Contents

MACIEJ PASZYŃSKI

Graph grammar-driven parallel adaptive PDE solvers

Summary

The main thesis of the work is to develop a formal description of a wide class of adaptive mesh-based algorithms, which will allow us to examine their concurrency, as well as to design and to implement an efficient parallel version of the algorithms.

The way to achieve this goal is to develop a graph grammar-based formal description of the adaptive mesh-based algorithms. The graph grammar model makes it possible to examine the concurrency of the algorithms by analysing the interdependence between the atomic tasks, tasks and super-tasks. The atomic tasks correspond to the graph grammar productions, representing basic undividable parts of the algorithms. The level of atomic tasks models the concurrency for the shared memory architectures. On the other hand, the tasks correspond to the groups of atomic tasks with predefined inter-task communication channels. They constitute the grain for the decomposition of the parallel algorithm for the distributed memory architecture. Finally, the super-tasks correspond to a group of tasks resulting from the execution of load balancing algorithm. The graph grammar-based model will be developed for the self-adaptive *hp* Finite Element Method (*hp*-FEM), which is the most complicated adaptive mesh-based algorithm, intended for high accuracy numerical solutions of a weak form of elliptic and Maxwell Partial Differential Equations (PDE). Most other mesh-based adaptive algorithms are embedded into the self-adaptive *hp*-FEM, thus the developed formalism can be also applied to other algorithms of this class.

The particular adaptive algorithms considered in this work employ the two-grid paradigm, with the coarse meshes and their corresponding fine meshes. The algorithms generate a sequence of coarse meshes, delivering convergence of the numerical error of a considered problem, with respect to the mesh size. They use the corresponding fine mesh to estimate the quality of the coarse mesh in order to select the optimal refinements to be performed. The creation of an efficient parallel version of the algorithm is critical for most applications, since the computational cost related to the adaptive computations is large.

The particular result of this work is the parallel adaptive software system that implements the self-adaptive *hp*-FEM algorithm. Many challenging engineering problems, including material science, geo-science and remote sensing, nano-lithography (micro-cheap production process) and wave propagation problems, require high accuracy of a numerical solution. It is impossible to find such a solution using other numerical methods. These problems will be finally solved by the developed software system, thanks to the exponential convergence of the self-adaptive *hp*-FEM algorithm. On the other hand, the applicability of the two-grid paradigm adaptive algorithms is not limited only to the PDE engineering problems. Any problem that requires a construction of the finite dimensional approximation space based on polynomials of different orders covering finite element mesh fits into the class of adaptive mesh-based algorithms.

MACIEJ PASZYŃSKI

Równoległe solvery adaptacyjne równań różniczkowych cząstkowych sterowane gramatyką grafową

Streszczenie

Głównym celem pracy jest stworzenie formalnego opisu dla klasy algorytmów adaptacyjnych, pozwalającego na zbadanie współbieżności tkwiącej w algorytmach, a także na zaprojektowanie i implementacje efektywnych wersji równoległych algorytmów.

Szczególnym celem pracy jest opracowanie formalizmu bazującego na gramatykach grafowych, opisującego algorytmy adaptacyjne wykorzystujące siatki obliczeniowe. Model gramatyk grafowych pozwala na analizę współbieżności tkwiącej w algorytmach, poprzez zbadanie zależności pomiędzy zadaniami podstawowymi, zadaniami i super-zadaniami. Zadania podstawowe to produkcje gramatyki grafowej reprezentujące niepodzielne części algorytmu adaptacyjnego, przeznaczone dla architektur o pamięci współdzielonej. Zadania zdefiniowane są poprzez zgrupowanie zadań podstawowych oraz określenie kanałów komunikacyjnych. Reprezentują one ziarno do podziału zadania obliczeniowego dla architektur z pamięcią rozproszoną. Super-zadania reprezentują grupy zadań uzyskane w wyniku wykonania algorytmu balansowania obciążeń.

Wspomniany formalizm oparty na gramatykach grafowych wykorzystany zostanie do opisu algorytmu automatycznej *hp* adaptacji, stanowiącego najbardziej zaawansowany algorytm adaptacyjny pracujący na siatkach obliczeniowych, stosowany do dokładnego rozwiązywania sformułowań wariacyjnych eliptycznych równań różniczkowych cząstkowych. Większość znanych algorytmów adaptacyjnych wykorzystujących siatki obliczeniowe stanowi uproszczone wersje wspomnianego algorytmu.

Algorytmy adaptacyjne rozważane w niniejszej pracy bazują na paradygmacie dwóch siatek obliczeniowych: siatki rzadkiej oraz siatki gęstej. Algorytmy generują ciąg siatek obliczeniowych dostarczających eksponencjalną zbieżność względnego błędu numerycznego względem rozmiaru siatki. Stworzenie efektywnej wersji algorytmów adaptacyjnych jest niezmiernie ważne dla większości aplikacji inżynieryjnych. Dzieje się tak z uwagi na duży koszt obliczeniowy wspomnianych algorytmów adaptacyjnych. Praktycznym wynikiem prezentowanej pracy jest równoległy kod obliczeniowy, implementujący algorytm automatycznej *hp* adaptacji. Wiele trudnych problemów inżynieryjnych, mających zastosowanie w dziedzinach takich jak inżynieria materiałowa, geologia, nano-litografia oraz zagadnienia propagacji fal elektromagnetycznych, zostało rozwiązanych z wymaganą dokładnością dzięki eksponencjalnej zbieżności algorytmów automatycznej *hp* adaptacji. Wymagana wysoka dokładność nie jest możliwa do uzyskania za pomocą klasycznych metod numerycznych. Opracowany formalizm może być z powodzeniem stosowany do opisu szerokiej klasy algorytmów bazujących na paradygmacie dwóch siatek obliczeniowych.

Index of symbols

Ω	–	Domain where the PDE solution is looked for
Γ	–	Boundary of Ω
Γ_D	–	Part of Γ where Dirichlet boundary conditions are defined
Γ_N	–	Part of Γ where Neumann boundary conditions are defined
Γ_C	–	Part of Γ where Cauchy boundary conditions are defined
u	–	Unknown scalar or vector field
R	–	Space of real numbers
u_D	–	Scalar or vector field assigned to the Dirichlet boundary
g	–	Scalar or vector field assigned to the Neumann boundary
$b(u,v)$	–	Bilinear form of the variational formulation
$l(v)$	–	Right-hand-side linear form of the variational formulation
V	–	The space of test functions (in chapters 1 and 3)
v	–	Test function $v \in V$ (in chapters 1 and 3)
$L^2(\Omega)$	–	The space of functions integrable with the second power (Lebesque space)
$H^1(\Omega)$	–	The subspace of $L^2(\Omega)$, with functions integrable with the second power and with the derivative (with the second power) (Sobolev space)
$\|v\|$	–	The norm in the Lebesque space
$\|v\|_{1,\Omega}$	–	The norm in the Sobolev space
V_{hp}	–	Coarse mesh approximation space $V_{hp} \subset V$
$\left\{e^{\,i}_{hp}\right\}_{i=1,\ldots,N_{hp}}$	–	Basis of the coarse mesh approximation space
u_{hp}	–	The approximate solution on the coarse mesh

u^{i}_{hp}	–	Components of the coarse mesh approximate solution, called *degrees of freedom*
N_{hp}	–	Dimension of the coarse mesh approximation space
$V_{h/2,p+1}$	–	Fine mesh approximation space
$\left\{ e^{i}_{h/2,p+1} \right\}_{i=1,\ldots,N_{h/2,p+1}}$	–	Basis of the fine mesh approximation space
$u_{h/2,p+1}$	–	The approximate solution on the fine mesh
$u^{i}_{h/2,p+1}$	–	Components of the fine mesh approximate solution, called *degrees of freedom*
$N_{h/2,p+1}$	–	Dimension of the fine mesh approximation space
K	–	A single finite element
$\|v\|_{1,K}$	–	The norm in the $H^1(K)$ space – the space restricted to a single element K
$\text{error}_{\text{rel}}$	–	Relative error – the $H^1(\Omega)$ norm of the difference between the coarse and fine mesh solutions
w	–	The solution corresponding to a considered refinement of the coarse mesh approximation space V_{hp} restriced to a single element K
$\text{rate}(w)$	–	Error decrease rate corresponding to w
$\Delta\,\text{nrdof}$	–	The increase in the number of degrees of freedom on the coarse mesh element K, resulting from execution of the considered mesh refinement strategy
(p_h, p_v)	–	Horizontal and vertical polynomial orders of approximations used for an element interior
$\text{error}_K(t)$	–	Relative error between the coarse and fine mesh solutions at time step t over an element K
$U^{p}_{K}(t)$	–	Coarse mesh solution for the uniform p coarse mesh at time step t for an element K
$U^{p+1}_{K}(t)$	–	Fine mesh solution for the uniform p+1 fine mesh at time step t for an element K
V	–	Set of graph vertices (in chapter 2)
A_V	–	Set of vertex labels
ξ_V	–	Vertices labeling function
$[i]_N$	–	Continuous integer numbering of vertex bounds
$B(V)$	–	Set of pairs – vertex bound number and the vertex itself
$\beta(\xi_V(v))$	–	Projection onto vertex bound number
E	–	Set of graph edges
A_E	–	Set of edge labels

ξ_E	–	Edge labeling function
att_V	–	Vertex attributing function
att_E	–	Edge attributing function
A_T	–	Set of vertex attributes
A_R	–	Set of edge attributes
W	–	Set of graph vertices with numbered bounds
P	–	Finite set of graph grammar productions
S	–	Axiom of the graph grammar
T, T1	–	Non-terminals of the graph grammar used during the initial mesh generation
Iel, Iel1, Iel2	–	Non-terminals of the graph grammar representing a single initial mesh element
iel	–	Non-terminal of the graph grammar representing a single initial mesh element with the topology already generated
v, v2	–	Non-terminals of the graph grammar representing finite element vertex
F	–	Non-terminal of the graph grammar representing finite element edge
I	–	Non-terminal of the graph grammar representing finite element interior
B	–	Non-terminal of the graph grammar representing boundary of the domain
fake	–	Non-terminal of the graph grammar used for the second fake initial mesh element, to which belongs an edge located on the boundary
N, S, W, E	–	Attributes of graph edges denoting north, south, west and east edges of an element
NW, NE, SW, SE	–	Attributes of graph edges denoting north-west, north-east, south-west and south-east vertices of an element
P1 – P6	–	Graph grammar productions generating the topology of an inital mesh
PII	–	Graph grammar production generating the structure of an inital mesh
PIC	–	Graph grammar production identifying common edges of two adjacent initial mesh elements
$\mathbf{A}_i$	–	Part of element local matrix related to interactions of element interior shape functions
$\mathbf{B}_i$, $\mathbf{C}_i$	–	Part of element local matrices related to interactions of element interior shape functions with common edge shape function
$\mathbf{A}_i^s$	–	Part of element local matrices related to interactions of common edge shape functions

$\mathbf{x}_i$	–	Degrees of freedom related to element interior
$\mathbf{b}_i$	–	Right-hand-side term related to element interior
$\mathbf{x}_i^s$	–	Degrees of freedom related to element interface (e.g. common edge between a pair of adjacent elements)
$\mathbf{b}_i^s$	–	Right-hand-side term related to element interface (e.g. common edge between a pair of adjacent elements)
$\mathbf{A}_i^{s*}$	–	Contributions to the interface problem related to the element interface (e.g. common edge between a pair of adjacent elements)
$\hat{\mathbf{A}}$	–	Matrix of the Schur complement
$\hat{\mathbf{b}}$	–	Right-hand-side term of the Schur complement
$\hat{\mathbf{x}}$	–	Degrees of freedom related to the Schur complement
$\mathbf{P}_i$	–	Permutation matrices, transforming element local ordering of degrees of freedom located on the interface into the global ordering on the interface
$n_{\text{interface}}^i$	–	Number of degrees of freedom for i^{th} element (sub-domain)
$n_{\text{interface}}$	–	Number of degrees of freedom for the global interface
(P aggregate boundary edge)	–	Graph grammar production for the aggregation of degrees of freedom related to element edge located on the domain boundary
(P aggregate corner vertex)	–	Graph grammar production for the aggregation of degrees of freedom related to element vertex located at the corner of domain
(P aggregate edge)	–	Graph grammar production for the aggregation of degrees of freedom related to elements common edge
(P aggregate shared vertex)	–	Graph grammar production for the aggregation of degrees of freedom related to elements common vertex
(P eliminate interior)	–	Graph grammar production for the elimination of internal degrees of freedom from *i*-th frontal matrix
(P eliminate boundary edge)	–	Graph grammar production for the elimination of degrees of freedom related to boundary edges
(P eliminate corner vertex)	–	Graph grammar production for the elimination of degrees of freedom related to corner vertices
(P eliminate edge)	–	Graph grammar production for the elimination of degrees of freedom related to elements common edge
(P eliminate common vertex)	–	Graph grammar production for the elimination of degrees of freedom related to elements common vertex
α^i	–	Symbol denoting aggregation of degrees of freedom to i^{th} frontal matrix
$\alpha^{i,j}$	–	Symbol denoting aggregation of degrees of freedom to both i^{th} and j^{th} frontal matrices
β^i	–	Symbol denoting elimination of degrees of freedom from i^{th} frontal matrix

$\beta^{i,j}$	–	Symbol denoting elimination of degrees of freedom from both i^{th} and j^{th} frontal matrices and merging the resulting Schur complement contributions
(P break interior)	–	Graph grammar production for breaking an element interior
(P break edge)	–	Graph grammar production for breaking an element edge
E		Non-terminal of the graph grammar denoting edge that can be broken
e, e2, e3	–	Non-terminals of the graph grammar denoting broken edge
F, Fi, F1, F2, Fe	–	Non-terminals of the graph grammar denoting edge that cannot be broken
i	–	Non-terminal of the graph grammar denoting broken interior
J, J1, J2, J3, J4	–	Non-terminals of the graph grammar denoting interiors that cannot be broken
(PFE1) – (PFE4)	–	Graph grammar productions allowing for breaking an element edge
(PJI)	–	Graph grammar production allowing for breaking an element interior
(Pwest), (Peast), (Pnorth), (Psouth)	–	Graph grammar productions for the reconstruction of the connectivities between edges and interiors in western, eastern, northern and southern directions
(P p-refine interior)	–	Graph grammar production performing local p refinement at graph vertex denoting an element interior
(P min-rule east edge),	–	Graph grammar productions enforcing the minimum rule on the eastern edges
(P min-rule west edge),	–	Graph grammar productions enforcing the minimum rule on the western edges
(P min-rule north edge),	–	Graph grammar productions enforcing the minimum rule on the northern edges
(P min-rule south edge)	–	Graph grammar productions enforcing the minimum rule on the southern edges
(P process son elements)	–	Graph grammar production for the aggregation on son elements of a broken element
(P process shared vertex)	–	Graph grammar production for the aggregation on shared vertex located on the domain boundary
(P aggregate interface)	–	Graph grammar production for the aggregation on the interface
(P propagate interface aggregation)	–	Graph grammar production propagating aggregation on the interface into father element of broken elements, adjacent to the interface
(P eliminate boundary edge)	–	Graph grammar production for the elimination of degrees of freedom related to boundary edges

(P eliminate corner vertex)	–	Graph grammar production for the elimination of degrees of freedom related to corner vertices
(P merge interiors horizontal pairs)	–	Graph grammar production for merging the Schur complement contributions from adjacent pairs of elements into the common edge submatrix
(P process shared vertex)	–	Graph grammar production for merging the Schur complement contributions from adjacent pairs of elements into the common vertices submatrix
(P merge into vertical pairs)	–	Graph grammar production for the elimination fully assembled edges and vertices, and merging the resulting Schur complements
(P merge sons problem)	–	Graph grammar production merging the Schur complements resulting from son elements
(P process interface)	–	Graph grammar production for the construction of the common interface problem
(P compute relative error)	–	Graph grammar production for the computation of relative error on an element interior
(P evaluate refinement)	–	Graph grammar production for the evaluation of a given refinement strategy
(P select optimal refinement)	–	Graph grammar production for the selection of the optimal refinement
(P virtual href)	–	Graph grammar production for the execution of optimal refinements
(P propagate virtual href)	–	Graph grammar production enforcing the 1-irregularity rule
V_w	–	The approximation space solution corresponding to a considered refinement of the coarse mesh approximation space V_{hp}
error_K	–	Relative error between the coarse and fine mesh solutions for an element K
max_error	–	Maximum relative error for all finite elements
BI	–	Non-terminal of the graph grammar representing interface between two sub-domains
Join	–	Non-terminal of the graph grammar for merging the two sub-domains into a single sub-domain
(DD1)	–	Graph grammar production for the partition of two adjacent elements
(DD2)	–	Graph grammar production for the partition of an edge with one son edge broken and the other one unbroken
(DD3)	–	Graph grammar production for the partition of an edge with two unbroken son edges
(DD4)	–	Graph grammar production for the partition of an edge with two broken son edges
(DD5)	–	Graph grammar production for the initialization of the merging process for two adjacent sub-domains

(DD6) – Graph grammar production for merging two broken edges

(DD7) – Graph grammar production for merging two unbroken edges

(DD8) – Graph grammar production for merging two broken son edges

(PIInt) – Graph grammar production for distinguishing the boundary from the interface

(P compute load) – Graph grammar production for the computation of load estimation an element

(P propagate compute load) – Graph grammar production for the propagation of load estimation

(P compute load init element) – Graph grammar production for the computation of the load estimation for an initial mesh element

$T^{i}_{comp1}\left(N_{elem}\right)$ – Time spend by *i*-th super-task on the computation of the computational cost

$T^{i}_{comm1}\left(N_{elem}\right)$ – Time spend by *i*-th super-task on the communication during the execution of the algorithhm estimating the computational cost

$T^{i}_{comp2}\left(N^{init}_{elem}, N^{ref}\right)$ – Time spend by *i*-th super-task on the computation in the parallel algorithm of the separation of all graphs representing a single initial mesh element

$T^{i}_{comm3}\left(N_{elem}\right)$ – Time spend by *i*-th super-task on the communication in the parallel algorithm of the mesh repartitioning

$T^{i}_{comp3}\left(N_{elem}\right)$ – Time spend by *i*-th super-task on the computation in the mesh repartitioning parallel algorithm

$T^{i}_{comm4}\left(N^{int}_{elem}\right)$ – Time spend by *i*-th super-task on the communication in the parallel algorithm of exchange of the ghost elements

$T^{i}_{comp4}\left(N^{int}_{elem}\right)$ – Time spend by *i*-th super-task on the computations in the parallel algorithm of exchange of the ghost elements

$T^{i}_{comm5}\left(N^{int}_{elem}\right)$ – Time spend by *i*-th super-task on the communication in the parallel algorithm of *h* refinements

$T^{i}_{comp5}\left(N_{elem}\right)$ – Time spend by *i*-th super-task on the computation in the parallel algorithm of *h* refinements

$T^{i}_{comm6}\left(N^{int}_{elem}\right)$ – Time spend by *i*-th super-task on the communication in the parallel algorithm of *p* refinements

$T^{i}_{comp6}\left(N_{elem}\right)$ – Time spend by *i*-th super-task on the computation in the parallel algorithm of *p* refinements

N_{elem} – Number of active elements located on initial mesh elements of a super-task

N^{init}_{elem} – Number of initial mesh elements of a super-task

N^{ref} – Depth of the refinement tree

N^{int}_{elem} – Number of active elements of a super task located on the initial mesh elements adjacent to the interface

$N_{elem}^{int,\,init}$	–	Number of initial mesh elements adjacent to the interface
P	–	Number of processors
Efficiency	–	Efficiency of parallel algorithm
Speedup	–	Speedup of parallel algorithm
T_1	–	Execution time for a single processor
T_P	–	Execution time for P processors
p	–	Uniform order of approximation
nrdof	–	Total number of degrees of freedom for a single element
interior nrdof	–	Number of degrees of freedom related to element interior node
interface nrdof	–	Number of degrees of freedom related to element edges
edge nrdof	–	Number of degrees of freedom on a single element edge
$T_{\text{comp solver}}\left(p, N_{elem}^{total}\right)$	–	Time spend on the computation in the sequential solver algorithm
$T_{\text{comp solver}}^{\max}\left(n, N_{elem}^{total}, P\right)$	–	Maximum for all processors P of the time spend on the computation in the parallel solver algorithm
$T_{\text{comm solver}}^{\max}\left(N_{elem}^{total}, p\right)$	–	Maximum for all processors P time spend on the communication in the parallel solver algorithm
t_{comp}	–	Execution time of a single instruction
t_{comm}	–	Time of transfer of a single double precision value
$T_{\text{comp solver}}^{i}$	–	Time spend by i-th processor on the computations in the parallel solver algorithm
$T_{\text{comm time}}^{i}$	–	Time spend by i-th processor on the communication in the parallel solver algorithm
$T_{\text{idle solver}}^{i}$	–	Time when the i-th processor is idle during the execution of the parallel solver algorithm
$T_{\text{comp re-solver l}}^{1}\left(N_{elem}^{total}, p\right)$	–	Time spend on the computation in the sequential solver algorithm with reutilization of partial LU factorizations
$T_{\text{re-solver r}}^{1}\left(N_{elem}^{total}, p, r\right)$	–	Total computational time of the sequential solver algorithm with reutilization of partial LU factorizations with r refined leafs (resulting from $r/4$ singularities)
$T_{\text{re-solver r}}^{P}\left(N_{elem}^{total}, P, p\right)$	–	Total execution time of the parallel solver algorithm with reutilization of LU factorizations after r elements have been refined
N_{elem}^{total}	–	Number of finite elements
computational cost(iel)	–	Estimated number of operations executed by the integration and elimination algorithms for a single initial mesh element, broken into multiple son elements (if necessary)
N	–	Size of matrix (number of degrees of freedom)

M	–	Size of the Schur complement matrix (number of degrees of freedom belonging to the Schur complement)
$K_{ij}^{(k)}$	–	Components of an anisotropic heat transfer tensor over material (k)
$f^{(k)}$	–	Heat source over material (k)
$g^{(k)}$	–	Scalar field from the Cauchy boundary condition for a material (k)
$\beta^{(k)}$	–	Constant from the Cauchy boundary condition for a material (k)
R	–	Resistance
Q	–	Generated heat
K	–	Thermal conductivity
H	–	Boundary convection
u_{env}	–	Ambient (environmental) temperature
σ_{ij}	–	Stress tensor
ε_{ij}	–	Green tensor
δ_{ij}	–	Kronecker delta
μ , λ	–	Lame coefficients
E	–	Young modulus (in chapter 3)
ν	–	Poisson ratio
α	–	Thermal expansion coefficient
T_0	–	Reference temperature
ΔT	–	Increase in the temperature
V	–	Volume
ΔV	–	Increase in the volume
E_{ijkl}	–	Tensor of elasticities
σ	–	Conductivity of the media
$div(J)$	–	Divergence of the prescribed impressed current
Jac	–	Jacobian matrix of the change of coordinates from quasi-cylindrical non-orthogonal system of coordinates to the Cartesian system of coordinates
(x_1, x_2, x_3)	–	Cartesian system of coordinates
$(\zeta_1, \zeta_2, \zeta_3)$	–	Quasi-cylindrical non-orthogonal system of coordinates
u_l	–	l^{th} component of the Fourier series expansion of the unknown scalar field
σ_m	–	m^{th} component of the Fourier series expansion of the conductivity scalar field
$div(J_l)$	–	l^{th} component of the Fourier series expansion of the divergence of the prescribed impressed current
F_l	–	l^{th} Fourier modal coefficient

$\hat{\sigma}$	–	Conductivity after change of variables into the Cartesian system of coordinates
$div\left(\hat{J}\right)$	–	Divergence of the prescribed impressed current after change of variables into the Cartesian system of coordinates
j	–	Imaginary unit
(ρ,φ,z)	–	Cylindrical system of coordinates
Ω_A, Ω_B, Ω_C, Ω_D, Ω_E	–	Parts of 3D domain Ω with different conductivities
$\hat{K}$	–	Either 1D or 2D or 3D reference finite element
$\hat{a}_i$	–	Node of reference finite element
$X\left(\hat{K}\right)$	–	Space of reference element shape functions
$\hat{\chi}_j$	–	1D reference element shape function
$P^p\left(\hat{K}\right)$	–	Polynomials of order p over $\hat{K}=[0,1]$
Π_p	–	Projection-based interpolant operator
$V^*\left(\hat{K}\right)$	–	Space of degrees of freedom (dual space to a functional space $V\left(\hat{K}\right)$)
$\{\psi_i\}_{i=1}^{p+1}$	–	Basis of $V^*\left(\hat{K}\right)$
K	–	Either 1D or 2D or 3D arbitrary finite element
a_i	–	Nodes of an arbitrary finite element
$X(K)$	–	Space of an arbitrary element shape functions
χ_j	–	Arbitrary element shape function
x_K	–	Map from the reference $\hat{K}$ to an arbitrary element K
$\hat{\phi}_i$, i=1,...,4	–	2D reference element vertex shape functions
$\hat{\phi}_{i,j}$ i=5,...,8, j=1,...,$p_{\text{i-4}}$	–	2D reference element edge shape functions
$\hat{\phi}_{9,i,j}$ i=1,...,p_{h}, j=1,...,p_{v}	–	2D reference element interior shape functions
T_{hp}	–	Set of *hp* finite elements defining the coarse mesh
$T_{h/2,p+1}$	–	Set of *hp* finite elements defining the fine mesh
ϕ_k	–	Local shape function
$i(k,K)$	–	Global number of degrees of freedom related to local shape function k from element K

For my wife Anna and daughter Joanna

Acknowledgements

First of all, I would like to express my gratitude to my wife and my daughter for all the support they gave me during these years, for their patience and understanding.

I wish to thank Prof. Leszek Demkowicz who was the first one to introduce me into the *hp* adaptive world and to Prof. Robert Schaefer for guiding me through the model of concurrent adaptive computations, for his help and support.

I am grateful to Prof. Carlos Torres-Verdin for supporting my research, to Prof. Jacek Kitowski for his creative ideas and to the entire *hp* team, including David Pardo, Jason Kurtz, Waldemar Rachowicz and Adam Zdunek.

I appreciate the support I received from the Foundation for Polish Science and from the Polish Ministry of Scientific Research and Information Technology.

1. Preface

1.1. Main thesis and the structure of the book

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

> "It is possible to develop a formal description of a wide class of adaptive mesh-based algorithms, which will allow us to examine their concurrency and to design parallel versions of the algorithms, as well as to develop their efficient parallel implementation."

The primary objective of this work is to develop a graph grammar-based formal description of the adaptive mesh-based algorithms. The graph grammar-based model will be developed for the self-adaptive *hp* Finite Element Method (*hp*-FEM), which is considered to be one of the most complicated adaptive mesh-based algorithms. Many other mesh-based adaptive algorithms are embedded into the self-adaptive *hp*-FEM, thus the developed formalism can be also successfully applied to other algorithms of this class.

The graph grammar model allows us to examine the concurrency of the self-adaptive *hp*-FEM algorithms. It is done by introducing the concepts of atomic tasks, tasks and super-tasks, and by analysing their interdependence. The atomic tasks correspond to the graph grammar productions, representing basic undividable parts of the algorithms. The analysis of the interdependence between the atomic tasks results in the control diagrams, setting the partial order of execution of the atomic tasks. The level of atomic tasks models the concurrency only for the shared memory architectures. This is because we assume that all atomic tasks work on a shared copy of the CP-graph representation of the computational mesh.

The shared memory architectures are not the most popular ones, thus the analysis is continued for the level of tasks, intended for the distributed memory architecture. A task corresponds to groups of atomic tasks working on the level of initial mesh element. The communication channels between tasks are defined and the parallel algorithms are redesigned for the level of tasks. The tasks represent the grain for the domain decomposition and load balancing algorithms. Finally, we introduce the level of super-tasks. A super-task corresponds to group of tasks resulting from the execution of the load balancing algorithm. The parallel algorithms are redefined again on the level of super-tasks.

The particular cognitive value of the formalism consists in a possibility of examining the algorithm's concurrency. Thus, the formal description enables the design of parallel versions of the algorithms as well as their efficient parallel implementation.

The adaptive algorithms considered in this work are based on the two-grid paradigm, with the coarse meshes and their corresponding fine meshes. The algorithms generate a sequence of coarse meshes to deliver convergence of the numerical error of a considered engineering problem, with respect to the mesh size. The algorithms make use of the corresponding fine mesh to estimate the quality of the coarse mesh in order to select the optimal refinements to be executed on the coarse mesh. The two-grid paradigm adaptive algorithms presented in this work are applied to the class of elliptic and Maxwell Partial Differential Equations (PDE) boundary-value problems. However, the applicability of the two-grid paradigm adaptive algorithms is not limited only to the PDE engineering problems. Any problem that requires a construction of the finite dimensional approximation space based on the polynomials of different orders span on finite element mesh can be included into the considered class of adaptive mesh-based algorithms.

The creation of an efficient parallel version of the algorithm is critical for most of the applications, since the computational cost related to the adaptive computations is high.

The practical result of this work is the parallel adaptive software system that implements the self-adaptive *hp*-FEM algorithm. Several challenging engineering problems require high accuracy of the numerical solution, which can be achieved only with huge computational grids when using other numerical methods. In other words, other numerical methods require enormous computational time and resources (a number of processors and a huge amount of computer memory). The parallel self-adaptive *hp*-FEM algorithm enables an accurate solution to these challenging problems in a relatively short computational time, using commonly available parallel clusters. Some examples of these challenging engineering problems include material science, geo-science and remote sensing, nano-lithography (micro-cheap production process) and wave propagation problems. Some of these problems, e.g. the problem of the 3D DC/AC borehole resistivity measurement simulation in borehole environment with steel casing, up to now has not been solved at all, because of the required accuracy of the order of 10^{-7} (seven significant digits after the dot) impossible to obtain by means of other numerical methods. All these problems can be finally solved by the software system described in this dissertation, thanks to the exponential convergence of the self-adaptive *hp*-FEM algorithm.

The structure of the presentation is as follows:

The presentation starts in Section 1.2 with the introduction of some basic definitions of the Finite Element Method. We use the example of the L-shape domain problem, a model engineering problem that requires the adaptive computations in order to obtain an accurate numerical solution. After defining the FEM terminology, the current state of art in the field of modeling sequential and parallel self-adaptive *hp*-FEM algorithms is presented in Section 1.3. The final part of this chapter includes a list of open problems related to this field, as well as an explanation of how this work will deal with most of those open problems.

In Chapter 2, the graph grammar model of the self-adaptive *hp*-FEM algorithm is introduced. In Section 2.1, the graph grammar model is employed to introduce all the details of the self-adaptive algorithm, such as the generation of initial mesh, the solution of numerical problem, the selection of optimal mesh refinements, and the execution of *h* and *p*

refinements. Furthermore, the CP-graphs (Composite Programmable graphs) are used to model the computational mesh, and the graph transformations (graph grammar productions) are used to express all parts of the self-adaptive *hp*-FEM algorithm. The graph grammar productions are in consequence understood as the atomic tasks, executed on the graph representation of the mesh and stored in the shared memory. That allows us to analyse the concurrency of the self-adaptive *hp*-FEM algorithm, by exploiting and exhibiting the partial causality order (Lamport relationships) among atomic tasks. Moreover, the production control diagrams, defined for each phase of the computations, act as randomized dynamic models of concurrent system similar to the Petri Net.

Section 2.2 is dedicated to the partition of the graph representation of the mesh into several sub-graphs related to the initial mesh elements. The atomic tasks introduced in Section 2.1 are then agglomerated into tasks. It is assumed that each task has its own local memory, where it stores its sub-graph representing a single initial mesh element. The self-adaptive *hp*-FEM algorithm is redefined on the level of tasks by updating the control diagrams executed now by particular tasks. A control diagram at this point defines the atomic tasks that can be executed by a task on its local sub-graph, with some minimum additional inter-task communication added.

The tasks defined in Section 2.2 constitute the grain for the load balancing and mesh repartitioning algorithms. The Section 2.3 contains the run-time management algorithms responsible for load balancing, mesh partitioning and mapping tasks into particular architectures of parallel machines. The input for the algorithms is the estimation of a computational load for each task, whereas the output is a new repartition of tasks. The repartition is performed at the end of each iteration of the self-adaptive *hp*-FEM algorithm. In Section 2.4, the tasks are again agglomerated into super-tasks. It is assumed that each task has its own local memory with a sub-graph representing a single sub-domain, with several initial mesh elements. The super-tasks are created by agglomerating several tasks resulting from the load balancing and mesh repartitioning algorithms. The self-adaptive *hp*-FEM algorithm is again redefined on the level of super-tasks. It is done by means of defining several control diagrams, which will assign the order of tasks executed by each super-task, and will indicate the required inter-super-task communication. The algorithm expressed on the level of super-tasks is suitable for the distributed or hybrid memory architectures.

Section 2.5 provides an analysis of the computational and communication complexities. The analysis is performed on the level of super-tasks. It concerns all parts of the self-adaptive *hp*-FEM algorithms. The complexities are first estimated for the mesh partitioning algorithm, including the exchange of ghost elements, and then for the mesh adaptation algorithm, including the *h* and *p* adaptations. The complexities are also estimated for the parallel recursive solver algorithm. Finally, we focus on the estimation of the computational and communication complexities for the algorithm of the solver with reutilization of partial LU factorizations.

In Section 2.6 a sequence of numerical experiments is carried out. The numerical experiments concern the parallel self-adaptive *hp*-FEM algorithm implemented for two-dimensional rectangular meshes and for three-dimensional quadrilateral meshes. The first sequence of experiments refers to the scalability of the parallel algorithm implemented for two-dimensional meshes, with the grain defined on the level of initial mesh elements. The second sequence of experiments refers to the parallel algorithm

implemented for three-dimensional meshes, again with the grain defined on the level of initial mesh elements. In the last part of this section we describe some numerical experiments performed to test the scalability of the proposed parallel direct solvers.

Chapter 3, entitled "Applications", describes numerous both academic and challenging engineering problems and provides solutions to them using the implemented parallel self-adaptive algorithms.

The presentation is concluded in Chapter 4. The conclusions are followed by Appendices, which include a precise definition of *hp* finite element, the mathematical details of the self-adaptive *hp*-FEM algorithm, the definition of the CP-graph grammar, as well as some implementation details.

1.2. Introduction

1.2.1. Introduction to the Finite Element Method

This section presents the necessary background of the Finite Element Method (FEM). The L-shape domain model problem is employed to illustrate the basic ideas and definitions related to the FEM. The FEM was first defined by Zienkiewicz 1967. Let us focus on the L-shape domain model problem (Babuška, Guo, 1986a, Babuška, Guo, 1986b). The problem consists in solving the Laplace equation

$$\Delta u = 0 \text{ in } \Omega \tag{1.1}$$

for the L-shape domain Ω, presented in Figure 1.1. The solution $u : R^2 \supset \Omega \ni u \to R$ may be interpreted as a temperature distribution inside the L-shape domain.

The zero Dirichlet boundary condition

$$u = 0 \text{ on } \Gamma_D \tag{1.2}$$

is assumed on the internal part of the boundary Γ_D. The Neumann boundary condition

$$\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N \tag{1.3}$$

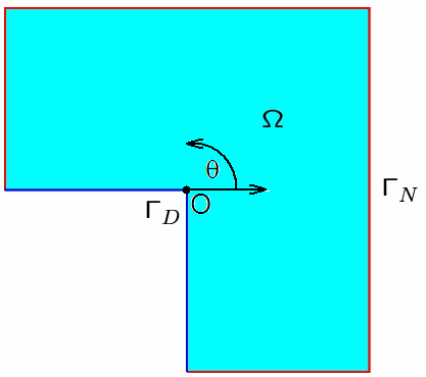


Fig.1.1. L-shape domain

is assumed on the external part of the boundary Γ_N. The temperature gradient in the direction normal to the boundary is defined in the radial system of coordinates with the origin point O presented in Figure 1.1

$$g(r,\theta)=r^{\frac{2}{3}}\sin^{\frac{2}{3}}\left(\theta+\frac{\Pi}{2}\right) \tag{1.4}$$

The so-called *strong form* of the partial differential equation (PDE) (1.1-1.3) is transformed into *weak (variational) form* of PDE

$$b(u,v)=l(v) \quad \forall v\in V \tag{1.5}$$

$$b(u,v)=\int_{\Omega}\nabla u\nabla v\,dx \tag{1.6}$$

$$l(v)=\int_{\Gamma_N} g\,v\,dS \tag{1.7}$$

by considering $L^2(\Omega)$ scalar products of (1.1) with test functions from functional space V

$$V=\left\{v\in L^2(\Omega): \int_{\Omega}\|v\|^2+\|\nabla v\|^2\,dx<\infty : \mathrm{tr}(v)=0 \text{ on } \Gamma_D\right\} \tag{1.8}$$

and integrating by parts as well as including boundary condition (1.3).

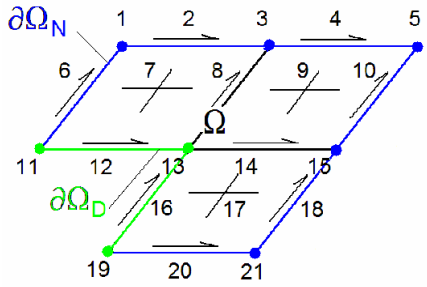


Fig.1.2. Partition of the L-shape domain into 3 finite elements

The *Finite Element Method* consists in constructing a subspace $V_{hp}\subset V$ with finite dimensional basis $\left\{e^{i}_{hp}\right\}_{i=1,\ldots,N_{hp}}$. The subspace V_{hp} is constructed by partitioning the domain Ω into finite number of elements and defining basis functions at finite element vertices, over edges and interiors.

The exemplary partition of the L-shape domain into 3 rectangular finite elements is presented in Figure 1.2. The elements of the basis e^{i}_{hp} are called *shape functions*. We define the first order shape functions (pyramids) at the element vertices, the second order shape-functions (edge-bubbles) over the element edges, and the second order shape functions over the elements interiors. That is presented in Figure 1.3.

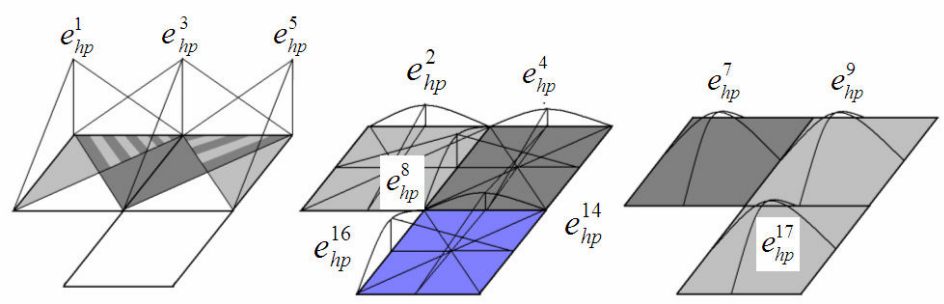


Fig.1.3. Examples of vertex, edge and interior shape-functions

The exact solution u of the weak formulation (1.5-1.8) is approximated in the subspace V_{hp} as a linear combination of the shape functions

$$u \approx u_{hp} = \sum_{i=1}^{N_{hp}} u^{i}_{hp} e^{i}_{hp} \tag{1.9}$$

where $N_{hp} = 21$ in this example. The coefficients u^{i}_{hp} are called *degrees of freedom*. The linear combination u_{hp} is called the *approximate solution*. The degrees of freedom are acquired by solving the following system of equations (*FEM discretization of the weak for of PDE*)

$$\sum_{i=1}^{N_{hp}} u^{i}_{hp} b\left(e^{i}_{hp}, e^{j}_{hp}\right) = l\left(e^{j}_{hp}\right) \quad j = 1,..., N_{hp} \tag{1.10}$$

$$b\left(e^{i}_{hp}, e^{i}_{hp}\right) = \int_{\Omega} \nabla e^{i}_{hp} \nabla e^{j}_{hp} \, dx \tag{1.11}$$

$$l\left(e^{j}_{hp}\right) = \int_{\Gamma_N} g e^{j}_{hp} \, dS \tag{1.12}$$

This system of equations results from the substitution of (1.9) for (1.5-1.7) and from the application of $\left\{e^{j}_{hp}\right\}_{j=1,...,21}$ test functions.

1.2.2. Introduction to the mesh adaptation techniques

The purpose of this section is to present different mesh adaptation techniques. In order to do it we apply the L-shape domain model problem presented in Section 1.2.1. The aim of the mesh adaptation methods is to increase the accuracy of the approximate FE solution u_{hp} by increasing the number of the shape functions N_{hp} in (1.9). Figure 1.4 illustrates the following mesh adaptation techniques:

a) *uniform h adaptation*: all finite elements are uniformly broken into smaller elements,
b) *uniform p adaptation*: the polynomial order of approximation is increased uniformly on the entire mesh, e.g. by adding bubble shape functions of the higher orders over element edges and interiors,

c) *non-uniform h adaptation*: some finite elements are broken into smaller elements, only in those parts of the mesh which have a high numerical error,
d) *non-uniform hp adaptation*: some finite elements are broken into smaller elements, and the polynomial orders of approximation are increased only in those parts of the mesh which have a high numerical error.

For non-uniform *h* or *hp* adaptation it is necessary to localize finite elements with high numerical error. In case of the non-uniform *hp* adaptation it is also necessary to select the optimal refinements for such finite elements. The non-uniform *h* or *hp* adaptation can be executed in the following ways:

a) the selection of the finite elements to be refined and a type of refinement depends on the user,
b) the selection of the finite elements to be refined and a type of refinement depends on an algorithm which is based on the knowledge of the structure of the solution,
c) the selection of the finite elements to be refined and a type of refinement depends on the self-adaptive algorithm which is designed without any particular knowledge of the structure of the solution, and works in a fully automatic mode, without any user's interaction.

The a) and b) algorithm are called the *non-automatic adaptation* while the c) algorithm is called the *automatic adaptation*. In particular, the non-uniform *hp* adaptation with the automatic adaptation is called the *self-adaptive hp Finite Element Method* (self-adaptive *hp*-FEM).

The comparison of the uniform *h*, uniform *p*, non-uniform *h* and non-uniform *hp* automatic adaptation algorithms (Demkowicz 2006) is presented in Figure 1.5. The horizontal axis denotes the number of degrees of freedom on particular meshes from the sequence of meshes generated by the considered mesh adaptation algorithm. The vertical axis denotes the relative error for particular meshes from the generated sequence. The relative error has been computed in the following way: for a given *hp* mesh, the approximate solution u_{hp} is obtained by solving the equations (1.10-1.12) (the FEM disretization of the weak form of PDE). The *hp* mesh is then *globally hp refined*, which means that each finite element is broken into four new finite elements and the polynomial order of approximation is uniformly raised by one on all finite elements. The new mesh, which is obtained by executing the global *hp* refinement, becomes the reference mesh employed to estimate the relative error for the original mesh. The new approximate solution $u_{h/2,p+1}$ is a result of solving the equations (1.10-1.12) on the new reference mesh.

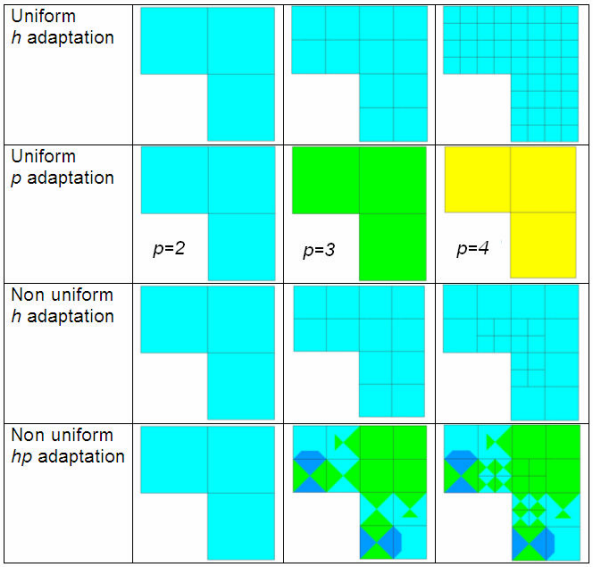


Fig.1.4. First three meshes from a sequence of meshes generated by uniform *h* adaptation, uniform *p* adaptation, non-uniform *h* adaptation and non-uniform *hp* adaptation. Different colours denote different polynomial orders of approximation on edges and interiors

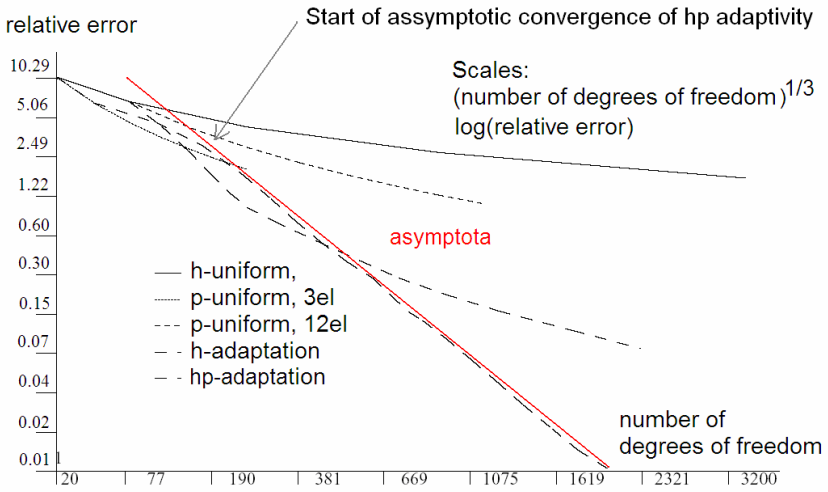


Fig.1.5. Comparison of different mesh adaptation strategies obtained from numerical experiments. Convergence history for a sequence of meshes generated by uniform *h* adaptation, uniform *p* adaptation, non-uniform *h* adaptation and non-uniform *hp* adaptation. The horizontal axis denotes number of degrees of freedom. The cube root square is used on the horizontal axis. The vertical axis denotes the relative error. Notice that the horizontal axis starts from 0.01 relative error (not from zero). The logarithmic scale is utilized on the vertical axis.

The relative error is computed as a difference between the original and the reference mesh solutions

$$\text{error}_{\text{rel}} = \left\| u_{hp} - u_{h/2,p+1} \right\|_{1,\Omega} \tag{1.13}$$

The difference is computed in the $H^1(\Omega)$ Sobolev space norm

$$\|v\|_{1,\Omega} = \int_{\Omega} \left(|\nabla v|^2 + |v|^2 dx \right)^{1/2} \tag{1.14}$$

The comparison of different mesh adaptation strategies for the L-shape domain problem discussed in this section is presented in Figure 1.5. The figure presents the convergence history for a sequence of meshes generated by uniform *h* adaptation, uniform *p* adaptation, non-uniform *h* adaptation and non-uniform *hp* adaptation. In Figure 1.5 the vertical axis uses the logarithmic scale of the relative error while the cube root of number of degrees of freedom is applied on the horizontal axis. The conclusion which we may draw from this comparison is that the uniform *h* adaptation is the most expensive strategy. It is followed by the uniform *p* adaptation strategies. The first considered strategy has been executed on the initial L-shape mesh with 3 finite elements whereas the second considered strategy has been executed on the initial L-shape mesh with 12 finite elements. The uniform *p* adaptation convergence curves are limited by the maximum polynomial order of approximation p=9.

