pérez-Sánchez. A performance/cost model for a cuda drug discovery application on physical and public cloud infrastructures. *Concurrency and Computation: Practice and Experience*, 2013.

[46] B. Guo. Approximation Theory for the p-Version of the Finite Element Method in Three Dimensions Part II: Convergence of the P Version of the Finite Element Method. *SIAM Journal on Numerical Analysis*, 47(4):2578–2611, 2009.

[47] B. Guo and I. Babuška. The hp version of the finite element method. *Computational Mechanics*, 1(1):21–41, 1986.

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

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

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

[51] P. Gurgul. A Linear Complexity Direct Solver for H-adaptive Grids With Point Singularities. *Procedia Computer Science*, 2014.

[52] P. Gurgul, A. Paszyńska, and M. Paszyński. Hypergraph grammar based linear computational cost solver for three dimensional grids with point singularities. *Procedia Computer Science*, 2014.

[53] P. Gurgul, M. Sieniek, K. Magiera, and M. Skotniczny. Application of multi-agent paradigm to hp-adaptive projection-based interpolation operator. *Journal of Computational Science*, 4(3):164–169, 2013.

[54] P. Gurgul, M. Sieniek, and M. Paszyński. Object-oriented multiscale hp-adaptive finite element method. *Computer Methods In Material Science*, 9(1):289–295, 2009.

[55] P. Gurgul, M. Sieniek, M. Paszyński, L. Madej, and N. Collier. Two dimensional hp-adaptive algorithm for continuous approximations of material data using space projections. *Computer Science*, 14(1):97–112, 2013.

[56] A. Habel and H. J. Kreowski. May We Introduce to You: Hyperedge Replacement. *Lecture Notes in Computer Science*, 291:5–26, 1987.

[57] A. Habel and H. J. Kreowski. Some Structural Aspects of Hypergraph Languages Generated by Hyperedge Replacement. *Lecture Notes in Computer Science*, 247:207–219, 1987.

[58] W. Hackbusch. Multi-grid methods for FEM and BEM applications. In E. Stein, editor, *Encyclopedia of computational mechanics. Vol 1 : Fundamentals*, pages 577–595. Wiley, Chichester, 2004.

[59] W. Hackbusch and B. Khoromskij. A Sparse H-Matrix Arithmetic. Part II: Application to Multi- Dimensional Problems. *Computing*, 2000:21–47, 2000.

[60] W. Hackbusch, B. N. Khoromskij, and R. Kriemann. Direct Schur complement method by domain decomposition based on H-matrix approximation. *Computing and visualization in science*, 8(3/4):179–188, 2005.

[61] R. Horn and R. Johnson. *Matrix Analysis*. Cambridge University Press, 1990.

[62] T. Hughes, J. Cottrell, and Y. Bazilevs. *Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement*, volume 194. Elsevier, 2005.

[63] T. Hughes and L. Franca. A new FEM for computational fluid dynamics: VII The Stokes problem with various well-posed boundary conditions symmetric formulations that converge for all velocity pressure spaces. *Comput. Methods Appl. Mech. Engrg.*, 65:85–96, 1987.

[64] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. V. Zee. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel Distrib. Comput.*, 72(9):1134–1143, 2012.

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

[66] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[67] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Siam Journal of Scientific Computing*, 20(1):359–392, 1998.

[68] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 1(20):359–391, 1999.

[69] D. Kelly, D. S. Gago, O. Zienkiewicz, I. Babuska, et al. A posteriori error analysis and adaptive processes in the finite element method: Part I - error analysis. *International Journal for Numerical Methods in Engineering*, 19(11):1593–1619, 1983.

[70] D. Komatitsch, D. Michea, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics card using CUDA. *J. Parallel Distrib. Computing*, 69(5):451–460, 2009.

[71] K. Kuźnik, M. Paszyński, and V. Calo. Grammar-Based Multi-Frontal Solver for One Dimensional Isogeometric Analysis with Multiple Right-Hand-Sides. *Procedia Computer Science*, 18:1574–1583, 2013.

[72] X. Li, J. Zheng, T. W. Sederberg, T. J. R. Hughes, and M. A. Scott. On Linear Independence of T-Splines. *ICES Report*, 10(40), 2010.

[73] H. Liu and D. Jiao. An H-LU Based Direct Finite Element Solver Accelerated by Nested Dissection for Large-scale Modeling of ICs and Packages. 6(7), 2010.

[74] C. Manni, F. Pelosi, and M. L. Sampoli. Generalized B-splines as a tool in isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 200(5–8):867 – 881, 2011.

[75] G. Markall and P. Kelly. Accelerating Unstructured Mesh Computational Fluid Dynamics Using the NVidia Tesla GPU Architecture. *ISO Report*, 2009.

[76] D. McRae. r-Refinement grid adaptation algorithms and issues. *Computer Methods in Applied Mechanics and Engineering*, 189(4):1161–1182, 2000. Adaptive Methods for Compressible CFD.