Two fastest mesh adaptation strategies are the non-uniform *h* adaptation and the non-uniform *hp* adaptation. The non-uniform *h* adaptation executes the *h* refinements in the direction of the central singularity. The non-uniform *hp* adaptation not only executes the *h* refinements, but also modifies the polynomial order of approximations. The maximum accuracy achieved by the non-uniform *h* adaptation is 0,07 % of the relative error (1.13), for 1700 degrees of freedom. The same accuracy for the non-uniform *hp* adaptation is achieved on a computationally less expensive mesh with 900 degrees of freedom. The *hp* adaptation is less expensive, since the computational cost of this solution is of the order of the cube of the number of degrees of freedom.

The non-uniform *hp* adaptation converges asymptotically, after some short pre-assymptotic range, where the non-uniform *h* adaptation is faster. This is illustrated in Figure 1.5 by the red sloped asymptote. The non-uniform *hp* adaptivity provides the fastest possible exponential convergence of the relative error $\text{error}_{\text{rel}}$, with respect to the number of degrees of freedom nrdof,

$$\log(\text{error}_{\text{rel}}) = C\,\text{nrdof}^{1/3} + \log(\text{error}_0) \tag{1.15}$$

$$\text{error}_{\text{rel}} = \text{error}_0 \exp\left(C\,\text{nrdof}^{1/3} \right) \tag{1.16}$$

where error_0 stands for the point of the cross-section of the asymptote with the vertical axis, and C stands for the slope of the curve. This exponential convergence of the non-uniform *hp* adaptivity expressed by the formulae (1.15-1.16) has been experimentally confirmed. It has been predicted in several theoretical works (Babuška, Guo, 1986a, Babuška, Guo, 1986b, Schwab 1998).

Note that all the adaptation techniques, except the non-uniform *hp* adaptation, have lost the exponential convergence. It implies that the high accuracy provided by the non-

uniform *hp* adaptation can be acquired by other mesh adaptation techniques only if they reach a huge number of degrees of freedom.

The numerous examples confirm also the exponential convergence of the non-uniform *hp* adaptation algorithm. In one of them, presented in Chapter 3.1.2, the convergence rate of the non-uniform *hp* adaptation is compared with the convergence of the non-uniform *h* adaptation. The required high 1% relative error accuracy of the solution is achieved by the non-uniform *hp* adaptation for 2392 degrees of freedom. On the other hand, in this example, such a high accuracy solution requires several millions of degrees of freedom or non-uniform *h* adaptation, so it can be achieved only by using the non-uniform *hp* adaptation algorithm.

1.3. State of art of adaptive mesh-based algorithms

This chapter provides a short presentation on the current state of art in the development of sequential and parallel self-adaptive *hp*-FEM algorithms.

1.3.1. Sequential algorithms for the self-adaptive *hp*-FEM

In the non-uniform *hp* adaptation, the decisions about which finite elements should be refined and about a type of refinement must be made by the self-adaptive *hp*-FEM algorithm. The sequential version of the self-adaptive *hp*-FEM algorithm has been designed and implemented by the group of Leszek Demkowicz (Demkowicz, Rachowicz, Devloo 2002, Rachowicz, Pardo, Demkowicz 2006, Demkowicz 2006, Demkowicz, Kurtz, Pardo, Paszyński, Rachowicz, Zdunek 2007). The serial version of the self-adaptive *hp*-FEM algorithm has been created for both two and three dimensions. The self-adaptive *hp*-FEM can be summarized in the following way. Let us focus on the two-dimensional case. The algorithm is based on the two-grid paradigm: a coarse mesh and a fine mesh obtained from the coarse mesh by executing the global *hp* refinement. This means that each finite element is broken into four new son elements and the polynomial order of approximation is uniformly raised by one for each finite element. The approximate solution u_{hp}, called the *coarse grid solution* is obtained by solving the equations (1.10-1.12) on the coarse mesh, and the *fine mesh solution* $u_{h/2,p+1}$ is obtained by solving the equations (1.10-1.12) on the fine mesh. The fine mesh solution $u_{h/2,p+1}$ is later used to mark the optimal refinements for the coarse grid in order to produce the next optimal grid.

The fine grid solution $u_{h/2,p+1}$ is projected onto a nested sequence of meshes that is locally embedded into the fine grid by using the projection-based interpolation technique (Demkowicz 2004). For each coarse grid element the sequence is dynamically constructed by testing possible types of local refinements. The refinement which provides the maximum error decrease rate

$$\text{rate}(w) = \frac{\left\| u_{h/2,p+1} - u_{h,p} \right\|_{1,K} - \left\| u_{h/2,p+1} - w \right\|_{1,K}}{\Delta \text{nrdof}} \tag{1.17}$$

is selected for each considered coarse mesh element. In this formula w denotes the projection of the fine mesh solution onto the considered coarse mesh refinement, where the projection is computed in H^1 norm on the considered coarse mesh element K. The Δnrdof denotes an increase in the number of degrees of freedom or the coarse mesh element, resulting from the execution of the considered mesh refinement strategy. This procedure is illustrated in Figure 1.6.

The selected refinements are executed on the coarse mesh elements and a resulting new mesh becomes the coarse mesh for the next iteration. The iterations are executed until the relative error estimation for the entire mesh is larger than the required accuracy of the solution.

For the mathematical details of the sequential algorithm of the self-adaptive *hp*-FEM, see Appendix B.

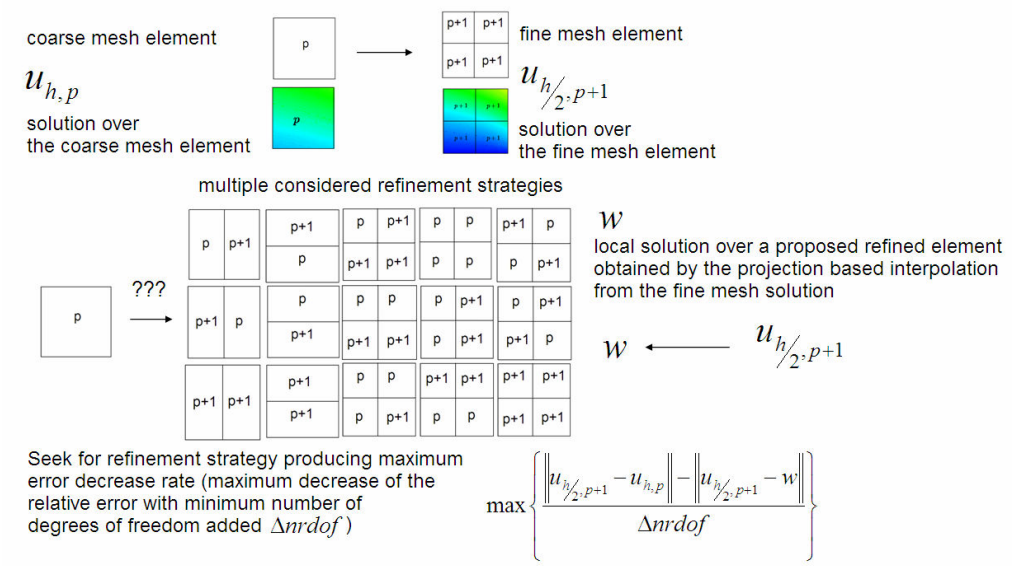


Fig.1.6. Example of different refinements for a single finite element from the coarse mesh

1.3.2. Parallel algorithms for non-uniform adaptive FEM

This section is a summary of the state of art on the non-uniform adaptive FEM algorithms. Most adaptive FEM algorithms are based on the domain decomposition paradigm where the computational mesh is partitioned into sub-domains. Thus, the presentation focuses on the following five layers:

1) *load balancing and mesh partitioning algorithms*: finding quasi-optimal partition of the computational domain into sub-domains,
2) *mesh transformation algorithms*: dealing with several mesh regularity rules on the computational mesh partitioned into sub-domains,
3) *optimal refinement algorithms*: selecting optimal refinements on the mesh,

4) *communication strategies*: dealing with an exchange of data between sub-domains resulting from the applied load balancing and mesh partitioning algorithms,
5) *implementation layer*: focusing on the implementation details of an algorithm.

1.3.2.1. Non-uniform non-automatic *h* refinements

Among major undertakings to develop the algorithms supporting *h* refinements for the computational mesh distributed into sub-domains, one has to list in the first place the SIERRA Environment (Edwards 1997, Edwards 2002, Edwards, Stewart, Zepper 2002, Stewart, Edwards 2002). This section describes the algorithms in terms of the above mentioned layers.

1.3.2.1.1 Load balancing and mesh partitioning algorithms

In case of the *h* adaptive algorithms the structure of the mesh changes dynamically during the mesh refinements. The mesh must be frequently redistributed to maintain a uniform load balancing. For the *h* adaptive algorithms the computational load related to each finite element is the same, since the polynomial order of approximation is uniform for the entire mesh. Thus, the algorithms from the SIERRA environment define the computational load equal to 1 for each finite element. The load balancing is achieved by interfacing with algorithms defined in the ZOLTAN library. The mesh can be partitioned on the level of initial mesh elements or on the level of active elements. The active elements are represented as leaves of the tree of nodes growing from initial mesh elements. Thus, the decomposition on the level of active mesh elements requires cutting the trees into pieces, which is technically complicated. However, the algorithms introduced by Edwards partition the mesh on the level of initial mesh elements in order to simplify the communication algorithms (if the mesh is partitioned on the level of active elements, an additional expensive communication will be required). The experiments performed by Edwards prove that the partition on the level of initial mesh elements is sufficient for most considered engineering applications.

1.3.2.1.2 Mesh transformation algorithms

The *h* adaptive parallel algorithms have to deal with the enforcement of the 1-irregularity rule. According to *the 1-irregularity rule* a finite element can be broken only once without breaking the adjacent large elements, which is illustrated in Figure 1.7. The rule prevents unbroken element edges from being adjacent to more than two finite elements on one side. 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 over the larger element. This is illustrated in Figure 1.8. To avoid a technical nightmare with constrained approximation over multiple constrained edges, the 1-irregularity rule is commonly used in the *h* adaptive algorithms. To maintain the 1-irregularity rule, several additional refinements on large adjacent elements may be required. When the computational mesh is distributed into multiple sub-domains, the parallel algorithms supporting *h* refinements require communication between neighboring sub-domains. The request for additional *h* refinement must be sent to the adjacent large finite element located on neighboring sub-domains.

The mesh transformation algorithms from the SIERRA provide a hierarchical *h*-refinement of meshes. The 1-irregularity rule is enforced for the mesh. That may cause some elements which have not been selected for refinement to be refined anyway. This enforcement may also require a communication step in order to force the elements from an adjacent processor to be refined, too. This is done by applying the communication specification notation, defined in Section 1.3.2.1.4.

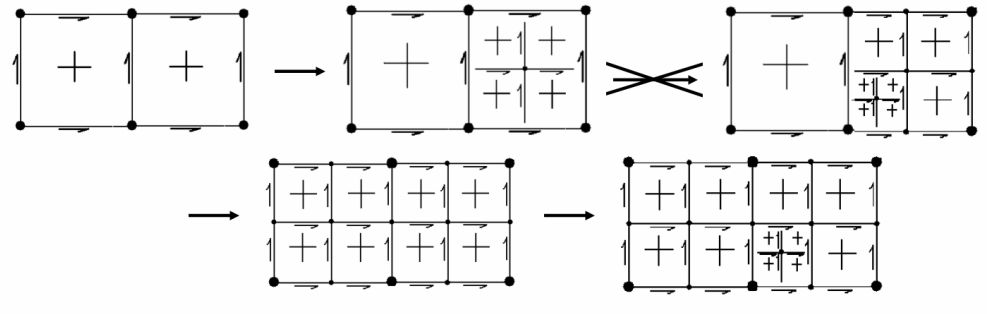


Fig.1.7. The 1-irregularity rule enforces breaking a large neighbor element before breaking a small element on the other side of edge

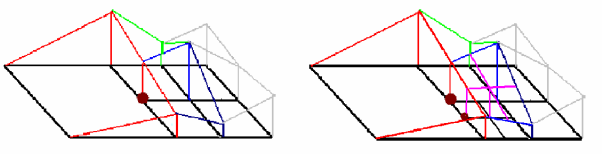


Fig.1.8. First order constrained approximation on the 1-irregular and 2-irregular meshes

1.3.2.1.3 Optimal refinement algorithms

The algorithms choose the elements for a refinement on the basis of some problem specific error estimators. The algorithms defined by SIERRA do not take into consideration the two mesh paradigm.

1.3.2.1.4 Communication strategies

The communication algorithms exchange data between pairs of specific mesh objects, such as node, edge, face or element. These communication operations are usually sparse, thus a given object only needs to communicate with a fraction of the total number of processors used by an application.

The communication algorithms from the SIERRA framework use *the communication specification*. It is defined by indicating pairs of communicating objects and the communication operations (e.g. sum, copy, construct). The communication specification is a topological relation between two potentially distributed mesh objects, and is defined by using the notations proposed by Edwards, 2002.

In general, a communication specification is non-symmetrical and non-local. Several communication specifications are further combined to form a multiple-communication specification by means of the algebraic operators.

1.3.2.1.5 Implementation layer

The algorithms implemented in the framework have been used to parallelize several finite element applications developed at Sandia National Laboratories (USA) (Edwards, Stewart 2001).

The object-oriented computational framework, called SIERRA, is designed to support the *h* adaptation on the unstructured meshes with multiple element types, such as hexahedra, tetrahedra etc. It provides an Application Programming Interface (API) for the definition of multi-physics models, dynamic load balancing, mesh transformation to support *h* adaptation and an interface to linear solvers. These are the following computational framework services:

a) mesh management,
b) degrees of freedom management,
c) material data management,
d) parallel communication operations,
e) infrastructure for h adaptivity, including the enforcement of the 1-irregularity rule,
f) dynamic load balancing.

All parallel communications in SIERRA are implemented using the MPI.

1.3.2.2. Non-uniform non-automatic *hp* refinements

The second group of the parallel algorithms is intended to support the non-uniform and non-automatic *hp* refinements. The only parallel *hp* adaptive algorithms implemented so far have been developed by Flaherty, Devine 1996, Remacle, Xiangrong, Shephard, Flaherty 2000 and Banaś 2003 in terms of Discontinuous Galerkin (DG) methods. The only parallel *hp* adaptive algorithms for Continuous Galerkin method that we are aware of, has been developed by Patra 1999, Laszlofy, Long, Patra, 2000, Bauer, Patra 2004. None of these algorithms supports the automatic *hp* adaptation.

1.3.2.2.1 Load balancing and mesh partitioning algorithms

The load balancing and mesh partitioning algorithms for the non-uniform *hp* refinements are much more complicated, since the computational complexity on a single finite element depends strongly on the polynomial orders of approximation of a finite element. The number of shape functions for an element with (p_h, p_v) orders of approximation in the horizontal and vertical directions is estimated as $(p_h+1)(p_v+1)$, thus the computational complexity of the integration and elimination of the degrees of freedom for this element is estimated as $O\left(p_h{}^3\, p_v{}^3\right)$. Note that for the elements with different polynomial orders of approximation, like those presented in Figure 1.9, this may lead to strong load imbalances. In this case, the mesh must be redistributed even after some local *p* refinements.

In the work by Remacle, Xiangrong, Shephard, Flaherty 2000 the load balancing is achieved by means of the tiling algorithm (Wheat, 1992). In this diffusive algorithm, the processors compare their estimated load with neighbors and exchange pieces of data with less loaded neighbors.

The load balancing problem can be also solved by the graph partitioning algorithm from the METIS library (Karypis, Kumar, 1999). The algorithm has been used by Banaś 2003 who interfaced the mesh partitioning algorithm with the METIS library.

Patra, 1999, Laszlofy, Long, Patra, 2003, and Bauer, Patra, 2004 have proposed an efficient load balancing and mesh partitioning algorithms based on the Hilbert Space Filling Curve (HSFC), presented in Figure 1.10.

The SFC maps the n-dimensional data in one-dimensional space, and fills the multidimensional space in the limit (Sagan 1984)

$$R^n \ni \mathbf{x} \to h_n(\mathbf{x}) \in R \tag{1.18}$$

The finite elements are identified by the coordinates of their central points, mapped into the SFC. The SFC provides a linear ordering for the n-dimensional finite elements. Laszlofy, Long, Patra, 2003 noticed that the SFC can be updated when the mesh is refined, which is illustrated in Figure 1.11.

The domain decomposition can be achieved by estimating the computational complexity for the finite elements along the SFC, and by partitioning the mesh on the basis of quasi-uniform partition of the SFC. The data structure necessary to store dynamically refined mesh consists in tree structures for storing nodal connectivities of the finite elements. The root of each tree denotes initial mesh element nodes, and the trees are expanded when the mesh is refined. These tree data-structure originated in the work of the group of Leszek Demkowicz and has been later developed in Demkowicz, 2006 and Demkowicz, Kurtz, Pardo, Paszyński, Rachowicz, Zdunek, 2007. The finite elements are localized by their interiors. To ensure a fast access to the finite elements and their nodes, the hash table is introduced. In this table the elements are identified by the keys defined on the basis of their locations on the SFC.

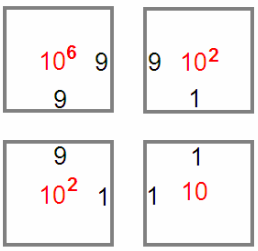


Fig.1.9. Four finite elements, the top left with both orders of approximation equal to 9, the bottom right with both orders of approximation equal to 1, and the two remaining elements matching orders of neighboring elements. This configuration results in a strong difference in computational complexity (five orders of magnitude).

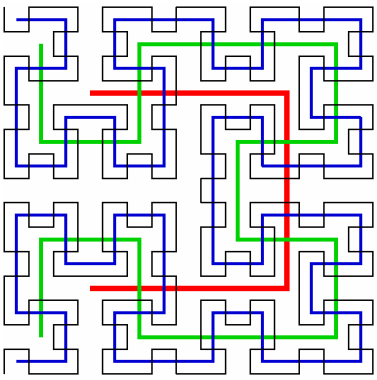


Fig.1.10. Four iterations of the Hilbert Space Filling Curve

This technique has the following advantages and disadvantages:

a) fast access to the elements, thanks to the hash tables with a key identifier based on the location on the SFC. This is possible only under the assumption that the mesh refinements are quite uniformly distributed on the entire mesh, which prevents multiple elements with the same hash table key,
b) easy load balancing, since the quasi-uniform load balancing can be acquired by cutting the SFC into equally loaded pieces,
c) easy mesh repartitioning after the mesh has been refined, since the data can be transferred through SFC, connecting all sub-domains,
d) the load balancing on the level of active finite elements is technically difficult to achieve. In order to avoid technical difficulties, the mesh can be distributed on the level of initial mesh elements, with a possible duplication of the entire initial mesh elements on sub-domains which share the newly refined elements,
e) the disadvantage of the proposed mesh partitioning technique is the fact that the SFC partition does not minimize the interface between the adjacent sub-domains, which may result in an expensive interface problem.

The order of elimination of the degrees of freedom for the direct solver can be obtained on the basis of the linear SFC, since the curve follows a local refinement pattern. However, the linear SFC ordering may be far from the optimal for a hierarchical multi-dimensional mesh structure. The quasi-optimal ordering can be achieved by means of a more sophisticated graph-based model of data structure.

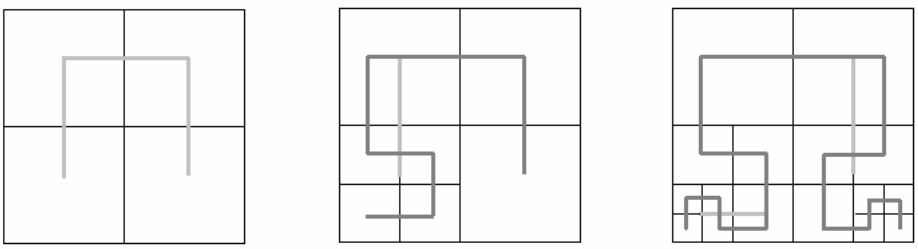


Fig.1.11. HSFC following the mesh refinements

1.3.2.2.2 Mesh transformation algorithms

The parallel *hp* adaptation algorithms have to overcome the same technical difficulties as the non-uniform *h* refinement algorithms, including the enforcement of the 1-irregularity rule. Moreover, we have to introduce an additional mesh regularity rule, *the minimum rule*.

The minimum rule implies that the polynomial order of approximation on an element edge must be equal to the minimum of corresponding orders of approximation from the element interiors. The minimum rule is illustrated in Figure 1.12.

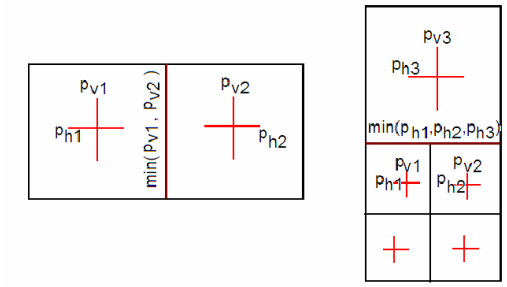


Fig.1.12.The minimum rule sets the polynomial order of approximation for edge as the minimum of corresponding orders from adjacent element interiors

The *h* refinement algorithm presented in Flaherty, Devine 1996 employs a hierarchical tree of meshes with AVL trees, allowing for a fast localization of mesh objects. Both, the refinement and the unrefinement strategies are applied.

1.3.2.2.3 Optimal refinement algorithms

Devine, Flaherty 1996 have developed an adaptive *hp*-refinement parallel algorithm for Discontinuous Galerkin simulations of hyperbolic systems of conservative laws on regular domains. The error estimator used in the algorithm is based on the relative error between the coarse and fine mesh solutions computed in the L^1-norm,

$$\text{error}_K(t) = \int_K \left| U_K^{p+1}(x,t) - U_K^p(x,t) \right| dx \tag{1.19}$$

where $U_K^p(x,t)$ denotes the coarse mesh solution, $U_K^{p+1}(x,t)$ denotes the fine mesh solution, with the fine mesh obtained by global *p* refinement (by increasing the polynomial orders of approximation by one for all finite elements). The refinement is executed on the finite elements with $\text{error}_K(t) \leq \text{TOL}$.

The local decision about a type of refinement (*h* or *p* refinement) is based on the following observation: the *p* refinement is more efficient in these areas of the mesh for which the solution is smooth, while the *h* refinement is more efficient in the areas near singularities, where the gradients are large.

The algorithm introduced in this dissertation differs from the one that has been proposed by Devine, Flaherty, 1996 in the following aspects: the decision about a type of refinement is based on the error decrease rate, with respect to the fine mesh resulting from the global *hp* refinement procedure, and the error decrease rate is measured in the H^1 norm,

whereas the algorithms described by Devine, Flaherty use the fine mesh resulting only from the p refinement procedure, and the error decrease rate is measured only in the L^1-norm.

This makes the strategy proposed in this dissertation universal. However, the price to pay is the following: we need to employ a computationally more expensive Continuous Galerkin approximation, which requires a computationally more expensive solver and implies some difficulties with the application of an iterative solver, due to worse conditioning of the system. Moreover, Flaherty, Devine, 1996 have not taken into consideration all possible refinement strategies and have not computed the error decrease rate, which is the most universal procedure suitable for all kind of engineering problems. On the other hand, they worked on the hyperbolic problems, which make their simulations very complex.

1.3.2.2.4 Communication strategies

In the work of Flaherty, Devine, 1996 the ghost elements are used to minimize the communication required by the parallel mesh refinements algorithms.

Banaś, 2003 has presented how the introduction of an overlap to the domain decomposition can simplify the parallel algorithms related to *hp* adaptive computations. The overlap consists of an additional layer of elements adjacent to the sub-domain, copied from neighboring sub-domains. The overlap is useful when the computation of some neighboring results is cheaper than their acquisition from the adjacent sub-domains. The parallel mesh refinement algorithms, including the enforcement of the 1-irregularity rule and the minimum rule, become easy to implement when the overlap is present. In this case, we can apply (up to some limit) the sequential refinement algorithms.

The disadvantage of the overlap method is a necessity of a frequent actualization of the overlap data, which results to be quite expensive.

1.3.2.3. Different parallelization methods for adaptive FE algorithms

Płażek, 1999, Płażek, 2000 and Płażek, Banaś, Kitowski, 2001 have compared two parallelization models for the adaptive FEM algorithm. The first one employs the explicit model with message-passing paradigm, whereas the other one employs the implicit model with data-parallel programming. In the implicit model, the parallelization is obtained by using compiler directives and pragmas when a loop over mesh data objects is encountered. The explicit model uses the master – slave strategy, where the master controls the solution procedure and several slave processes work in parallel. The master process is responsible for the partition of the mesh into several sub-domains, while each slave process executes the solver-related computations, such as vector-vector and matrix-vector (BLAS level 3) operations, on a single sub-domain. However, the mesh refinements are executed by the master process before the mesh is partitioned. Thus, the h refinement algorithms are actually serial, and only the iterative solver (GMRES) is executed in parallel.

The algorithm has been executed on a well known benchmark problem – the ramp problem (Woodward, Colella, 1984). The conclusion of their numerical experiments is that in case of the explicit model the efficiency does not depend strongly on the parallel machine architecture, which results from the applied master-slave strategy. However, in case of the implicit model, the resulting scalability is much worse. The last part of their

work emphasizes a need for further research on the explicit model, which is actually a subject of this dissertation.

In their second paper, Płażek, Banaś, Kitowski, Boryczko 1997, the Authors emphasize also the advantages of the two-level parallelism in adaptive FEM applications. The first level is understood as the domain decomposition and the second level as the functional decomposition. The two-level parallelism is especially important for cc-NUMA architectures where the hypernodes with high local bandwidth are interconnected by lower bandwidth properties, using the cache coherency features.

1.3.3. Parallel solvers for adaptive FE algorithms

The core part of FE adaptive algorithm is an efficient parallel solver. It is very difficult to develop such an iterative solver for adaptive FEM that could converge on different kinds of meshes and problems. There are several convergence problems for the meshes with elongated elements and mixed polynomial orders of approximation. Even if an iterative solver converges for a given variational formulation on a current *hp* mesh, it very often causes the convergence problems for other meshes and other variational formulations. On the other hand, the direct solver can always solve correctly formulated variational problems, and it never causes any convergence problems. Nevertheless, the direct solvers are much more expensive in terms of their computational complexity and memory usage.

An efficient parallel iterative multi-grid solver for DG method has been proposed by Banaś, Płażek 1997. The solver employs an overlap for the domain decomposition to compute local preconditioners with patches of elements which may be possibly located on the adjacent sub-domains. An efficient mesh partitioning algorithm for the multi-grid solvers has been also proposed by Bastian 1998, Bastian, Birken, Johanssen, Lang, Nuess, Rentz-Reicher, Wieners 1997. However, these solvers are not suitable for the general self-adaptive *hp* adaptivity, since they do not converge on multi-grids with various polynomial orders of approximation *p*. The automatic *hp* adaptivity generates a sequence of coarse meshes and their corresponding fine meshes, with higher orders of approximation, so the most suitable iterative solver should replace the expensive fine mesh solution by its projection onto several coarse mesh solutions. The sequential implementation of such two-grid solver is presented by Pardo, 2004. However, this solver requires various pre-conditioning methods for different problems.

The current state of art for sequential and parallel direct solvers for the domain decomposition – based FEM computations, can be summarized in the following way:

a) *frontal solver*. The solver browses the finite elements in the order determined by the user. It aggregates the degrees of freedom to the so-called frontal matrix. On the basis of the element connectivity information, it recognizes the fully assembled degrees of freedom and eliminates them from the frontal matrix (Irons 1970, Duff, Reid, 1983, Duff, Reid, 1984). This is done to keep the size of the frontal matrix as small as possible. The key for an efficient work of the frontal solver is the optimal ordering of finite elements,
b) *multi-frontal solver*. The solver constructs the tree of connectivity for the degrees of freedom by analysing the geometry of computational domain (Duff, Reid, 1983). It is usually done by means of the graph representation of the computational domain and graph partitioning algorithm. The frontal elimination pattern is applied

for each tree branch. The finite elements are joined into pairs and the degrees of freedom are assembled into a frontal matrix assigned to a branch. The process is repeated until we reach the root of the assembly tree. Finally, the common dense problem is solved and partial backward substitutions are recursively executed on the assembly tree,

c) *sub-structuring method solver*. This is a parallel solver working on a computational domain partitioned into multiple sub-domains (Giraud, Marocco, Rioual, 2005). It works in such a way that first, the sub-domains' internal degrees of freedom are eliminated with respect to the interface degrees of freedom. Second, the interface problem is solved. Finally, the internal problems are solved by executing backward substitution on each sub-domain, and by applying the interface problem solution computed in the second step,
d) *multiple fronts solver*. This is the simplest implementation of the sub-structuring method solver (Scott 2003, Giraud, Marocco, Rioual 2005). It executes a partial frontal decomposition on each sub-domain. Next, it sums up the contributions from particular sub-domains into one common interface problem. Finally, it solves the common interface problem by applying a sequential frontal solver,
e) *direct sub-structuring method solver*. In this version of the sub-structuring method solver, the interface problem is solved by means of the parallel solver (Smith, Bjőrstad, Gropp, 1996, Giraud, Marocco, Rioual 2005),
f) *sparse direct method solver*. This is a parallel implementation of the multi-frontal solver. An example of the sparse direct method solver has been presented in Geng, Oden, van de Geijn, 2006. Another example of the sparse direct method solver is the MUlti frontal Massively Parallel sparse direct Solver (MUMPS) (Amestoy, Duff, L'Excellent 2000, Amestoy, Duff, Koster, L'Excellent 2001, Amestoy, Guermouche, L'Excellent, Pralet 2006, MUMPS).

1.3.4. Graph grammar models

The topological structure of the finite element mesh, with a hierarchy of vertices, edges, faces and regions has been proposed by Beall, Shephard, 1997 to illustrate mesh generation and data storage.

The first attempt to model mesh transformations by applying the graph grammar concept has been proposed by Schaefer, Flasiński, 1996, for the regular triangular two-dimensional meshes with the *h* adaptation. This has been done using the quasi context sensitive graph grammar. However, the application of the quasi-context sensitive graph grammar for *hp* adaptive mesh transformation seems to be limited. The reason is that the mesh transformations, such as the enforcement of 1-irregularity rule or the minimum rule, are context dependable and cannot be modeled by the the quasi-context sensitive graph grammar .

The Composite Programmable graph grammar (CP-graph grammar) has been introduced by (Grabska 1993a, Grabska 1993b, Grabska, Hliniak 1993) as a tool for a formal description of various design processes. The CP-graph grammar describes a design process by means of the graph transformations executed on the CP-graph representation of the designed objects. In this dissertation the CP-graph grammar is used to model the mesh transformations which result from the self-adaptive *hp*-FEM computations.

1.3.5. Analysis of computational and communication complexities of adaptive algorithms

The computational complexities of iterative solvers employed by the adaptive *hp*-FEM algorithms have been analysed by Banaś 2006. In particular, this analysis takes into consideration the computational complexities of GMRES solver, with the emphasis on the algorithm computing the residuum of linear system of equations, the algorithm computing the projections between meshes used by the multi-grid solver, the block Gauss-Seidl algorithm and the algorithm for partial LU factorizations.

The communication and computational complexities of the parallel adaptive algorithms as well as its scalability have been discussed by Flasiński, Schaefer, Toporkiewicz 1996. The paper defines the scalability of the parallel adaptive algorithms as a possibility of dividing both the computational job and data that have to be processed into an arbitrary set of tasks, which can be run in parallel.

The computational complexities of the frontal solver have been discussed by Irons 2003. The computational complexity of the multiple front solver has been described by Walsh, Demkowicz 199. Geng, Oden, van de Geijn 2006 have presented the computational and communication complexities for the multi-frontal solver. These analyses do not take into account the influence of the *h* and *p* adaptation on the scalability of the solver, which is discussed in detail in this dissertation.

The computational and communication complexities for load balancing algorithms, such as HSFC and nested dissection algorithms, have been already described by Bauer, Patra 2004 and Khaira, Miller, Sheffler, 1992.

1.4. Summary of open problems

This section provides a list of the open problems in the field of research related to the parallelization of the self-adaptive *hp*-FEM algorithms.

a) there is no universal methodology for the parallelization of all components of the self-adaptive *hp*-FEM algorithms,
b) there is no mathematical model to formalize sequential and parallel mesh transformations,
c) there is no formal model of concurrency for the adaptive *hp*-FEM algorithm,
d) it is necessary to define and implement an efficient parallel direct solver for the adaptive *hp*-FEM,
e) it is necessary to define and implement the parallel self-adaptive *hp*-FEM algorithms for two- and three-dimensional elliptic problems,
f) it is necessary to define and implement an efficient parallel iterative solver re-applying coarse mesh solution for the fine mesh solution.

1.5. Summary of my research

The following scientific research of the Author provides solutions to the open problems:

a) the introduction of the graph grammar based model which enables a formal definition, verification and control of the sequential and parallel self-adaptive *hp*-FEM algorithms,
b) the description of sequential and parallel mesh transformations by means of the graph grammar productions,
c) the formal definition of the algorithm of the sequential self-adaptive *hp*-FEM based on the graph grammar and control diagrams,
d) the definition of the model of concurrency describing all parts of the self-adaptive *hp*-FEM algorithm,
e) the definition of several parallel processing models intended for the distributed and hybrid memory architectures,
f) the theoretical analysis of the created parallel processing models,
g) the execution of several numerical experiments for the proposed parallel processing models,
h) the implementation of an efficient parallel direct solver for the automatic *hp* adaptive FEM.

The parallel iterative solver for the self-adaptive *hp*-FEM has not been created yet, but the work is actually in progress (see Chapter 4).

2. Graph grammar-driven PDE solvers

This chapter introduces the concept of Composite Programmable Graph Grammar (CP-graph grammar) controling the execution of the self-adaptive *hp*-FEM algorithm. The CP-graph grammar is suitable for modeling all aspects of the self-adaptive algorithm. In this model the computational mesh is represented by the attributed CP-graph. The presentation is restricted to two-dimensional rectangular finite elements, however it can be generalized to two-dimensional triangular elements, as well as to three-dimensional meshes. Besides, in order to simplify the presentation and to reduce the number of graph grammar productions and the complexity of control diagrams, the presented graph grammar has been restricted to a row of initial mesh elements. Thus, the presented graph grammar assumes that the initial mesh is limited to the one-dimensional sequence of two-dimensional rectangular finite elements. The graph grammar can be generalized to more complex initial meshes and to other types of elements as well as to three-dimensional computations, which will be the subject of the future work.

The original definition of the CP graph grammar (Grabska 1993a, Grabska 1993, Grabska, Hliniak 1993) used the directed graphs. However, for the purposes of modeling the *hp* meshes, undirected graphs seem to be more appropriate. Thus, the original definitions presented by Grabska 1993a, Grabska 1993, Grabska, Hliniak 1993 have been transformed in such a way that we can apply them to non-directed graphs.

Definition 2.1.
Composite programmable graph (CP-graph) over $W = A_V \times [i]_N$ and A_E is defined as

$$\left(\left(V, E \right), \xi_V, \xi_E, att_V, att_E \right) \tag{2.1}$$

where

- V stands for a set of graph vertices,
- $\xi_V : V \to A_V \times [i]_N$ is a function labeling vertices, with A_V being the set of vertex labels
- $[i]_N = \{ \{1\}, \{1,2\}, \{1,2,3\}, ..., \{1,2,...,i\} \}$ is a continuous integer numbering of vertex bounds.
- $B(V) = \bigcup_{v \in V} \beta(\xi_V(v)) \times \{v\}$ is a set of pairs – vertex bound number and the vertex itself,
- $\beta(\xi_V(v))$ is the projection onto vertex bound number

- $E \subseteq B(V) \times B(V)$ stands for a set of graph edges such that
  - there are no loop edges $\forall((v,j),(i,u)) \in E, v \neq u$
  - there are no free bound edges $\forall(v,j) \in B(V)$ there is no more than one bound $(i,u) \in B(V)$ such that $((v,j),(i,u)) \in E$
  - the symmetric condition $\forall(v,j) \in B(V)$ there is no more than one bound $(i,u) \in B(V)$ such that $((u,i),(j,v)) \in E$
- A_E denotes the set of edge labels
- $\xi_E : E \to A_E$ denotes the edge labeling function, with A_E being the set of edge labels
- $att_V : V \to A_T$ is the vertex attributing function, with A_T being the set of vertex attributes
- $att_E : E \to A_R$ is the edge label attributing function, with A_R being the set of edge attributes
- $W = A_V \times [i]_N$ denotes the set of graph vertices with numbered bounds

Definition 2.2.
Composite programmable graph grammar (CP-graph grammar) over $W = A_V \times [i]_N$ and A_E is a pair

$$(P, S) \tag{2.2}$$

where

- P is a finite set of productions of the form (l,r) where l and r are CP-graphs with the same number of free bounds
- S called the axiom symbol, such that there is at least one production of the form (S,r)

The CP-graph grammar models the initial mesh generation, the procedure of the *h* refinement (breaking selected finite elements into son elements) and the *p* refinement (adjusting polynomial orders of approximation on selected element edges and interiors). The mesh regularity rules are automatically enforced on the level of grammar syntax. The order of execution of graph transformations (graph grammar productions) is defined by the control diagrams. The graph grammar models also the algorithm of the multi-frontal direct solver, used in the self-adaptive algorithm. In the following definitions we introduce the CP graphs used for modeling particular parts of the self-adaptive *hp*-FEM algorithm.

Definition 2.3
The CP-graph modeling an initial mesh with polynomial orders of approximation lower than 10, is defined using the following sets of graph vertices labels A_V, graph edges labels A_E and graph vertices attributes A_T

$$A_V = A_V^1 = \{\mathbf{x, y, y1, Iel1, Iel2, Iel, , iel, F, I, v, v2, B, fake}\} \tag{2.3}$$

$$A_T = A_T^1 = \{(1,1),(1,2),(2,1),(2,2),...,(9,9)\} \cup \{1,2,....,9\} \tag{2.4}$$

$$A_E = A_E^1 = \{\mathbf{N,S,W,E,NW,NE,SW,SE}\} \tag{2.5}$$

and the empty set of graph edge attributes $A_R = A_R^1 = \varnothing$.

Definition 2.4

The CP-graph modeling the solver execution on an initial mesh, represented by CP-graph defined by Definition 2.3, is defined by the following sets of graph vertices labels A_V, graph edges labels A_E and graph vertices attributes A_T

$$\begin{aligned} &A_V = A_V^2 = \\ &A_V^1 \cup \left\{\alpha^{\mathrm{i}}\,\mathbf{v}, \alpha^{\mathrm{i}}\,\mathbf{F}, \alpha^{\mathrm{i}}\,\mathbf{I}, \alpha^{\mathrm{i,j}}\,\mathbf{v2}, \alpha^{\mathrm{i,j}}\,\mathbf{F}\right\}_{i,j=1,\ldots,N} \cup \\ &\left\{\beta^{\mathrm{i}}\,\mathbf{v2}, \beta^{\mathrm{i,j}}\,\mathbf{F}, \beta^{\mathrm{i}}\,\mathbf{I}, \beta^{\mathrm{i,j}}\,\mathbf{v2}\right\}_{i,j=1,\ldots,N} \end{aligned} \tag{2.6}$$

$$A_T = A_T^2 = A_T^1 \tag{2.7}$$

$$A_E = A_E^2 = A_E^1 \tag{2.8}$$

where N denotes the maximum number of local matrices (maximum number of elements) and the set of graph edge attributes $A_R = A_R^2 = \varnothing$ is empty.

Definition 2.5

The CP-graph modeling the *h* refinement process on a mesh, represented by CP-graph described in Definition 2.3, is defined by the following sets of graph vertices labels A_V, graph edges labels A_E and graph vertices attributes A_T

$$A_V = A_V^3 = A_V^1 \cup \{\mathbf{i,J,Fi,e,Fe,E,E2,E3,F1,F2,J2,J3,J4}\} \tag{2.9}$$

$$A_T = A_T^3 = A_T^1 \tag{2.10}$$

$$A_E = A_E^3 = A_E^1 \cup \{\mathbf{1^{st}, 2^{nd}}\} \tag{2.11}$$

and the empty set of graph edge attributes $A_R = A_R^3 = \varnothing$.

Definition 2.6

The CP-graph modeling the *p* refinement process on an *h* refined mesh is equal to the CP-graph defined in Definition 2.5.

$$A_V = A_V^4 = A_V^3 \tag{2.12}$$

$$A_T = A_T^4 = A_T^3 \tag{2.13}$$

$$A_E = A_E^4 = A_E^3 \tag{2.14}$$

$$A_R = A_R^4 = A_R^4 \tag{2.15}$$

Definition 2.7

The CP-graph modeling the execution of the solver on an *hp* refined mesh, represented by CP-graph described in Definition 2.6, is defined by the following sets of graph vertices labels A_V, graph edges labels A_E and graph vertices attributes A_T

$$A_V = A_V^5 =$$
$$A_V^4 \cup \left\{ \alpha^{\mathrm{i}}\,\mathbf{F1}, \alpha^{\mathrm{i,j}}\,\mathbf{Fi}, \alpha^{\mathrm{i,j,k,l}}\,\mathbf{v}, \alpha^{\mathrm{i,j}}\,\mathbf{e}, \alpha^{\mathrm{i,j,k,l}}\,\mathbf{e} \right\}_{i,j,k,l=1,\ldots,N} \cup \tag{2.16}$$
$$\left\{ \beta^{\mathrm{i}}\,\mathbf{F1}, \beta^{\mathrm{i,j}}\,\mathbf{Fi}, \beta^{\mathrm{i,j,k,l}}\,\mathbf{v}, \beta^{\mathrm{i,j}}\,\mathbf{e}, \beta^{\mathrm{i,j,k,l}}\,\mathbf{e} \right\}_{i,j,k,l=1,\ldots,N}$$

$$A_T = A_T^5 = A_T^4 \tag{2.17}$$

$$A_E = A_E^5 = A_E^4 \tag{2.18}$$

where N denotes the maximum number of local matrices (maximum number of elements) and the set of graph edge attributes $A_R = A_R^5 = \varnothing$ is empty.

Definition 2.8
The CP-graph modeling the partitioning and merging of *hp* refined mesh, represented by CP-graph described in Definition 2.6, is defined by the following sets of graph vertices labels A_V, graph edges labels A_E and graph vertices attributes A_T

$$A_V = A_V^6 = A_V^4 \cup \{ \mathbf{BI, eI, Int, eJ, Join} \} \tag{2.19}$$

$$A_T = A_T^6 = A_T^4 \tag{2.20}$$

$$A_E = A_E^6 = A_E^4 \tag{2.21}$$

and the set of graph edge attributes $A_R = A_R^6 = \varnothing$ is empty.

The following definitions list graph grammar productions responsible for modeling particular parts of the self-adaptive *hp*-FEM algorithm. The productions are illustrated in several Figures, presented later in this chapter.

Definition 2.9
The CP-graph grammar modeling the generation of an initial mesh represented by the CP-graph, introduced in Definition 2.3, is defined by the following set of graph grammar productions

$$P = P_1 = \{ \quad \mathbf{P1, P2, P3, P4, P5, P6, PII, PIC} \quad \} \tag{2.22}$$

Definition 2.10
The CP-graph grammar modeling the execution of the solver on an initial mesh represented by the CP-graph, introduced in Definition 2.3, is defined by the following set of graph grammar productions

$$P = P_2 = \{ \quad \mathbf{P\ aggregate\ interior,\ P\ aggregate\ boundary\ edge,}$$
$$\mathbf{P\ aggregate\ corner\ vertex,\ P\ aggregate\ edge,\ P\ aggregate\ shared\ vertex,}$$
$$\mathbf{P\ aggregate\ interior,\ P\ eliminate\ boundary\ edge,\ P\ eliminate\ corner\ vertex,}$$
$$\mathbf{P\ eliminate\ edge,\ P\ eliminate\ common\ vertex} \quad \} \tag{2.23}$$

Definition 2.11
The CP-graph grammar modeling the execution of *h* refinements on a mesh represented by the CP-graph, introduced in Definition 2.6, is defined by the following set of graph grammar productions

$$P = P_3 = \{ \text{ P break interior, P break edge, PFE1, PFE2, PFE3, PFE4, Pwest, Pnorth, Psouth, Peast } \} \quad (2.24)$$

Definition 2.12
The CP-graph grammar modeling the execution of *p* refinements on a mesh represented by the CP-graph, introduced in Definition 2.6, is defined by the following set of graph grammar productions

$$P = P_4 = \{ \text{ P p-refine interior, P min-rule east edge, P min-rule west edge, P min-rule south edge, P min-rule north edge } \} \quad (2.25)$$

Definition 2.13
The CP-graph grammar modeling the execution of the solver on an *hp* refined mesh represented by the CP-graph, introduced in Definition 2.6, is defined by the following set of graph grammar productions

$$P = P_5 = P_2 \cup \{ \text{ P aggregate corner vertex, P process son elements, P process shared vertex, P aggregate interface, P propagate interface aggregation, P merge interiors into horizontal pairs, P merge interiors into vertical pairs, P mergo sons problem, P process interface } \} \quad (2.26)$$

Definition 2.14
The CP-graph grammar modeling the partitioning and merging of an *hp* refined mesh represented by the CP-graph, introduced in Definition 2.6, is defined by the following set of graph grammar productions

$$P = P_6 = \{ \text{ DD1, DD2, DD3, DD4, DD5, DD6 } \} \quad (2.27)$$

The first section of this chapter, entitled "Graph grammar model with atomic tasks for the self-adaptive algorithm", presents the applicability of the CP-graph grammar for the modeling of the self-adaptive *hp*-FEM algorithm. The presentation refers to an exemplary variational formulation (1.10-1.12) for the heat transfer problem. However, all the presented considerations remain valid if we replace the heat transfer problem (1.10-1.12) by the general variational formulation of the 2D elliptic boundary-value problem introduced in Definition B.2 in Appendix B.

The CP-graph grammar model examines the concurrency of the self-adaptive algorithm, by using the atomic tasks, tasks and super-tasks, introduced in the following definitions.

Definition 2.15
An *atomic task* is a single graph grammar production. This is a basic undividable computational task which must be executed sequentially.

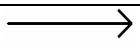
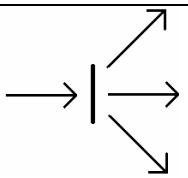
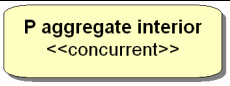
Name	Exemplary graphical representation	Description
Initial state		Represents the starting point of execution
Final state		Represents the end of execution
State		Represents the execution of a single atomic task (graph grammar production) on some part of the graph representation of the mesh
Transition		Sets the partial order of execution of atomic tasks (graph grammar productions)
Synchronization		Represents the synchronization point, after which several atomic tasks (graph grammar productions) are executed on different parts of the graph representation of the mesh
Concurrent state		Represents the concurrent execution of a single atomic task (graph grammar production) on several parts of the graph representation of the mesh

Table 2.1. UML symbols used on control diagrams on the level of atomic tasks

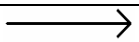
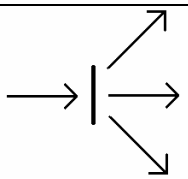
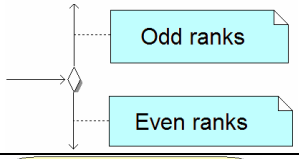
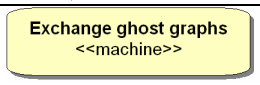
Name	Exemplary graphical representation	Description
Initial state		Represents the starting point of execution
Final state		Represents the end of execution
State		Represents the execution of a single atomic task (graph grammar production) within a task on some part of its local graph representation of the mesh
Transition		Sets the order of execution of atomic tasks (graph grammar productions)
Synchronization		Represents the inter-task synchronization point, after which several tasks execute some atomic tasks (graph grammar productions) on different parts of their local graphs
Send		Represents the send operation performed by a task
Receive		Represents the receive operation performed by a task
Communication channel		An arrow represents the communication channels, from the sender task to the receiver task
Choice with predicates of applicability		Exemplary predicates of applicability – the tasks with odd rank and the tasks with even rank select different execution paths
Machine		Exemplary reference to another control diagram, called “Exchange ghost elements”

Table 2.2. UML symbols used on control diagrams on the level of tasks

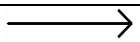
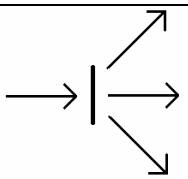
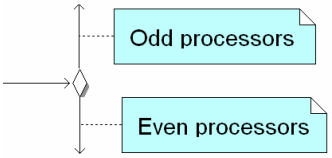
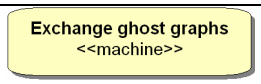
Name	Exemplary graphical representation	Description
Initial state		Represents the starting point of execution
Final state		Represents the end of the execution
State		Represents the execution of a single atomic task (graph grammar production) within a super-task on some part of its local graph representation of the mesh
Transition		Sets the order of execution of atomic tasks (graph grammar productions)
Synchronization		Represents the inter-super-task synchronization point, after which several super-tasks execute some atomic tasks (graph grammar productions) on different parts of their local graphs
Send		Represents the send operation performed by a super-task
Receive		Represents the receive operation performed by a super-task
Communication channel		An arrow represents the communication channels, from the sender super-task to the receiver super-task
Choice with predicates of applicability		Exemplary predicates of applicability – the super-tasks with odd number and the super-tasks with even number select different execution paths
Machine		Exemplary reference to another control diagram, called “Exchange ghost elements”

Table 2.3. UML symbols used on control diagrams on the level of super-tasks

Definition 2.16
A *control diagram* for the level of atomic tasks is a graphic representation of the partial order of execution of atomic tasks. The control diagram is obtained by analysing the logical independence between atomic tasks. This analysis is based on the identification of the minimal causality relations, and is followed by the denotation of the atomic tasks that can be executed at the same time. The control diagram consists of states representing an execution of a single atomic task, and of transitions which determine the partial order of execution of atomic tasks. The control diagram is represented by UML state diagrams, with the symbols explained in Table 2.1. The abstract concurrent self-adaptive *hp*-FEM algorithm defined by means of the control diagrams can be executed for the shared memory architecture, with the graph representation of the mesh shared between multiple concurrent atomic tasks. The control diagrams are executed by all atomic tasks on one global graph representation of the mesh.

Definition 2.17
A *task* is defined as an execution of multiple atomic tasks on its local sub-graph representation of a part of the mesh. A sub-graph is identified with an initial mesh element. Each task has its own *rank* – a unique global identifier.

Definition 2.18
A *control diagram* for the level of tasks is a graphic representation of the partial order of execution of atomic tasks, agglomerated into a task, working on a local graph representing a single initial mesh element. The control diagram for the level of tasks represents a sequential execution of atomic tasks within a task on its local sub-graph, representing initial mesh element. Thus, each task executes the control diagram on its local sub-graph, with some additional inter-task communication added. The control diagram is represented by UML state diagrams, with the symbols explained in Table 2.2. Some transitions on the control diagrams are enriched now with predicates of applicability. If a transition does not have a predicate of applicability, it can be used by all tasks. However, if a transition has a predicate of applicability assigned, it can be used only by tasks fullfilling this predicate. Now, each task has its own *rank* – a unique global identifier. Usually, the predicates of applicability contain some expressions with task ranks. In other words, the adaptive algorithms are defined now by means of multiple tasks working on the graph representation of a mesh distributed into several sub-graphs. The algorithms require now some communication between tasks working on sub-graphs assigned to adjacent parts of a mesh. Thus, the communication channels are defined between adjacent tasks. In order to express the communication on the control diagrams, some states represent send / receive operations, with predicates of applicability clearly defining the sender's and receiver's tasks.

At this point the model is still analysed on the abstract level and we neither introduce particular parallel machine architecture nor do we consider a particular number of processors. The algorithms are defined in the most general manner, to enable an efficient execution in case when there are many more tasks than processors. In order to do so, the redundancy of the algorithms executed for particular tasks is reduced as much as possible.

It should be emphasized that the tasks defined in Definition 2.18 constitute the grain for the load balancing and scheduling algorithms. The algorithms are introduced in Section

2.3. At this point, it is assumed that the architecture of the parallel machine is already known. The input for the algorithms (the grain) is a list of tasks with estimated computational load for each task, related to the most expensive parts of the self-adaptive *hp*-FEM algorithm. As the output we obtain a map that assigns the tasks to processors.

Definition 2.19
A *super-task* is defined as an execution of multiple atomic tasks on its local graph representation of a part of the mesh, representing a single sub-domain. The partition of the mesh into sub-domains results from the execution of load balancing and mesh partitioning algorithms. Each super-task is associated with a single sub-domain assigned to a single processor, and has its own global number.

Definition 2.20
A *control diagram* for the level of super-tasks is a graphic representation of the partial order of execution of atomic tasks, agglomerated into a super-task, working on a local graph representing a single sub-domain. The control diagram for the level of super-tasks represents a sequential execution of atomic tasks within a super-task on its local sub-graph, representing a single sub-domain. Thus, each task executes the control diagram on its local sub-graph, with some additional inter-super-task communication added. The control diagram is represented by UML state diagrams, with symbols explained in Table 2.3. Some transitions on the control diagrams are enriched now with predicates of applicability. If a transition does not have a predicate of applicability, it can be used by all super-tasks. However, if a transition has a predicate of applicability assigned, it can be used only by super-tasks fullfilling this predicate. Now, each task has its own number – a unique global identifier. At this point, we assume that the distributed memory architecture is used, and each super-task is assigned to a single processor. Thus, the super-task number becomes automatically the processor number. Usually, the predicates of applicability contain some expressions with super-task numbers. In other words, the adaptive algorithms are defined now by means of multiple super-tasks working on the graph representation of a mesh distributed into several sub-graphs. The algorithms require now some communication between super-tasks working on sub-graphs assigned to adjacent sub-domains. Thus, the communication channels are defined between the adjacent super-tasks. In order to express the communication on the control diagrams, some states represent send / receive operations, with predicates of applicability clearly defining the sender's and receiver's super-tasks.

The presented analysis is based on the PCAM (Partitioning, Communication, Agglomeration and Mapping) model introduced by Foster, but it differs from his definition in several aspects. There are three levels in the hierarchy of tasks: the atomic tasks, the tasks and the super-tasks. The communication channels are defined between tasks, in order to obtain abstract parallel algorithms on the level of grains for the load balancing and mesh (graph) partitioning algorithms. The parallel algorithms are redefined on the level of super-tasks, resulting from the execution of the load balancing and partitioning algorithms. The communication channels are redefined on the level of super-tasks, since the amount of transferred data differs from the amount defined on the level of tasks, and the parallel algorithms are slightly modified to express the new decomposition of the graphs.

2.1. Graph grammar model with atomic tasks for the self-adaptive algorithm

The original self-adaptive *hp*-FEM algorithm introduced by Demkowicz 2006 is described in Appendix B. It can be summarized in the following eight steps:

1) **generate an initial mesh**. The initial mesh becomes the so-called "coarse mesh" for the first iteration,
2) **solve the coarse mesh problem** by using the direct solver for *hp*-FEM,
3) **generate the fine mesh** from the coarse mesh. Each finite element from the coarse mesh is broken into four elements and the polynomial order of approximation is uniformly raised by one,
4) **solve the fine mesh problem** by using the direct solver for *hp*- FEM,
5) **select the optimal refinement** strategy for each finite element from the coarse mesh. That should be based on the relative error estimations computed using the coarse and fine mesh solutions,
6) **execute all required *h* refinements**,
7) **execute all required *p* refinements**,
8) **if** the maximum relative error of the solution is greater than the required accuracy, **then go to Step 2**. The new optimal mesh becomes the coarse mesh for the next iteration.

The following meta-definitions are introduced for better understanding of the algorithm:

a) *FE mesh*. Partition of a domain into several finite elements, e.g. partition of L-shape domain from Figure 1.1 into FE mesh with 3 finite elements, as presented in Figure 1.2,
b) *initial mesh*. FE mesh, input to the algorithm,
c) *FE mesh problem*. Solve the system of linear equations resulting from FE discretization of a weak variational form, e.g. solve the system of equations (1.10-1.12) resulting from the FE discretization of the weak formulation (1.5-1.8),
d) *solution of the FE mesh problem*. The degrees of freedom u_{hp}^{i} obtained by solving FE mesh problem (1.10-1.12),
e) *mesh refinements – h*, *p* or *hp* refinements executed on selected finite elements: *h refinement* consists in breaking some finite element into smaller elements; *p refinement* consists in changing polynomial order of approximation for some finite element edges or interiors; *hp refinement* is a mixture of *h* and *p* refinements,
f) *optimal mesh*. The mesh providing the higher possible error decrease rate (1.17), obtained by executing a sequence of *h*, *p*, or *hp* refinements on the initial mesh.

For more precise mathematical definitions, see Appendix B.

The particular steps of the algorithm will be formulated on the basis of the CP-graph grammar in the next part of this chapter. This graph grammar model is intended for the two-dimensional rectangular finite elements. For the detailed mathematical definition of the finite element, see Appendix A. In order to simplify and to reduce the number of graph grammar productions, we assume that the initial mesh is a linear sequence of rectangular finite elements.

2.1.1. Algorithm of initial mesh generation

The self-adaptive *hp*-FEM algorithm starts with an arbitrary initial mesh. The initial mesh can be either generated by executing a sequence of graph transformations provided by the user, or reconstructed from the initial data obtained from some external mesh generator (e.g. NASTRAN).

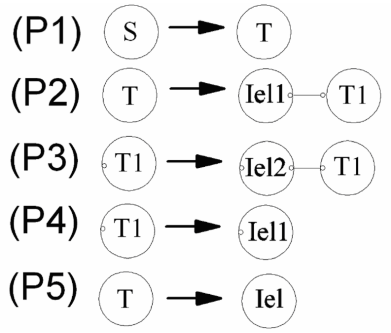


Fig.2.1. Graph transformations generating structure of initial mesh

Let us focus first on the subset of graph transformations modeling the generation of an arbitrary initial mesh. The presentation is limited to the horizontal row of initial mesh elements, for the sake of simplicity. However, the presented graph transformations can be generalized as one case of an arbitrary two-dimensional initial mesh. The process of the initial mesh generation is expressed by the graph transformations, presented in Figure 2.1.

The Figure introduces several graph grammar productions. Each graph grammar production consists of two CP-graphs, one on the left and the other one on the right-hand side. The first graph grammar production denoted by **(P1)** has the single vertex CP-graph, denoted by **S** symbol, on its left-hand side. This is the initial CP-graph called the graph grammar axiom. The graph grammar productions are executed on a current CP-graph representation of the computational mesh (called the current CP-graph in the following explanations).

The initial CP-graph representation of the mesh is supposed to be equal to the single **S** vertex axiom graph. The execution of a graph grammar production consists of the following steps:

a) the graph on the left-hand side of the production must be localized on the current CP-graph representation of the mesh,
b) the graph is removed from the current CP-graph,
c) the new graph, on the right-hand side of the production becomes a replacement for the removed graph,
d) the process of removing the left-hand-side graph is very often accompanied by releasing some bounds which connect the vertices of the removed graph with some vertices of the current CP-graph,
e) a number of bounds on the left-hand-side graphs and on the right-hand-side graphs of the production must be equal. The new right-hand-side graph of the production is connected to the corresponding vertices of the current graph,
f) formally, the bounds on the left-hand-side graphs and on the right-hand-side graphs should be always numbered in order to establish a rule for the connection

of the right-hand-side graph bounds with the remaining vertices of the current graph. However, for the sake of simplicity, if the connection rule can be concluded from the context of the production, the numbering is omitted.

The presented subset of the graph transformations allows us to generate a horizontal sequence of initial mesh elements. An exemplary application of the sequence of graph transformations defining an initial mesh with two finite elements is illustrated in Figure 2.2.

Fig.2.2. The sequence of graph transformation **(P1)-(P2)-(P4)** generating an initial mesh with two finite elements

The generation of a row of elements starts from the axiom graph. First, the **(P1)** production is executed. The production replaces the single **S** vertex graph by the single **T** vertex graph. Second, the **(P2)** production is executed. The production replaces the single **T** vertex graph by a new graph, with two vertices, denoted by **Iel** and **T1** symbols, connected by one edge. The last executed production **(P4)** removes the sub-graph with the single **T1** vertex and adds a new graph with the single **Iel1** vertex. The new graph is connected to the second **Iel** vertex in place of the removed graph.

The subset of graph transformations presented in Figure 2.1 generates only a topology of the initial mesh. Once the topology of the initial mesh is generated, we can generate also the structure of each element.

It is assumed that each initial mesh element is a two-dimensional rectangular *hp* finite element. The structure of the *hp* finite element is as follows: it consists of four vertices, four edges and the interior. The first order shape functions are defined at the element vertices, the hierarchical higher order shape functions are defined at the element edges, and the higher order bubble shape functions are defined at the element interior. For the formal definition of the *hp* finite element, see Appendix A. The structure of an exemplary *hp* finite element, with the second order of approximation used in the vertical direction and the third order of approximation used in the horizontal direction, is illustrated in Figure 2.3. The polynomial orders of approximation are denoted in the CP-graph by attributing the graph vertices. The location of the finite element vertices and edges is identified by **N**, **S**, **W**, **E**, **NW**, **NE**, **SW** and **SE** labels assigned to the graph edges.

The process of generation of the structure of initial mesh elements is described by two graph transformations, presented in Figures 2.4 and 2.5. The first graph transformation generates the structure of a single initial mesh element, and the second graph transformation identifies the edges of adjacent initial mesh elements. Note that these graph transformations must be defined also for **Iel** label replaced by **Iel1** and **Iel2**, since the initial mesh elements are denoted by these three labels. A type of label to be applied depends on the number of neighboring initial mesh elements.

The sequence of graph transformations, which generate the structure of the two initial mesh elements from Figure 2.2 is presented in Figure 2.6. This corresponds to two executions of **(PII)** graph transformation and one execution of **(PCI)** graph transformation and is denoted by the sequence $\mathbf{(PCI)^2\text{-}(PII)}$. In particular, the whole sequence of the graph grammar productions, executed from the graph grammar axiom, is the following: $\mathbf{(P1)\text{-}(P2)\text{-}(P5)\text{-}(PCI)^2\text{-}(PII)}$. The execution of **(P1)-(P2)-(P5)** productions has been summarized in Figure 2.2. The first execution of the **(PCI)** production removes the first

sub-graph with the single **Iel1** vertex from the current graph (presented on the last panel in Figure 2.2). The removed graph is replaced by a new graph, on the right-hand side of the production, with the vertices representing the structure of the first element connected as son vertices to the **iel** vertex. The second execution of the **(PCI)** production removes the second sub-graph with the single **Iel1** vertex from the current graph. The removed graph is replaced by a new graph, on the right-hand side of the production, with the vertices representing the structure of the second element connected as son vertices to the second **iel** vertex. The resulting graph is presented on the top panel in Figure 2.6. Finally, the **(PIC)** production is executed. The production localizes a slightly complicated sub-graph on its left-hand side on the current graph and removes the graph, leaving two bounds unconnected (the bounds connecting the **F** vertices with the **I** vertices). Then, a new graph, on the right-hand side of the production is added to the current graph, and connected with two previously cut bounds. The resulting graph is presented on the bottom panel in Figure 2.6.

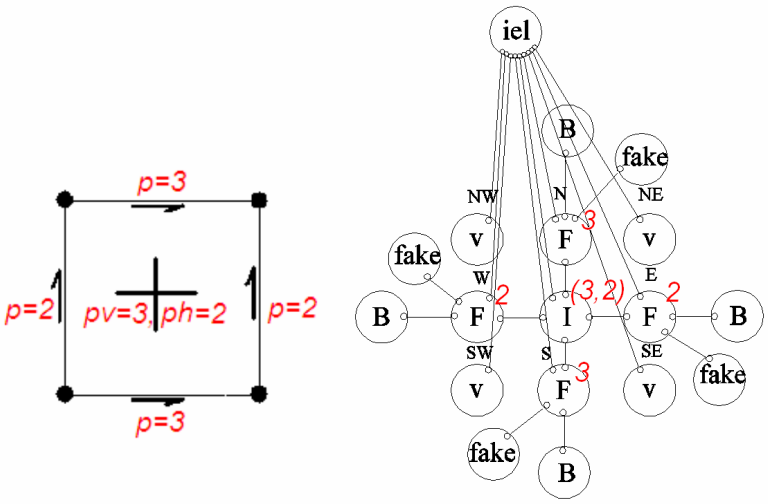


Fig.2.3. Structure of exemplary *hp* finite element, together with the corresponding CP-graph representation. The third order of approximation is used in the horizontal direction and the second order of approximation is used in the vertical direction

At this point, we can examine the concurrency of the algorithm for an initial mesh generation. This can be done by denoting those atomic tasks (defined as graph grammar productions) that can be executed in the concurrent way.

Note that the state of the parallel algorithm is determined by its position on the control diagram and by the current graph representation of the computational mesh. Each state of the control diagram denotes some atomic tasks, defined as the execution of graph grammar production on the graph representation of the mesh. Each production is executed on a sub-graph of the graph representation of the mesh. The sub-graph is determined by the left-hand side of the production. If a sub-graph on the left-hand side of the production is localized in several positions on the current graph representation of the mesh, it is possible to execute the production in the concurrent way.

The initial mesh generation consists of two parts. The first part is related to the generation of the topology of the initial mesh. This has been denoted by graph grammar productions **(P1-P5)** presented in Figure 2.1. The second part is related to the generation of the structure of initial mesh elements as well as to the identification of common edges of adjacent initial mesh elements, which has been denoted by the productions **(PII)** and **(PIC)** in Figures 2.4 and 2.5.

The presented graph grammar has been limited to the simplified case of the linear sequence of initial mesh elements, so the generation of the topology of the mesh is a purely sequential operation. This is because each graph grammar production (atomic task) employs the sub-graph generated by the previous production (the left-hand side of each production employs graph vertices denoted by **T** or **T1** symbols generated by the previous production).

However, the generation of the mesh structure can be executed in the concurrent way. The structure of each initial mesh element can be generated in a way that is independent of its adjacent elements. The process of identification of edges can be executed in the concurrent way, if the mesh elements have a structure of a linear sequence. This is summarized in Figure 2.7. These states of the control diagram on which the graph grammar productions can be executed in the concurrent way, are denoted by the `<<concurrent>>` stereotype. The additional synchronization points before the concurrent execution of **PII** and **PIC** have been added.

The alternative way to generate an initial mesh is to provide the input data coming from an external mesh generator. Such input data usually have the following format:

```
List of vertices (geometrical coordinates)
List of edges (defined by two vertices)
List of elements (defined by four edges)
List of polynomial orders of approximation
 for elements interiors
```

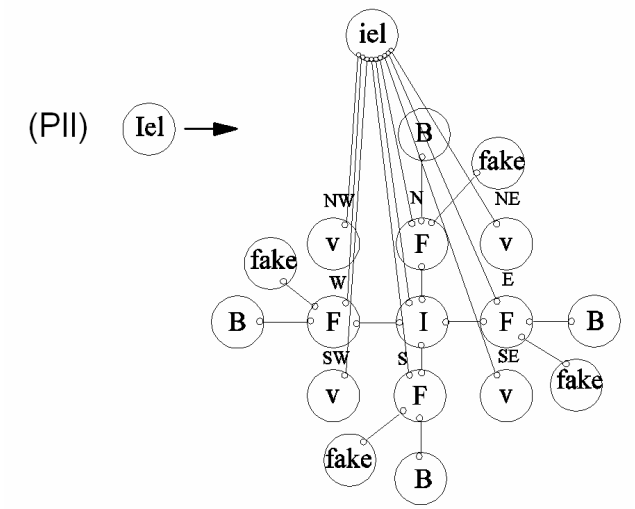


Fig.2.4. Graph grammar production generating structure of initial mesh element

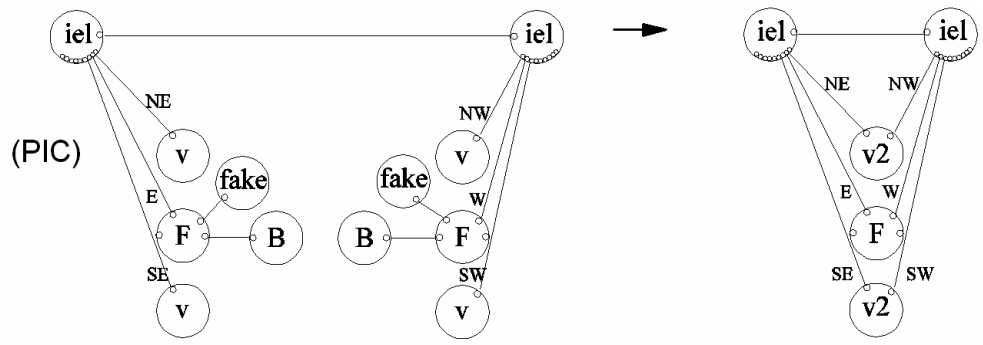


Fig.2.5. Graph grammar production identifying common edges for two adjacent initial mesh elements

In this case, the graph representation of the initial mesh results from the execution of the following algorithm:

```
Generate graph vertices representing element vertices
  (denoted by v)
Attribute vertices (assign coordinates to vertices)
Generate graph vertices representing element edges
  (denoted by F)
Connect graph vertices representing element edges with graph
  vertices representing element vertices
Generate graph vertices representing element interiors
  (denoted by I)
Connect graph vertices representing element interiors with
  graph vertices representing element edges
Generate graph vertices representing initial elements
  (denoted by iel)
Connect graph vertices representing initial element with
  graph vertices representing element vertices, edges and
  interiors
Generate graph vertices representing the boundary
  (denoted by B and fake)
Attribute graph vertices representing element interiors
  by polynomial orders of approximation
```

Algorithm 1. Algorithm for the generation of initial mesh

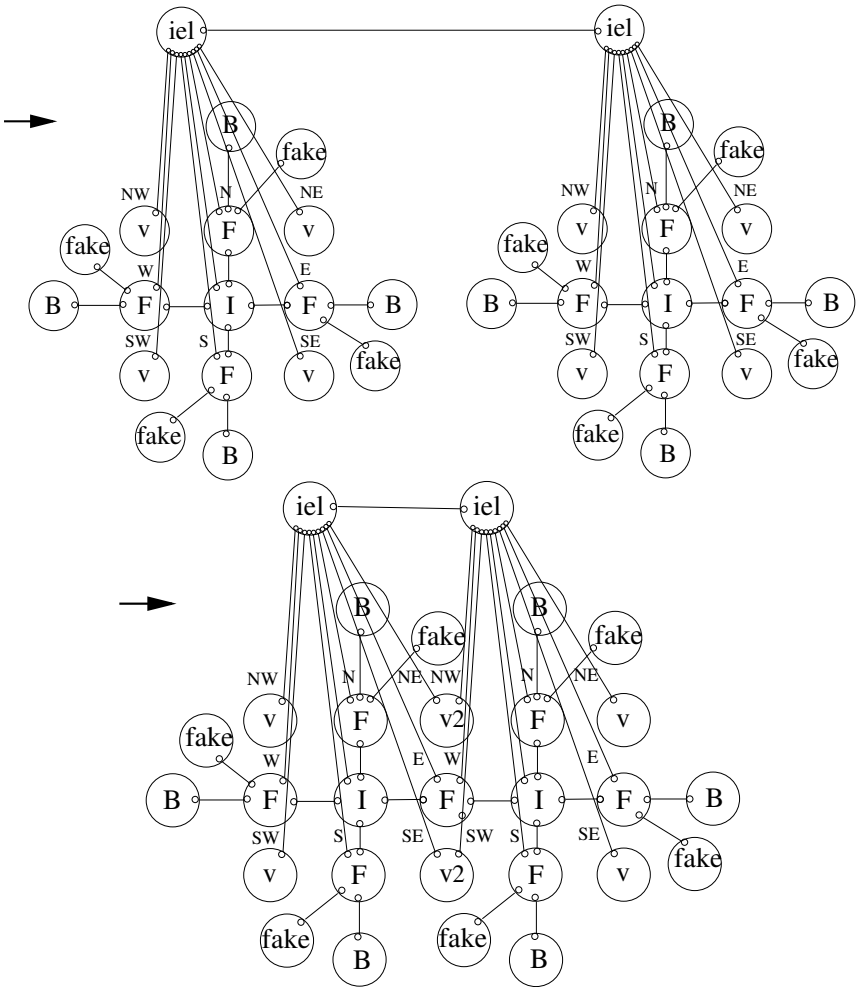


Fig.2.6. The sequence of graph transformations **(PII)^2-(PIC)** generating the structure of the two initial mesh elements from Figure 2.2

The initial mesh generated in one of the above ways is called the *coarse* mesh.

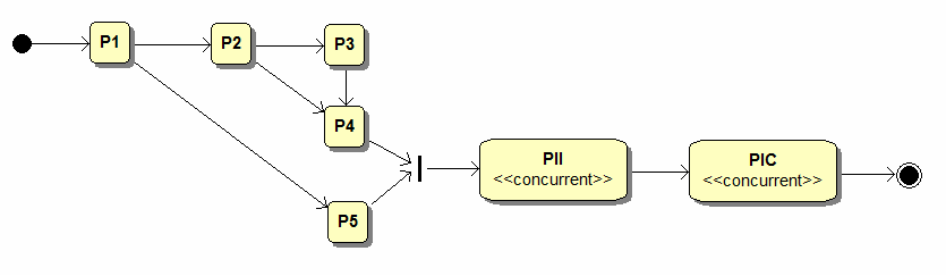


Fig.2.7. Control diagram for the generation of initial mesh on the level of atomic tasks

2.1.2. Algorithm of solution of coarse mesh problem

In the next step of the self-adaptive *hp*-FEM algorithm, the FEM discretization of the weak form of PDE (1.10-1.12) is solved for the coarse mesh. It is necessary to define an efficient direct solver working on the graph representation of the computational mesh. In the first part of this section we introduce the direct solver algorithm for the *hp* finite element mesh. In the second part of this section, the direct solver algorithm is expressed in terms of graph transformations and control diagram, which defines the order of execution of the graph transformations related to the solver operations.

The coarse mesh approximation is obtained with the basis of the coarse mesh approximation space V_{hp}. The basis is constructed by means of the graph representation of the coarse mesh. Each graph vertex denoted by label **v** corresponds to a finite element vertex, and contributes a single first order shape function to the basis. Each graph vertex denoted by **F** label corresponds to a finite element edge, and contributes $p-1$ higher order edge shape functions to the basis. Here *p* denotes the polynomial order of approximation of the edge. Each graph vertex denoted by **I** label corresponds to a finite element interior, and contributes $(p_h-1)(p_v-1)$ higher order interior shape functions to the basis. Here (p_h, p_v) denote the polynomial orders of approximation in the horizontal and vertical directions, used for the element interior. For the formal definition of a hierarchical shape functions used for the *hp* finite element, see Appendix A.

To facilitate the discussion on the proposed solver, let us consider a simple two finite element problem presented on panel (a) in Figure 2.8. The *hp*-FEM discretization of the weak form of considered PDE leads to the following system of equations

$$\sum_{i=1}^{N_{hp}} u^i_{hp}\, b\left(e^i_{hp}, e^j_{hp}\right) = l\left(e^j_{hp}\right) \quad j=1,\ldots,N_{hp} \tag{2.28}$$

where $b\left(e^i_{hp}, e^j_{hp}\right)$ and $l\left(e^j_{hp}\right)$ are problem-dependent integrals of shape functions e^i_{hp} and e^j_{hp} on the domain Ω presented on panel a) in Figure 2.8. N_{hp}=15 is the total number of shape functions, and u^i_{hp} are the unknown degrees of freedom – coordinates of the approximated solution in the shape function basis $\left\{e^i_{hp}\right\}_{i=1}^{N_{hp}}$. For the sake of simplicity, we

assume that the polynomial orders of approximation in both directions are equal to two, for all edges and for the element interiors.

The first order shape functions related to the top vertices are presented on panel (b) in Figure 2.8. The support of the first and fifth shape function, denoted by light grey colour, covers a half of the interior of a single element. The support of the third shape function, denoted by dark grey colour, covers both elements. The other first order shape functions are defined in a similar way.

The second order shape functions related to the edges are presented on panel (c) in Figure 2.8. The support of the second shape function, denoted by light grey colour, covers only the first element, while the support of the fourth shape function, denoted by dark grey colour, covers only the second element. However, the support of the eighth shape function covers both elements.

The shape functions of the element interior are illustrated on panel (d) in Figure 2.8. Their support covers only a single element interior. The *hp*-FEM employs also the higher order shape functions for the element edges and interiors (Demkowicz 2006). However, their support covers the same fragments of the mesh as in case of the second order shape functions.

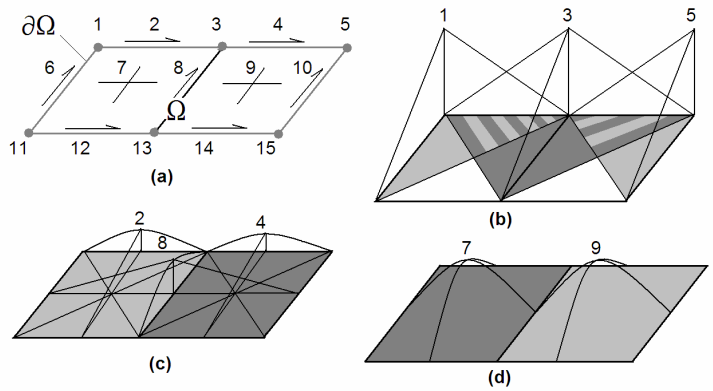


Fig.2.8. **(a)** Two finite element mesh. **(b)** Three examples of the first order shape function assigned to element vertices. **(c)** Three examples of the second order shape functions assigned to element edges. **(c)** Two examples of the second order shape functions assigned to element interiors

The solution of the system of equations (2.28) consists of two steps:

1) to generate a system of equations by computing stiffness matrix and load vector contributions,
2) to solve the generated system of equations by executing the forward elimination and the backward substitution.

For the finite elements with the polynomial orders of approximation equal to $(p_h-1)(p_v-1)$ in the element interior, and the polynomial orders of approximation equal to p_h and p_v in the element horizontal and vertical edges, respectively, there are $\text{nrdof}=(p_h+1)(p_v+1)$ unknowns, so there are nrdof^2 matrix integral contributions. Each

contribution requires $(p_h+1)(p_v+1)$ Gaussian quadrature integration points. It implies the following integration algorithm:

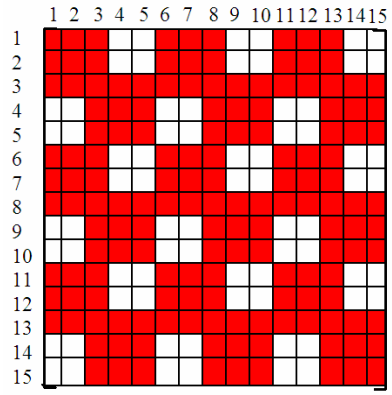


Fig.2.9. Global stiffness matrix resulting from the integration on two finite element mesh from Figure 2.8

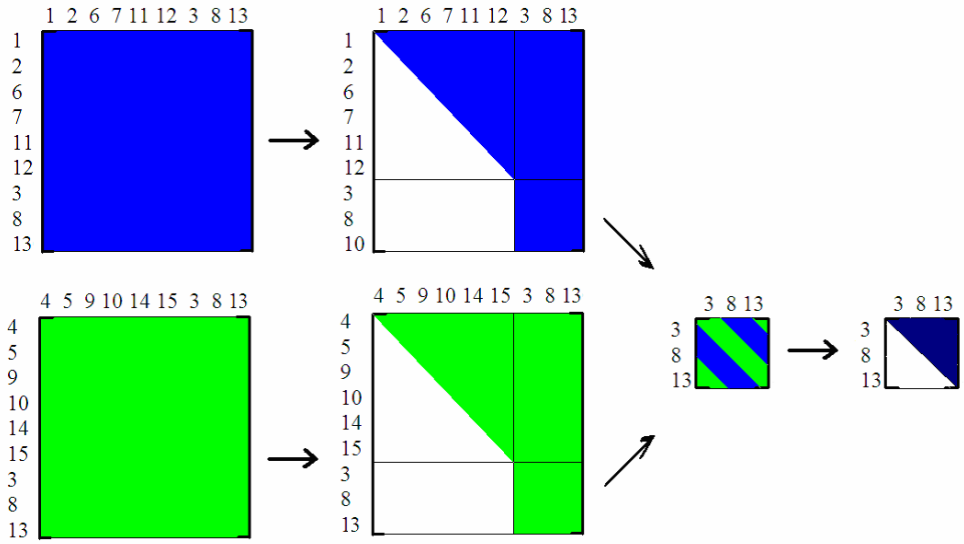


Fig.2.10. Two element local matrices resulting from the integration algorithm executed on the first and the second element independently. Elimination of fully assembled degrees of freedom related to element interior and boundary edges. Merging interface (common edge) problem contributions, followed by full elimination of the interface degrees of freedom.

```
for i=1, p_h+1
  for j=1, p_v+1
    for m=1,nrdof
      for n=1,nrdof
        aggregate a(m,n) into element stiffness matrix
      end
      aggregate b(m) into element load vector
    end
  end
end
```

Algorithm 2. Sequential integration for a single finite element

Note that the computational cost of this algorithm is $O\left((p+1)^6\right)$ in case of 2D meshes and $O\left((p+1)^9\right)$ in case of the 3D meshes (Paszyński 2007b). Thus, for the high polynomial orders of approximation $\left(p_h = p_v = 9\right)$ on a single finite element, the computational cost increases to 10^6 in 2D or 10^9 in 3D.

The stiffness matrix for the two element mesh from Figure 2.8 is illustrated in Figure 2.9. The row and column indices refer to the shape functions. Each time the support of *i*-th row shape function and *j*-th column shape function has a common part, (*i, j*) matrix entry is non-zero. These non-zero entries are denoted by dark grey colour. The straightforward elimination of the whole matrix involves 15^3 operations.

But let us consider an alternative approach presented in Figure 2.10.

We create two separate matrices, the first and the second element matrices, with such an ordering that the degrees of freedom assigned to common edge are located at the end. Now, all matrix contributions are fully assembled, except the common edge contributions. The partial forward elimination can be executed on each matrix, and stopped before processing these degrees of freedom assigned to common edge which are not fully assembled.

This procedure can be expressed in the following way:

$$\begin{bmatrix} \mathbf{A}_1 & \mathbf{B}_1 \\ \mathbf{C}_1 & \mathbf{A}_1^s \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \mathbf{x}_1^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_1 \\ \mathbf{b}_1^s \end{Bmatrix} \rightarrow \begin{bmatrix} \mathbf{U}_1 & \mathbf{B}_1^* \\ \mathbf{0} & \mathbf{A}_1^{s\,*} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \mathbf{x}_1^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_1^* \\ \mathbf{b}_1^{s\,*} \end{Bmatrix} \tag{2.29}$$

$$\begin{bmatrix} \mathbf{A}_2 & \mathbf{B}_2 \\ \mathbf{C}_2 & \mathbf{A}_2^s \end{bmatrix} \begin{Bmatrix} \mathbf{x}_2 \\ \mathbf{x}_2^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_2 \\ \mathbf{b}_2^s \end{Bmatrix} \rightarrow \begin{bmatrix} \mathbf{U}_2 & \mathbf{B}_2^* \\ \mathbf{0} & \mathbf{A}_2^{s\,*} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_2 \\ \mathbf{x}_2^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_2^* \\ \mathbf{b}_2^{s\,*} \end{Bmatrix} \tag{2.30}$$

Here, $\mathbf{A}_i$ stands for this part of element local matrices which is related to the interactions of the element interior and boundary edges shape functions, $\mathbf{B}_i$ and $\mathbf{C}_i$ stand for these parts of element local matrices which are related to the interactions between the element interior shape functions and the common edge shape function, and $\mathbf{A}_i^s$ stands for this part of element local matrices which is related to the interactions between the common edge shape functions. Moreover, $\mathbf{x}_i$ and $\mathbf{b}_i$ stand for the degrees of freedom and the right-hand-side terms related to the element interior and boundary edges, while $\mathbf{x}_i^s$ and $\mathbf{b}_i^s$ stand for the degrees of freedom and the right-hand-side terms related to the common edge.

The partial forward eliminations executed on both systems result in $\mathbf{A}_i^{s^*}$ two contributions to the interface problem of the common edge. The entire procedure is called the *Schur complement* of the internal degrees of freedom with respect to the interface degrees of freedom.

$$\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}} \tag{2.31}$$

$$\hat{\mathbf{A}} = \mathbf{P}_1\mathbf{A}_1^{s^*}\mathbf{P}_1^T + \mathbf{P}_2\mathbf{A}_2^{s^*}\mathbf{P}_2^T \tag{2.32}$$

$$\hat{\mathbf{b}} = \mathbf{P}_1\mathbf{b}_1^{s^*}\mathbf{P}_1^T + \mathbf{P}_2\mathbf{b}_2^{s^*}\mathbf{P}_2^T \tag{2.33}$$

Here, $\mathbf{P}_i$ stand for the permutation matrices transforming an element local ordering of the degrees of freedom located on the interface (the common edge in this example) into the global ordering on the interface.

$$\mathbf{P}_i : \mathbf{M}\left(n^i_{\text{interface}} \times n^i_{\text{interface}}\right) \rightarrow \mathbf{M}\left(n^i_{\text{interface}} \times n^i_{\text{interface}}\right) \tag{2.34}$$

In other words, both sub-matrices are merged, as presented in Figure 2.10, and we obtain a fully assembled matrix assigned to the common edge. The interface problem – the common edge problem (2.31) – has been solved now, and we obtain the interface problem solution $\hat{\mathbf{x}}$. Finally, the global problem can be solved by executing partial backward substitutions, based on the solutions propagating from the previous level systems. It may be achieved by the replacement of the Schur complement contributions $\mathbf{A}_i^{s^*}$ in (2.29) and (2.30) by the identity matrices, replacing the right-hand-side parts $\mathbf{b}_i^s$ by the obtained solution (remapped into the element local ordering of the interface degrees of freedom), and executing backward substitution on both systems.

$$\begin{bmatrix} \mathbf{U}_1 & \mathbf{B}_1^* \\ \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \mathbf{x}_1^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_1^* \\ \mathbf{P}_1^{-T}\hat{\mathbf{x}}\,\mathbf{P}_1^{-1} \end{Bmatrix} \tag{2.35}$$

$$\begin{bmatrix} \mathbf{U}_2 & \mathbf{B}_2^* \\ \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_2 \\ \mathbf{x}_2^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_2^* \\ \mathbf{P}_1^{-T}\hat{\mathbf{x}}\,\mathbf{P}_1^{-1} \end{Bmatrix} \tag{2.36}$$

The computational cost of this alternative approach involves two partial forward eliminations with $6 \cdot 9^2$ operations and one full forward elimination with 3^3 operations. This approach is over three times faster than straightforward elimination of the whole matrix, in case of the two finite element mesh.

In the final part of the section, the solver algorithm is expressed by the graph transformations working on the graph representation of the computational mesh. Let us focus on the example of the initial mesh with two finite elements represented by the graph presented in Figure 2.6.

There are three subsets of the graph transformations modeling the direct solver:

1) graph transformations executing the aggregation of the degrees of freedom into multiple front matrices,
2) graph transformations executing the partial forward eliminations,
3) graph transformations executing the partial backward substitutions.

The process starts with the aggregation of the active elements interior nodes into two so-called front matrices. This is expressed by the graph transformation presented in Figure

2.11. The α symbol denotes the aggregation process and the *i* symbol denotes the global index of the front matrix. Actually, the $\alpha^i \mathbf{I}$ symbol is meant to create a new front matrix and to aggregate the degrees of freedom of the element interior to the front matrix.

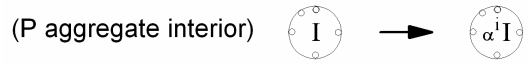


Fig.2.11. Graph transformation enforcing the aggregation of degrees of freedom of element interior

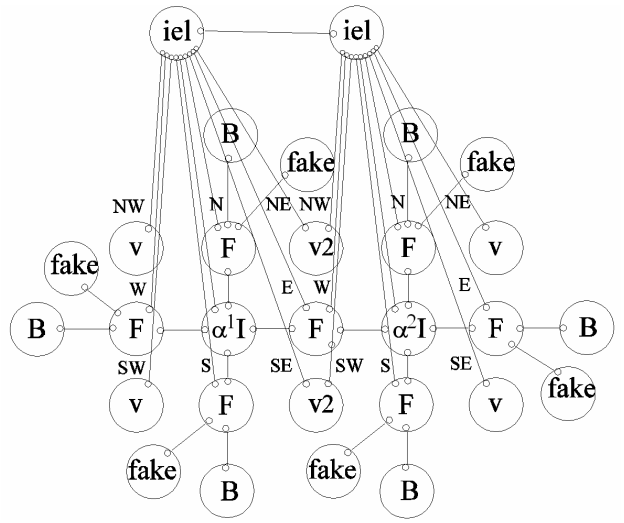


Fig.2.12. Graph representation of the mesh after execution of **(P aggregate interior)^2**

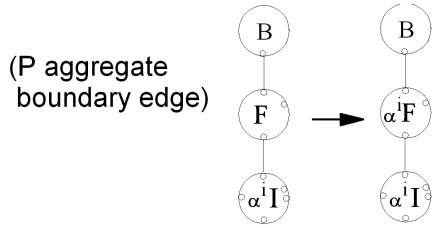


Fig.2.13. Graph transformation aggregating edges located on the boundary

The graph representation of the mesh after the execution of **(P aggregate interior)** is presented in Figure 2.12. Two front matrices related to the two initial mesh elements have been created. The next step is to apply the graph transformations aggregating the element edges and the corner vertices, which are located on the boundary of the domain. These transformations are presented in Figure 2.13 and 2.14. The graph representation of the

mesh, after these transformation have been executed, is presented in Figure 2.15. The final step is to aggregate the common edge contributions to both local front matrices. This is expressed by graph transformations presented in Figures 2.16 and 2.17. First, the common edge contributions, and then the common vertices are aggregated. The $\alpha^{i,j}$ symbol expresses the aggregation of two contributions, related to the common edge or vertex, to two frontal matrices assigned to both adjacent elements. The resulting graph representation of the mesh is presented in Figure 2.18.

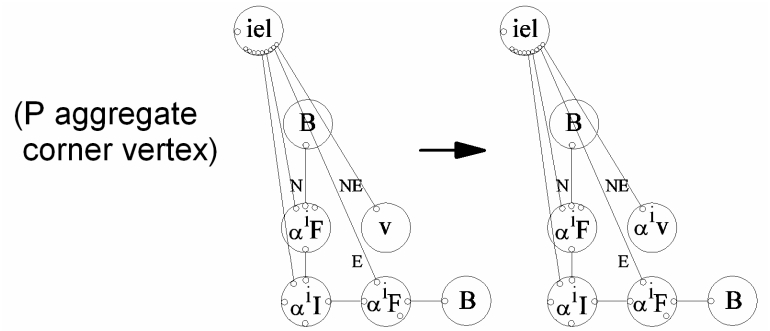


Fig.2.14. Graph transformation aggregating corner vertices located on the boundary. The transformation is cloned for 4 directions representing **NE, NW, SE, SW** locations of graph vertices

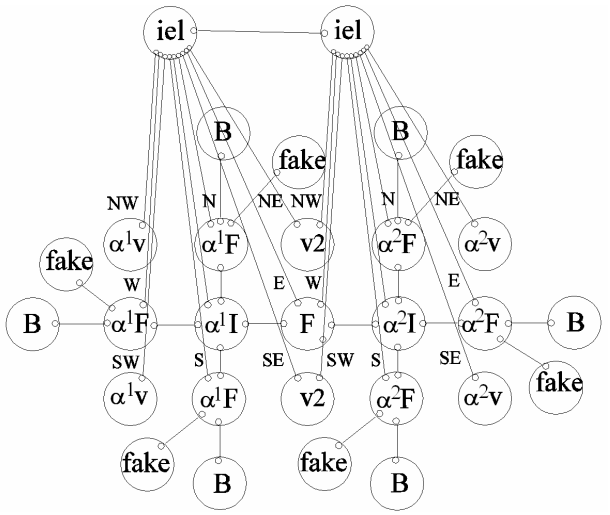


Fig.2.15. Graph representation of the mesh after execution of – **(P aggregate interior)^2 – (P aggregate boundary edge)^6 – (P aggregate corner vertex)^4**

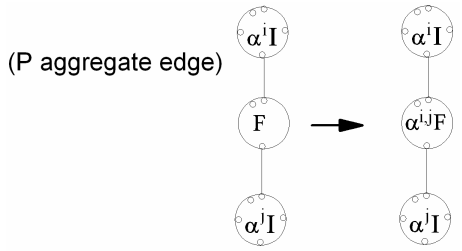


Fig.2.16. Graph transformation aggregating degrees of freedom of the common edge

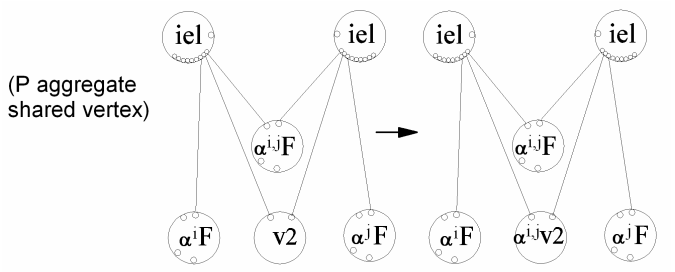


Fig.2.17. Graph transformation aggregating degrees of freedom of the common vertices

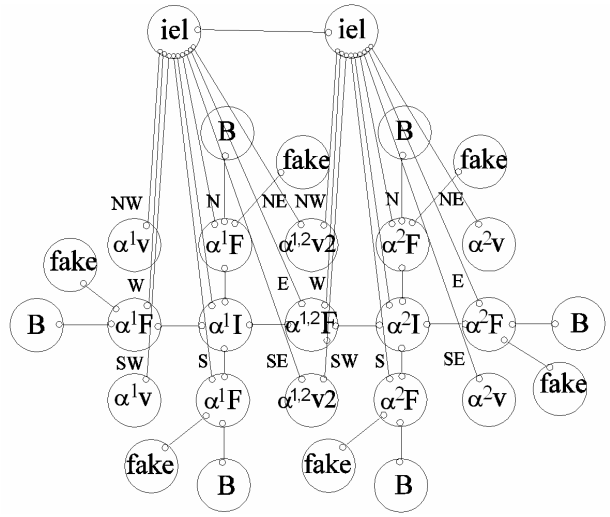


Fig.2.18. Graph representation of the mesh after execution of – **(P aggregate interior)^2 – (P aggregate boundary edge)^6 – (P aggregate corner vertex)^4 – (P aggregate edge) – (P aggregate common vertex)**

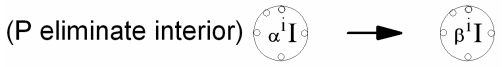


Fig.2.19. Graph transformation executing the elimination of internal degrees of freedom from *i*-th frontal matrix

At this point, the aggregation process is finished. The two local frontal matrices correspond to those presented in (2.2-2.3) before the elimination process starts. The following set of graph transformations executes the elimination process.