[77] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[78] J. T. Oden, L. Demkowicz, and T. Strouboulis. *Adaptive methods for problems in solid and fluid mechanics*. John Wiley and Sons Ltd., 1986.

[79] D. Pardo. *Integration of hp-adaptavity with a two grid solver: applications to electromagnetics*. PhD thesis, The University of Texas at Austin, 2004.

[80] D. Pardo, V. Calo, C. Torres-Verdin, and M. Nam. Fourier series expansion in a non-orthogonal system of coordinates for the simulation of 3D-DC borehole resistivity measurements. *Computer Methods in Applied Mechanics and Engineering*, 197(21):1906–1925, 2008.

[81] D. Pardo, L. Demkowicz, C. Torres-Verdin, and M. Paszyński. Two-dimensional high-accuracy simulation of resistivity logging-while-drilling (LWD) measurements using a self-adaptive goal-oriented hp finite element method. *SIAM Journal on Applied Mathematics*, 66(6):2085–2106, 2006.

[82] D. Pardo, L. Demkowicz, C. Torres-Verdin, and M. Paszyński. A self-adaptive goal-oriented hp-finite element method with electromagnetic applications. Part II: Electrodynamics. *Computer Methods in Applied Mechanics and Engineering*, 196(37):3585–3597, 2007.

[83] D. Pardo, L. Demkowicz, C. Torres-Verdín, and L. Tabarovsky. A goal-oriented hp-adaptive finite element method with electromagnetic applications. Part I: Electrostatics. *International journal for numerical methods in engineering*, 65(8):1269–1309, 2006.

[84] D. Pardo, C. Torres-Verdin, and M. Paszyński. Simulations of 3d dc borehole resistivity measurements with a goal-oriented hp finite-element method. Part II: Throughcasing resistivity instruments. *Computational Geosciences*, 12(1):83–89, 2008.

[85] A. Paszyńska, M. Paszyński, and E. Grabska. Graph transformations for modeling hp-adaptive finite element method with mixed triangular and rectangular elements. *Lecture Notes in Computer Science*, 5545:875–884, 2009.

[86] M. Paszyńska, P. Gurgul, M. Sieniek, and M. Paszyński. Linear computational cost graph grammar based direct solver for 3D adaptive finite element method simulations. *IJMMM International Journal of Materials, Mechanics and Manufacturing*, 2013(1):225–230, 2013.

[87] M. Paszyński. *Graph Grammar-Driven Parallel Adaptive PDE Solvers*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH, Krakow, 2009.

[88] M. Paszyński. On the parallelization of self-adaptive hp Finite Element Methods. Part I. Composite Programmable Graph Grammar Model. *Fundamenta Informaticae*, (93):411–434, 2009.

[89] M. Paszyński and A. Paszyńska. Graph transformations for modeling parallel hp-adaptive finite element method. *Lecture Notes in Computer Science*, (4967):1313–1322, 2008.

[90] B. Patzak. Object Oriented Finite Element Modeling. *Acta Polytechnica*, 39(2):93–113, 2010.

[91] L. Piegl and W. Tiller. *The NURBS Book*. Springer, 1995.

[92] A. Polyanin. *Handbook of Linear Partial Differential Equations for Engineers and Scientists*. CRC Press, 2002.

[93] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw.*, 39(2):13, 2013.

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

[95] M. Scott, X. Li, T. Sederberg, and T. Hughes. Local refinement of analysis-suitable T-splines. *Computer Methods in Applied Mechanics and Engineering*, 213:206–222, 2012.

[96] M. A. Scott, M. J. Borden, C. V. Verhoosel, T. W. Sederberg, and T. J. Hughes. Isogeometric finite element data structures based on Bézier extraction of T-splines. *International Journal for Numerical Methods in Engineering*, 2011.

[97] R. Sevilla, S. Fernández-Méndez, and A. Huerta. NURBS-Enhanced Finite Element Method (NEFEM). *Archives of Computational Methods in Engineering*, 18(4):441–484, 2011.

[98] M. Sieniek, P. Gurgul, and M. Paszyński. Employing adaptive finite elements to model squeezing of a layered material in 3D. *IJMMM International Journal of Materials, Mechanics and Manufacturing*, 2013(1):319–323, 2013.

[99] M. Sieniek, P. Gurgul, M. Skotniczny, K. Magiera, and M. Paszyński. Agent-oriented image processing with the hp-adaptive projection-based interpolation method. *Procedia Computer Science*, 4(1):1844–1853, 2011.

[100] G. Ślusarczyk and A. Paszyńska. Hypergraph Grammars in hp-adaptive Finite Element Method. *Procedia Computer Science*, 18:1545–1554, 2013.

[101] N. Staniforth, L. Herschel, and J. Mitchell. A Variable-Resolution Finite-Element Technique for Regional Forecasting with the Primitive Equations. *Monthly Weather Review*, 106:439–447, 1978.

[102] G. Strang. *Linear Algebra and its Applications*. Saunders HBJ, 1988.

[103] A. Thom and C. Apelt. *Field Computations in Engineering and Physics*. D. Van Nostrad, 1961.