The process of elimination follows the order identical to the process of aggregation of degrees of freedom. First of all, the internal degrees of freedom are eliminated from both frontal matrices, which is expressed by the graph transformation presented in Figure 2.19. The process of elimination of degrees of freedom, related to the element edges and corner vertices located on the boundary of the domain, is executed next. This is expressed by the graph transformations presented in Figures 2.20 and 2.21.

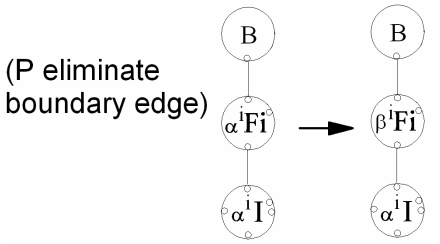


Fig.2.20. Graph transformation executing the elimination of degrees of freedom related to boundary edges

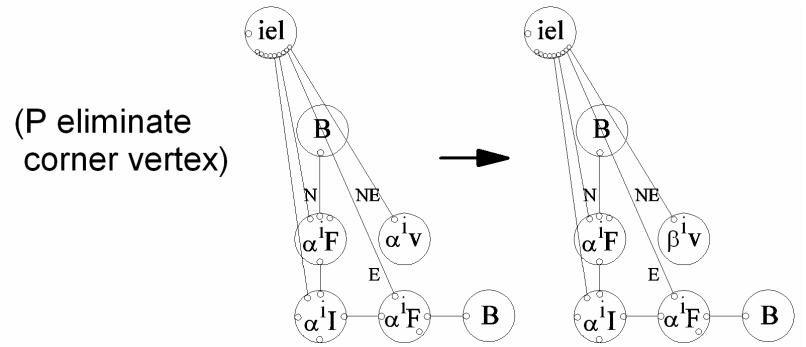


Fig.2.21. Graph transformation executing the elimination of degrees of freedom related to corner vertices

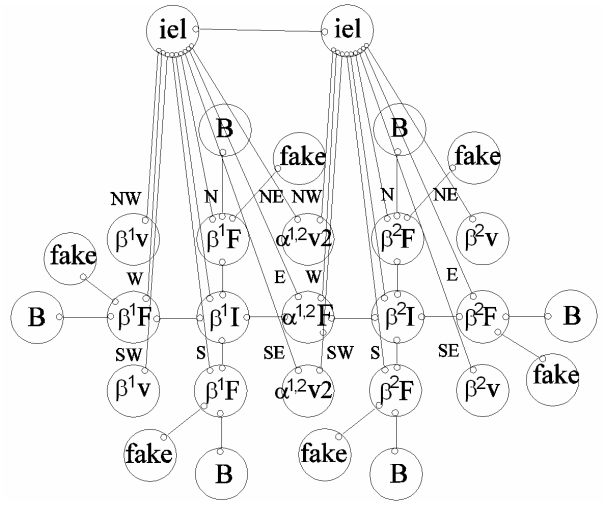


Fig.2.22. Graph representation of the mesh after the elimination of degrees of freedom related to element interiors as well as to boundary edges and vertices

The graph representation of the mesh after the elimination of degrees of freedom, related to the element interior and boundary edges, is presented in Figure 2.22. The degrees of freedom which have been already eliminated are denoted by the β symbol.

At this point, the two local matrices correspond to the equations (2.29)-(2.30) presenting the system after the elimination. Finally, the graph transformations responsible for the forward elimination of the common interface are applied. These transformations actually merge the two contributions, and implement the elimination process. The transformations are presented in Figures 2.23 and 2.24.

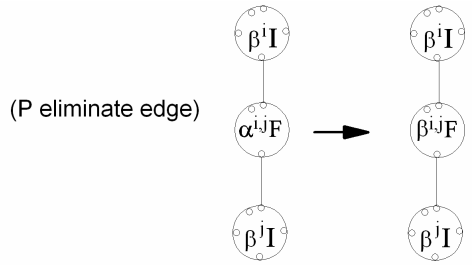


Fig.2.23. Graph transformation executing the elimination of degrees of freedom related to common edges

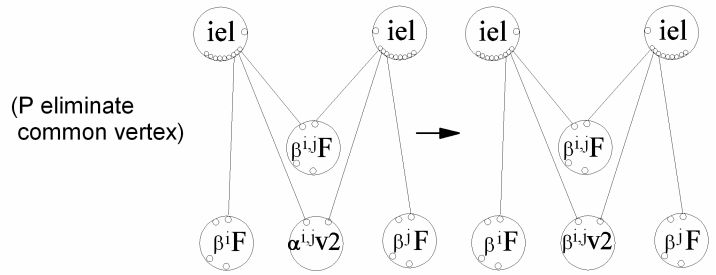


Fig.2.24. Graph transformation executing the elimination of degrees of freedom related to common vertices

The process of the solution of the coarse mesh problem can be expressed by the control diagram presented in Figure 2.25. The first part of the diagram refers to the aggregation process while the second part of the diagram refers to the elimination. The process of the coarse mesh solver has been expressed as a sequence of atomic tasks, defined as the execution of graph grammar productions on the graph representation of the mesh.

The first atomic task, represented by the production (**P aggregate interior)**, generates a part of an element local matrix related to degrees of freedom of an element interior. The finite element is identified by the graph vertex denoted by **I** symbol on the left-hand side of the production. Usually, there are many elements, so there are many sub-graphs representing different elements interiors, thus the atomic task can be executed in the concurrent way. In fact, these atomic tasks create several element matrices and number them by index *i* assigned to the right-hand side $\alpha^i I$ of the production.

The following atomic tasks are executed after all interiors have been aggregated (after all **(P aggregate interior)** atomic tasks - productions have been executed). The following atomic tasks concern the production **(P aggregate corner vertex)** which aggregates the degrees of freedom related to element vertices located on the corners of the domain and the production **(P aggregate boundary edge)** which aggregates the degrees of freedom assigned to the edges located on the boundary. It is obvious that there are many such boundary vertices and edges, and all these aggregations can be executed in the concurrent way. Moreover, the executions can be mixed, which is denoted in Figure 2.25, by adding the synchronization points before and after the execution of both atomic tasks. Note that the element matrices are identified on the basis of index read from element interior node from the left-hand side of the production.

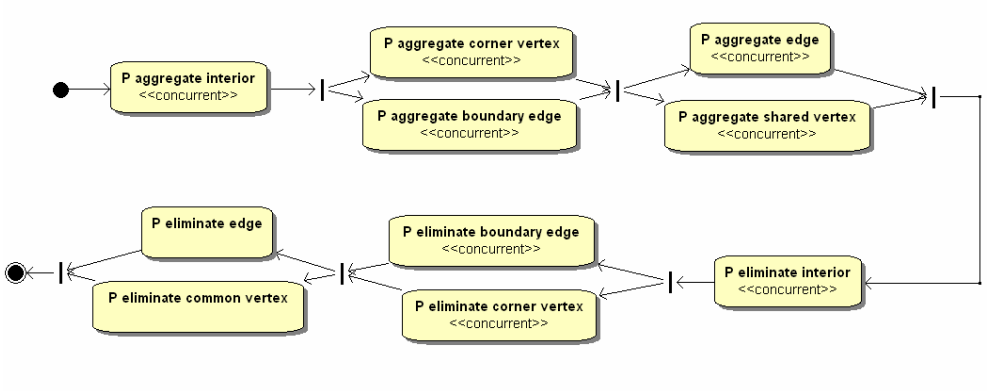


Fig. 2.25. Control diagram for coarse mesh solver algorithm on the level of atomic tasks

The following atomic tasks are executed after corner vertices and all boundary edges have been aggregated. Note that boundary vertices which are not located on the corners of the domain are still not aggregated. The next atomic tasks concern the productions **(P aggregate edge)** and **(P aggregate shared vertex)**, responsible for the aggregation of edges and vertices shared between some adjacent elements. These atomic tasks can be again executed in the concurrent way, and they can be mixed. The element matrices are also identified on the basis of the index read from element interior node from the left-hand side of the production.

The aggregation process is now completed. It is followed by partial forward eliminations. The first atomic task refers to the production **(P eliminate interior)** and is responsible for the elimination of the element interior degrees of freedom. The matrix on which the elimination is executed is identified by the *i* index stored as attribute $\alpha^i I$ at the interior node vertex on the graph representation of the mesh. Note that these eliminations can be performed in the concurrent way, since they are executed on separate matrices.

The following atomic tasks are executed after all element interiors have been eliminated. The next atomic tasks concern the production **(P eliminate boundary edge)**, eliminating the degrees of freedom assigned to the edges located on the boundary, and the production **(P eliminate corner vertex)**, eliminating the degrees of freedom assigned to vertices located at the corners of the domain. The atomic tasks can be executed in the concurrent way and they can be mixed (the elimination of boundary edges can be executed at the same time as the elimination of corner vertices). Again, the element matrix is identified by indices read from graph vertices.

The following atomic tasks are executed after all boundary edges and corner vertices have been eliminated. The next atomic tasks refer to productions **(P eliminate edge)**, eliminating the degrees of freedom related to the shared edges and productions **(P eliminate common vertex)**, eliminating the degrees of freedom related to the shared vertices. This time the elimination is executed on two matrices, identified by graph nodes attributes. As a result of these two eliminations, we obtain two Schur complements, which are merged. These atomic tasks can be executed in the concurrent way, providing that the sub-graphs from the left-hand side of the production do not overlap. Note that these atomic tasks can be also mixed.

After the partial forward eliminations, the backward substitutions are executed. They follow the same pattern as the aggregation and the elimination. Finally, all α and β symbols are removed from the mesh.

This algorithm can be generalized in such a way that it may be applied to more complex meshes. This will be discussed further in this chapter.

We assume that the coarse mesh solution is stored at the graph vertices. The solution represents the shape function coefficients. It is stored in the form of graph attributes u^{i}_{hp} distributed among the graph vertices. The number of shape functions at the graph vertices representing the finite element vertices is equal to 1. The number of shape functions at the graph vertices representing the element edges is equal to p-1, where p denotes the polynomial order of approximation at the edge. Finally, the number of shape functions at the graph vertices representing the element interiors is equal to (p_h-1)(p_v-1), where (p_h , p_v) denote the polynomial orders of approximation in the horizontal and vertical directions.

2.1.3. Algorithm of global *hp* refinement

In the next step of the self-adaptive *hp*-FEM algorithm, the so-called *fine* mesh is generated from the copy of the coarse mesh. Thus, the coarse mesh is duplicated, and the so-called global *hp* refinement is executed on the copy of the coarse mesh. The global *hp* refinement involves the following two steps:

1) application of the global *h* refinement,
2) application of the global *p* refinement.

The global *h* refinement employs the execution of the *h* refinement for each element of the copied coarse mesh. The *h* refinement is expressed by breaking an element's interior and edges.

To break an element's interior means to generate a new son vertex, four new edges and four new interiors. From the point of view of the graph representation of the mesh, the breaking of an element's interior is expressed by the creation of a new graph vertex denoting the newly created son vertex, four new graph vertices denoting newly created edges, and four new graph vertices denoting newly created interiors. These newly created graph vertices are connected to the broken interior. The locations of a finite element vertices and edges are identified by **N**, **S**, **W**, **E**, **NW**, **NE**, **SW** and **SE** labels assigned to the graph edges.

To break an element edge means to generate a new vertex and two new edges. From the point of view of the graph representation of the mesh, the breaking of an element edge is expressed by the creation of a new graph vertex denoting the newly created son vertex and two new graph vertices denoting the newly created edges.

These procedures are expressed by **(P break interior)** and **(P break edge)** graph transformations presented in Figures 2.26 and 2.27.

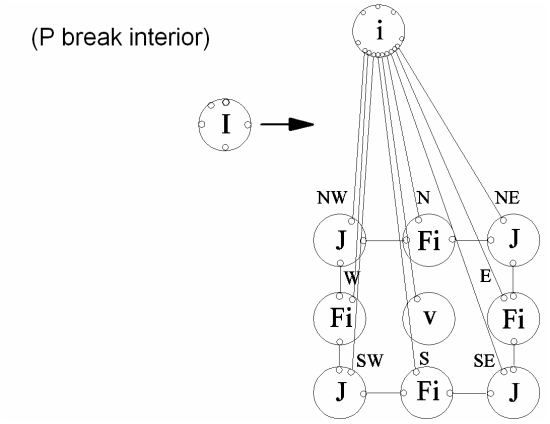


Fig.2.26. Graph transformation for breaking an element interior

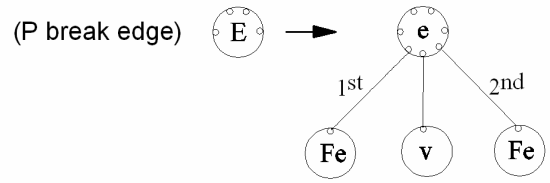


Fig.2.27. Graph transformation for breaking an element edge

The following two mesh regularity rules are enforced during the process of mesh transformation:

1) the 1-irregularity rule „*a finite element can be broken only once without breaking the adjacent large elements*",
2) the minimum rule "*the polynomial order of approximation of an element edge must be equal to the minimum of corresponding orders of approximation from the element interiors*".

The 1-irregularity rule enforces breaking the unbroken large adjacent elements before breaking a small element for the second time, which has been illustrated in Figure 1.7. The purpose of this mesh regularity rule is to avoid multiple constrained edges, which leads to the problems with approximation on such edges. In fact, breaking of an element consists in two steps – to break element interior and to break element edges. The 1-irregularity rule can be reformulated in the following way, to express this strategy of two steps.

An element edge can be broken only if two adjacent interiors have been already broken, or the edge is adjacent to the boundary. An element interior can be broken only if all adjacent interiors are of the same size as this interior, or smaller.

This is expressed by the graph transformations **(PFE1-PFE4)** presented in Figure 2.28, and the production **(PJI)** presented in Figure 2.29.

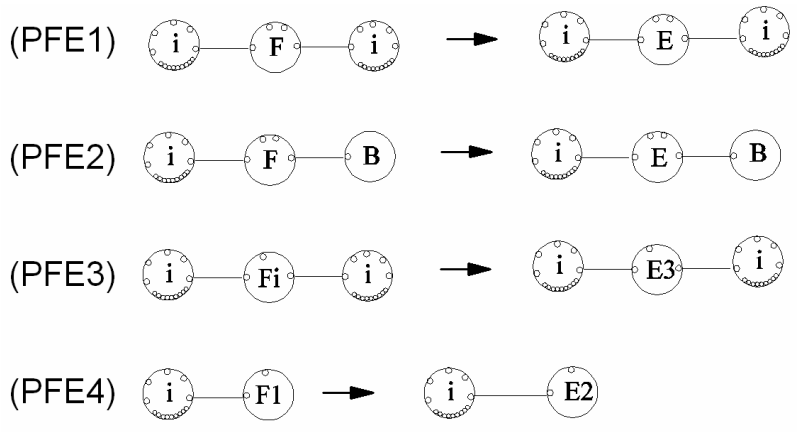


Fig.2.28. Graph transformations allowing for breaking an element edge

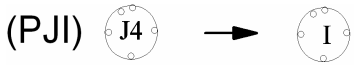


Fig.2.29. Graph transformation allowing for breaking an element interior

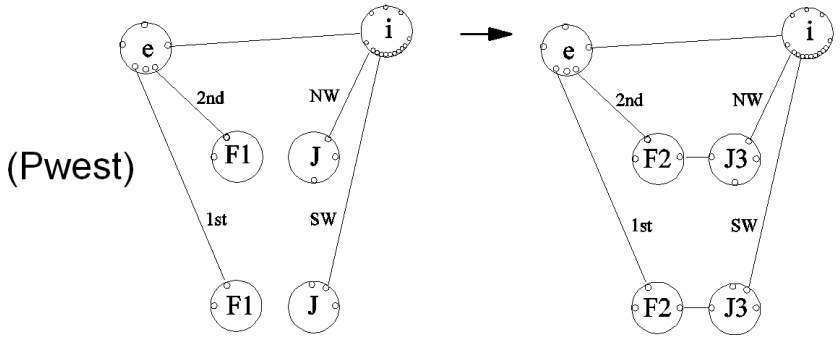


Fig.2.30. Graph transformation for reconstructing connectivities between edges and interiors in the western direction. The transformation is cloned for **F1** replaced by **Fe**, **F2** by **F1** and **e** replaced by **e2** or **e3**

The 1-irregularity rule is enforced on the level of graph grammar syntax. The interiors which can be broken without violating the mesh regularity rule are denoted by the capital **I** symbol. There is a graph transformation **(P break interior)** which allows for breaking these interiors. The newly created interiors are denoted by the capital **J** symbol, however there is no graph grammar transformation which allows for breaking **J** interiors. Thus, the adjacent large elements must be broken first. Then, the element edges surrounded by already broken interiors must be broken, too. There are several graph transformations **(PFE1-PFE4)** which make it possible to break those element edges which are either

surrounded by two broken interiors or are surrounded by one broken interior and are adjacent to the boundary. These transformations change the edge symbol from **F** to **E** (or to **F2** or **E2**). These edges can be broken now using the graph transformation **(P break edge)**.

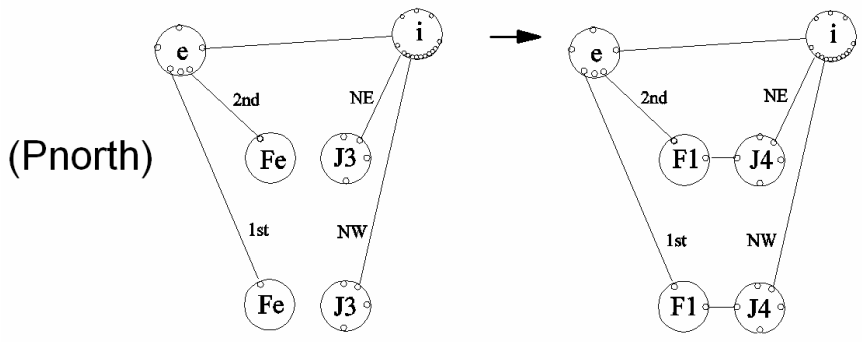


Fig.2.31. Graph transformation for reconstructing connectivities between edges and interiors in the northern direction. The transformation is cloned for **e** replaced by **e2** or **e3**

The following set of graph transformations **(Peast)**, **(Pwest)**, **(Pnorth)**, **(Psouth)**, presented in Figures 2.30-2.33, reconstructs the connectivities between edges and interiors. The number of edges which surround an element interior is coded in the symbol of each interior. Here the **J** symbol stands for an interior with no adjacent edges of the same size, **J2**, **J3** and **J4** stand for an interior with, respectively, two, three or four adjacent edges of the same size. An element interior can be broken only if it is surrounded by four edges of the same size. The graph transformation **(PJI)** changes the **J4** symbol to the **I** symbol, allowing for the application of graph transformation **(Pbreak interior)**.

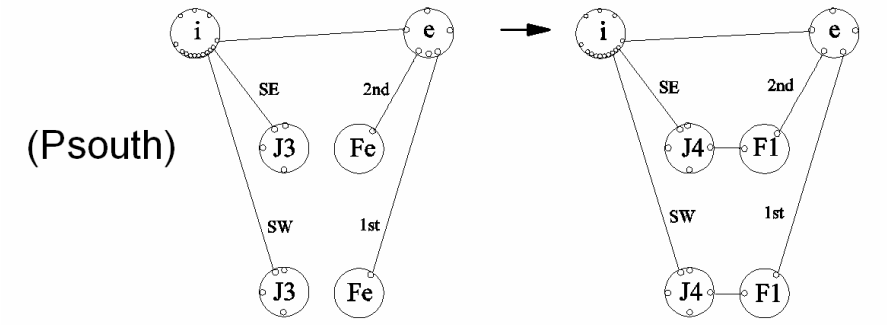


Fig.2.32. Graph transformation for reconstructing connectivities between edges and interiors in the southern direction. The transformation is cloned for **e** replaced by **e2** or **e3**

Let us conclude this section with an exemplary execution of the global *h* refinement on the graph representation of the two finite element mesh presented in Figure 2.34. The corresponding graph representation of the mesh is presented in Figure 2.6.

First, it is possible to break both element interiors, since they are surrounded by the edges of the same size. This is illustrated in Figure 2.35. Second, it is possible to break all the element edges, since they are either surrounded by two broken interiors, or by a broken interior and the boundary. This is illustrated in Figure 2.36.

Next, the adjacency data are propagated from the parent graph vertices to the son graph vertices. We execute all **(Pwest)** transformations, all **(Peast)** transformations, then all **(Psouth)** transformations and **(Pnorth)** transformations. At this point, all the interiors are denoted by **J4** symbols. The interior symbols can be updated to the **I** symbols, to allow for future refinements. The entire procedure is illustrated in Figure 2.37. The resulting *fine* mesh corresponds to the following sequence of graph transformations **(P break interior)^2-(P break edge)^7-(Pwest)^2-(Peast)^2-(Pnorth)^2-(Psouth)^2-(PJI)^8** executed on the coarse mesh. This is presented in Figure 2.38.

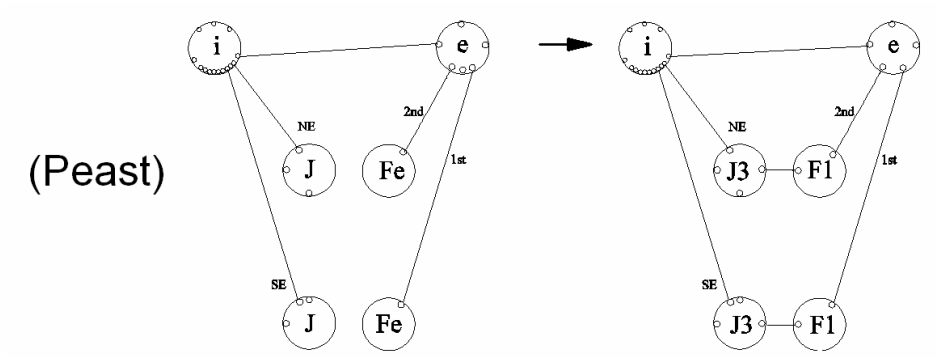


Fig.2.33. Graph transformation for reconstructing connectivities between edges and interiors in the eastern direction. The transformation is cloned for **e** replaced by **e2** or **e3**

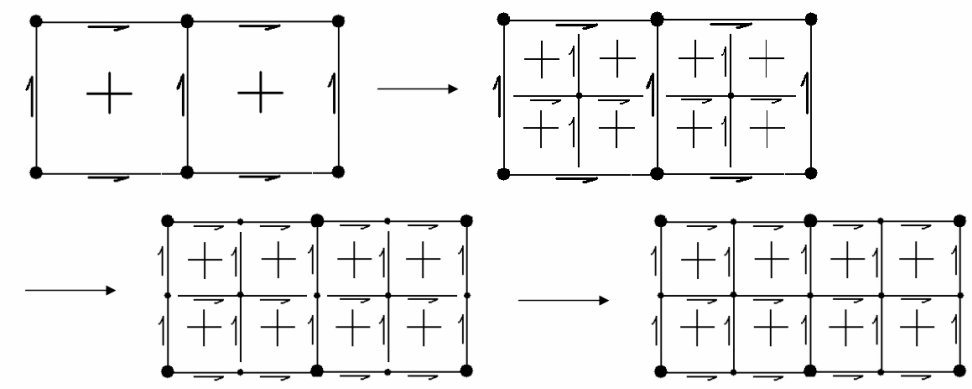


Fig.2.34. Execution of the global *h* refinement on the two finite elements mesh

The process of *h* refinement can be now described in terms of the atomic tasks. The order of execution of the atomic tasks for the *h* refinement is determined by the control diagram, presented in Figure 2.39.

The execution starts from the atomic tasks which refer to productions **(PJI)** and **(P break interior)**, with changing a label of graph vertex representing an element interior node from **J** to **I** (since only vertices denoted by **I** symbol can be broken), and with breaking element interior. All these atomic tasks can be executed in the concurrent way.

The following atomic tasks are executed after all the required interiors have been broken. The atomic tasks refer to productions **(PFE1-4)**, changing labels of graph vertices representing an element edge node from **F** to **E** (since only vertices denoted by **E** symbol can be broken). All these atomic tasks can be executed in the concurrent way.

After all symbols of graph vertices representing element edges have been updated, the following atomic tasks related to production **(P break edge)** can be executed. The atomic tasks break an element edge and can be executed in the concurrent way, since all edges are independent.

The remaining atomic tasks are responsible for updating the adjacency data in the eastern, western, northern and southern directions. All these atomic tasks can be executed in the concurrent way, however they cannot be mixed. In other words, first all atomic tasks related to **(Peast)** productions are executed in the concurrent way, next all atomic tasks related to **(Pwest)** productions are executed in the concurrent way, then all atomic tasks related to **(Pnorth)** productions are executed in the concurrent way and finally all atomic tasks related to **(Psouth)** productions are executed, also in the concurrent way.

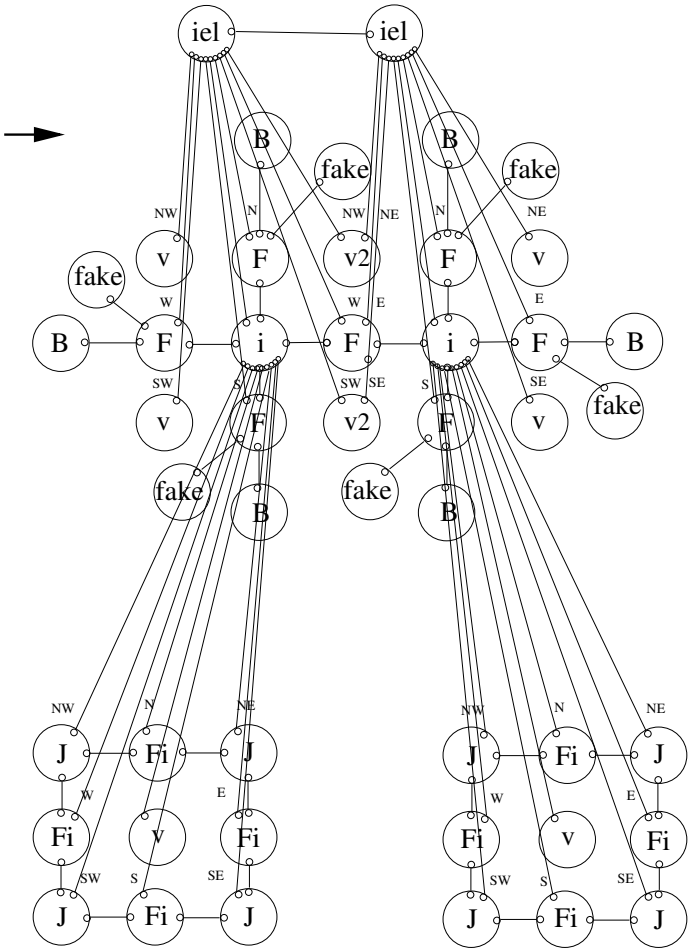


Fig.2.35. Breaking two element interiors by two executions of **(P break interior)**

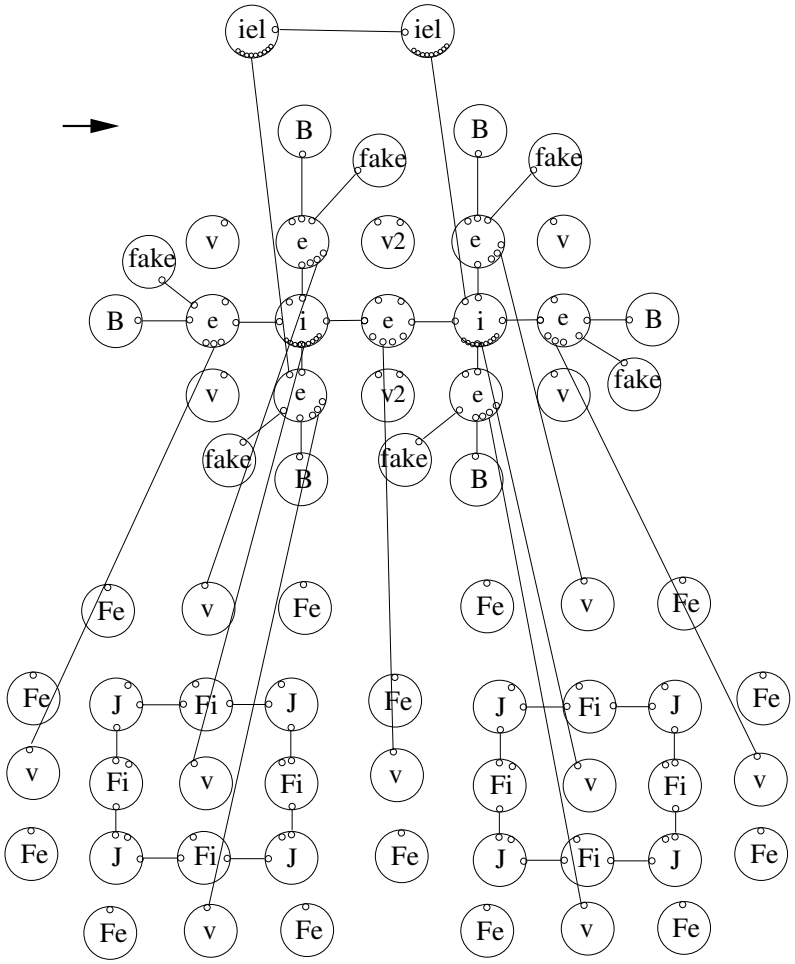


Fig.2.36. Breaking all edges by seven executions of **(P break edge)**. For simplicity, only single father-son links are drawn and edge attributes are not present

The global *h* refinement is followed by the global *p* refinement. This involves an increase in the polynomial orders of approximation for each element interior, followed by the enforcement of the minimum rule. The polynomial orders of approximation are stored as attributes of graph vertices (not presented in the above figures).

The attributes (p_h, p_v) which denote the polynomial orders of approximation assigned to the element interiors in the horizontal and vertical directions are simply increased by one to (p_h+1, p_v+1). This is expressed by the graph transformation **(P p-refinement)** presented in Figure 2.40.

The minimum rule, executed in the next step, determines the polynomial orders of approximation of the element edges. The order of an edge must be equal to the minimal order of element interiors adjacent to the edge. This can be expressed by graph transformations presented in Figures 2.41 – 2.44. The graph transformations must be

defined for graph vertices which represent the element edges located in the northern, southern, eastern and western directions, with respect to the graph vertex representing the element interior.

The process of *p* refinement can be also described in terms of the atomic tasks. The *p* refinement procedure has been summarized in the control diagram presented in Figure 2.45. The procedure consists in attributing element interiors by means of the required polynomial orders of approximation and in executing the minimum rule for edges. All these atomic tasks with associated graph grammar productions can be executed in the concurrent way, but they cannot be mixed.

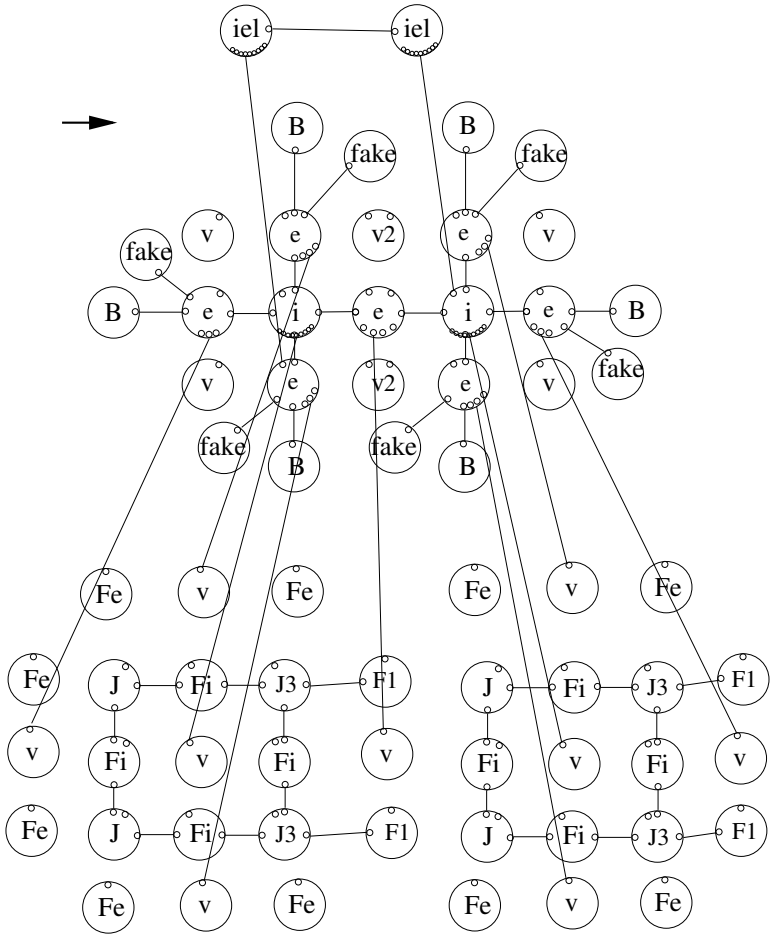


Fig.2.37. Setting connectivities in the western direction by two executions of **(Pwest)**

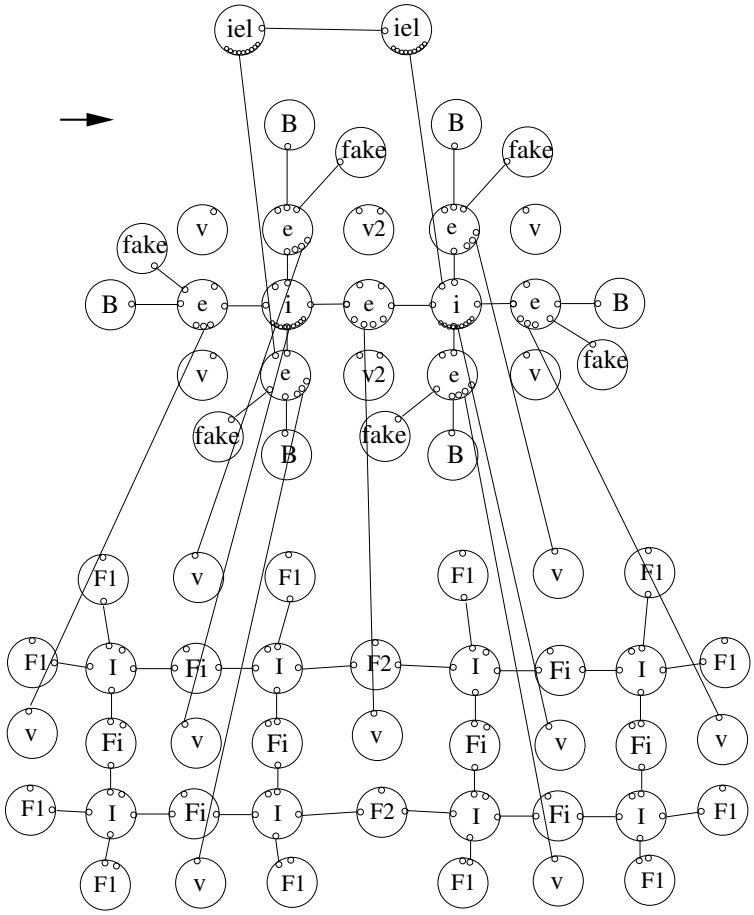


Fig.2.38. Fine mesh obtained from coarse mesh from Figure 2.8 after executing the following sequence of graph transformations **(P break interior)^2-(P break edge)^7-(Pwest)^2-(Peast)^2-(Pnorth)^2-(Psouth)^2-(PJI)^8**

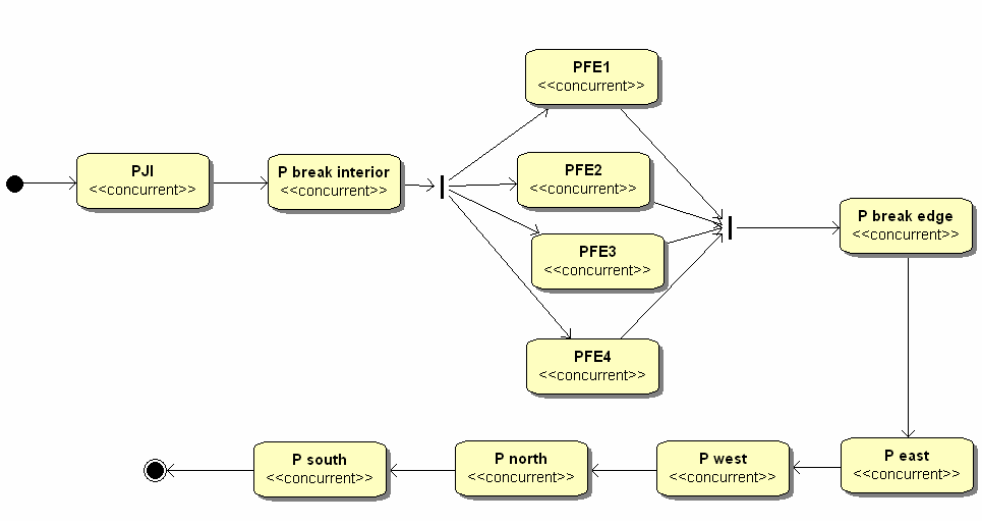


Fig.2.39. Control diagram for *h* refinement algorithm on the level of atomic tasks

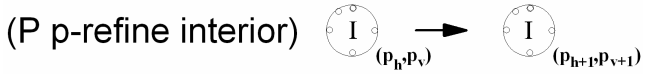


Fig.2.40. Graph transformation for performing local *p* refinement at graph vertex denoting an element interior

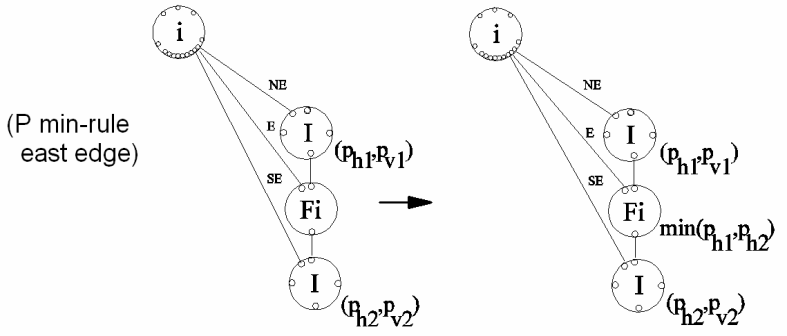


Fig.2.41. Graph transformation enforcing the minimum rule on the eastern edge

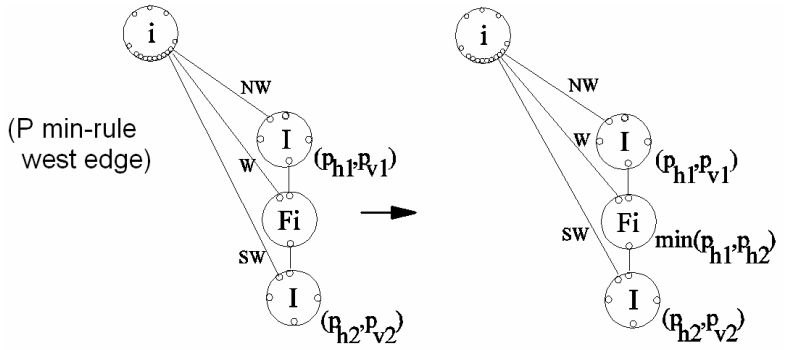


Fig.2.42. Graph transformation enforcing the minimum rule on the western edge

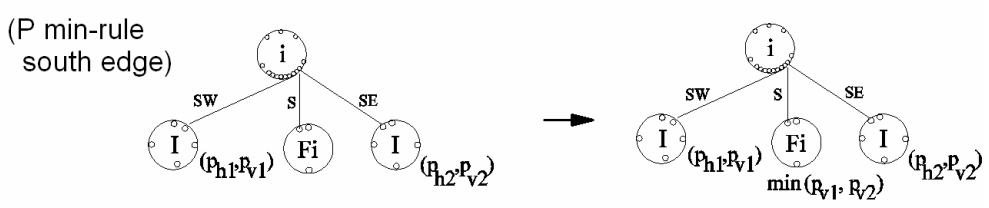


Fig.2.43. Graph transformation enforcing the minimum rule on the southern edge

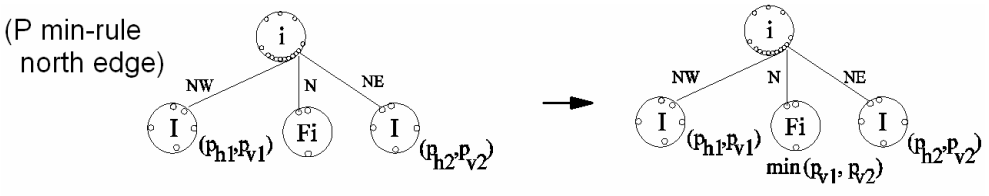


Fig.2.44. Graph transformation enforcing the minimum rule on the northern edge

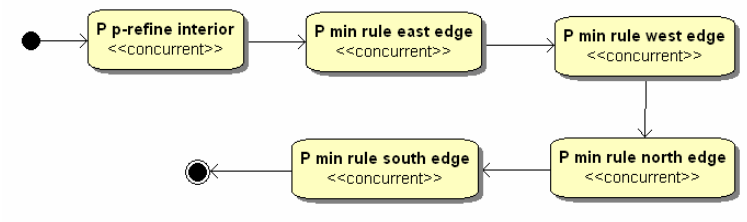


Fig.2.45. Control diagram for algorithm of *p* refinements
and enforcement of the minimum rule on the level of atomic tasks

2.1.4. Algorithm of solution of fine mesh problem

In the next step of the self-adaptive *hp*-FEM, we solve the FEM discretization of the weak form of the PDE (1.10-1.12) on the fine mesh.

The same algorithm of the direct solver which has been used in case of the coarse mesh is applied now to the fine mesh. However, this time the finite elements from the coarse mesh have been *h* refined (broken into new smaller son elements) during the global *hp* refinement.

The solver algorithm can be generalized to an arbitrary mesh. This can be achieved by considering the elimination tree constructed by the nested dissection algorithm (Khaira, Miller, Sheffler 1992) executed for the initial mesh. The simplest example of the initial mesh and its elimination tree is presented in Figure 2.46.

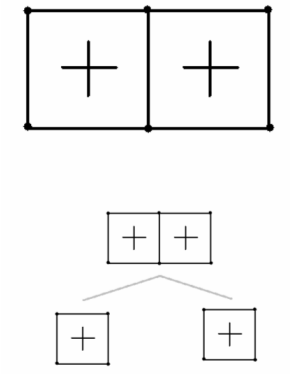


Fig.2.46. Elimination tree constructed for exemplary initial mesh with eight initial mesh elements

The elimination tree created for the initial mesh is updated when the mesh is refined (elimination tree follows the refinements executed on the mesh), see Figure 2.47.

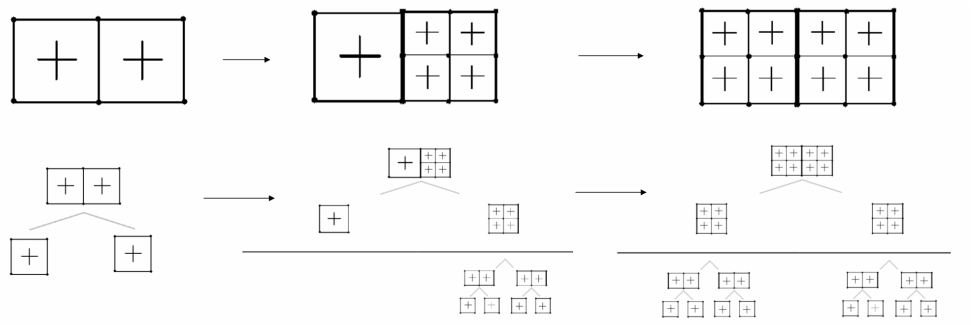


Fig.2.47. Elimination tree created for the initial mesh is updated when the mesh is refined (elimination tree is constructed dynamically, during mesh refinements)

The solver starts with the elimination of the most expensive interior degrees of freedom, as presented on panel (a) in Figure 2.48. The solver computes the local matrices

related to the active elements and eliminates the degrees of freedom related to an element interior and the boundary of the domain. The remaining degrees of freedom are related to the edges shared with adjacent finite elements.

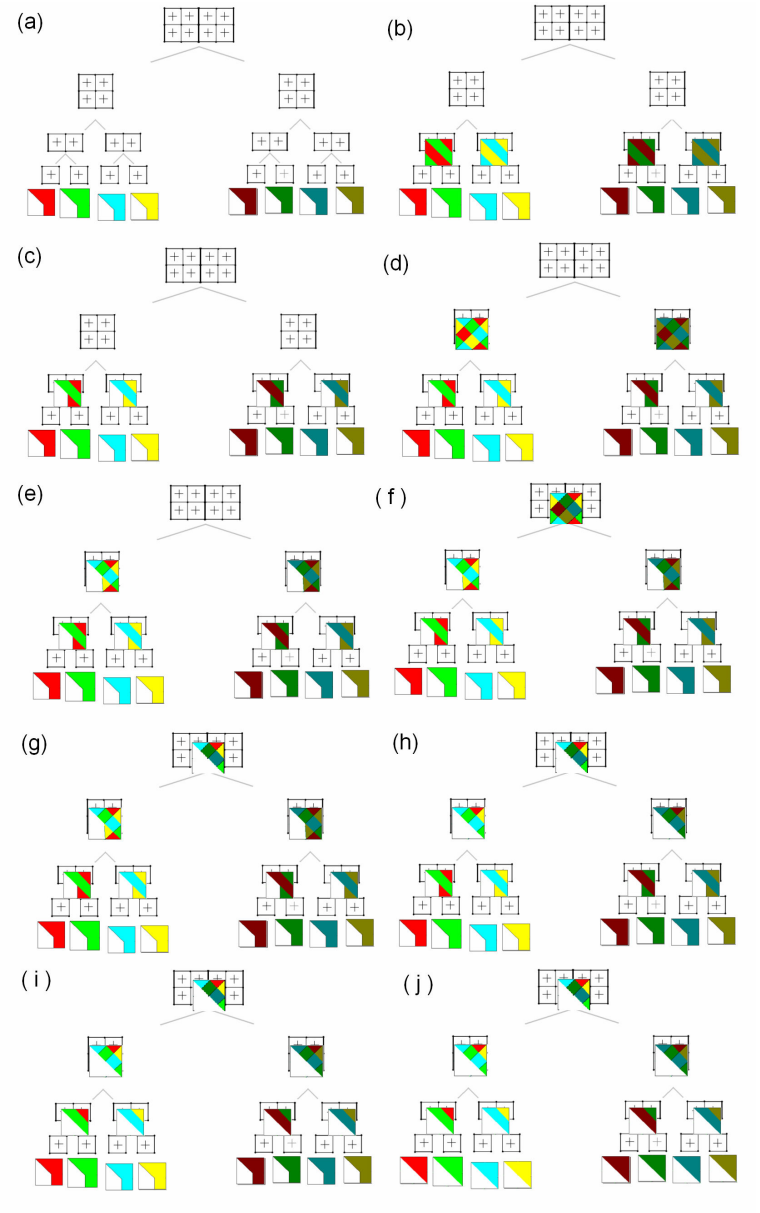


Fig.2.48. Execution of the solver on the eight finite element mesh

In the next step (presented on panels (b) and (c) in Figure 2.48) the solver sums up the contributions to the Schur complement and eliminates the degrees of freedom related to the common edge shared between two adjacent finite elements. The process is repeated recursively, as presented on panels (d) – (e), until we reach the root of the elimination tree. This is presented on panel (f) in Figure 2.48. The interface problem is solved, which is illustrated on panel (g) in Figure 2.48. The size of this interface problem is related to the cross-section of the domain, going through the element edges. This process is followed by the recursive backward substitutions presented on panels (h-j) in Figure 2.48. Thus, a proper ordering defined by the elimination tree is based on the knowledge of the structure of the initial mesh and on the history of mesh refinements. The solver algorithm can be expressed by the following recursive routine:

```
matrix function recursive_forward_elimination(tree_node)
if tree_node has no son nodes then
  eliminate leaf element stiffness matrix internal nodes
  return Schur complement sub-matrix
else if tree_node has son nodes then
  do for each tree_node_son
    son_matrix = recursive_forward_elimination(tree_node_son)
    merge son_matrix into new_matrix
  enddo
  decide which unknowns of new_matrix can be eliminated
  perform partial forward elimination on new_matrix
  store the local system at tree_node
  return Schur complement sub-matrix
endif
```

Algorithm 3. Sequential recursive algorithm for forward elimination

The forward elimination is followed by analogous recursive backward substitution:

```
function recursive_backward_substitution(tree_node,
                                         partial_solution)
retrieve the local system from tree_node
replace the Schur complement sub-matrix by identity matrix
substitute partial solution at interface degrees of freedom
execute backward substitution of the local system
if tree_node has son nodes then
  do for each tree_node_son
    retrieve the partial_solution related to interface
      variables at tree_node_son
    call recursive_forward_elimination(tree_node_son,
                                       partial_solution)
  enddo
endif
```

Algorithm 4. Sequential recursive algorithm for backward substitution

The generalized algorithm can be also expressed by the graph grammar transformations. The set of graph transformations, presented in Figures 2.11-2.25, must be extended to add new graph productions, defined for the labels of the new graph vertices. These new graph productions can be applied to a new recursive structure of the graph representation of the fine mesh.

Let us trace the process of the execution of the graph grammar-driven solver on the fine mesh example. The process starts with the creation of several matrices, called the *frontal-matrices*, each matrix for each element interior. It is followed by the aggregation of degrees of freedom for each element interior. This has been expressed by the production **(P aggregate interior),** presented in Figure 2.11. The graph representation of the fine mesh after the first step is presented in Figure 2.50. In this way we create eight local frontal matrices.

The process is followed by the aggregation of element edges located on the boundary, and by the aggregation of the corner vertices. This has been expressed by the graph transformations **(P aggregate boundary edge)** and **(P aggregate corner vertex)** presented in Figures 2.13 and 2.14. However, this time the context is different: the edges of active elements, located on the boundary, are denoted now by the **F1** symbol, and the corner vertices are located on the level of parent elements. Two new graph transformations are presented in Figures 2.49 and 2.51. The graph representation of the fine mesh after these transformations is presented in Figure 2.54.

The next step is to aggregate the common edges and vertices, which is expressed in the graph transformation presented in Figure 2.52. First, two pairs of elements in the horizontal directions are grouped together, and the common edge is aggregated to be eliminated later. Second, the two sets of elements are joined into a new set of four elements, and one common edge is aggregated to be eliminated later. In other words, the degrees of freedom of a shared edge must be aggregated to two frontal matrices $\mathbf{A}_i$ and $\mathbf{A}_j$ related to the elements which share an edge (this is denoted by $\alpha^{i,j}$ symbol), while the degrees of freedom of a shared vertex must be aggregated to four frontal matrices $\mathbf{A}_i$, $\mathbf{A}_j$, $\mathbf{A}_k$ and $\mathbf{A}_l$, of the four elements which share a vertex (this is denoted by $\alpha^{i,j,k,l}$ symbol). It is also necessary to process the shared vertices which are located on the boundary of the domain, on the level of son elements. This is expressed in the graph transformation presented in Figure 2.53. The graph representation of the fine mesh after these transformations is presented in Figure 2.55.

At this point, the only degrees of freedom which have not been aggregated are situated at the common interface, located on the vertical cross-section of the domain. In order to aggregate them, we apply the graph transformations presented in Figures 2.56 and 2.57. The resulting graph representation of the mesh is illustrated in Figure 2.60.

Now the aggregation process is finished. The degrees of freedom related to the element interiors have been aggregated to a single frontal matrix. The degrees of freedom related to the element edges have been aggregated to two frontal matrices. Finally, the degrees of freedom related to the element vertices have been aggregated to four frontal matrices.

The process of aggregation of degrees of freedom is followed by the process of partial forward eliminations. The elimination starts at the level of active elements, with the elimination of the degrees of freedom related to element interiors. This is expressed by the

graph transformation for the coarse mesh solver **(P eliminate interior)**, presented in Figure 2.19. The next step is to eliminate the degrees of freedom related to the boundary edges and corner vertices, which results in two new graph transformations **(P eliminate boundary edge)** and **(P eliminate vertex),** presented in Figures 2.58 and 2.59.

The graph representation of the mesh after these eliminations is illustrated in Figure 2.61. This corresponds to the panel (a) in Figure 2.48. In the following steps we merge into pairs the Schur complements related to the uneliminated edges and vertices of the active elements. The elimination is continued for fully assembled edges and vertices.

In a more detailed way, the next step is to join the elements into pairs in horizontal directions, which is expressed by the graph transformations presented in Figure 2.62 and 2.63. The application of these transformations corresponds to merging the Schur complements presented on panel (b) in Figure 2.48. The resulting graph representation of the mesh is presented in Figure 2.65.

The next step is to eliminate the fully assembled edges and vertices from the frontal matrices. This is presented on panel (c) in Figure 2.48 and reflected by the application of the graph transformation presented in Figure 2.64, followed by the application of the graph transformation **(P merge shared vertex)** for two external vertices.

The resulting graph representation of the mesh is presented in Figure 2.66. Again, the resulting Schur complements are merged, which corresponds to panel (d) in Figure 2.48. Next, the fully aggregated degrees of freedom are eliminated, which is reflected in panel (e) in Figure 2.48 and denoted by the graph transformation presented in Figure 2.67. The last step is to formulate and solve the common interface problem. This is done by the graph transformation presented in Figures 2.68 and 2.69.

The partial forward eliminations are followed by recursive backward substitutions, which repeat the same pattern as in case of the aggregation and elimination.

The presented solver algorithm can be generalized to an arbitrary mesh, since the graph transformation follows the recursive structure of the graph representation of a mesh.

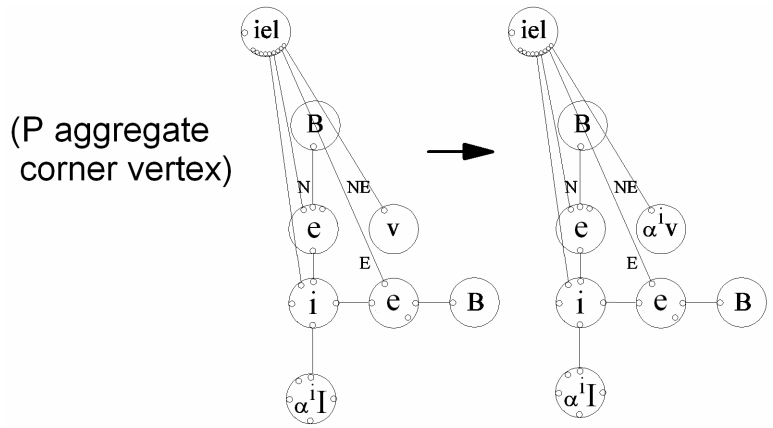


Fig.2.49. Graph transformation executing the aggregation of degrees of freedom related to corner vertices located at the initial mesh element level, after the mesh has been refined

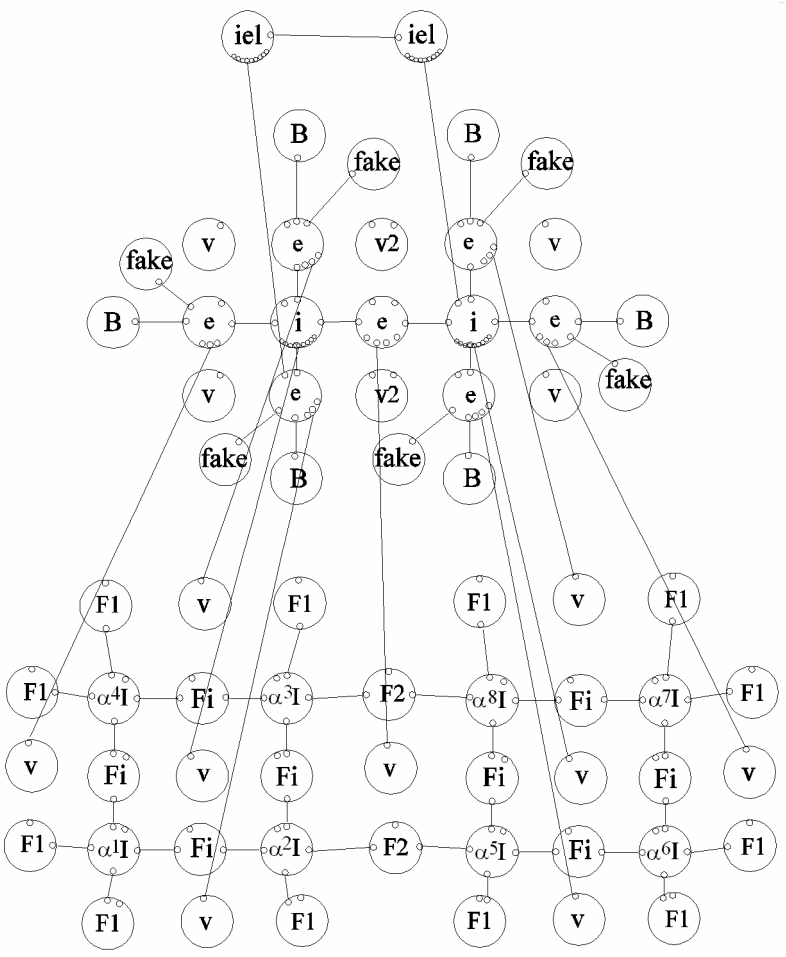


Fig.2.50. Graph representation of the fine mesh after the execution of **(P aggregate interior)**[8]

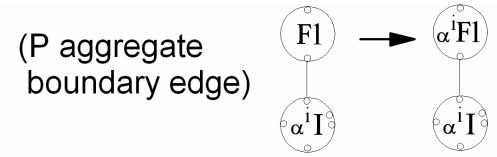


Fig.2.51. Graph transformation executing the aggregation of degrees of freedom related to boundary edges, on son elements

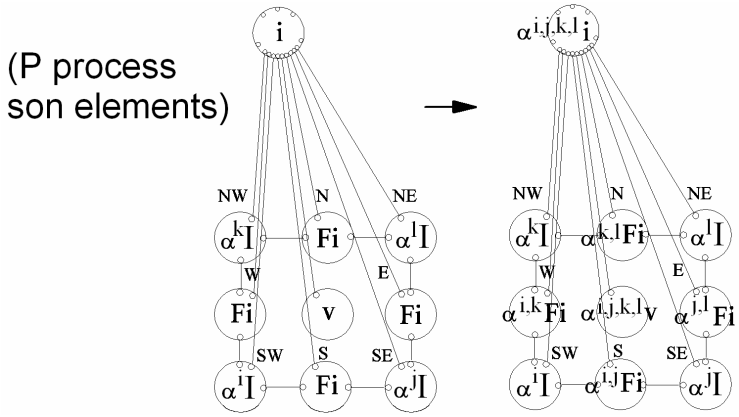


Fig.2.52. Graph transformation executing the aggregation on son elements

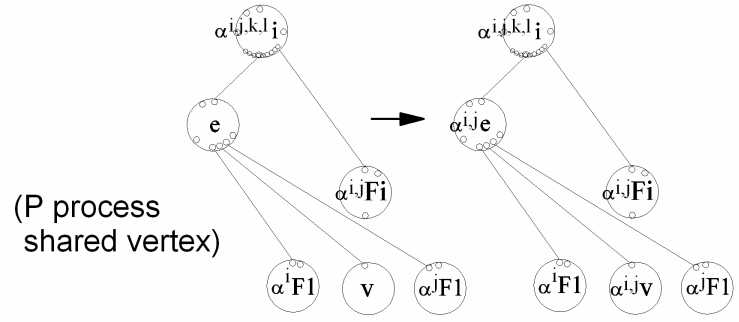


Fig.2.53. Graph transformation executing the aggregation for shared vertex located on the boundary

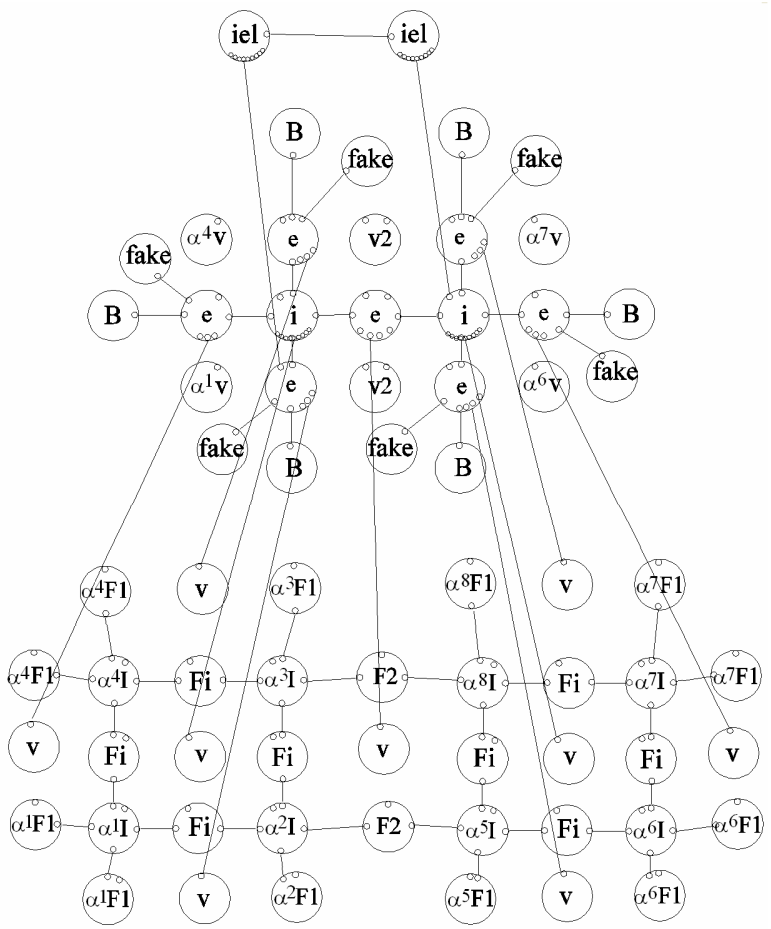


Fig.2.54. Graph representation of the fine mesh after the execution of **(P aggregate interior)^{8} – (P aggregate boundary edge)^{12} – (P aggregate corner vertex)^{4}**

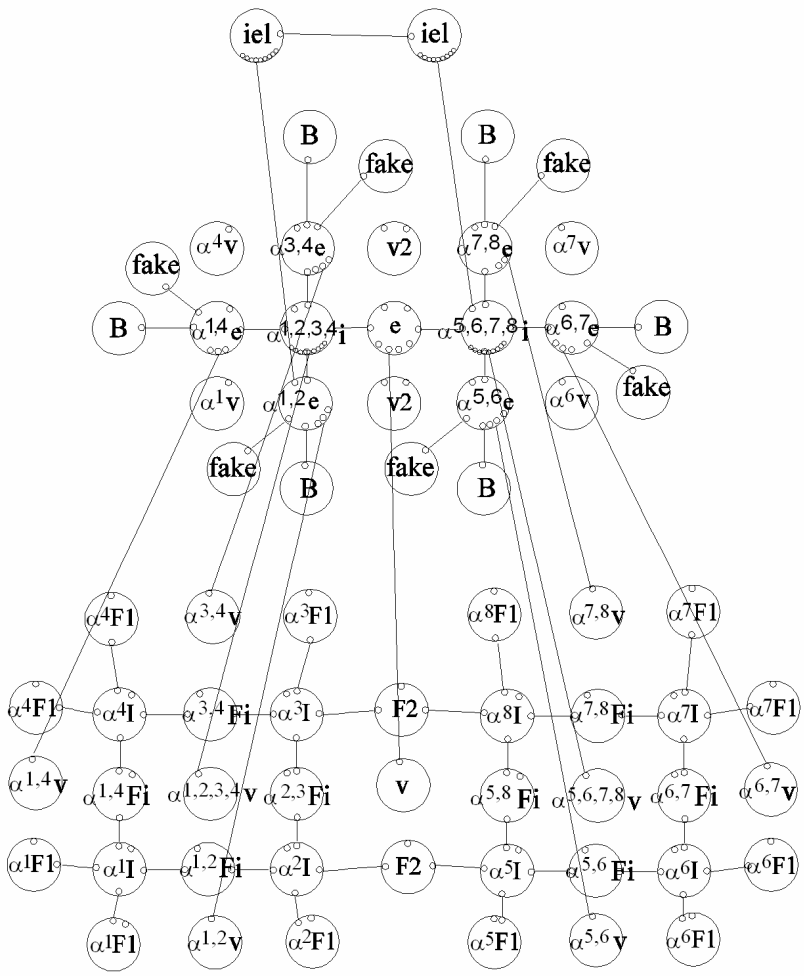


Fig.2.55. Graph representation of the fine mesh after the execution of **(P aggregate interior)^{8} – (P aggregate boundary edge)^{12} – (P aggregate corner vertex)^{4} – (P process son elements)^{2} – (P process shared vertex)^{6}**

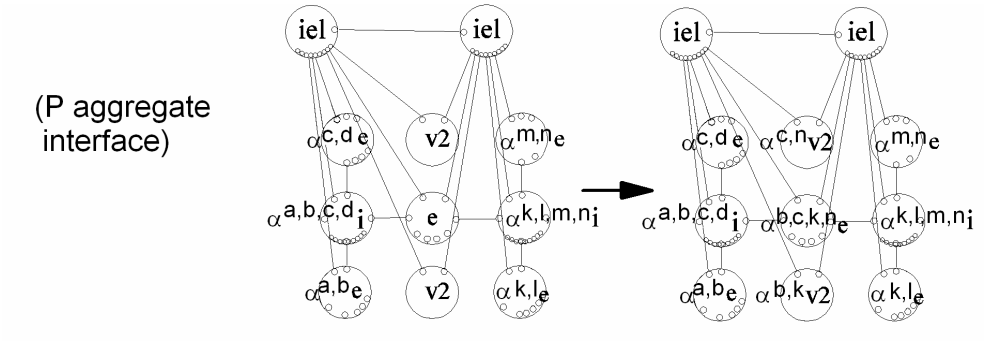


Fig.2.56. Graph transformation executing the aggregation of interface

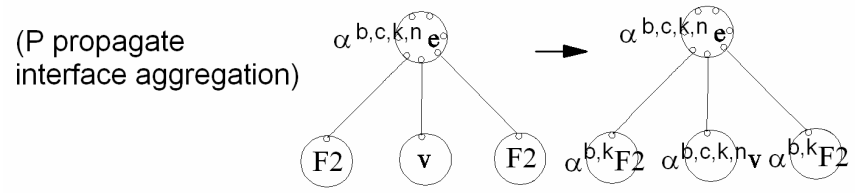


Fig.2.57. Graph transformation propagating the aggregation on interface

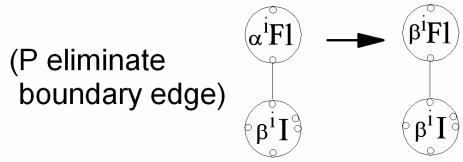


Fig.2.58. Graph transformation eliminating degrees of freedom related to boundary edges

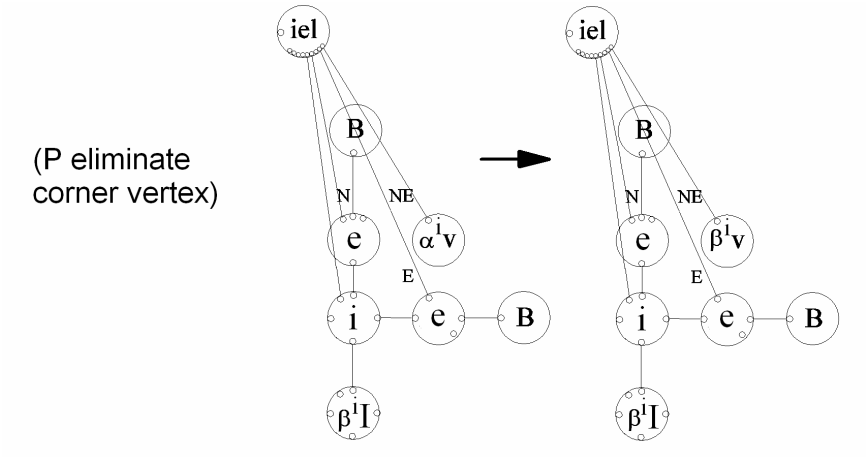


Fig.2.59. Graph transformation executing the elimination of degrees of freedom related to corner vertices

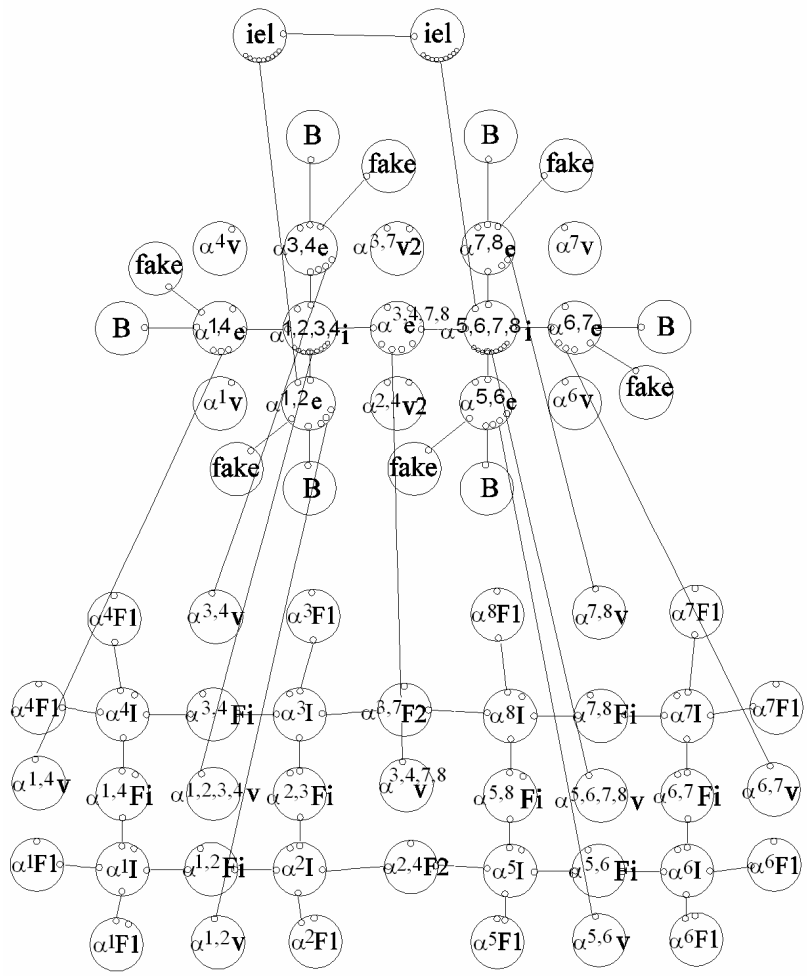


Fig.2.60. Graph representation of the fine mesh after the execution of **(P aggregate interior)^{8} – (P aggregate boundary edge)^{12} – (P aggregate corner vertex)^{4} – (P process son elements)^{2} – (P process shared vertex)^{6}– (P aggregate interface) – (P propagate interface aggregation)**

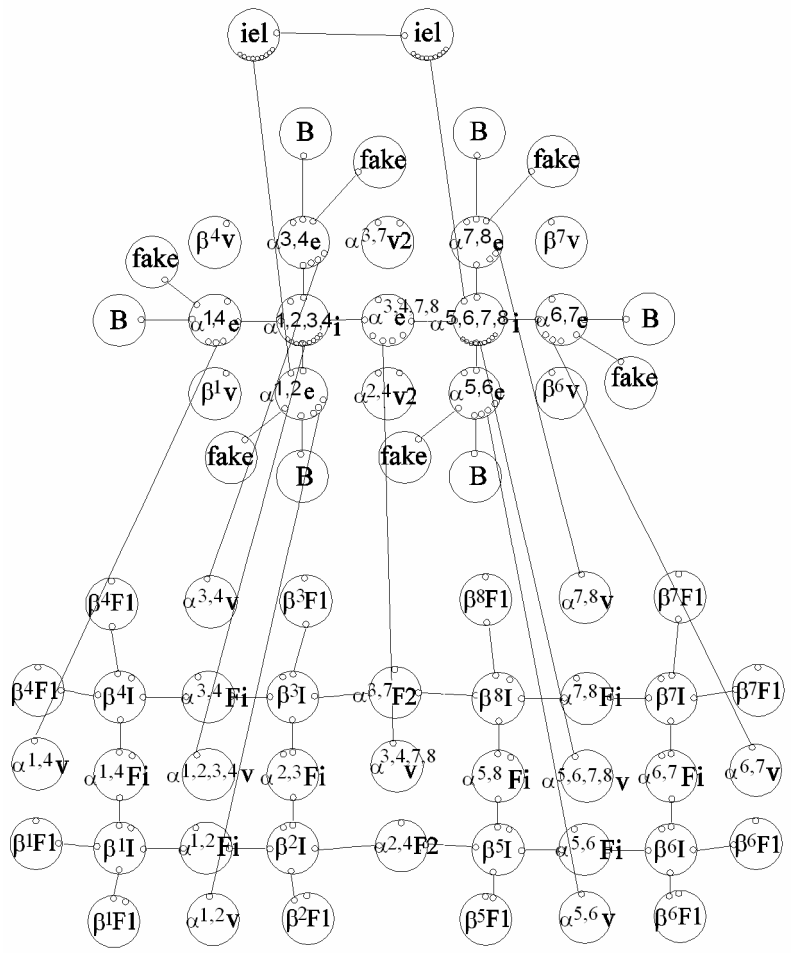


Fig.2.61. Graph representation of the fine mesh after the execution of **(P aggregate interior)^8 – (P aggregate boundary edge)^8 – (P aggregate corner vertex)^4 – (P process son elements)^2 – (P process shared vertex)^6– (P aggregate interface) – (P propagate interface aggregation) – (P eliminate interior)^8 – (P eliminate boundary edge)^8 – (P eliminate corner vertex)^4**

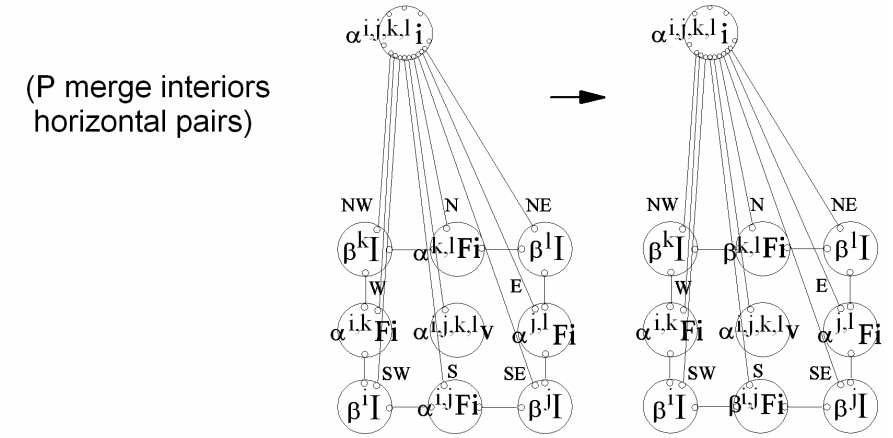


Fig.2.62. Graph transformation executing the merging of Schur complement contributions from adjacent pairs of elements, into the common edge sub-matrix

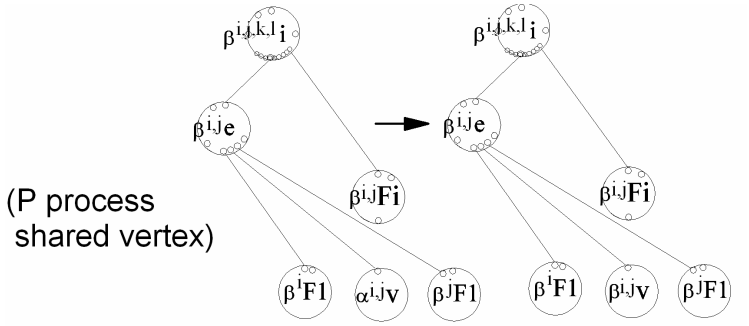


Fig.2.63. Graph transformation executing the merging of Schur complement contributions from adjacent pairs of elements into the common vertices sub-matrix

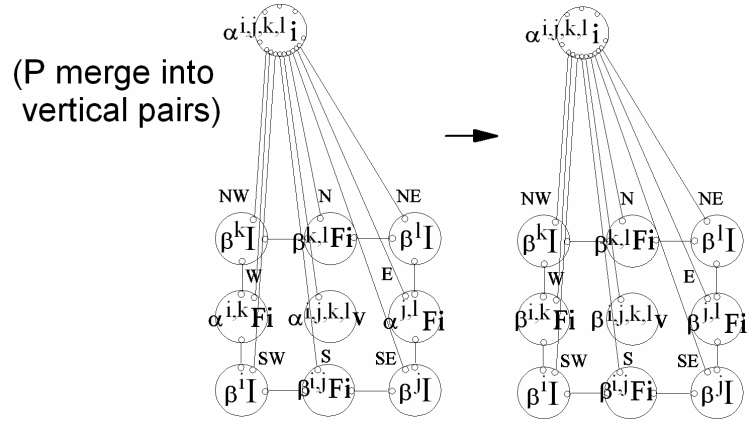


Fig.2.64. Graph transformation eliminating fully assembled edges and vertices, and merging the resulting Schur complements

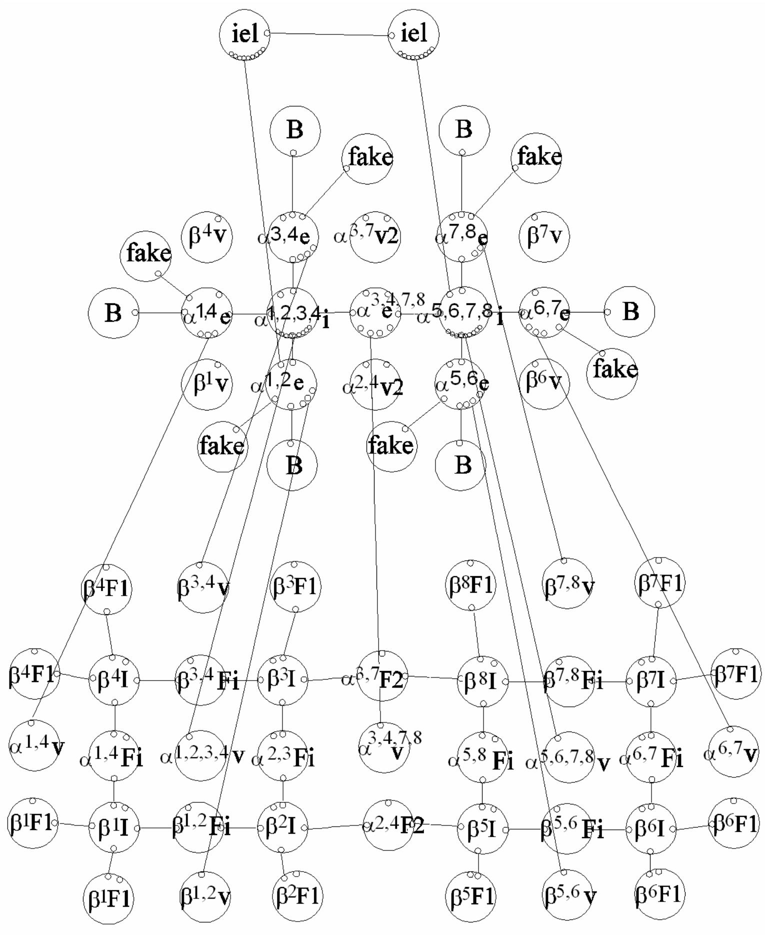


Fig.2.65. Graph representation of the fine mesh after the execution of **(P aggregate interior)^8** – **(P aggregate boundary edge)^8** – **(P aggregate corner vertex)^4** – **(P process son elements)^2** – **(P process shared vertex)^6** – **(P aggregate interface)** – **(P propagate interface aggregation)** – **(P eliminate interior)^8** – **(P eliminate boundary edge)^8** – **(P eliminate corner vertex)^4** – **(P merge interiors horizontal pairs)^4** – **(P process shared vertices)^4**

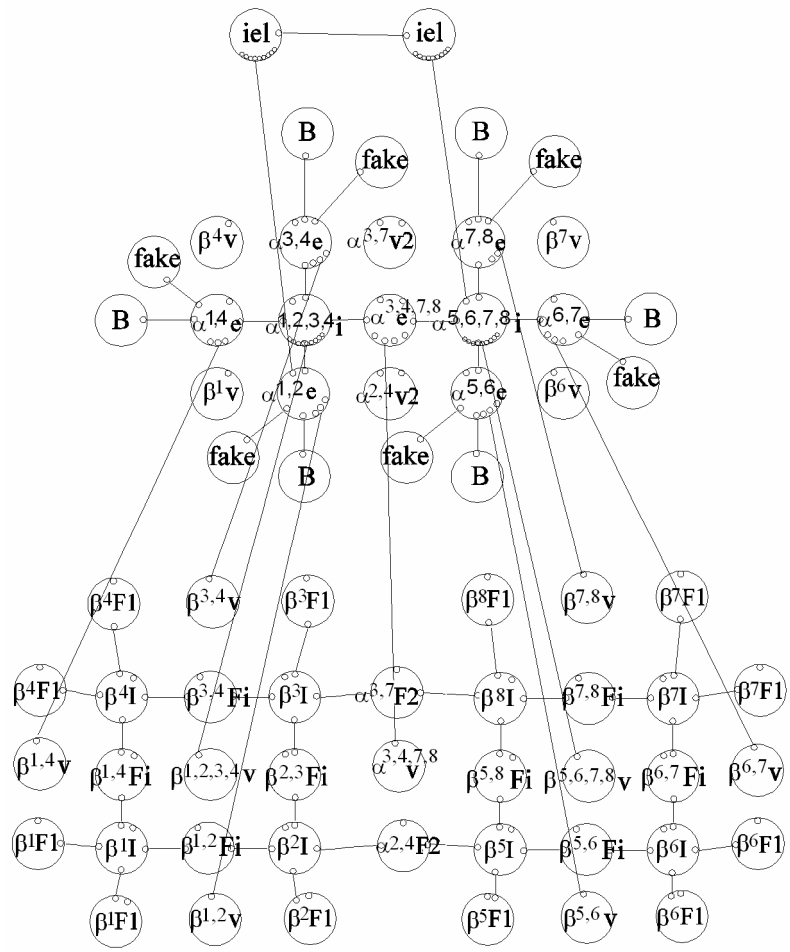


Fig.2.66. Graph representation of the fine mesh after the execution of **(P aggregate interior)^8 – (P aggregate boundary edge)^8 – (P aggregate corner vertex)^4 – (P process son elements)^2 – (P process shared vertex)^6– (P aggregate interface) – (P propagate interface aggregation) – (P eliminate interior)^8 – (P eliminate boundary edge)^8 – (P eliminate corner vertex)^4– (P merge interiors horizontal pairs)^4– (P process shared vertices)^4– (P merge into vertical)^2– (P process shared vertices)^2**

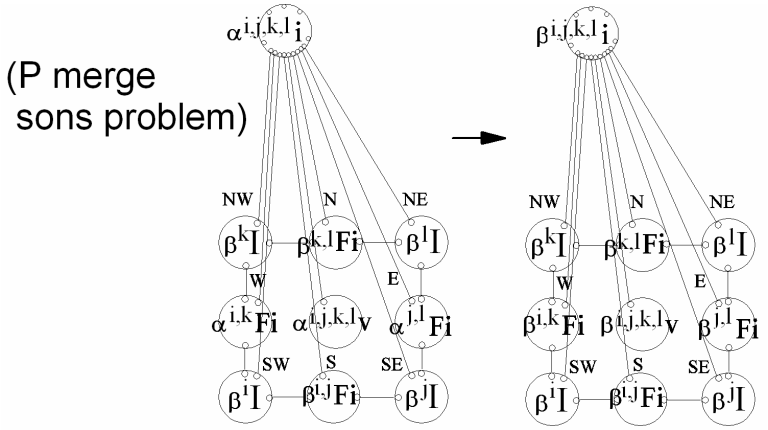


Fig.2.67. Graph transformation merging Schur complements resulting from son elements

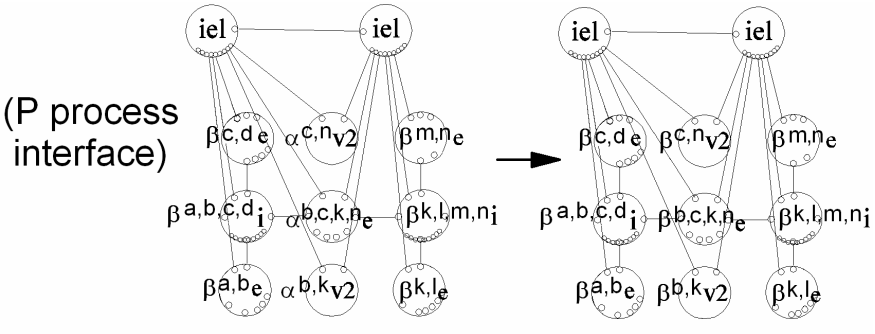


Fig.2.68. Graph transformation for the construction of common interface problem

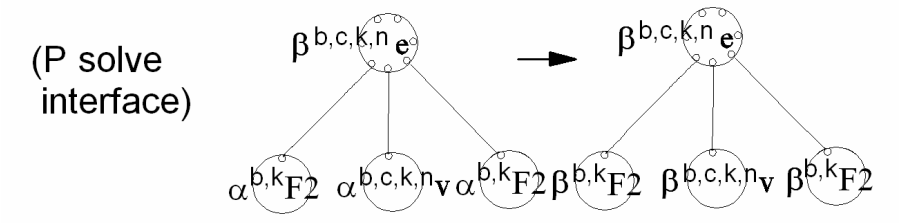


Fig.2.69. Graph transformation for the solution of common interface problem

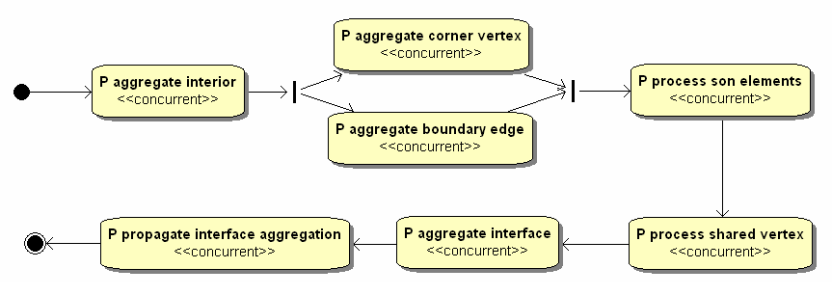


Fig.2.70. Control diagram for aggregation on fine mesh on the level of atomic tasks

It is assumed that the fine mesh solution is stored at the graph vertices. The solution represents the coefficients of the fine mesh shape functions. It is stored in the form of graph attributes $u^{i}_{h/2,p+1}$ distributed among the graph vertices. The number of shape functions at the graph vertices, representing finite element vertices, is equal to 1. The number of shape functions at the graph vertices, representing element edges, is equal to p-1, where p denotes the polynomial order of approximation at the edge. Finally, the number of shape functions at the graph vertices, representing element interiors, is equal to (p_h-1)(p_v-1), where (p_h , p_v) denote the polynomial orders of approximation in the horizontal and vertical directions.

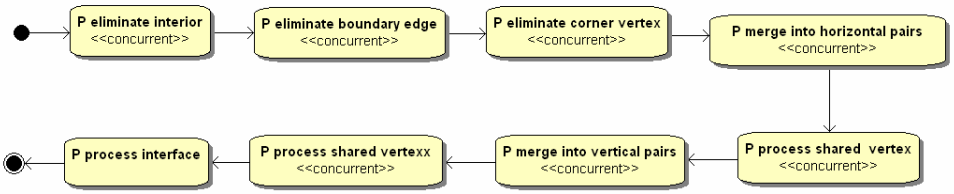


Fig.2.71. Control diagram for elimination on fine mesh on the level of atomic tasks

The process of the fine mesh problem solution can be now described by a sequence of atomic tasks. The control diagrams presenting the order of execution of atomic tasks with assigned graph grammar productions are presented in Figures 2.70 and 2.71. Figure 2.70 presents the control diagram for the aggregation while Figure 2.71 presents the control diagram for the elimination. All atomic tasks can be executed in the concurrent way, some of them can be also grouped together (their execution can be mixed), which is denoted by `<<concurrent>>` stereotype and by adding synchronization points before and after the each group of atomic tasks. The solver algorithm and the control diagrams can be also applied to an arbitrary multi-level graph representation of an arbitrary fine mesh.

2.1.5. Algorithm of selection and execution of the optimal mesh refinements

The next step of the self-adaptive *hp*-FEM algorithm is to choose the optimal refinements. The algorithm employs the graph representation of the fine mesh, obtained by copying the graph representation of the coarse mesh, and by performing the global *hp* refinement. It is assumed that the coarse and fine mesh solutions are stored at the graph vertices in the form of u^{i}_{hp} and $u^{i}_{h/2,p+1}$ graph attributes. In fact, the graph vertices which represent active finite elements and constitute the leaves of refinement tree, contain the graph attributes $u^{i}_{h/2,p+1}$ for the fine mesh solution, and their parent vertices contain the graph attributes u^{i}_{hp} for the coarse mesh solution.

The general algorithm can be expressed in two steps. First, the estimation of the relative error in the energy norm is computed for each active finite element,

$$\text{error}_K = \left\| u_{h/2,p+1} - u_{h,p} \right\|_{1,K} \tag{2.37}$$

where $u_{hp} = \sum_i u^{i}_{hp} e^{i}_{hp}$ denotes the coarse mesh solution, and $u_{h/2,p+1} = \sum_i u^{i}_{h/2,p+1} e^{i}_{h/2,p+1}$ denotes the fine mesh solution, while e^{i}_{hp} and $e^{i}_{h/2,p+1}$ denote the coarse and fine mesh shape functions (compare Section 1.2.1). The norm is computed on the element *K*. This operation is expressed by the graph grammar production **(P compute relative error),** presented in Figure 2.72. The production is executed for all element interiors.

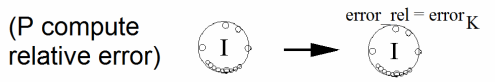


Fig.2.72. Graph grammar production responsible for computing relative error on element interior

Second, the maximum relative error is computed

$$\text{max\_error} = \max_K \left\{ \text{error}_K \right\} \tag{2.38}$$

and all active elements with a relative error greater than 33% of the maximum relative error are selected for a refinement.

Next, we choose the optimal refinement for each selected element. This is achieved by analysing various possible strategies of the refinement. An element can be either *h* refined (in horizontal or vertical directions, or both) or *p* refined (the polynomial order of approximation can be modified in horizontal or vertical directions) or *hp* refined.

Now, for each analysed refinement strategy, for each element, we have to compute a local interpolant *w* of the fine mesh solution $u_{h/2,p+1}$. This is done using the *projection-based interpolation* algorithm (Demkowicz 2006).

Let us make the following observation. The coarse mesh approximation space V_{hp} is a subset of the approximation space V_w, corresponding to a proposed refinement, which is also a subset of the fine mesh approximation space $V_{h/2,p+1}$

$$V_{h,p} \subset V_w \subset V_{h/2,p+1} \tag{2.39}$$

The projection-based interpolant w of the fine mesh solution for the element K is computed by the application of the principles of *locality, global continuity* and *optimality*. The principle of locality means that the interpolant is defined entirely in terms of the restriction of the function to the element only. The principle of global continuity implies that the union of element interpolants is globally conforming. The principle of optimality implies that the interpolation errors must behave asymptotically, both in h and p, in the same way as a current approximation error. First of all, the interpolant w must match interpolated function $u_{h/2,p+1}$ at element vertices v

$$w(v) = u_{h/2,p+1}(v) \tag{2.40}$$

With the vertex values fixed, locality and global continuity imply that the restriction of the interpolant to an element edge should be calculated using the restriction of function $u_{h/2,p+1}$ only. The optimality implies that the appropriate edge norm should be applied for the projection

$$\left\| u_{h/2,p+1} - w \right\|_e \to \min \tag{2.41}$$

for each edge e.