[104] M. Uhruski, M. Grochowski, and R. Schaefer. A two-layer agent-based system for large-scale distributed computation. *Computational Intelligence*, 24(3):191–212, 2008.

[105] R. A. Van de Geijn. *Using PLAPACK - parallel linear algebra package*. The MIT Press, 1997.

[106] A. Vuong, C. Gianelli, B. Juttler, and B. Simeon. A Hierarchical Approach to Adaptive Local Refinement in Isogeometric Analysis. 2011.

[107] I. Yamazaki, T. Dong, R. Solca, S. Tomov, J. Dongarra, and T. Schulthess. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*, 2013(1), 2013.

[108] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals (Sixth ed.)*. Butterworth-Heinemann, 2005.

[109] J. Zitelli, I. Muga, L. Demkowicz, J. Gopalakrishnan, D. Pardo, and V. M. Calo. A Class of discontinuous Petrov-Galerkin methods. Part IV: The optimal test norm and time-harmonic wave propagation in 1D. *Journal of Computational Physics*, 230(7):2406–2432, 2011.

[ISI] – ISI Master Journal List
[WoS] – Web of Science

List of papers

1. [WoS] **Gurgul P.**, 2014, A Linear Complexity Direct Solver for H-adaptive Grids With Point Singularities, accepted to Procedia Computer Science.

2. [WoS] **Gurgul P.**, Paszyńska A., Paszyński M., 2014, Hypergraph grammar based linear computational cost solver for three dimensional grids with point singularities, accepted to Procedia Computer Science.

3. Abou-Eisha H., **Gurgul P.**, Paszyńska A., Paszyński M., Kuźnik K., Moshkov M., 2013, An automatic way of finding robust elimination trees for a multi-frontal sparse solver for radical 2D hierarchical meshes, accepted to Lectures Notes in Computer Science.

4. Sieniek M., **Gurgul P.,** Paszyński M., Fast multi-scale simulations of a Step-and-Flash Imprint Lithography, Proceedings of 5th Asia Pacific Congress on Computational Mechanics & 4th International Symposium on Computational Mechanics, in press.

5. [ISI] [WoS] **Gurgul P.**, Sieniek M., Magiera K., Skotniczny M., 2013, Application of multi-agent paradigm to hp-adaptive projection-based interpolation operator, Journal of Computational Science, 3(4):164–169.

6. [WoS] Szymczak A., Paszyńska A., **Gurgul P.**, Paszyński M., 2013, Graph grammar based direct solver for hp-adaptive finite element method with point singularities, Procedia Computer Science, 18(2013):1594-1603.

7. Sieniek M., **Gurgul P.**, Paszyński M., 2013, Employing adaptive finite elements to model squeezing of a layered material in 3D, IJMMM International Journal of Materials, Mechanics and Manufacturing, 1(2013):319–323.

8. Paszyńska A., **Gurgul P.,** Sieniek M., Paszyński M., 2013, Linear computational cost graph grammar based direct solver for 3D adaptive finite element method simulations, IJMMM International Journal of Materials, Mechanics and Manufacturing, 1(2013):225–230.

9. **Gurgul P.,** Syrek R., 2013, Testing of dependencies between stock returns and trading volume by high frequency data, Managing Global Transitions : international research journal, 11(4):353–373.

10. **Gurgul P.**, Sieniek M., Paszyński M., Madej Ł., Collier N., 2012, Two-dimensional hp-adaptive algorithm for continuous approximations of material data using space projection, Computer Science, 14(2012):97–112.

11. **Gurgul P.**, Sieniek M., Paszyński M., Madej Ł., Three-dimensional adaptive algorithm for continuous approximations of material data using space projection, Computer Methods in Materials Science : quarterly, 13(2013):245–250.

12. **Gurgul P.**, Zając P., 2012, Forecasting of migration matrices in business demography, Statistics in Transition: new series: an international journal of the Polish Statistical Association, 13(2012):387–404.

13. **Gurgul P.,** 2012, New mobile marketing capabilities of the Android platform, Managerial Economics, 12(2012): 119-129.

14. **Gurgul P.**, 2012, Marketing mobilny na smartfonach, Zeszyty Naukowe Wyższej Szkoły Ekonomii i Informatyki w Krakowie, 8(2012):141-149.

15. [WoS] **Gurgul P.**, Sieniek M., Magiera K., Skotniczny M., Paszyński M., 2011, Agent-oriented image processing with the hp-adaptive projection-based interpolation method, 2011, Procedia Computer Science, Elsevier, 4(2011):1844-1853.

16. [WoS] Sieniek M., **Gurgul P.**, Kołodziejczyk P., Paszyński M., 2010, Agent-based parallel system for numerical computations, Procedia Computer Science, Elsevier, 1(2010):1971-1981.

17. **Gurgul P.**, Sieniek M., Paszyński M., 2009, Object-oriented Multiscale HP-Adaptive Finite Element Method, Computer Methods in Materials Science, 9(2009):289-295.