Finally, the projection for the element K in appropriate element norm is computed

$$\left\| u_{h/2,p+1} - w \right\|_K \to \min \tag{2.42}$$

to complete the definition of the projection-based interpolant w. Following Demkowicz 2006, the edge norm is selected as $H^{1/2}(e)$ norm, and the element norm is selected as $H^1(K)$ norm. We refer to Demkowicz 2006 for more details on the projection-based interpolation procedure. Having the local projection-based interpolant w of the fine mesh solution $u_{h/2,p+1}$ for considered element refinement, we can compute the *error decrease rate*

$$\text{rate}(w) = \frac{\left\| u_{h/2,p+1} - u_{h,p} \right\|_{1,K} - \left\| u_{h/2,p+1} - w \right\|_{1,K}}{\Delta \text{nrdof}} \tag{2.43}$$

where Δnrdof denotes an increase in the number of degrees of freedom on the coarse mesh element resulting from the execution of the analysed mesh refinement strategy. The nominator expresses a decrease in the relative error associated with a proposed refinement of element K, the denominator expresses the corresponding increase in the number of degrees of freedom.

The procedure of evaluating different possible refinement strategies for an element is expressed by the graph grammar production **(P evaluate refinement),** presented in Figure 2.73. The production is executed for all element interiors and for all considered refinement strategies *w*.

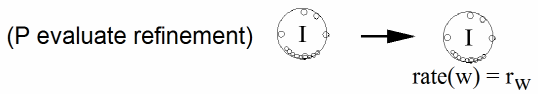


Fig.2.73. Graph grammar production responsible for evaluation of a given refinement strategy *w*

The refinement strategy providing the maximum error decrease rate is selected for the element *K*. This is expressed by the production **(P select optimal refinement)**, presented in Figure 2.74.

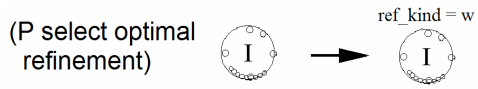


Fig.2.74. Graph grammar production responsible for selection of the optimal refinement

The algorithm can be summarized in the form of the control diagram presented in Figure 2.75.

The algorithm has been partitioned into multiple atomic tasks. The algorithm starts with the execution of atomic tasks related to the graph grammar productions **(P compute relative error)**. All these atomic tasks can be executed in the concurrent way. The next atomic tasks compute the global maximum error. Then, the atomic tasks related to the production **(P evaluate refinement)** are executed. Note that these atomic tasks can be also executed in the concurrent way. In fact, there are many more such atomic tasks than graph vertices representing finite element interiors, since there are many possible refinement strategies for each graph vertex representing element interior.

There are several possibilities to limit a number of refinement strategies analysed for an element. The quasi-optimal solution to this problem was proposed by Demkowicz, 2006. The process of selection of the refinement strategy consists of two steps. First, we select the optimal refinements for each element edge. The projection-based interpolation for the local solution corresponding to the proposed edge refinement is based on the edge projections (2.41). The number of possible edge refinements is quite limited (actually only one *h* refinement and several *p* refinements may be considered for an edge). Second, we select a type of refinement for an element interior. This selection is limited by the optimal refinement previously selected for these element edges.

At this point, we execute the optimal refinement which have been selected for the elements with a high relative error. This procedure consists of the following two steps:

1) to perform all the requested *h* refinements,
2) to perform all the requested *p* refinements.

The so-called 1-irregularity rule (as defined in Section 2.1.3) may result in some additional *h* refinements. Let us describe the procedure in a more detailed way. An element presented in Figure 2.76 has been selected for the isotropic *h* refinement. The element is broken into four new son elements, in horizontal and vertical directions. Two nodes and

one vertex on the common edge are constrained by the large element edge, with no new nodes generated. The mesh is called irregular.

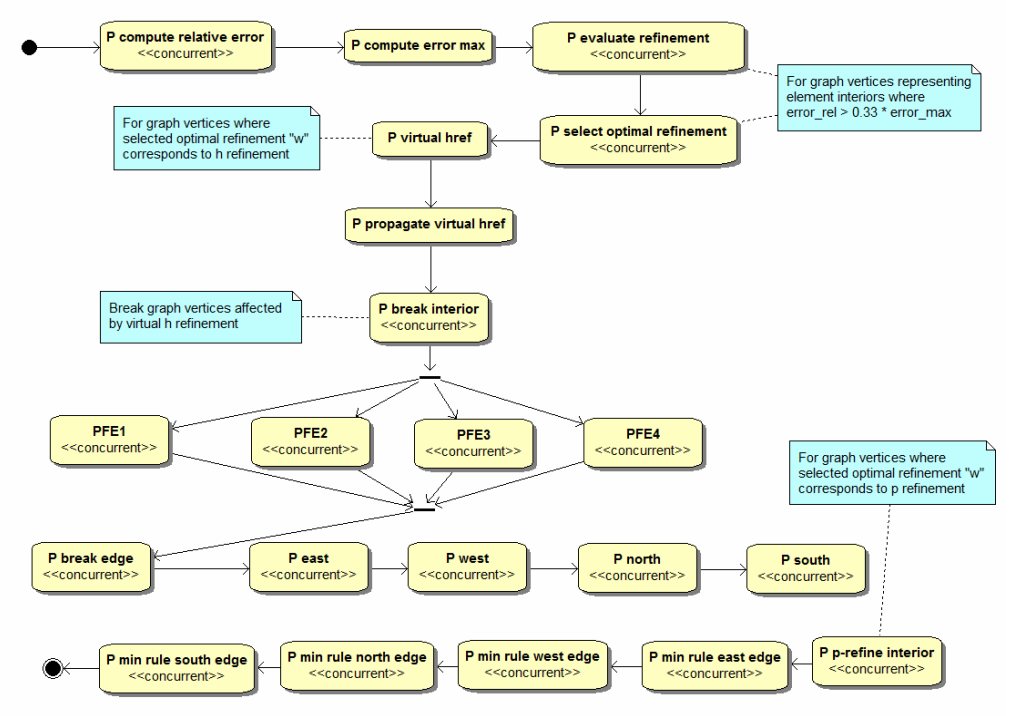


Fig.2.75. Control diagram for algorithm of selection and execution of optimal refinements on the level of atomic tasks

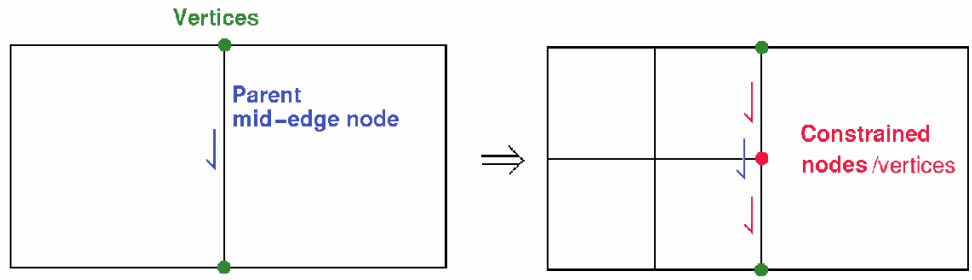


Fig.2.76. Isotropic *h* refinement of an element

Next, one of the two small elements adjacent to the common edge is selected for an additional isotropic *h* refinement. The execution of this new *h* refinement will lead to multiple constrained nodes, as presented in Figure 2.77. In this case, there are three nodes and two vertices constrained by the adjacent large element. These nodes and vertices will not be generated until the adjacent large element is broken twice. This situation is

extremely inconvenient from the technical point of view, since the approximation is highly complicated in this case.

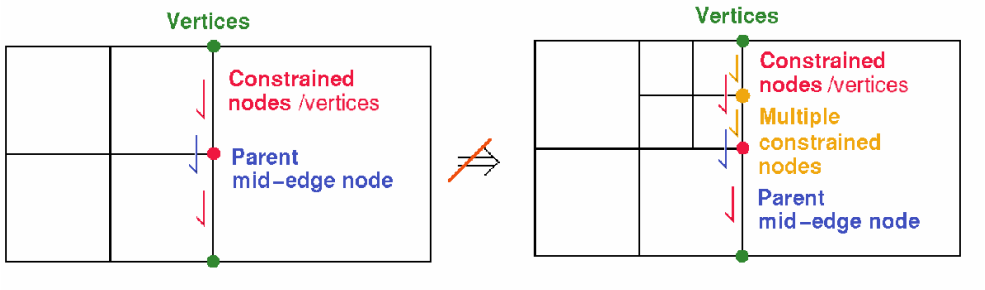


Fig.2.77. Isotropic *h* refinement of one of newly created son elements

The common practice in the engineering community is to enforce the isotropic *h* refinement of the adjacent large element before the execution of the second *h* refinement of a small element. In this example the large element must be broken first, which is illustrated in Figure 2.78, and two nodes and one central vertex are created on the common edge.

However, two nodes and one vertex of the two smallest elements remain constrained on the common edge. One of the two nodes which have been created is employed for the constrained approximation over the two smallest elements adjacent to the common edge.

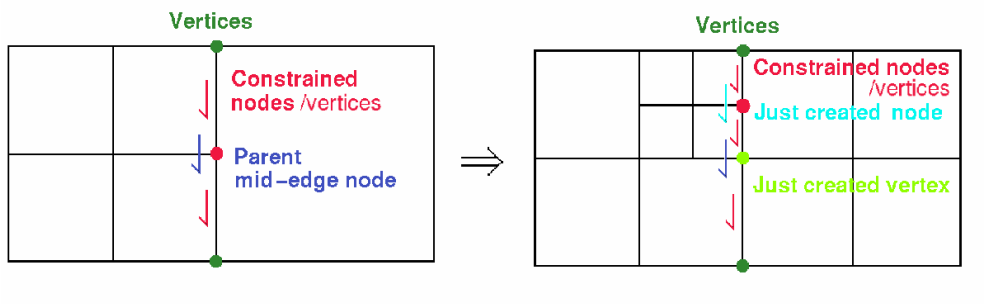


Fig.2.78. Additional isotropic *h* refinement of adjacent large element

Let us focus on the process of *h* and *p* refinements, expressed by the control diagram presented in Figure 2.79. We start with the execution of selected *h* refinements. Graph vertices representing finite elements to be broken are attributed by *h* refinement flags. This is done by executing atomic tasks related to the production **(P virtual href)**, presented in Figure 2.79.

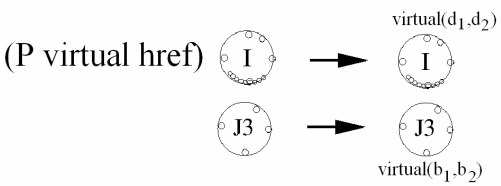


Fig.2.79. Graph grammar productions attributing graph vertices, for element interiors intended for *h* refinement (for graph vertices **I** and **J3**)

Now, the procedure of virtual *h* refinement is executed. Before breaking selected elements, the graph vertices intended for *h* refinements are denoted by virtual refinement flags. The virtual refinement flags propagate from small elements up to adjacent large elements, in order to fullfill the 1-irregularity mesh rule. This is done by executing the productions **(P propage virtual href)**, presented in Figure 2.80. The atomic tasks assigned to the execution of virtual refinements are performed in serial, since some of them may be related to the same graph vertices. In this case, we use the logical "OR" between current and new refinement flags.

At this point, the elements denoted by the virtual refinement flags are broken. This is called the *physical refinement*. The process of the physical refinement employs the same atomic tasks and graph grammar productions as for the global *h* refinement introduced in Section 2.1.3. First we execute the atomic tasks related to the productions **(P break interior)**. They may be executed in the concurrent way. Then, we execute several atomic tasks related to the productions **(PFE1-4)**. They can be also executed in the concurrent way, which is denoted by adding the synchronization points before and after these atomic tasks. Finally, we execute a sequence of atomic tasks related to the productions **(P break edge)**, **(P east)**, **(P west)**, **(P north)** and **(P south)**, also in the concurrent way.

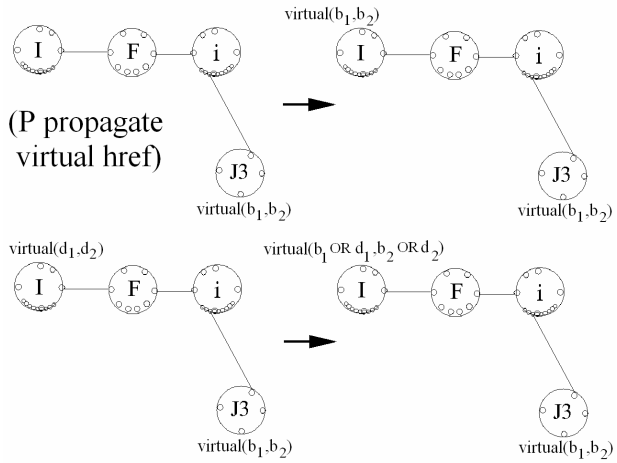


Fig.2.80. Graph grammar productions enforcing 1-irregularity mesh rule by the propagation of virtual refinements

After all the selected *h* refinements have been executed, we focus on the execution of selected *p* refinements. The *p* refinement consists in assigning the selected polynomial orders of approximation to an element interior. This is done by executing atomic tasks related to **(P p-refine interior)** productions, which attribute graph vertices representing element interiors. The order of approximation for element edges is indicated by the application of the minimum rule, defined in Section 2.1.3. The minimum rule is applied by the execution of four atomic tasks related to the productions **(P min rule east edge)**, **(P min rule west edge)**, **(P min rule north edge)** and **(P min rule south edge)**. They enforce the minimum rule for vertices representing the element edges located in the eastern, western, northern and southern directions of element interiors with updated polynomial orders of approximations. All these atomic tasks can be executed in the concurrent way, which is expressed by the `<<concurrent>>` stereotype in control diagram in Figure 2.75.

2.1.6. The stopping condition

The optimal mesh resulting from the previous step becomes the coarse mesh for the next iteration of the self-adaptive *hp*-FEM algorithm. The iterations are repeated until the maximum relative error (2.37) is smaller than the required accuracy of the solution.

$$\text{max\_error} \leq \text{tolerance} \tag{2.44}$$

2.2. Definition of computational tasks (grains) for parallel processing model of the self-adaptive *hp*-FEM

In this section, the atomic tasks introduced in Definition 2.15 are agglomerated into tasks, presented in Definition 2.17. Moreover, we assume that each task has its own local graph representation of the mesh and works on a local sub-graph which represents a single initial mesh element. A task is understood as the execution of several atomic tasks on a local graph representing a single initial mesh element.

The control diagrams introduced on the level of atomic tasks in Definition 2.16 are now redefined for the level of tasks (see Definition 2.18), since some algorithms require inter-task communication. Thus, it is possible to define several communication channels between tasks. Each task has its own global identificator, called *rank.* The control diagrams defined for tasks are executed by all tasks on their own local graph representing a single initial mesh element. The tasks constitute the grain for the load balancing and mesh partitioning algorithms, introduced in the next section. The graph representation of the computational mesh is stored in a partitioned manner, with each sub-graph representing a single initial mesh element, assigned to a single task.

Summing up, the task is defined as an execution of several atomic tasks on a local graph representation of a single initial mesh element, stored in the task's local memory. The execution of the atomic tasks within the task is managed by the control diagrams. However, this time the control diagrams contain the predicates of applicability, clearly defining task ranks that can execute this part of the control diagrams. More precisely, the predicates of applicability are assigned to control diagram states.

If a state from control diagram does not have a predicate of applicability, it is executed by all tasks. However, if a state does contain the predicate of applicability, it is executed only by the task with a given rank.

To express the self-adaptive algorithm on the level of tasks, with a graph representation of the mesh distributed into tasks, it is necessary to introduce new graph grammar productions for partitioning and merging of the graph representation of the mesh.

The first set of graph transformations is intended for mesh partitioning. These transformations use the recursive propagation of the partition through the refinement trees. The first production presented in Figure 2.81 initializes the partition process by splitting two adjacent finite elements and adding a new graph vertex **Int** for the propagation of the mesh partition.

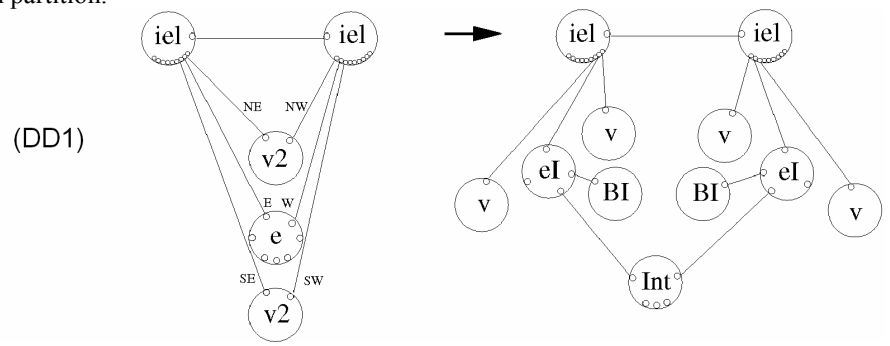


Fig.2.81. Production for partitioning two adjacent elements.
The new ***Int*** vertex is created to propagate the partition

The **Int** vertex propagates the partition through the refinement trees. There are different structures of the refinement trees to be considered. In case of the graph transformation presented in Figure 2.82, we need to partition an element edge broken into two son edges, with the first son edge, denoted by **e** symbol, being broken, and the second son edge, denoted by **F2** symbol, remaining unbroken.

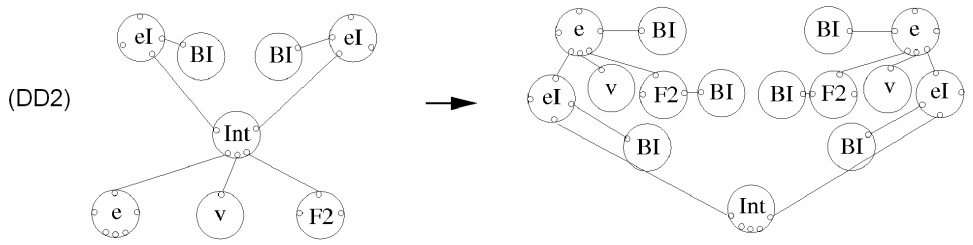


Fig.2.82. Partition of edge with one son edge broken and the other one unbroken

In the second case, presented in Figure 2.83 we need to partition one element edge node with two son edges, where both son edges denoted by **F2** symbol are not broken. This is the end of the partition of a branch of tree.

In the third case, presented in Figure 2.84, we need to partition one element edge with two son edges, where both son edges, denoted by **e** symbol, are broken. Two new graph vertices labeled by **Int** symbol are created to propagate the partition into both branches.

The second set of graph transformations shows the merging of two sub-graphs, previously partitioned by transformations **(DD1)-(DD4)**.

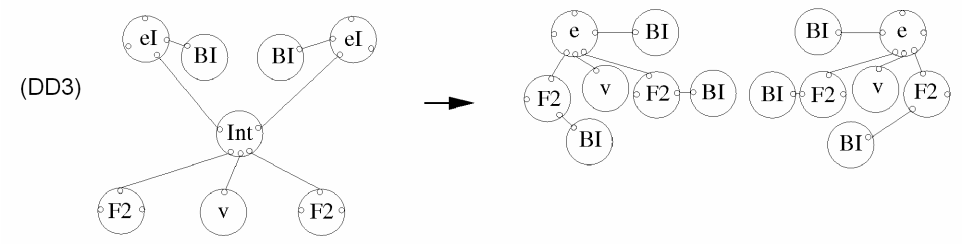


Fig.2.83. Partition of edge with two unbroken son edges

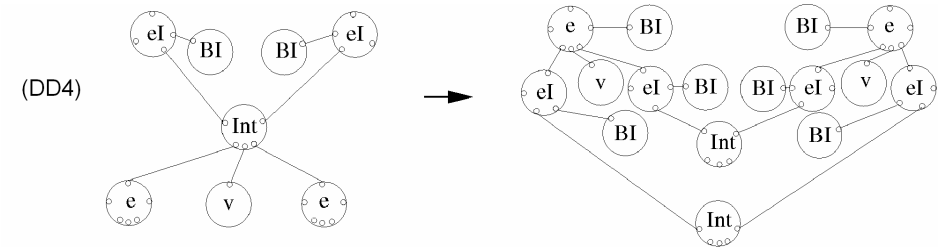


Fig.2.84. Partition of edge with two broken son edges

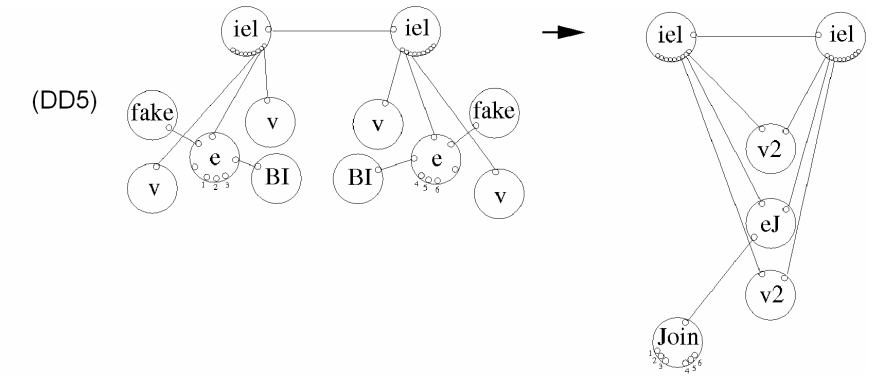


Fig.2.85. Initialization of merging process

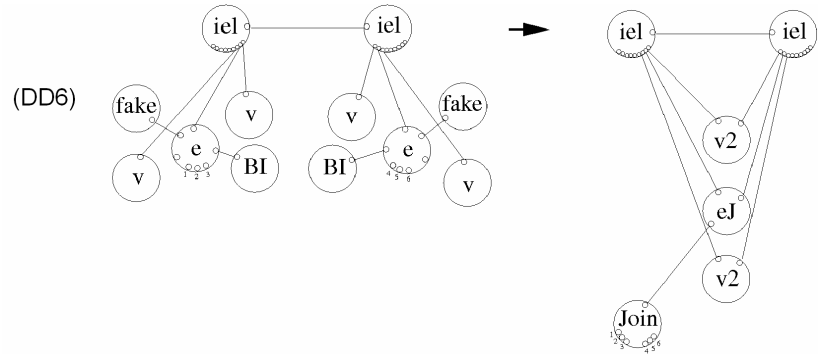


Fig.2.86. Merging of two broken edges

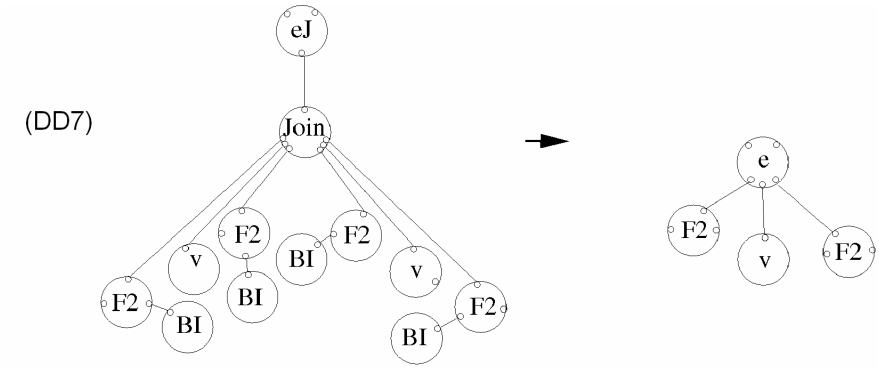


Fig.2.87. Merging of two unbroken edges. The transformation is cloned with **eJ** replaced by **eJ2**

The first production presented in Figure 2.85 initializes the merging process by joining two adjacent finite elements and adding a new graph vertex **Joint** for propagation of the merging process. It's necessary to analyse different structures of the refinement tree.

In the first case presented in Figure 2.86 we need to join two identical broken edges, denoted by **e** symbol. We need to maintain the corresponding numbering of bounds. In the second case presented in Figure 2.87 we have to join two unbroken edges, while in the last case, presented in Figure 2.88, we have to join two broken son edges. The joining process propagates into both branches. Thus, two new graph vertices labeled with **Join** symbols are created. We need to maintain the corresponding numbering of bounds.

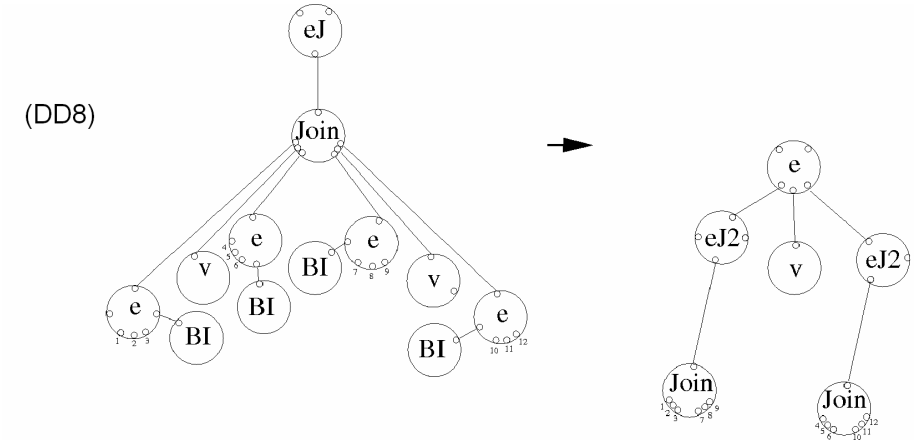


Fig.2.88. Merging of two broken son edges. The transformation is cloned with **eJ** replaced by **eJ2**

In the following sub-section, the control diagrams are redefined to describe all parts of the self-adaptive *hp*-FEM algorithm executed on the level of tasks.

2.2.1. Generation of the initial mesh

The generation of the initial mesh is now partitioned into multiple tasks. We assume that each task is assigned to a single initial mesh element and has a global identifier called the rank. In order to simplify the communication between tasks, it is assumed that each task generates the topology of the entire mesh, with global numbering of initial mesh elements. Next, each task generates a structure of initial mesh element with the number equal to the task rank. The set of graph grammar productions for the generation of an intial mesh has been updated to assign attributes to the initial mesh elements (see Figure 2.89).

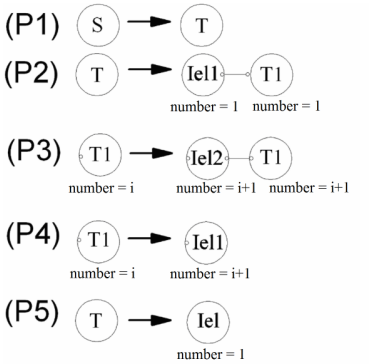


Fig 2.89. Updated graph grammar productions for the generation of initial mesh

As a result, we generate the topology of the initial mesh, with initial mesh elements numbered. The process of the mesh generation is managed by the control diagram presented in Figure 2.90.

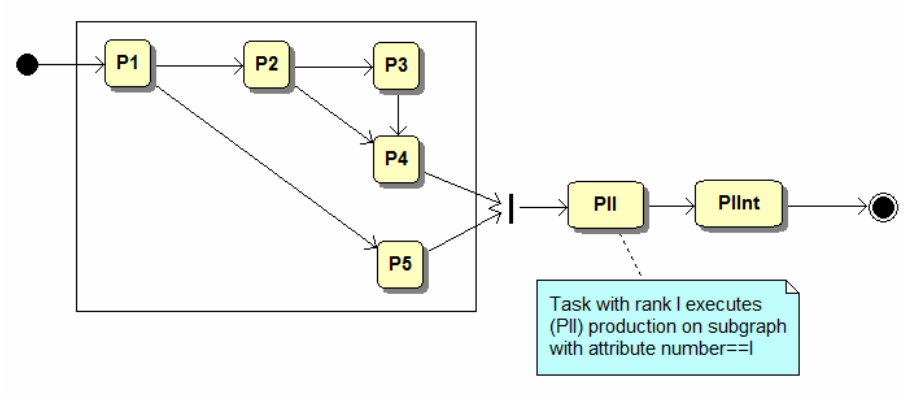


Fig 2.90. Control diagram for the generation of initial mesh on the level of tasks

To simplify this example we assume that the structure of the initial mesh is linear. In other words, an initial mesh consists of a row of initial mesh elements. The control diagram presented in Figure 2.90 can generate initial meshes with any number of elements, in the linear sequence. Thus, the input for the initial mesh generation algorithm is the initial mesh coded as the order of production. For the initial mesh with a single initial mesh element, only the productions **(P1)-(P5)** are executed. For the initial mesh with two initial mesh elements, the productions **(P1)-(P2)-(P4)** are executed. For the initial mesh with *m* initial mesh elements, the productions **(P1)-(P2)-(P3)**$^{m-2}$**-(P4)** are executed. Thus, the initial mesh is coded by the number of **(P3)** productions to be executed.

The topology of the initial mesh is supposed to be generated for all tasks. Then, only a task with the rank equal to the number of initial mesh elements, executes the production **(PII)** generating the structure of the initial mesh. Finally, all tasks execute twice the new production **(PIInt)**, presented in Figure 2.91. The production distinguishes the boundaries of the domain (graph vertices with **B** symbols) from the interface between initial mesh elements, denoted now by graph vertices with **BI** symbols.

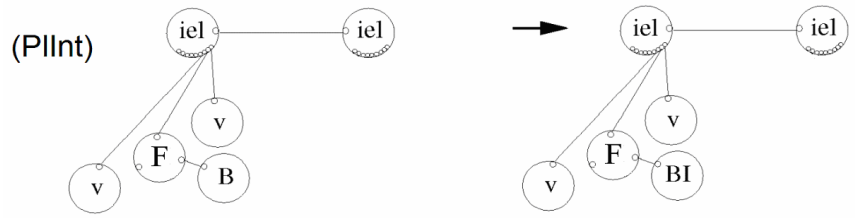


Fig 2.91. Graph grammar production distinguishing boundary from the interface

It should be emphasized that the generation of the topology of the initial mesh is actually a redundant operation. If the tasks are used in the distributed memory or hybrid

parallel machine architectures, and the number of processors is comparable to the number of tasks, it is not a problem. However, if the tasks are used in the shared memory architecture or the number of tasks is much higher than the number of processors, this may slow down significantly the generation process. In such case, an alternative strategy algorithm can be employed. The topology of the entire mesh can be generated only by the first task, and sent later to all other tasks, since in the shared memory architecture, the send/receiver operations are implemented by rewritting data from one part of the memory to the other.

Let us conclude this section with the example presented in Figure 2.92. It shows the two initial element mesh distributed to two tasks. The first task contains the topology of the entire mesh and the structure of the first element, while the second task contains the topology of the entire mesh and the structure of the second element.

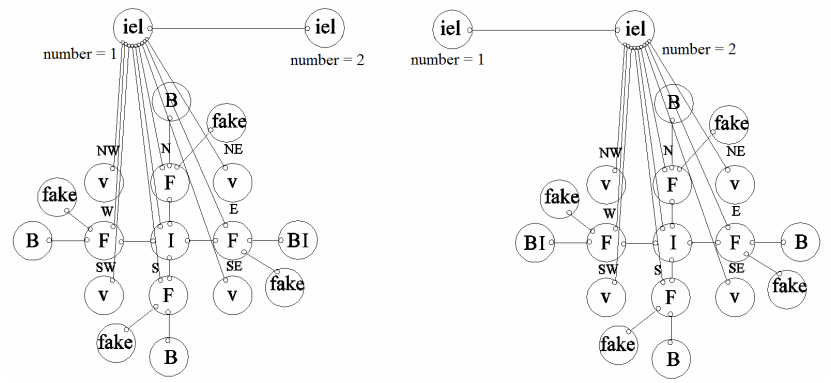


Fig 2.92. The two initial element mesh genereted for two tasks

When the process of the initial mesh generation is finished, we obtain multiple tasks, each of them containing the topology of the entire mesh and the structure of the single initial mesh element assigned to this task.

2.2.2. Solution of coarse mesh problem

At this point, we can solve the coarse mesh problem. In order to execute the coarse mesh solver algorithm it is necessary to define communication channels between tasks. In case of the simplified linear sequence of graph grammar productions, the communication channels are defined only between two consecutive tasks. Then, the coarse mesh solver algorithm can be summarized by the control diagram presented in Figure 2.93.

Let us trace the execution of the coarse mesh solver on the exemplary two initial element mesh distributed to two tasks, presented in Figure 2.92.

First, each task executes the productions **(P aggregate interior)-(P aggregate boundary edge)^3-(P aggregate corner vertex)^2**. The result of the execution of this sequence of productions for the first and the second task is presented in Figure 2.95. The graph grammar productions create local frontal matrices, one for each task. The frontal matrices have global numbering, and the number of the frontal matrix is inherited from the

task rank. The productions aggregate to the local frontal matrix the degrees of freedom related to element interior and boundary edges.

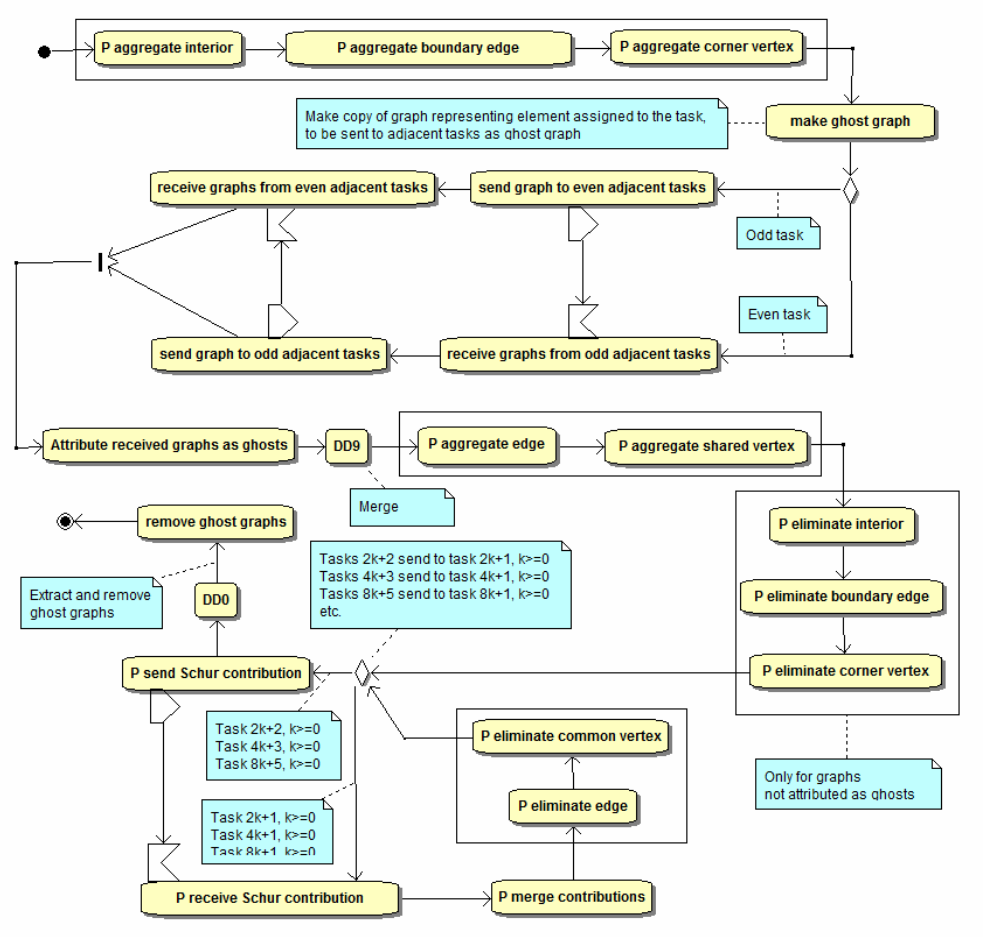


Fig 2.93. Control diagram for the execution of coarse mesh solver on the level of tasks

Next, each task makes a copy of the sub-graph representing its initial mesh element, which is illustrated in Figure 2.96. At this point, the control diagram makes a distinction between the odd and even tasks. The odd tasks send the copy of the sub-graph representing their initial mesh element to adjacent even tasks. Since the structure of the considered initial mesh is linear, the tasks 2k+1, for k>0, send the copy of sub-graph to tasks 2k+2, and to tasks 2k. Then, the even tasks send the copy of sub-graph representing their initial mesh element to odd tasks. Again, it means that tasks 2k for k>0 send the sub-graph to tasks 2k+1 and 2k-1. In other words, the tasks exchange sub-graphs representing initial mesh elements with adjacent tasks. Note that the tasks send only sub-graphs, without exchanging the frontal matrices. Now, each task attributes the received sub-graphs as ghost sub-graphs with ghost initial mesh elements. The tasks merge the received sub-graphs with their local

graphs. The resulting graphs for the simplest possible two-task configuration are presented in Figure 2.96.

It is possible to aggregate now the degrees of freedom related to the common edge. There are two frontal matrices, the first one assigned to the first task, with the interior of the first element already aggregated, and the second one assigned to the second task, with the interior of the second element already aggregated. The first task aggregates the contribution related to the common edge from the first element to the first frontal matrix, and the second task aggregates the contribution related to the common edge from the second element to the second frontal matrix. This is expressed by graph grammar productions **(P aggregate edge)-(P aggregate shared vertex)^2**.

At this point, the aggregation process is finished. The first task stores the fully aggregated frontal matrix of the first element, while the second task stores the fully aggregated frontal matrix of the second element. This is illustrated in Figure 2.97.

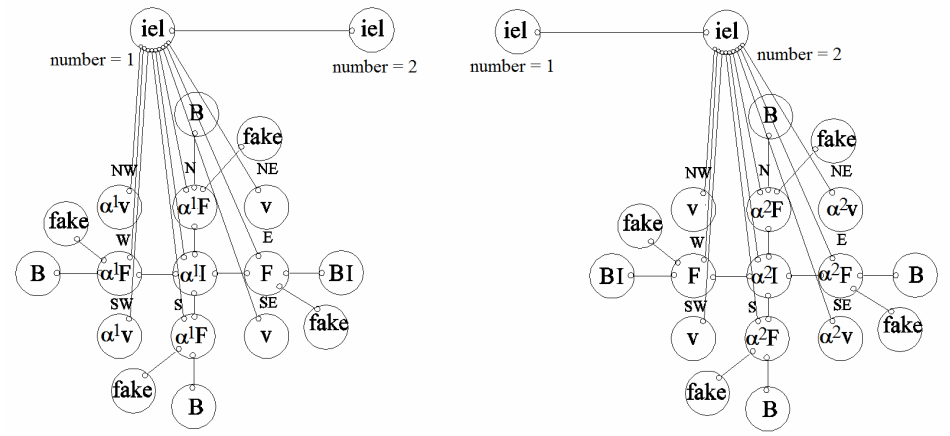


Fig 2.94. The two tasks after the aggregation of internal degrees of freedom

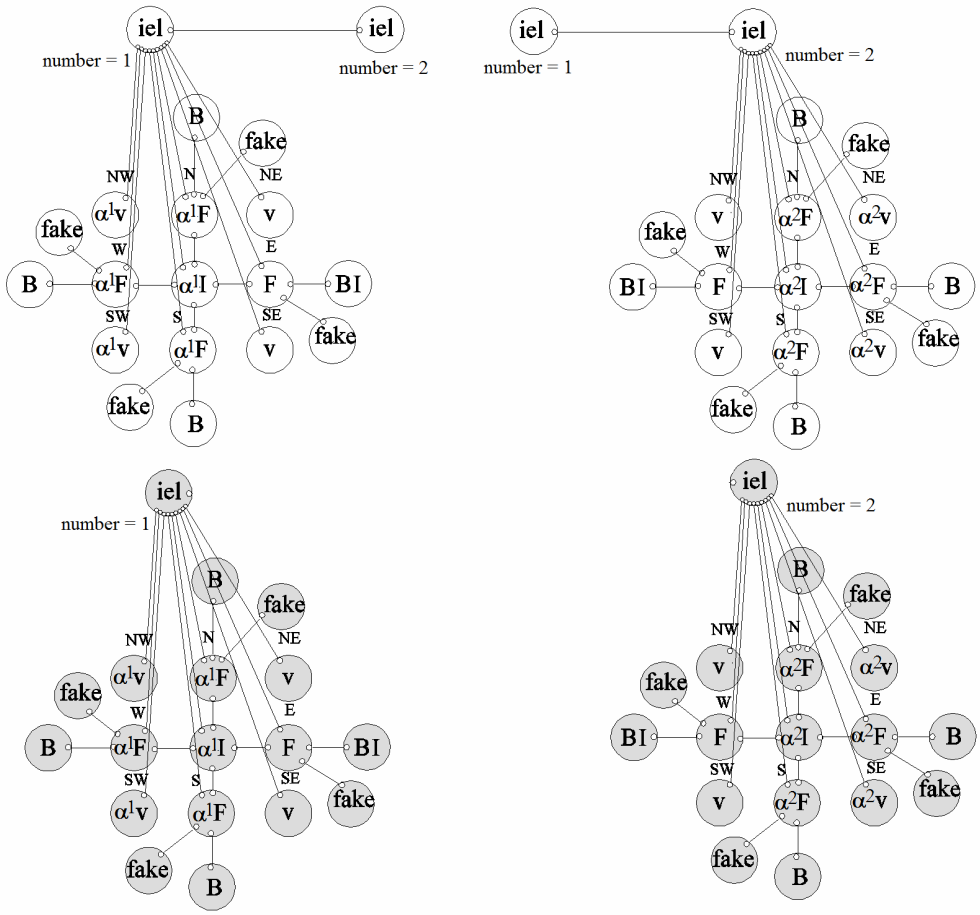


Fig. 2.95. Each task makes a copy of its initial mesh element

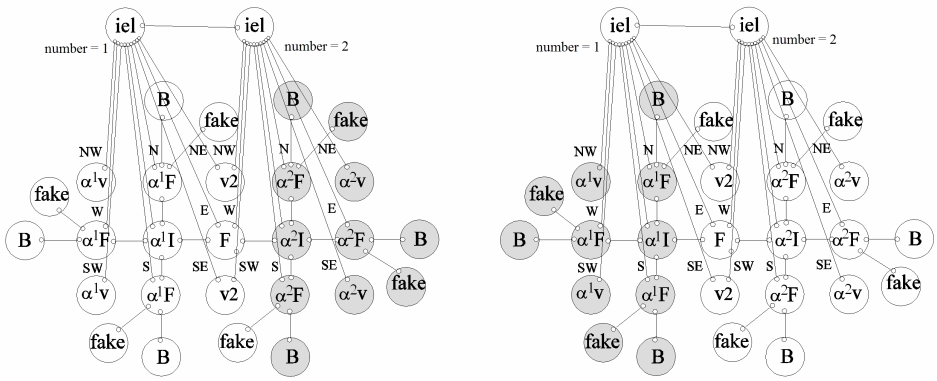


Fig. 2.96. Each task merges the received ghost sub-graphs representing adjacent initial mesh element

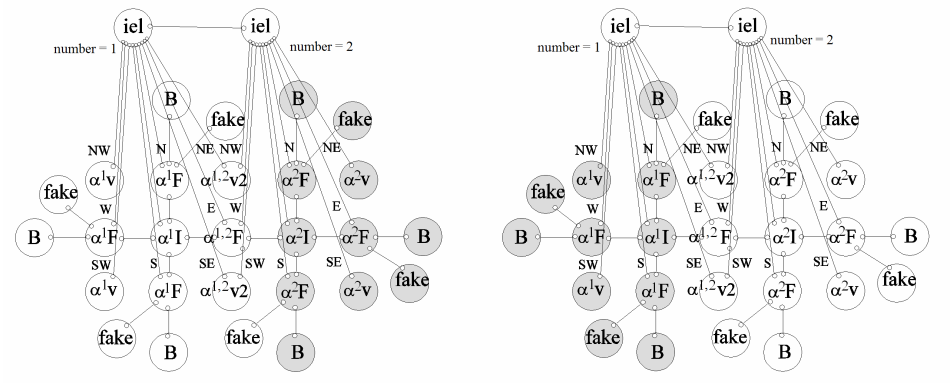


Fig. 2.97. Each task aggregates the common edge contribution to its frontal matrix

Now, each task executes the productions **(P eliminate interior)–(P eliminate boundary edge)^3–(P eliminate corner vertex)^2** on its graph. The degrees of freedom assigned to the element interiors and boundary edges are eliminated from both frontal matrices. This is illustrated in Figure 2.98. As a result, each matrix contains a local contribution to the common edge problem.

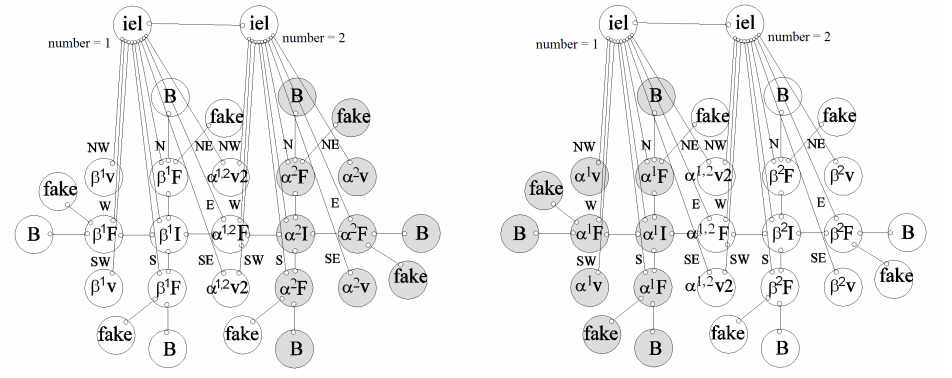


Fig 2.98. Each task eliminates the element interior from its frontal matrix

Now, we should formulate and solve the common interface problem. It is necessary to send the common interface contribution from one task to the other. The second task sends the matrix contribution to the first task, which receives the matrix and merges both contributions. Then, the common interface problem is solved by the first task by executing the productions **(P eliminate edge)-(P eliminate shared vertices)^2**. This sequence is illustrated in Figures 2.99 and 2.100. Finally, all tasks execute analogous backward substitutions (which follow the reverse execution pattern, and are omitted for simplicity) and remove the ghost elements.

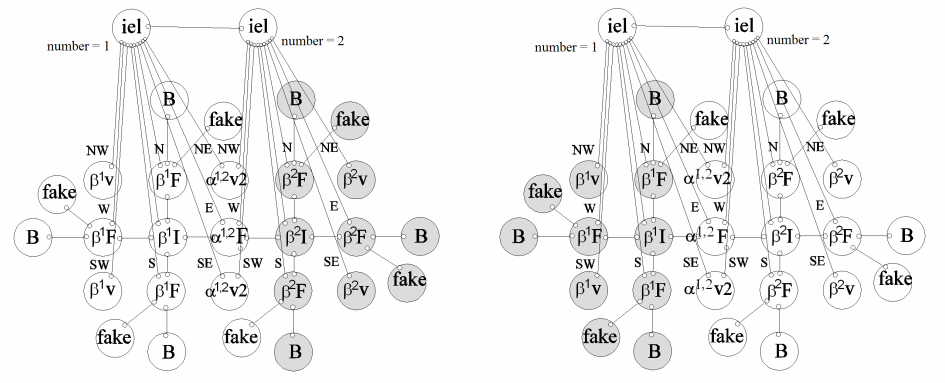


Fig 2.99. The second task sents its matrix contribution to the first task

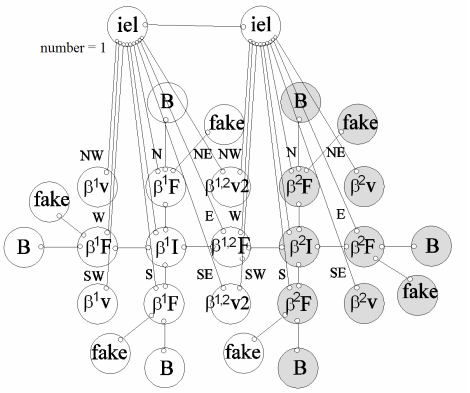


Fig 2.100. Solution of common interface problem performed by the first task

2.2.3. *h* refinement

The *h* refinement algorithm is executed either during the global *hp* refinement, where all finite elements are broken into four new elements, or after the selection of the optimal refinement, where only some selected elements are broken, which sometimes requires a refinement of some additional elements in order to fullfill the 1-irregularity rule. It is assumed that there are several tasks, each of them with a sub-graph representing a single initial mesh element, possibly refined into several smaller finite elements.

There are two steps of the *h* refinement algorithm. In the first step the virtual refinements are executed, in order to enforce the 1-irregularity rule, by attributing finite elements to be broken. On the level of tasks, the algorithm requires some additional communication between tasks assigned to adjacent initial mesh elements. The control diagram executed by all tasks for the *h* refinement procedure is presented in Figure 2.101.

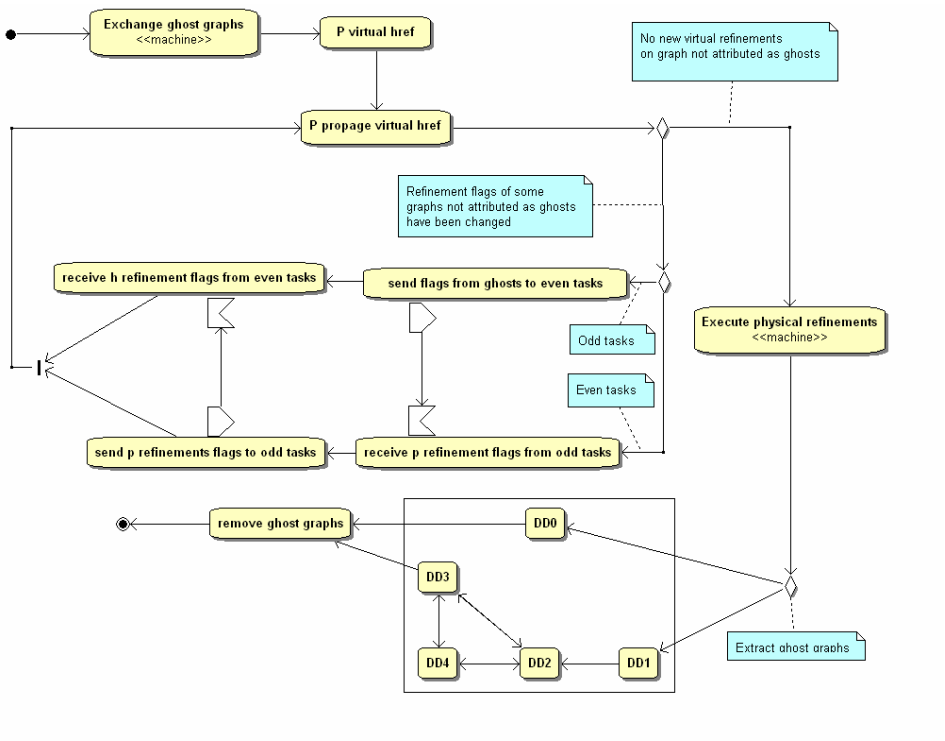


Fig 2.101. Control diagram for *h* refinement executed on the level of tasks

First, the tasks exchange graphs with adjacent tasks, attribute them as ghost graphs, and merge with local graphs. The procedure is called "Exchange ghost graphs" and follows a similar pattern as the procedure introduced at the beginning of Section 2.2.2. All the atomic tasks and communications executed during the exchange of ghost graphs are summarized in the control diagram presented in Figure 2.102.

After the ghost elements have been exchanged, the virtual *h* refinements are executed. The productions **(P virtual href)** are executed by all tasks for graph nodes representing the element interiors which are to be broken. Note that some initial mesh elements may have been already broken, and there may be several finite elements in a single task. In case of the global *hp* refinement, the production is executed for all elements. In case of the optimal *h* refinement, the production is executed only for the selected graph vertices (this aspect of the *h* refinement will be explained later in Section 2.2.6).

Next, the production **(P propagate virtual href)** is executed in order to attribute some additional graph vertices representing the element interiors which should be also broken to enforce the 1-irregularity rule.

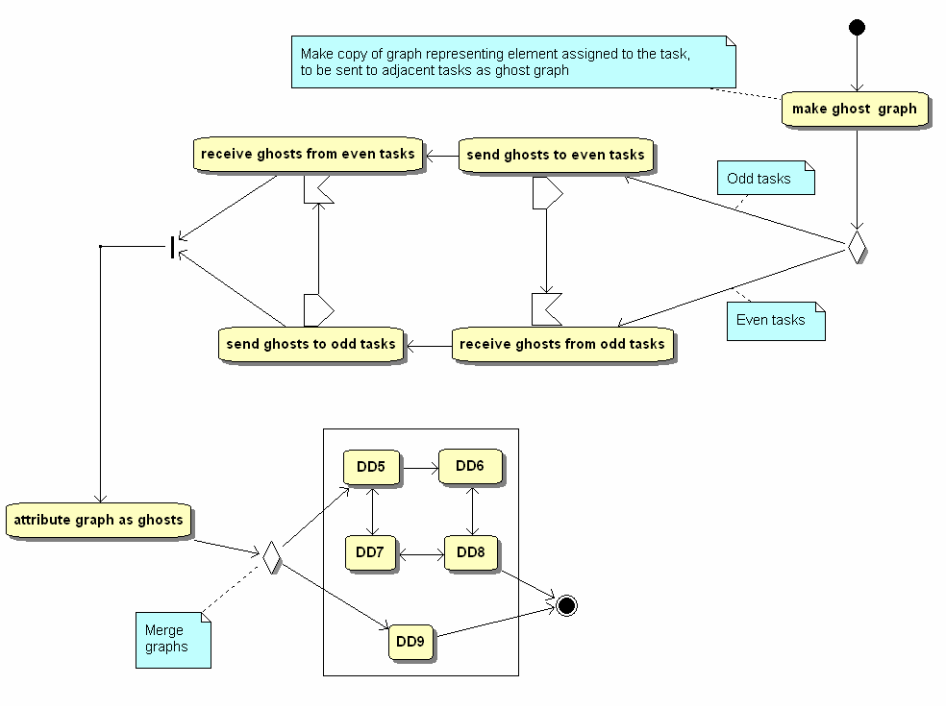


Fig 2.102. Control diagram for the exchange of ghost graphs

After all virtual refinement flags have been set for all tasks, each task sends its virtual refinement flags to the adjacent tasks, to attribute a corresponding ghost graph. The communication is managed on the control diagram by the predicates of applicability with odd and even tasks. After the exchange of the virtual refinements, the productions **(P propagate virtual href)** are executed again, to propagate virtual refinement flags from ghost graphs into the graph representing the initial mesh element assigned to the task. The procedure must be repeated until there are no changes in the non-ghost graphs attributes.

Finally, the physical *h* refinements are executed, which is summarized on the control diagram presented in Figure 2.103. The last step is to remove the ghost graphs from all tasks.

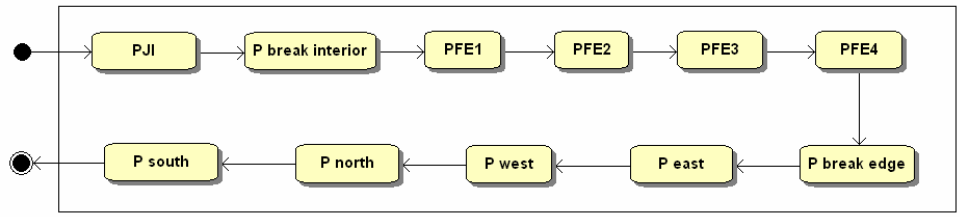


Fig 2.103. Control diagram for the execution of physical *h* refinements

2.2.4. *p* refinement

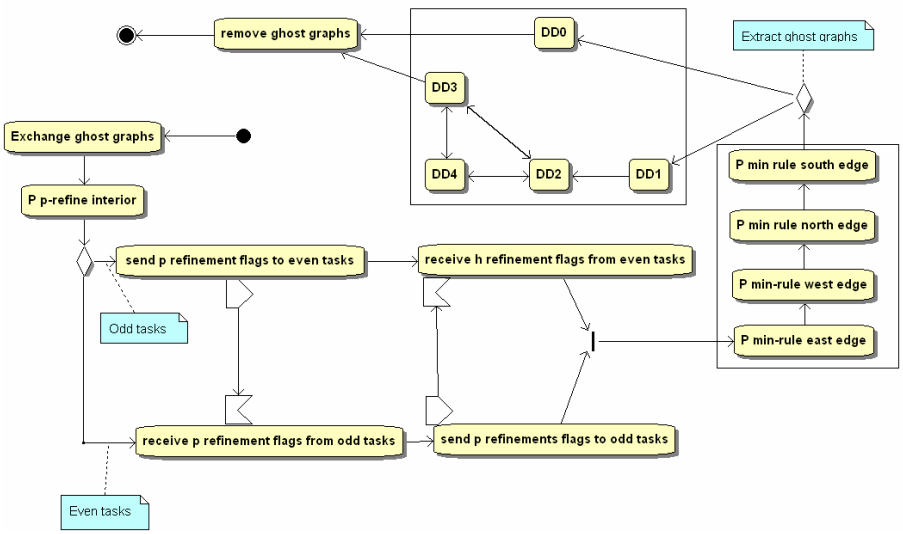


Fig 2.104. Control diagram for the execution of *p* refinements

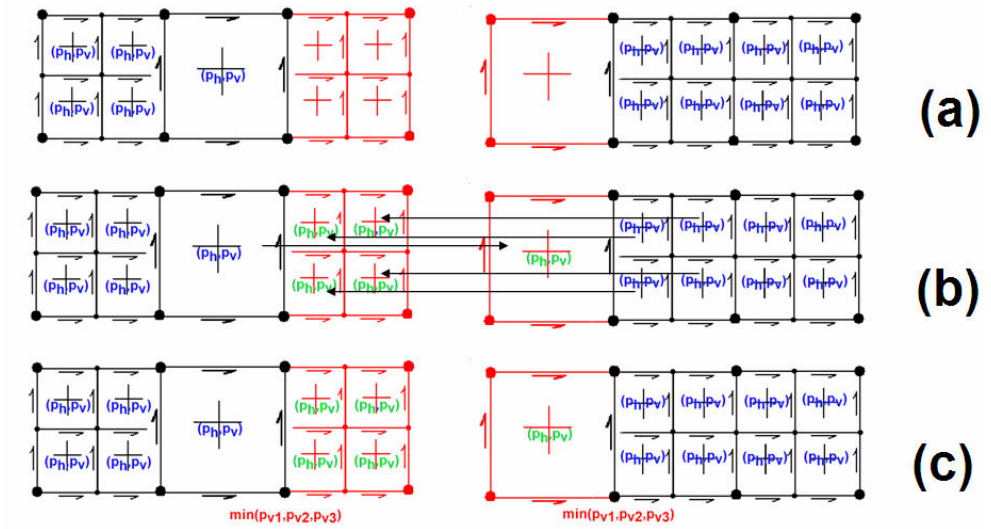


Fig.2.105. Illustration of parallel *p* refinement algorithm

The execution of the *p* refinement procedure is very similar to the execution of the *h* refinement procedure, as presented in Figure 2.104. The ghost graphs are exchanged and the graph vertices representing element interiors which are to be *p* refined are attributed by the productions **(P p-refined interior)**. The *p* refinement flags are sent to two adjacent

graphs, to be used for the ghost graphs. Then, a sequence of graph grammar productions responsible for the enforcement of the minimum rule for edges is executed. The procedure is illustrated in Figure 2.105.

2.2.5. Solution of the fine mesh problem

The coarse mesh solver algorithm is generalized to the case of the refined mesh. There are two control diagrams, presented in Figure 2.106 and 2.107. The first control diagram describes the process of the aggregation, while the second control diagram describes the process of the elimination, both working now on the level of tasks.

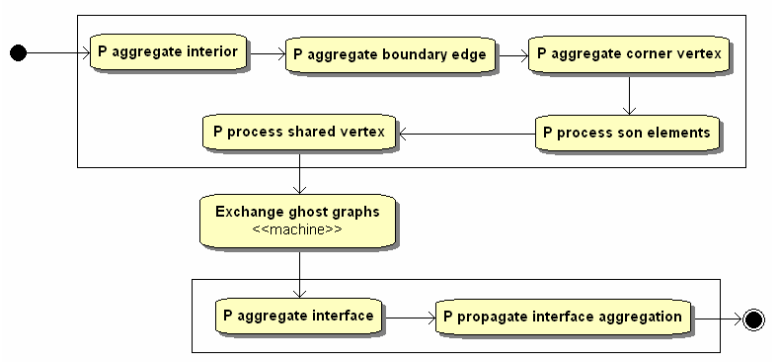


Fig 2.106. Control diagram for the aggregation on fine mesh

The aggregation and the elimination on the fine mesh follow a similar pattern as in case of the coarse mesh solver. The control diagram expresses the execution of the solver algorithm on the elimination tree introduced in Section 2.1.5. The contributions to the Schur complements are sent from even to odd tasks. The tasks that have sent the Schur complements are finished. The tasks that have received the Schur complements merge them with their local contributions and execute the partial forward elimination of the fully assembled degrees of freedom. The process is repeated until there is only one task with the common interface problem fully assembled. Finally, the common interface problem is solved and the backward substitutions are executed, following the reverse pattern (for the sake of simplicity this is omitted in our presentation).

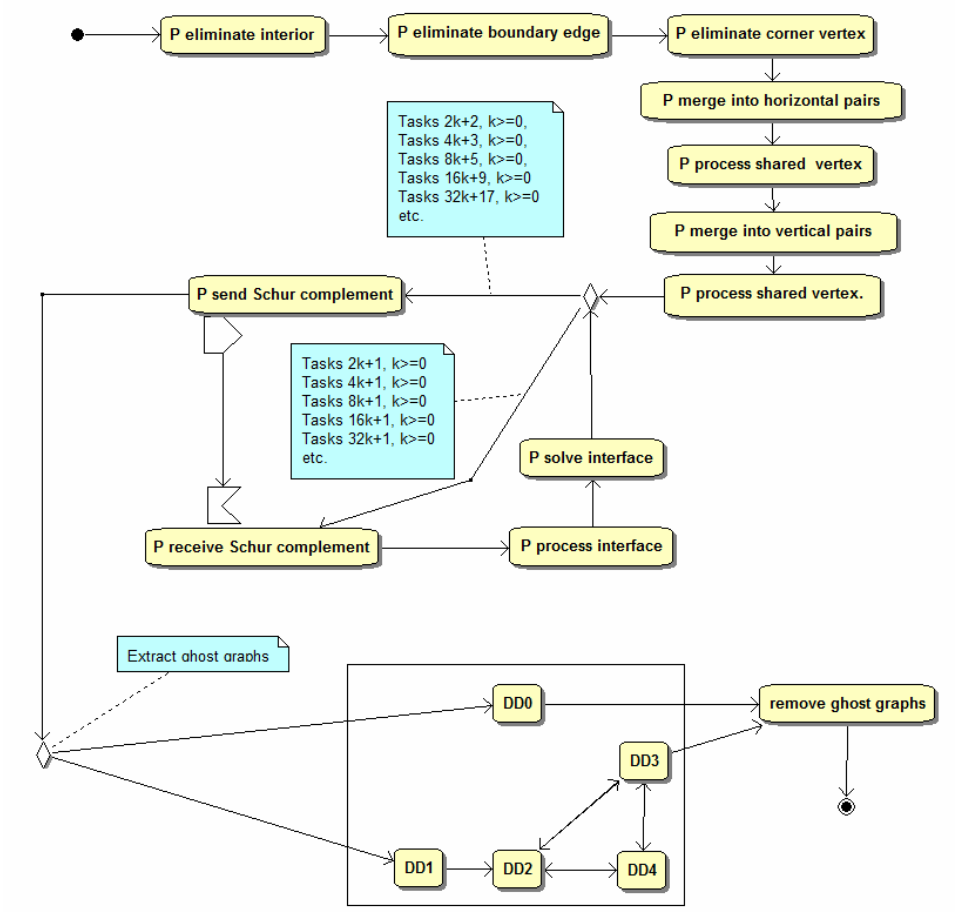


Fig 2.107. Control diagram for the elimination on fine mesh

2.2.6. Selection of the optimal refinements

After both the coarse and the fine mesh problems have been solved, it is necessary to select the optimal refinements on the level of tasks, which is summarized in the control diagram presented in Figure 2.108. This control diagram is similar to the diagram presented in Figure 2.79 in Section 2.1.5 for the level of atomic tasks. This time the selection of the optimal refinements is performed locally by each task, on its local graph. The control diagram includes also the execution of the selected *h* and *p* refinements, which follow the pattern introduced in Sections 2.2.3-2.2.4. However, this time the *h* and *p* refinements coded by the productions **(P virtual href)** and **(P p-refine interior)** are executed only for some selected elements.

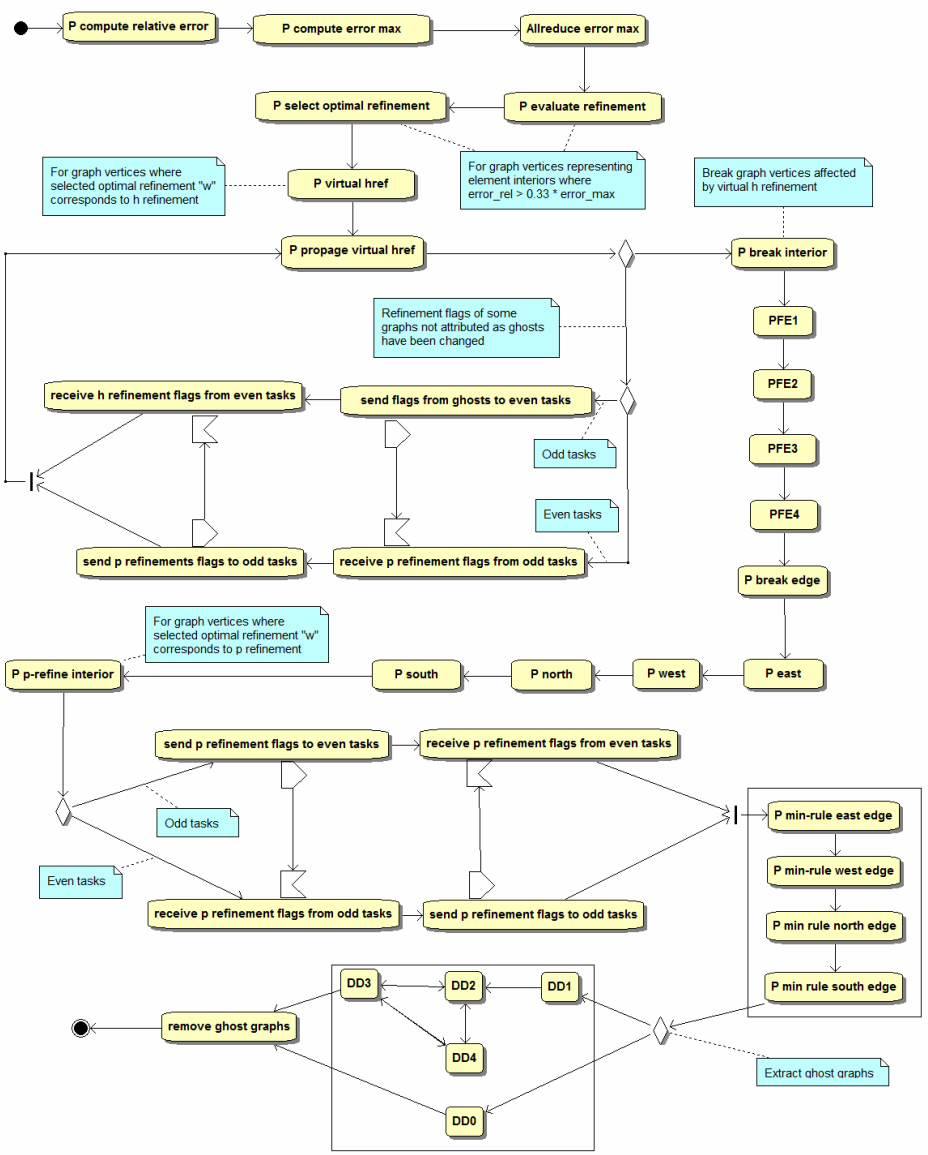


Fig 2.108. Control diagram for the selection of optimal refinements

2.3. Run-time management algorithms

This section introduces scheduling, load balancing, mesh partitioning and merging as well as mapping algorithms, defined as follows:

Definition 2.21
The *scheduling* algorithm is the method by which computational tasks defined in Section 2.2 are given access to system resources (processors). The scheduling algorithm is consists in the graph partitioning, load balancing, graph merging and mapping algorithms.

Definition 2.22
The *graph partitioning* algorithm is the method by which the graph representation of a single sub-domain is partitioned into several sub-graphs, each of them associated with a single initial mesh element.

Definition 2.23
The *load balancing* algorithm is the method by which given set of computational tasks defined in Section 2.2, with assigned estimations of the computational cost, is agglomerated into several almost equally loaded groups called sub-domains.

Definition 2.24
The *graph merging* algorithm is the method by which several sub-graphs, each of them representing a single initial mesh element, are agglomerated together into a graph representing a single sub-domain.

Definition 2.25
The *mapping* algorithm is the method by which the sub-domains resulting from the load balancing agorithm are assigned to particular parallel machine architecture.

2.3.1. Load balancing and graph partitioning algorithms

The load balancing is performed on the level of tasks introduced in Section 2.2. In other words, the tasks defined on the level of initial mesh elements become now the grains used for the load balancing procedure. The partition of the mesh on the level of initial mesh elements is sufficient for large problems and for problems with many singularities. However, it is not suitable for small problems with a small number of singularities. On the other hand, this is not a problem, because they are not computationally expensive. This observation is described in the following remarks.

Remark 2.1. If a number of tasks defined on the level of initial mesh elements and a number of local singularities are large enough, the load balancing can be effectively performed.

Remark 2.2. The computational cost for a single task defined on the level of the initial mesh element is equal to

$$\text{computational\_cost}(iel) = \sum_{K} \left({p_h}^K + 1 \right)^3 \left({p_v}^K + 1 \right)^3 \tag{2.45}$$

for two-dimensional meshes, and

$$\text{computational\_cost}(iel) = \sum_{K} \left({p_x}^K + 1 \right)^3 \left({p_y}^K + 1 \right)^3 \left({p_z}^K + 1 \right)^3 \tag{2.46}$$

for three-dimensional meshes.

In this context, the computational cost is the total sum of the computational complexities of the integration or elimination of degrees of freedom on active elements K located inside the initial mesh element *iel*.

With the above definition of the grain and the computational cost, we can introduce now the scheduling algorithm. The scheduling algorithm is understood here as the agglomeration of tasks into so-called super-tasks, in order to assign each super-task to a single processor.

We introduce the following new graph grammar productions to estimate the computational cost for each task. The production **(P compute load)** presented in Figure 2.109, is responsible for the evaluation of the computational cost according to (2.45), for either the integration or elimination of the degrees of freedom on a single finite element.

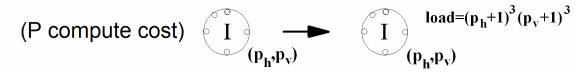


Fig 2.109. Graph grammar production estimating the computational cost of integration or elimination

The following graph grammar production **(P propagate compute load)** presented in Figure 2.110 is responsible for the recursive summing up of computational cost estimations along the refinement trees. There are several similar graph grammar productions, with capital **I** replaced by small **i**, in order to include all possible configurations of refinements on the trees.

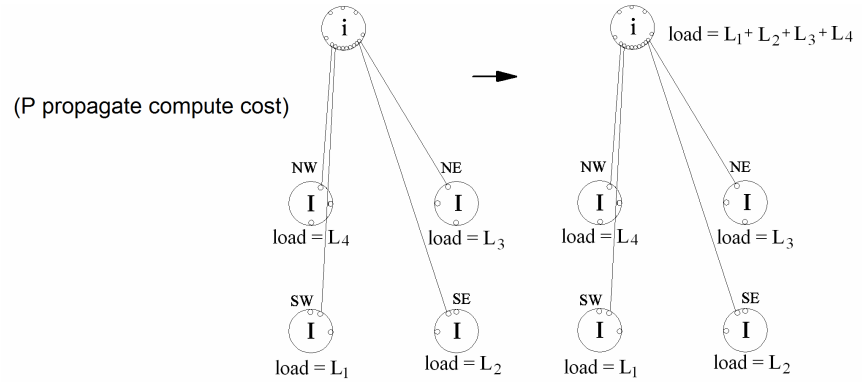


Fig 2.110. Graph grammar production propagating the estimation of computational cost along the refinement tree

Finally, we can execute the production **(P propagate compute cost)** which attributes graph vertices by means of the estimation of computational cost on the topology level representing a single initial mesh element (see Figure 2.111). After all these graph grammar productions are executed, each initial mesh element has its computational cost estimated.

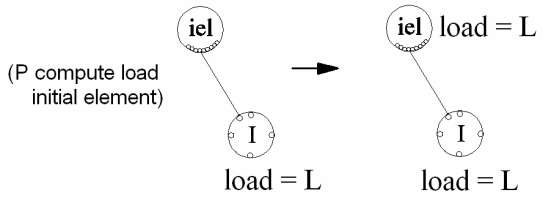


Fig 2.111. Graph grammar production attributing initial mesh element by estimating its load

The scheduling algorithm requires the following input data:

a) the global rank of the task,
b) the estimation of the computational cost for the initial mesh element assigned to the task,
c) the number of available processors.

The scheduling algorithm assigns the tasks (grains) to the processors. We assume the 1—1 relation between super-tasks and processors. There are two possible realizations of the scheduling algorithm.

In the first version of the algorithm we assume that the entire initial mesh has been generated in the first super-task, on the first processor. Then, the load balancing algorithm is executed on the first processor, and several sub-graphs representing initial mesh elements are sent to other super-tasks, assigned to other processors.

The algorithm is summarized in the following control diagrams executed by super-tasks. The first control diagram, presented in Figure 2.112, is executed only by the first super-task, which generates the entire initial mesh.

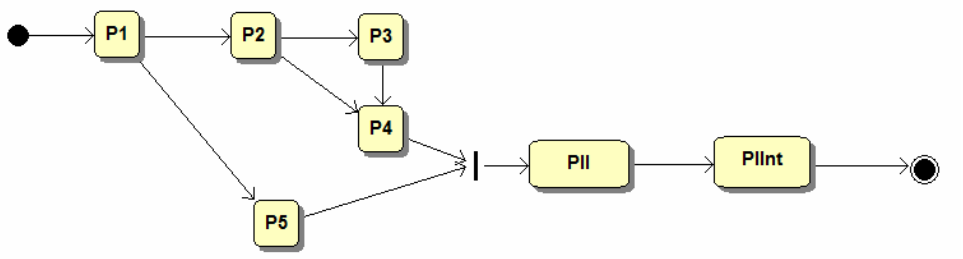


Fig 2.112. Control diagram for the first super-task, for the generation of the entire initial mesh

Next, the first task executes the newly introduced graph grammar productions for the estimation of computational cost for all initial mesh elements, as illustrated in Figure 2.113. The load balancing procedure is executed later. This is summarized in the “balance load” and “assign sub-graphs to processors” states on the control diagram. These are the two load balancing algorithms considered here:

1) Hilbert Space Filling Curve (HSFC),
2) nested dissections.

For the detailed description of these algorithms we refer to the documentation of the ZOLTAN library.

After the decision about the quasi-optimal redistribution of the graph is made, we execute several mesh partitioning productions, in order to split the graph into such number of sub-graphs that will correspond to all initial mesh elements. Those sub-graphs are later sent to the destination super-tasks, according to the decisions made by the load balancing algorithms. Thus, all other super-tasks execute the control diagram presented in Figure 2.114.

Each super-task merges all received sub-graphs into a single graph. Such realization of the scheduling algorithm is expensive from the point of view of the communication. However, as it is presented later in the numerical experiments, the amount of data exchanged during the load balancing and mesh partitioning algorithms constitutes only 10% of the communication needed while executing the fine mesh solver algorithm. The reason for this is that the Schur complement contributions exchanged between super-tasks are much larger than sub-graphs.

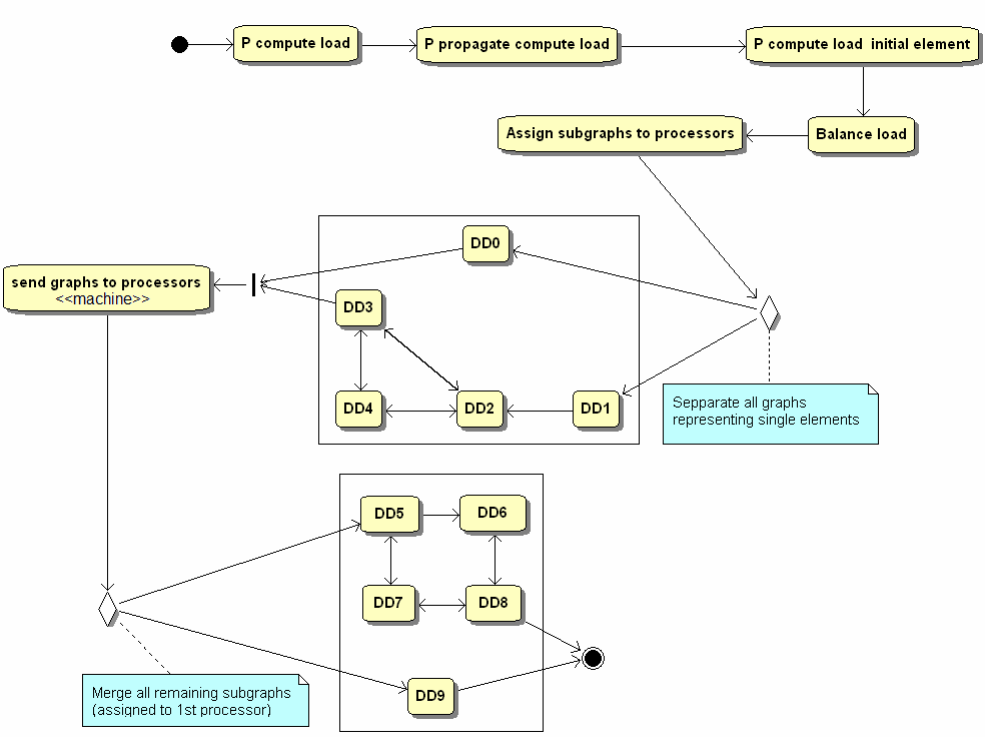


Fig 2.113. Control diagram for the first super-task, for the load balancing and mesh partitioning

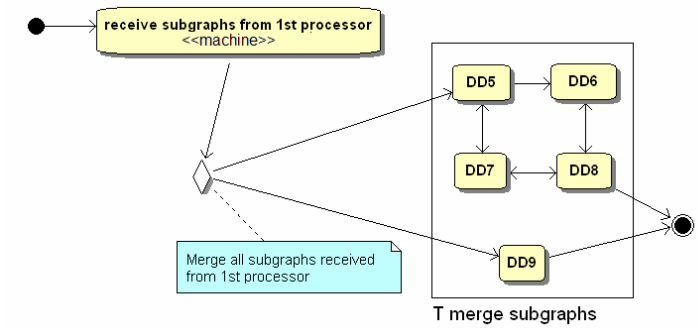


Fig 2.114. Control diagram for all but the first super-tasks, for the load balancing and mesh partitioning

After each iteration of the self-adaptive *hp*-FEM algorithm, the distribution of the computational cost may significantly change, since some elements may be refined. The graph representation of the mesh, stored in a distributed manner between super-tasks, must be redistributed now, to maintain quasi-uniform load balancing. Thus, we need to define a new scheduling algorithm. Each super-task executes the same control diagram, presented in Figure 2.115. It is assumed now that each super-task estimates the computational cost for its own graph. The data for the load balancing algorithm are provided in a distributed manner. The load balancing algorithm collects data from all super-tasks, makes decision about a new quasi-optimal distribution of the mesh, and notifies each super-task about the tasks that must be sent to other super-tasks and about tasks that must be received from other super-graphs. An example of the load balancing algorithms implemented in this way is the ZOLTAN library.

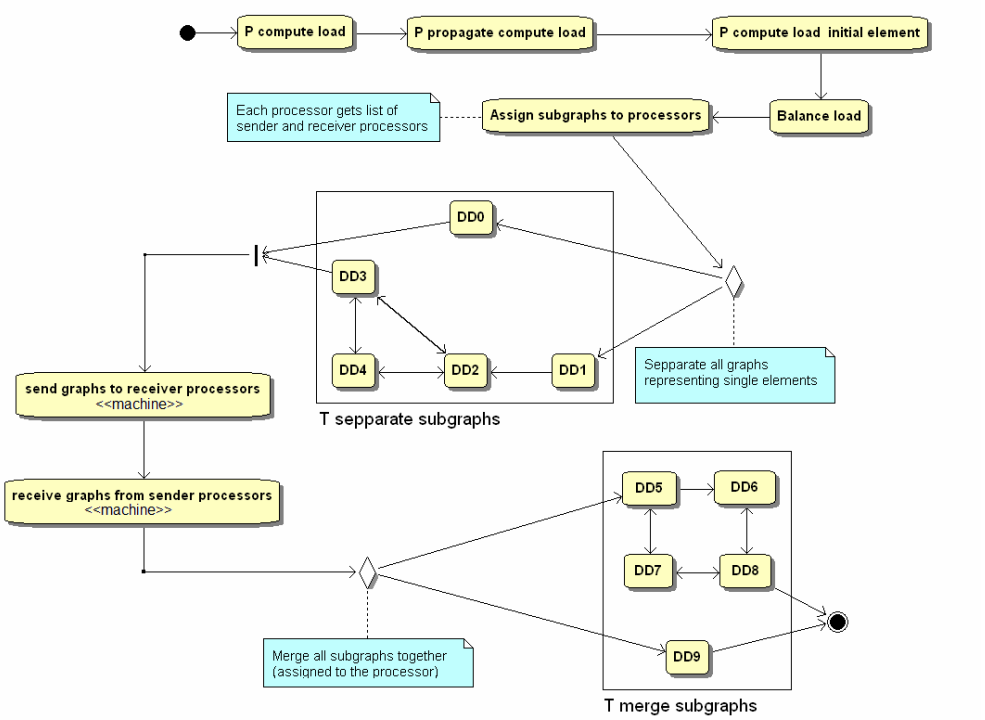


Fig 2.115. Control diagram for all super-tasks, for the mesh repartitioning

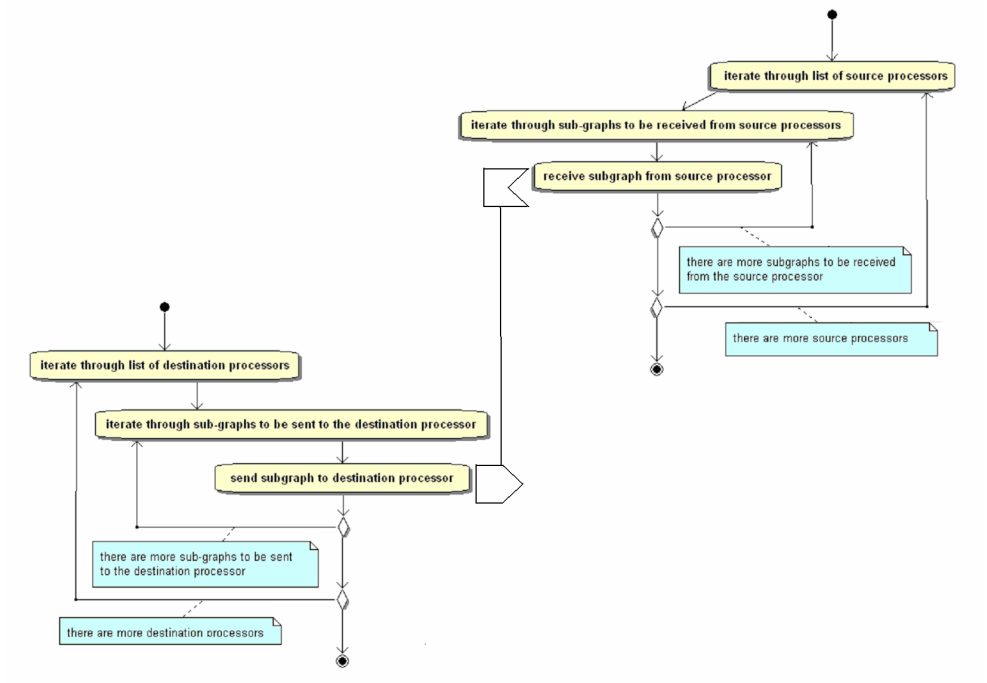


Fig 2.116. Control diagram for super-tasks,
for send/receive of graphs from source/to receiver processors

2.3.2. Mapping algorithm

Remark 2.3. The optimal mapping of computational tasks into processors is achieved when each computational task is assigned to the leaves of elimination tree, in a consecutive manner.

Proof : The computational tasks should be mapped into processors in such a way that the resulting communication bandwidth will be minimal. In general, the mapping problem is NP-complete. To solve the mapping problem in a quasi-optimal way, we have to localize these parts of the algorithm which are the most expensive ones from the point of view of the communication cost. We need to identify also the architecture of the destination parallel machine.

The most expensive part of the self-adaptive *hp*-FEM algorithm is the parallel solver. It involves multiple independent point-to-point communications. The independent communications mean that all pairs of communicating processors are separated. The messages in the solver are transmitted between separate pairs of processors (between child nodes in the elimination tree). In today's HPC clusters there are nodes of several (usually 2 to 16) processors. These nodes are interconnected by high performance switches or hierarchy of switches (e.g. Clos topology Clos 1984). Figure 2.117 presents the performance profiles for inside-node processor-to-processor communications and between-

nodes processor-to-processor communications, measured on the CHAMPION linux cluster. The inside-nodes communications are cheaper than the between-nodes communications. The effective bandwidth for concurrent messages between multiple pairs of separate processors is presented in Figure 2.118.

On the CHAMPION cluster, there are 8 processors at each node. The effective bandwidth decreases when the number of communicating processors is the multiplication of eight.

The best performance can be reached when the tasks assigned to the eliminated elements are sorted along the elimination tree leaves and assigned to consecutive processors. If for the elimination tree presented in Figure 2.119, the tasks 0-7 are agglomerated to one super-task and assigned to the first node, and the tasks 8-15 are agglomerated to another super-task and assigned to the second node, there will be only one between-nodes communication: a message between the sons of tree root □

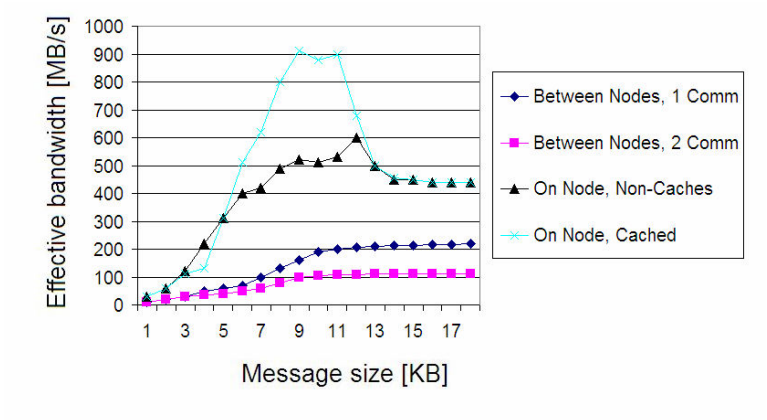


Fig.2.117. Performance profiles for dual-processor communications.
Intra-processor: On Node (Non-Cached removes data from cache before timing).
Inter-processor: Between Nodes (1 and 2 simultaneous communications)

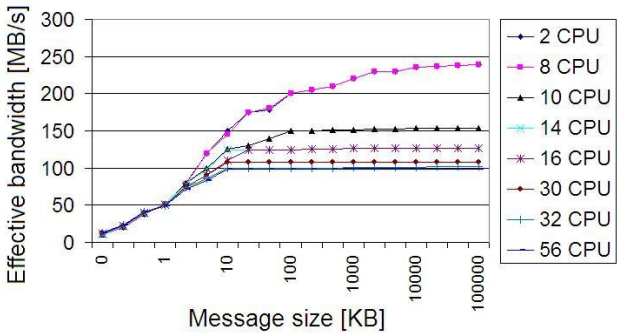


Fig.2.118. Effective bandwidth profiles for concurrent communication through a switch hierarchy

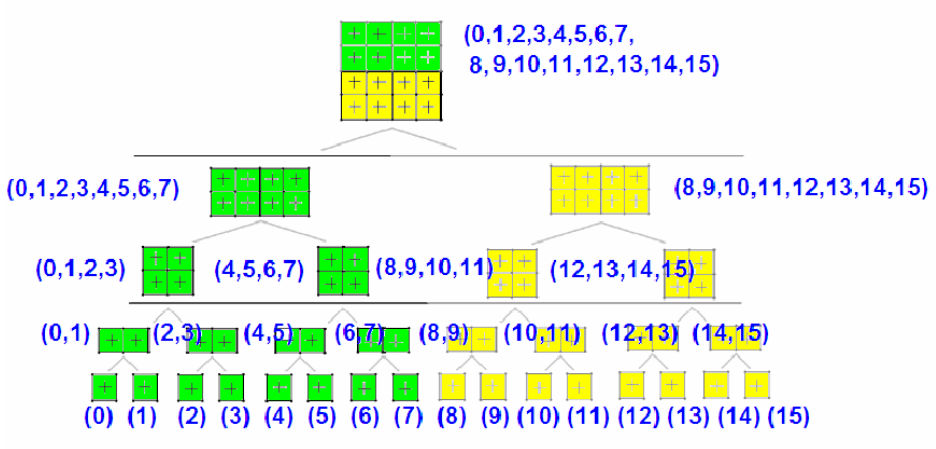


Fig.2.119. Mapping the solver elimination tree into processors

2.4. Parallel processing model with super-tasks for the self-adaptive algorithm

Finally, we have reached the level of super-tasks. Several tasks introduced by Definition 2.17 have been agglomerated by the scheduling algorithms to super-tasks, introduced by Definition 2.19. The self-adaptive *hp*-FEM algorithm must be redefined now on the level of super-tasks. All the control diagrams, previously defined for the level of tasks (see Definition 2.18), are rewritten now for the level of super-tasks (see Definition 2.20). Each super-task works now on its local graph, obtained by merging several graphs, which belong to all tasks agglomerated to a single super-task. The graph assigned to a single super-task corresponds now to many initial mesh elements. It is assumed that each super-task is assigned to a single processor. We define also the communication channels between super-tasks. Since the assumption about the linear structure of initial mesh elements is still valid, the communication channels are established again between consecutive super-tasks. This is our simplification, made in order to reduce the number of technical details in this presentation.

Figures 2.120-2.128 present the control diagrams updated from the level of tasks to the level of super-tasks. In fact, the diagrams are quite similar to those defined on the task level. The predicates of applicability in these control diagrams use global identifiers for super-tasks. It is assumed that the global identifier for a super-task is a number of processor assigned to this task, since there is the 1—1 relation between super-tasks and processors. The super-tasks work on larger graphs than the tasks. Thus, the number of operations executed by a super-task on its local graph is larger than the number of operations executed by a task on its local graph.

The main difference is in the coarse and fine mesh solvers. The solvers on the level of super tasks, presented in Figures 2.4.1, 2.4.7 and 2.4.8, must eliminate the degrees of freedom assigned to common edges of the adjacent finite elements of the super-tasks. Thus, the first part of the control diagram contains more productions to be executed, but the other

part of the control diagram, responsible for the elimination of the interface problem, is actually the same as for the control diagram on the level of tasks.

The number of operations performed by super-tasks is usually larger than the number of operations performed by tasks, since the super-tasks possess larger graphs. A super-task must evaluate the computational cost for all sub-graphs representing initial mesh elements assigned to this super-task.

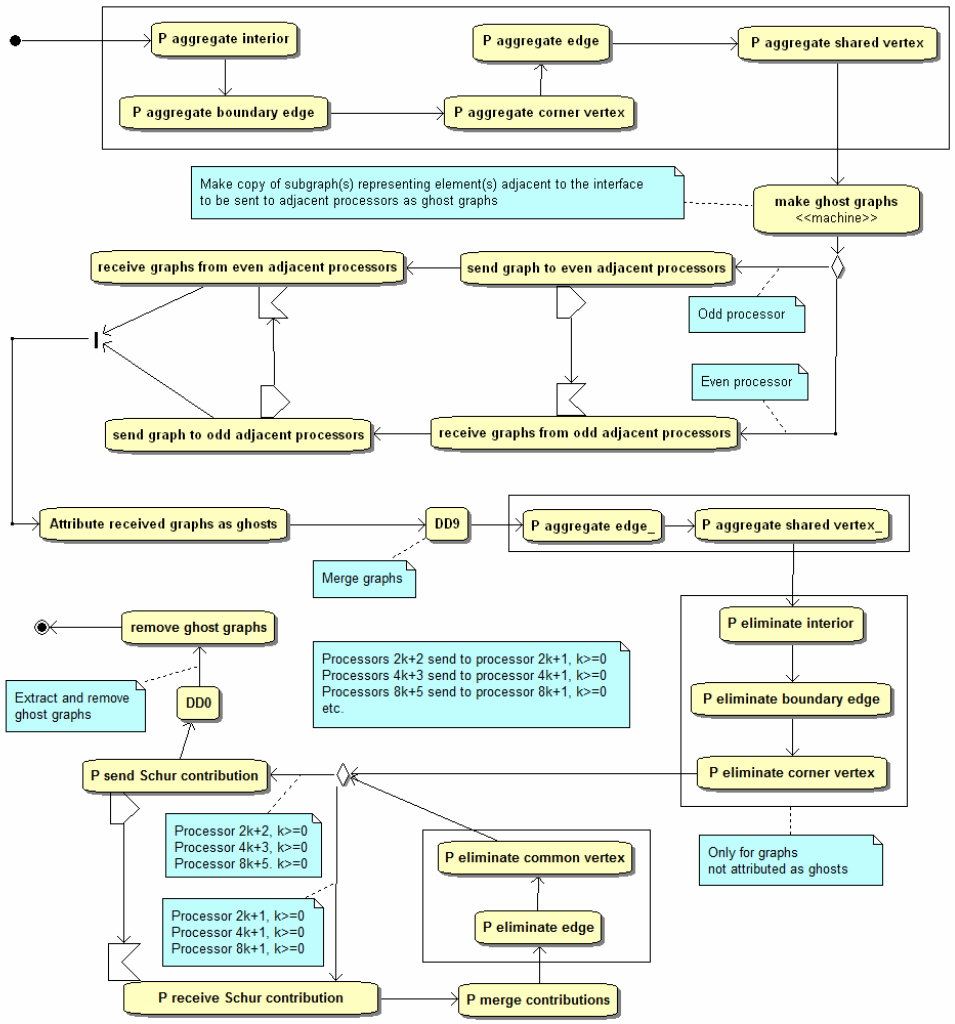


Fig.2.120. Control diagram for the execution of the coarse mesh solver on the level of super-tasks

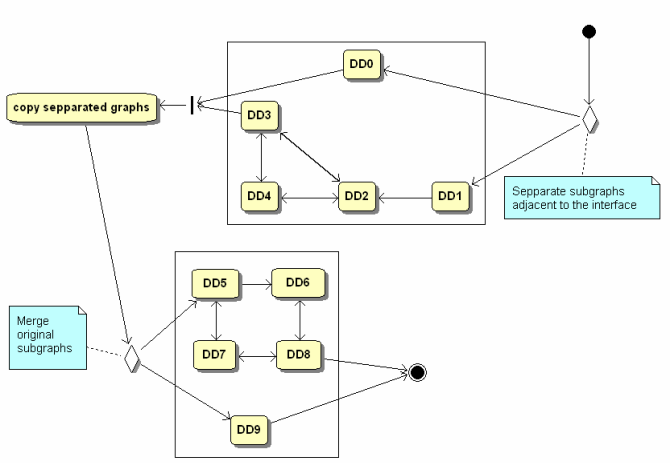


Fig.2.121. Control diagram for the extraction of ghost elements on the level of super-tasks (corresponding to the "make ghost graphs" state from the control diagram presented in Figure 2.4.1)

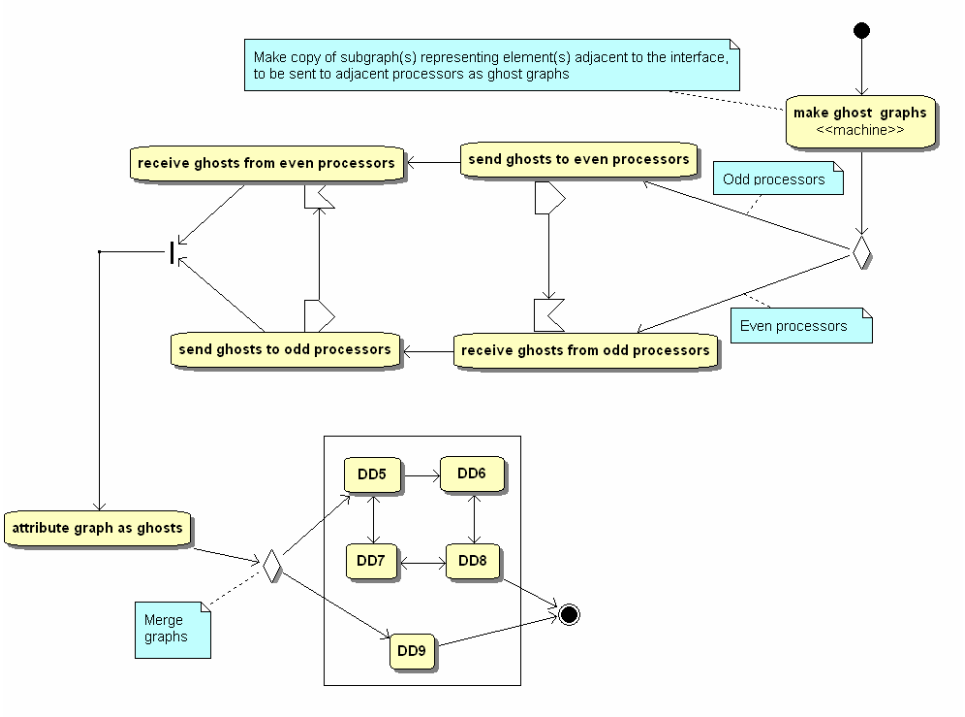


Fig.2.122. Control diagram for the exchange of ghost elements on the level of super-tasks

The control diagrams for the super-tasks are almost the same as for the tasks. Each state from the control diagram represents a production that is executed on a super-task graph. The left-hand side of a production identifies the sub-graph to which the production can be applied. In case of the super-tasks, each production is executed in the concurrent way, in different places on the super-task graph.

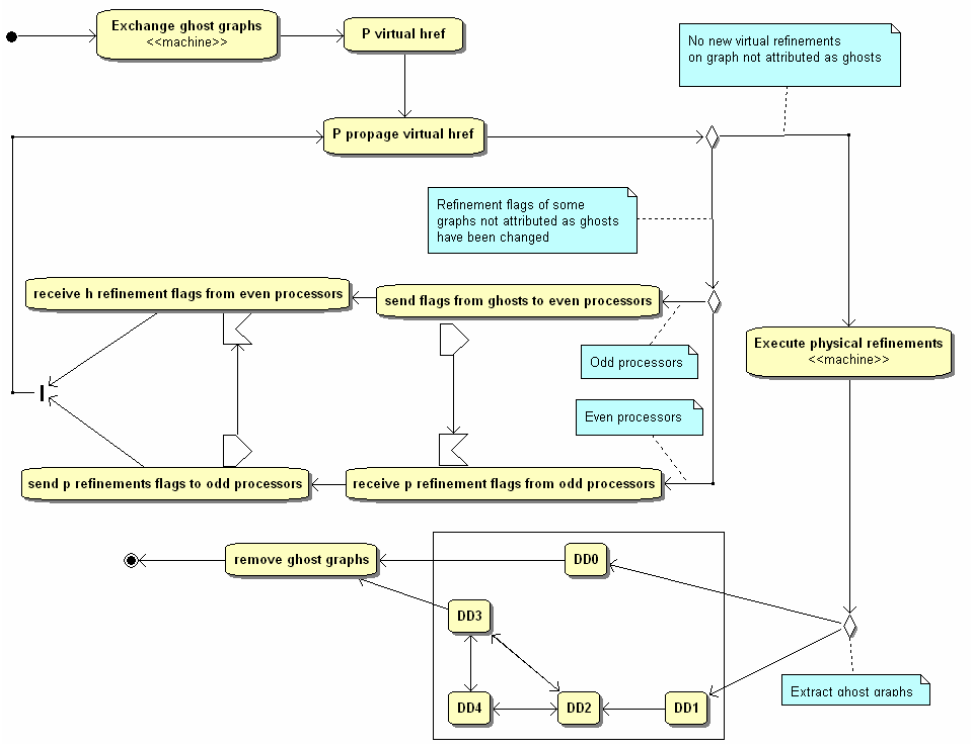


Fig.2.123. Control diagram for the execution of *h* refinements on the level of super-tasks

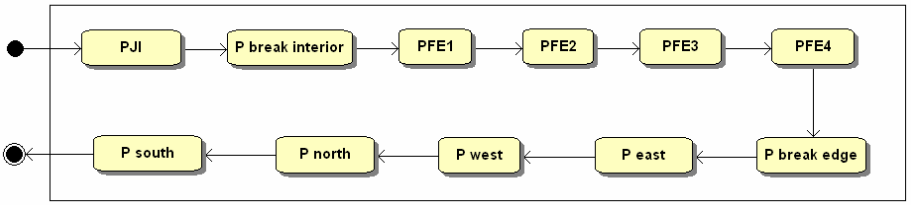


Fig.2.124. Control diagram for the execution of physical *h* refinements on the level of super-tasks

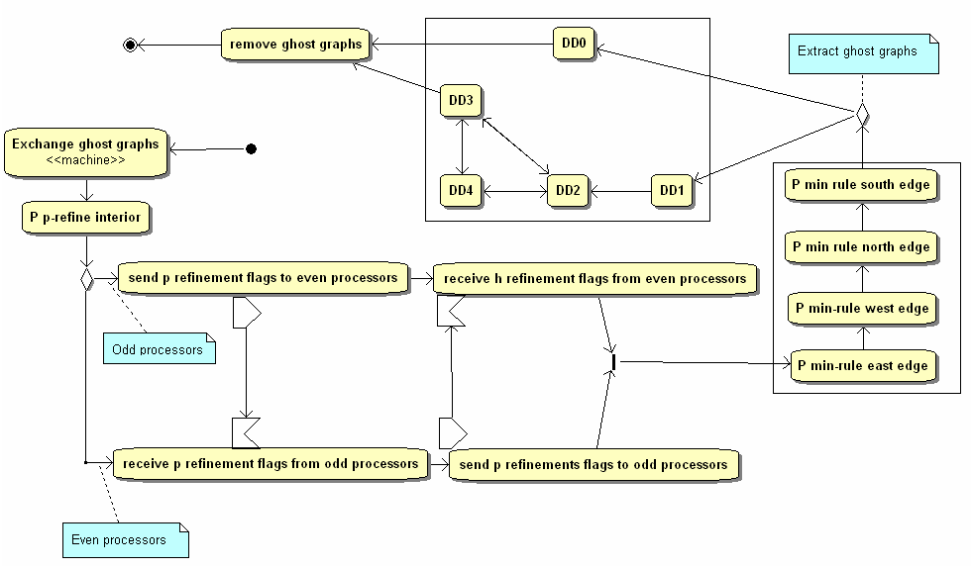


Fig.2.125. Control diagram for the execution of *p* refinements on the level of super-tasks

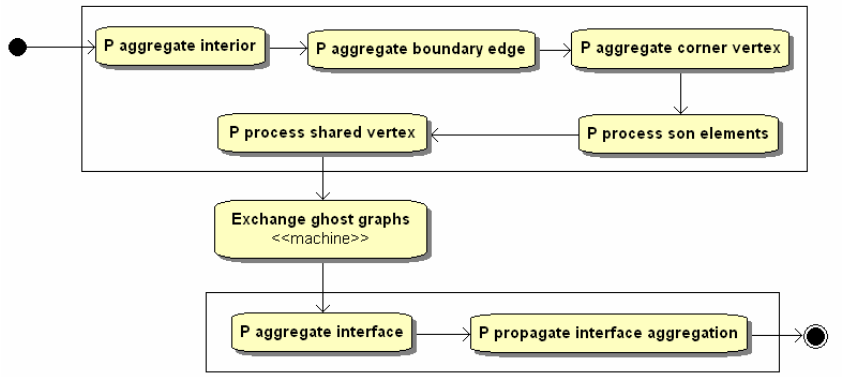


Fig.2.126. Control diagram for the agglomeration on the fine mesh on the level of super-tasks

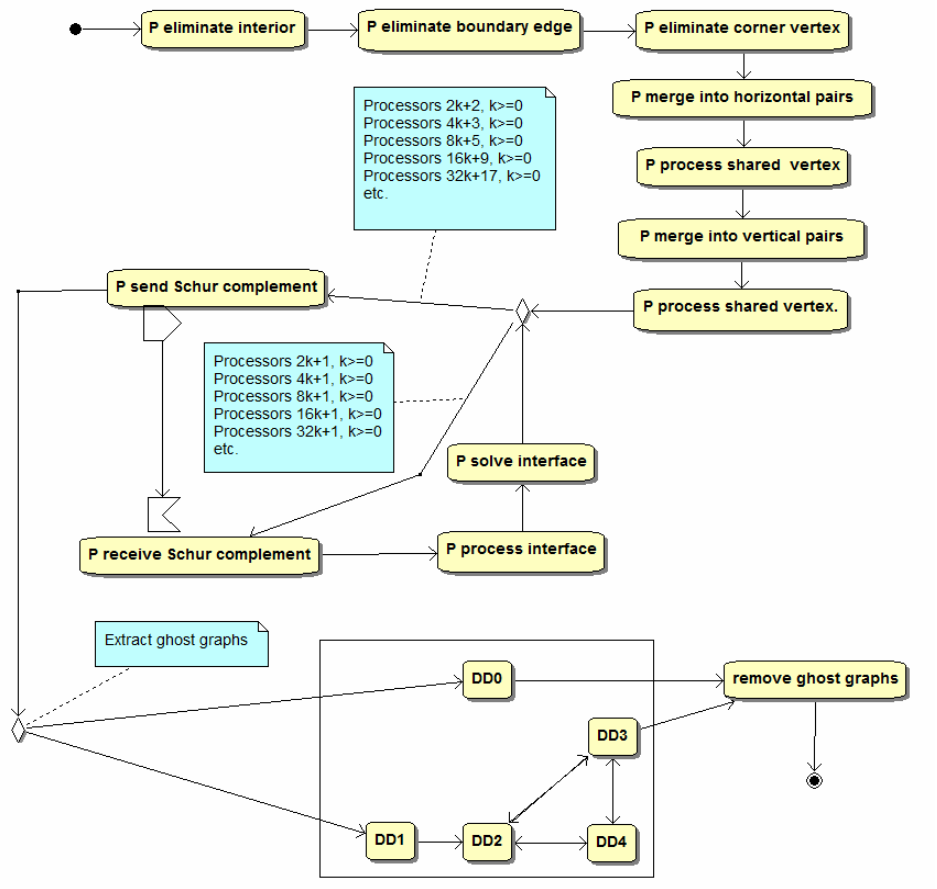


Fig.2.127. Control diagram for the elimination on the fine mesh on the level of super-tasks

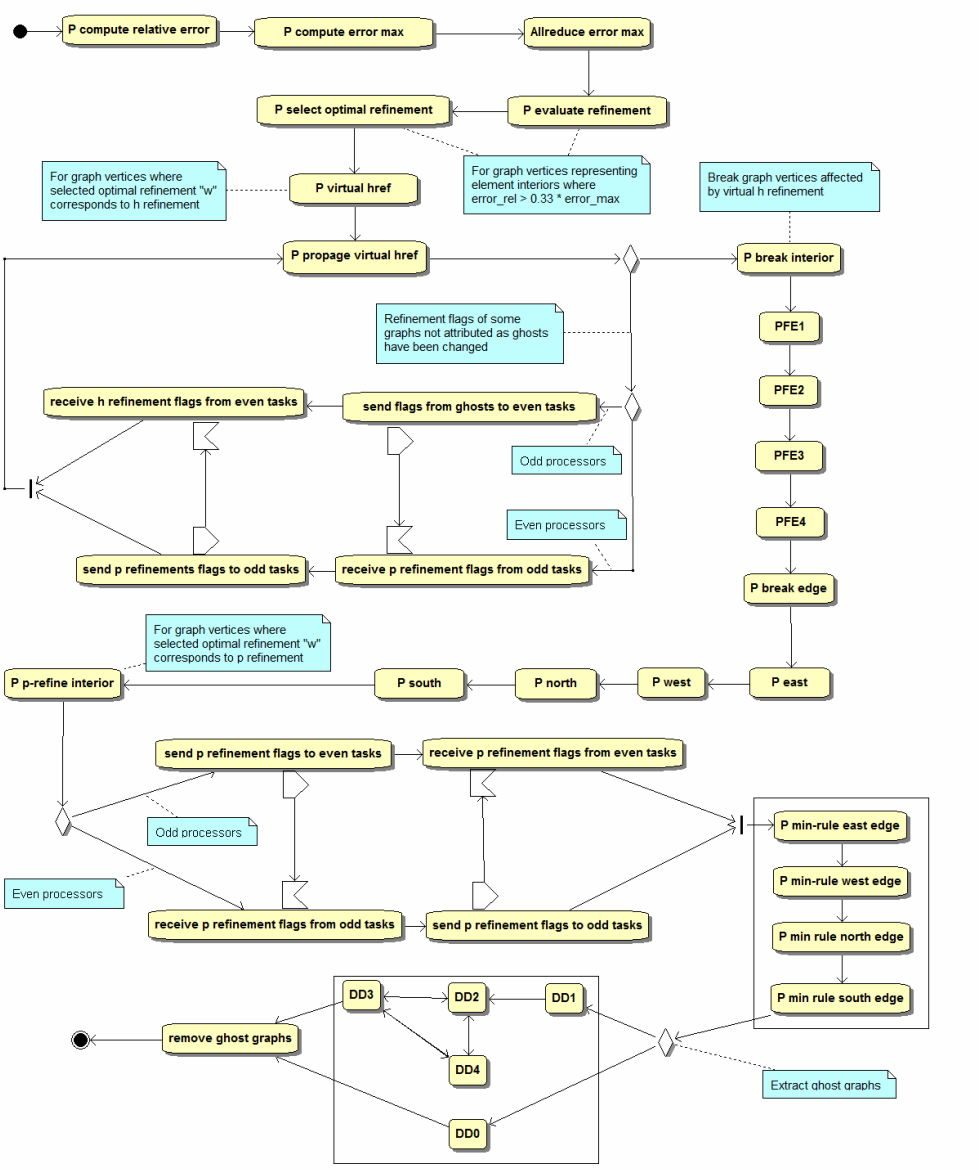


Fig.2.128. Control diagram for the selection of optimal refinements on the level of super-tasks

2.5. Theoretical analysis of the computational and communication complexities

The theoretical analysis presented in this section refers to the self-adaptive *hp*-FEM algorithm defined on the level of super-tasks. We focus on the following parts of the algorithm.

The first section is related to the computational and communication times of the mesh partitioning algorithms. In particular, we analyse the computational and communication times for the exchange of ghost elements. The computational and communication times for the load balancing algorithms are not included here, since we employ the load balancing algorithms, such as HSFC and nested dissection algorithms, and their computational and communication times have been already described by Bauer, Patra 2004 and Khaira, Miller, Sheffler, 1992.

In the following sections, we examine the computational and communication times of the mesh adaptation and solver algorithms.

Finally, the times are evaluated for the proposed extension of the solver algorithm, with the reutilization of partial LU factorizations.

2.5.1. Mesh partitioning algorithms

Remark 2.4.
The time spend by *i*-th super-task on the computation of the computational cost (2.45-2.46) over the graph representation of the sub-domain is

$$T^{i}_{comp1}\left(N_{elem}\right)=O\left(N_{elem}\right) \tag{2.47}$$

where N_{elem} is a number of active elements located on the initial mesh elements of a super-task. This can be roughly estimated as a number of *h* refinements multiplied by a number of created son elements, that is

$$T^{i}_{comp1}\left(N^{init}_{elem},N^{ref}\right)=O\left(N^{init}_{elem}\sum_{j=1}^{N^{ref}}4^{j-1}\right) \tag{2.48}$$

where N^{init}_{elem} is a number of initial mesh elements of a super-task, N^{ref} is the depth of the refinement tree that is less or equal to a number of iterations of the self-adaptive *hp*-FEM algorithm.

Proof: The computational time concerns the execution of the productions **(P compute load), (P propagate compute load)** and **(P compute load initial element)**, presented in Figure 2.115 □

Remark 2.5.
The time spend by *i*-th super-task on the communication in the parallel algorithm for estimation of computational cost for a super-task is equal to zero

$$T^{i}_{comm1}\left(N_{elem}\right)=0 \tag{2.49}$$

Proof: This is a straightforward observation, since the algorithm is purely local □

Remark 2.6.
The time spend by *i*-th super-task on the computation in the parallel algorithm of the separation of all graphs representing a single initial mesh element is

$$T^{\mathrm{i}}_{\mathrm{comp}\,2}\left(\mathrm{N}^{\mathrm{init}}_{\mathrm{elem}}, \mathrm{N}^{\mathrm{ref}}\right) = O\left(\mathrm{N}^{\mathrm{init}}_{\mathrm{elem}} 4 \sum_{\mathrm{i}=1}^{\mathrm{N}^{\mathrm{ref}}} 2^{\mathrm{i}-1}\right) \tag{2.50}$$

where $\mathrm{N}^{\mathrm{init}}_{\mathrm{elem}}$ is a number of initial mesh elements of a super-task, 4 stands for four edges of an initial mesh element, and the last term $\sum_{\mathrm{i}=1}^{\mathrm{N}^{\mathrm{ref}}} 2^{\mathrm{i}-1}$ is equal to a number of edges to be separated on the refinement tree, with its root located on one of the four edges of the initial mesh element.

Proof: The computational time concerns the execution of the productions **(DD0)-(DD4)**, presented in Figure 2.115 □

Remark 2.7.
The time spend by *i*-th super-task on the communication in the parallel algorithm of the mesh repartitioning is

$$T^{\mathrm{i}}_{\mathrm{comm}\,3}\left(\mathrm{N}_{\mathrm{elem}}\right) = O\left(\mathrm{N}_{\mathrm{elem}}\right) \tag{2.51}$$

where $\mathrm{N}_{\mathrm{elem}}$ is a number of active elements located on the initial mesh elements of a super-task. This can be roughly estimated by (2.48).

Proof: This estimation corresponds to the execution of the control diagram presented in Figure 2.116, for a linear sequence of initial mesh elements, where all super-tasks are connected by the loop-like communication channels. In such case, the quasi-optimal partition can be preserved by exchanging the sub-graphs related to the initial mesh elements with two adjacent super-tasks. In a more general case (Bauer, Patra 2004) if the HSFC load balancing algorithm is applied, it is possible to create a communication loop along the space filling curve. In this case, we can also preserve a quasi-uniform load balancing, by exchanging data between two adjacent super-tasks along this curve. Thus, on average, each super-task sends and receives the amount of data equal to the size of its graph □

Remark 2.8.
The time spend by *i*-th super-task on the computation in the mesh repartitioning parallel algorithm is

$$T^{\mathrm{i}}_{\mathrm{comp}\,3}\left(\mathrm{N}_{\mathrm{elem}}\right) = O\left(\mathrm{N}_{\mathrm{elem}}\right) \tag{2.52}$$

Proof: The computations related to the mesh repartitioning algorithm, presented in Figures 2.115 and 2.116 involve packing and unpacking of sub-graphs to be exchanged, thus the computation time is of the same order as the communication time □

Remark 2.9.
The time spend by *i*-th super-task on the communication in the parallel algorithm of exchange of the ghost elements is

$$T^{\mathrm{i}}_{\mathrm{comm}\,4}\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right)=O\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right) \tag{2.53}$$

where $\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}$ is a number of active elements of a super task, located on the initial mesh elements adjacent to the interface, which can be roughly estimated as a number of *h* refinements multiplied by a number of created son elements, that is

$$\mathrm{N}^{\mathrm{int}}_{elem}=O\left(\mathrm{N}^{\mathrm{int,\,init}}_{\mathrm{elem}}\sum_{\mathrm{i}=1}^{\mathrm{N}^{\mathrm{ref}}}4^{\mathrm{i}\text{-}1}\right) \tag{2.54}$$

where $\mathrm{N}^{\mathrm{int,\,init}}_{\mathrm{elem}}$ is a number of initial mesh elements adjacent to the interface and $\mathrm{N}^{\mathrm{ref}}$ is the depth of the refinement tree that is less or equal to a number of iterations of the self-adaptive *hp*-FEM algorithm.

Remark 2.10.
The time spend by *i*-th super-task on the computations in the parallel algorithm of exchange of the ghost elements is

$$T^{\mathrm{i}}_{\mathrm{comp}\,4}\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right)=O\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right) \tag{2.55}$$

Proof: The computational time related to the exchange of ghost elements, presented in Figures 2.121 and 2.122 involve packing and unpacking of the sub-graphs to be exchanged, thus the computational time is of the same order as the communication time □

2.5.2. Mesh adaptation algorithms

Remark 2.11.
The time spend by *i*-th super-task on the communication in the parallel algorithm of *h* refinements is

$$T^{\mathrm{i}}_{\mathrm{comm}\,5}\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right)=O\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right) \tag{2.56}$$

Proof: This corresponds to the control diagram presented in Figure 2.123. The virtual refinement flags are exchanged between the adjacent super-tasks. The number of exchanged virtual refinements corresponds to the number of graph vertices representing the interiors of active elements located on the ghost sub-graphs adjacent to the interface □

Remark 2.12.
The time spend by *i*-th super-task on the computation in the parallel algorithm of *h* refinements is

$$T^{i}_{\mathrm{comp}\,5}\left(\mathrm{N}_{\mathrm{elem}}\right)=O\left(\mathrm{N}_{\mathrm{elem}}\right) \tag{2.57}$$

Proof: This corresponds to the execution of the productions **(P virtual href)** and **(P propagate virtual href)**, presented on the control diagram in Figure 2.123. The execution of these productions attributes the graph vertices representing active element interiors. All active elements are affected by the algorithm in order to check the applicability of 1-irregularity rule □

Remark 2.13.
The time spend by *i*-th super-task on the communication in the parallel algorithm of *p* refinements is

$$T^{\mathrm{i}}_{\mathrm{comm}\,6}\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right)=O\left(\mathrm{N}^{\mathrm{int}}_{\mathrm{elem}}\right) \tag{2.58}$$

Proof: This corresponds to the control diagram presented in Figure 2.125. The *p* refinement flags are exchanged between the adjacent super-tasks. A number of exchanged *p* refinements corresponds to a number of graph vertices representing the interiors of active elements located on the ghost sub-graphs adjacent to the interface □

Remark 2.14.
The time spend by *i*-th super-task on the computation in the parallel algorithm of *p* refinements is

$$T^{\mathrm{i}}_{\mathrm{comp}\,6}\left(\mathrm{N}_{\mathrm{elem}}\right)=O\left(\mathrm{N}_{\mathrm{elem}}\right) \tag{2.59}$$

Proof: This corresponds to the execution of the productions **(P p-refine interior)**, presented on the control diagram in Figure 2.125. The execution of these productions attributes the graph vertices representing active element interiors. All active elements are affected by the algorithm □

2.5.3. Solver algorithms

In this section, we desrcibe the theoretical analysis of the efficiency of the parallel recursive solver algorithm described in Appendix D.4, expressed by graph grammar productions introduced in Sections 2.1.4 and 2.4. It is assumed that the solver is executed on a square 2D finite element mesh with $N^{total}_{elem}=2^{2n}$ finite elements, as presented in Figure 2.129 for n=2. In other words, now we take into consideration a more general computational mesh, not only a row of initial mesh elements, as in case of the graph grammar model.

We assume that the order of approximation in the interior of all elements is equal to (p_h, p_v). The total number of the degrees of freedom in such an element is equal to

$$\mathrm{nrdof}=(p_h+1)(p_v+1)+2(p_h+1)+2(p_v+1) \tag{2.60}$$

We assume that $p_h=p_v=p$, to simplify the theoretical analysis. Thus, a number of degrees of freedom is

$$\mathrm{nrdof}=(p+1)^2+4(p+1)=O\left(p^2\right) \tag{2.61}$$

a number of degrees of freedom on element interior is

$$\mathrm{interior\ nrdof}=(p+1)^2=O\left(p^2\right) \tag{2.62}$$

a number of degrees of freedom on interface is

$$\mathrm{interface\ nrdof}=4(p+1)=O(p) \tag{2.63}$$

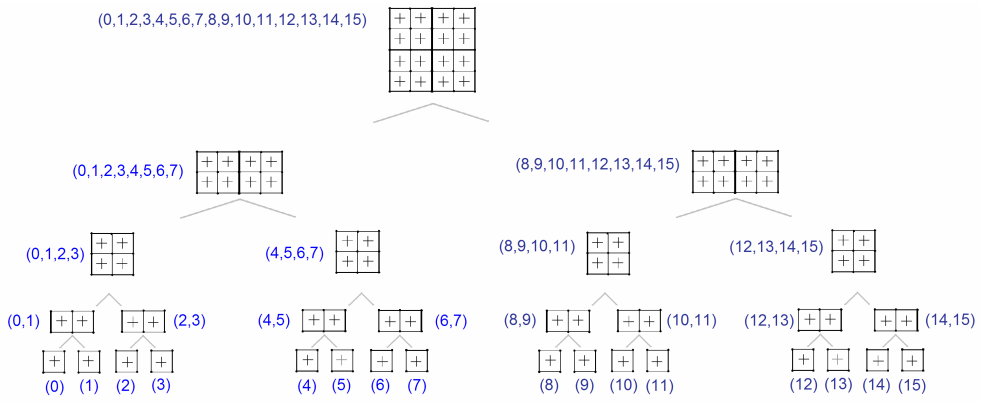


Fig.2.129. Step-by-step elimination of degrees of freedom on a square domain.
The processor numbers are in brackets, and the grey lines denote the point-to-point communications.

Remark 2.15.
The time spend on the computation in the sequential solver algorithm is

$$T_{\text{comp solver}}\left(p, N_{elem}^{total}\right) = O\left(N_{elem}^{total} p^6 + N_{elem}^{total\,3/2} p^3\right) = O\left(2^{2n} p^6 + 2^{3n} p^3\right)$$

where $N_{elem}^{total} = 2^{2n}$ is the total number of elements, and p is the polynomial order of approximation assumed to be uniform over the mesh.

Proof: On the level of elimination tree leaves, the solver eliminates the interior degrees of freedom together with the element edges located on the boundary. The time spend on computation in this operation for a single element can be estimated as

$$\left(\text{interface nrdof} + \text{interface nrdof}\right)^2 \text{interior nrdof}\; \text{t}_{\text{comp}} \tag{2.64}$$

for internal element,

$$\left(\text{interior nrdof} + \text{interface nrdof}\right)^2 \left(\text{interior nrdof} + \text{edge nrdof}\right) \text{t}_{\text{comp}} \tag{2.65}$$

for an element adjacent to the boundary,

$$\left(\text{interior nrdof} + \text{interface nrdof}\right)^2 \left(\text{interior nrdof} + 2\,\text{edge nrdof}\right) \text{t}_{\text{comp}} \tag{2.66}$$

for a corner element. Here t_{comp} denotes the time of execution of a single arithmetic operation. In all these cases, the computation time is $O\left(p^6\right)$. There are $2^n\, 2^n = 2^{2n}$ such elements.

The next step is to join the elements into pairs. The common edges are eliminated. The computation time of the elimination of a common edge for a pair of elements is

$$\left((2\cdot 1 + 2\cdot 2 + 1)\text{edge nrdof}\right)^2 (2\cdot 1 + 2\cdot 2)\text{edge nrdof}\; \text{t}_{\text{comp}} \tag{2.67}$$

since there are 2 external vertical edges, 4 external horizontal edges, and one common edge to be eliminated. There are $2^n \big/ 2 = 2^{2n-1}$ such pairs of elements.

Then, the elements are joined into sets of four, and two common edges are eliminated. The computation time is

$$\left((4\cdot 2+2)\text{edge nrdof}\right)^2(4\cdot 2)\text{edge nrdof}\ \mathrm{t}_{\text{comp}} \tag{2.68}$$

since there are 4 external horizontal edges and 4 external vertical edges to be eliminated. All internal edges have been already eliminated, and there are 2 remaining common edges to be eliminated. There are $2^{n-1}2^{n-1}=2^{2n-2}$ such sets of elements.

The procedure illustrated in Fig.2.129 is repeated until all elements are grouped in one set only.

The total computation time is of the order of $2^n p^6$ (elimination of elements interiors) plus $2^{2n-1}\left((2+4+1)p\right)^2(2+4)p$ (elimination of one common edge for a pair of elements) plus $2^{2n-2}\left((4\cdot 2+2)p\right)^2(4\cdot 2)p$ (elimination of two common edges for a set of four elements) plus $2^{2n-3}\left((2\cdot 2+2\cdot 4+2)p\right)^2(2\cdot 2+2\cdot 4)p$ (elimination of two common edges for a set of eight elements), plus further contributions. This can be expressed by the following sum:

$$\begin{aligned}&2^{2n}p^6+2^{2n-1}\left((2+4+1)p\right)^2(2+4)p+\\&\sum_{k=1}^{n}\left[2^{2n-2k-1}\left(2\cdot 2^{k+1}+3\cdot 2^k\right)^2 p^2\left(2\cdot 2^{k+1}+2\cdot 2^k\right)p+2^{2n-2k}\left(5\cdot 2^k\right)^2 p^2\left(4\cdot 2^k\right)p\right]\\&\approx O\left(2^{2n}p^6+2^{3n}p^3\right)\end{aligned} \tag{2.69}$$

□

Remark 2.16.
The maximum for all processors P of the time spend on the computation in the parallel solver algorithm is

$$T^{\max}_{\text{comp solver}}\left(n,N^{total}_{elem},P\right)=O\left(\frac{N^{total}_{elem}}{P}p^6+\left(\frac{N^{total}_{elem}}{P}\right)^{3/2}p^3+N^{total\,3/2}_{elem}p^3\right)=$$

$$O\left(2^{2(n-m)}p^6+2^{3(n-m)}p^3+2^{3n}p^3\right)$$

where $N^{total}_{elem}=2^{2n}$ is the total number of elements, $P=2^{2m}$ is the number of processors, and p is the polynomial order of approximation assumed to be uniform over the mesh.

Proof: To estimate the computational time, let us assume that the number of processors is $P=2^{2m}$. Each processor performs the elimination on its part of the elimination tree. If $m=n$, then number of processors is equal to the number ef elements and each processor is assigned to a single leaf (see Figure 2.129). If $m<n$, then each processor is assigned to the equal branch of the tree. The computational time of this operation for $P=2^{2m}$ processors is

$$2^{2(n-m)}p^6+2^{3(n-m)}p^3\ \mathrm{t}_{\text{comp}} \tag{2.70}$$

After this step, each processor performs elimination up to some level of the elimination tree, e.g. processor 4 in Figure 2.129 performs eliminations up to the third level

of the tree. Only processor 0 performs all eliminations up to the root of the tree. The total execution time spend by processor 0 on these operations is of the order of

$$\sum_{k=m+1}^{n}\left[\left(2\cdot 2^{2k+1}+2\cdot 2^{2k}+2^{k}\right)^{2} p^{2}\left(2\cdot 2^{k+1}+2\cdot 2^{k}\right)p+2^{2n-2k}\left(5\cdot 2^{k}\right)^{2} p^{2}\left(4\cdot 2^{k}\right)p\right] \quad (2.71)$$
$$\approx O\left(2^{3n}p^{3}\right)$$

□

Remark 2.17.
The maximum for all processors *P* time spend on the communication in the parallel solver algorithm is

$$T^{\max}_{\text{comm solver}}\left(N^{total}_{elem},p\right)=O\left(N^{total}_{elem}p^{2}\right)=O\left(2^{2n}p^{2}\right)$$

where $N^{total}_{elem}=2^{2n}$ is the total number of elements, and p is the polynomial order of approximation assumed to be uniform over the mesh.

Proof: The communication involves no more than $2\left(n-m+1\right)$ parallel point to point communications where Schur complement contributions are sent (compare Figure 2.129). This can be estimated as

$$\sum_{k=m+1}^{n} 2\cdot\left(t_{\text{startup}}+t_{\text{comm}}\left(2^{k}p\right)^{2}\right) \quad (2.72)$$

since the size of every contribution matrix is $2^{k}\cdot p$. Here t_{startup} is the message startup time (the time required to initiate the communication) and t_{comm} is the time of transfer of a double precision value. We neglect t_{startup} since the amount of transfered data is large. This communication time is of the order of $O\left(2^{2n}p^{2}\right)$. The size of each message can be reduced by sending only a list of non-zero matrix entries □

Remark 2.18.

$$T^{i}_{\text{comp solver}}+T^{i}_{\text{comm solver}}+T^{i}_{\text{idle solver}}=T^{\max}_{\text{comp solver}}+T^{\max}_{\text{comm solver}}\quad\forall i$$

where $T^{i}_{\text{comp solver}}$ is the computational time for *i*-th processor, $T^{i}_{\text{comm time}}$ is the communication time for *i*-th processor, $T^{i}_{\text{idle solver}}$ is the idle time for *i*-th processor, $T^{\max}_{\text{comp solver}}$ is the maximum for all processors *P* computational time computed in Remark 2.16, $T^{\max}_{\text{comm solver}}$ is the maximum for all processors *P* communication time computed in Remark 2.17 and *i* is the processor number.

Proof: According to Figure 2.129 the maximum computational and communication time is equal to the execution time for processor 0. All other processors perform computations only on a part of the elimination tree, send the resulting Schur complement contribution to some

other processor, and then become idle. Thus, the execution time plus the communication time plus idle time for all processor is equal to $T^{\max}_{\text{comp solver}} + T^{\max}_{\text{comm solver}}$ □

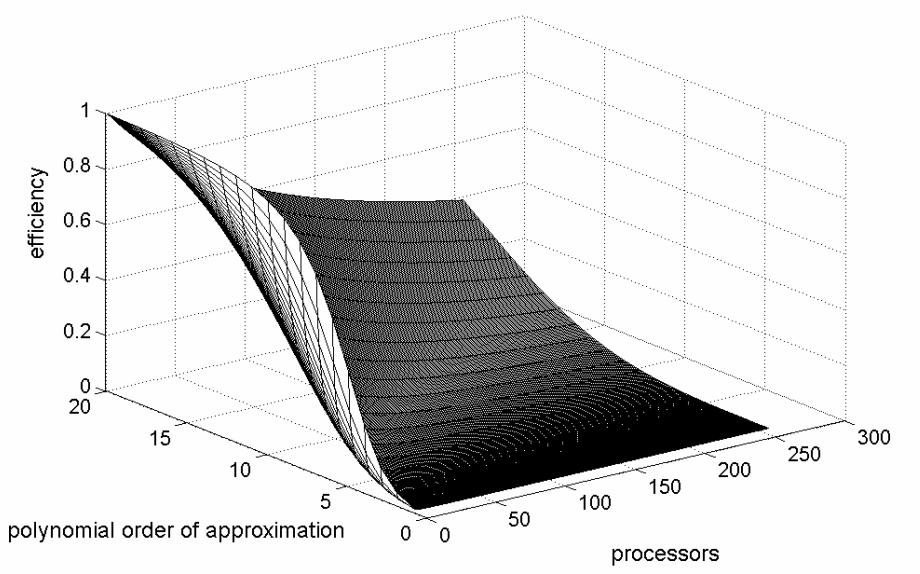


Fig.2.130. Efficiency of the parallel solver as a function of number of processors and polynomial order of approximation

Remark 2.19.
The relative efficiency of the solver grows when the polynomial order of approximation is increased (when the mesh is *p* refined).

$$\lim_{p\to\infty} Efficiency = 1 \tag{2.73}$$

Proof: The relative efficiency for $P = 2^{2m}$ processors is

$$\begin{aligned} Efficiency &= \frac{T_1}{PT_P} = \frac{T^1_{\text{comp solver}}}{P\left(T^{\text{i}}_{\text{comp solver}} + T^i_{\text{comm solver}} + T^i_{\text{idle solver}}\right)} = \\ &= \frac{\left(2^{2n}p^6 + 2^{3n}p^3\right)t_{comp}}{\left(2^{2(n-m)}p^6 + 2^{3(n-m)}p^3 + 2^{3n}p^3\right)t_{comp} + 2^{2n}p^2\ t_{comm}} = \\ &= \frac{\left(2^{2n}p^6 + 2^{3n}p^3\right)t_{comp}}{\left(2^{2n}p^6 + 2^{3n-m}p^3 + 2^{3n+2m}p^3\right)t_{comp} + 2^{2n+2m}p^2\ t_{comm}} \end{aligned} \tag{2.74}$$

where $T^1_{\text{comp solver}}$ is the computation time for a sequential solver, $T^i_{\text{comp solver}}$ is the computational time for *i*-th processor, $T^i_{\text{comm time}}$ is the communication time for *i*-th processor, $T^i_{\text{idle solver}}$ is the idle time for *i*-th processor, t_{comp} stands for the execution time of a single instruction and t_{comm} is the time of transfering a single double precision value.

The efficiency plot for different numbers of processor and different values of polynomial orders of approximation is presented in Figure 2.130 □

2.5.4. Reutilization of partial LU factorizations

This section is a theoretical analysis of a possible extension of the solver algorithm, with the reutilization of partial LU factorizations. For the technical details on the algorithm of the solver with reutilizations of partial LU factorizations, see Appendix D. In the next part of this section, we analyse the computational and communication times of the algorithm of the solver, with the extension for the reutilization of partial LU factorizations.

Remark 2.20. The time spend on the computation in the sequential solver algorithm with reutilization of partial LU factorizations is

$$T^1_{\text{comp re-solver 1}}\left(N^{total}_{elem}, p\right) = O\left(p^6 + N^{total\,3/2}_{elem} p^3 \right) = O\left(p^6 + 2^{3n} p^3 \right)$$

Proof: Let us consider a square computational mesh, presented in Figure 2.129, with $\mathrm{N}^{\text{total}}_{\text{elem}} = 2^{2n}$ elements. We assume that the problem for the computational mesh has been already solved, and only one element has been *h* refined in the direction of a mesh corner singularity. In this case, there is a need to compute all LU factorizations related to the elimination sub-tree assigned to a broken corner element. It is also necessary to recompute all LU factorizations on a single path of the refined element (represented by a leaf in the original elimination tree), up to the root of the tree. The computational time for a broken element is

$$\left\{ 4p^6 + 2\left(2+4+1\right)^2 p^2 \left(2+4\right) p + \left(4\cdot 2+2\right)^2 p^2 \left(4\cdot 2\right) p \right\} t_{\text{comp}} \quad (2.75)$$

since there are four element interiors, two single common edges and one double edge. The computational complexity of the recomputation of the whole path, from the refined leaf up to the elimination tree root, can be estimated as in the Remark 2.16. The difference is that now there is only one set of elements on each level of the tree. Thus, the computational time, without the computations on the level of leaf element, already estimated in (2.75), is

$$\begin{aligned} &\left(2+4+1\right)^2 p^2 \left(2+4\right) p + \\ &\sum_{k=1}^{n} \left[\left(2\cdot 2^{k+1} + 2\cdot 2^k + 2^k\right)^2 p^2 \left(2\cdot 2^{k+1} + 2\cdot 2^k\right) p + \right. \\ &\qquad \left. \left(\left(2\cdot 2^k + 2\cdot 2^k + 2^k\right)^2 p^2 \left(2\cdot 2^k + 2\cdot 2^k\right) p \right) \right] \end{aligned} \quad (2.76)$$

The total computational time of the solver reutilizing LU factorization is equal to the sum of (2.75) and (2.76), that is

$$\begin{aligned} T^1_{\text{comp re-solver 1}} &= O\left(p^6\right) + O\left(p^3\right) + O\left(\sum_{k=1}^{n} 2^{3k+6} p^3 \right) = \\ &= O\left(p^6 + \left(1 + 2^{3n+6} - 2^6\right) p^3 \right) = O\left(p^6 + 2^{3n} p^3 \right) \end{aligned} \quad (2.77)$$

□

Remark 2.21. The total computational time of the sequential solver algorithm with r refined leafs (resulting from $r/4$ singularities) is

$$T^1_{\text{re-solver r}}\left(N^{total}_{elem}, p, r\right) = O\left(r\, p^6 + r\, N^{total\,3/2}_{elem} p^3 \right) = O\left(r\, p^6 + r\, 2^{3n} p^3 \right).$$

Proof: In case of multiple refined leaves, the pessimistic estimation is that each leaf will generate a separate path to be totally recomputed. Thus, the total computational time with r refined leafs (resulting from $r/4$ singularities) is

$$T^1_{\text{re-solver r}} = O\left(r\, p^6 + r\left(1 + 2^{3n+6} - 2^6\right) p^3 \right) = O\left(r\, p^6 + r\, 2^{3n} p^3 \right) \tag{2.78}$$

□

Remark 2.22. The sequential solver algorithm with reutilization of partial LU factorizations is $O\left(\frac{\mathrm{N}^{\text{total}}_{\text{elem}}}{r} \right)$ times faster than the solver without the reutilization, where $\mathrm{N}^{\text{total}}_{\text{elem}}$ is a number of elements.

$$\frac{T_1}{T^1_{\text{re-solver r}}} = O\left(\frac{2^{2n}}{r} \right) = O\left(\frac{\mathrm{N}^{\text{total}}_{\text{elem}}}{r} \right) \tag{2.79}$$

Remark 2.23. The total execution time of the parallel solver algorithm with reutilization of LU factorizations after *r* elements have been refined is

$$T^P_{\text{re-solver r}}\left(N^{total}_{elem}, P, p\right) = \left(p^6 + N^{total\,3/2}_{elem} p^3 \right) t_{comp} + \frac{N^{total}_{elem}}{P} p^2\, t_{comm}$$

$$= \left(p^6 + 2^{3n} p^3 \right) t_{comp} + 2^{2(n-m)} p^2\, t_{comm} \tag{2.80}$$

This is ”the best parallel time” that can be achieved by the parallel solver with reutilization of partial LU factorizations, provided that we have enough available processors $P = 2^{2m} \geq r$. In other words, the number of the processors we use cannot be higher than the number of refined elements *r*.

Proof: In case of the parallelization of the solver with reutilization of partial LU factorizations, the maximum number of processors that can be used is equal to *r* (a number of elements refined on the current mesh). Each refinement requires a recomputation of the entire path from the refined leaf up to the tree root. This is a purely sequential operation.

If the number of processors is larger or equal to the number of executed refinements $P = 2^{2m} \geq r$, the total computational time can be roughly estimated as a parallel execution of computations for r paths, from a leaf up to the root of the tree, which corresponds to (2.77). The communication time remains unchanged, since there is still a need to exchange the LU factorization, even if they are taken from local tree nodes. Thus the communication time can be estimated as in the Remark 2.17 □

Remark 2.24. The parallel solver with reutilization of partial LU factorizations is $O\left(\frac{\mathrm{N}_{\mathrm{elem}}^{\mathrm{total}}}{r}\right)$ times faster than the solver without the reutilization, where N is a number of elements.

Proof: If $P = 2^{2m} \geq r$ then

$$\frac{T_P}{T^P_{\text{re-solver } r}} = O\left(2^{2(n-m)}\right) = O\left(\frac{\mathrm{N}_{\mathrm{elem}}^{\mathrm{total}}}{P}\right) \leq O\left(\frac{\mathrm{N}_{\mathrm{elem}}^{\mathrm{total}}}{r}\right) \tag{2.81}$$

□

2.6. Simulational study of scalability

In this section we describe several numerical experiments performed in order to test the scalability of the parallel self-adaptive *hp*-FEM algorithm. The numerical tests have been executed with the use of the *hp2Dpar* and *hp3Dpar* implementations of the parallel self-adaptive *hp*-FEM algorithm (for the details on the implementation, see Appendix C). The implementations employ the self-adaptive *hp*-FEM algorithm extended to an arbitrary two- and three-dimensional rectangular finite element meshes.

2.6.1. Scalability of the parallel self-adaptive *hp*-FEM algorithm in two dimensions, with the grain defined on the level of initial mesh elements and with multiple front parallel solver

The first numerical test concerns the scalability of the self-adaptive *hp*-FEM algorithm with rectangular finite elements implemented for two dimensions in the *hp2Dpar* code, interfaced with the multiple front parallel solver developed by Walsh, Demkowicz 1999 (for the technical details on the implementation, see Appendix C; for the multiple front solver algorithm, see Appendix D).

There are two goals of the first numerical test:

1. to test the scalability of the parallel self-adaptive *hp*-FEM algorithm implemented in two dimensions on the level of super-tasks, with load balancing and mesh partitioning, for which the grains are defined on the level of tasks,
2. to test the scalability of the multiple front parallel solver applied to two-dimensional *hp* adaptive computations.

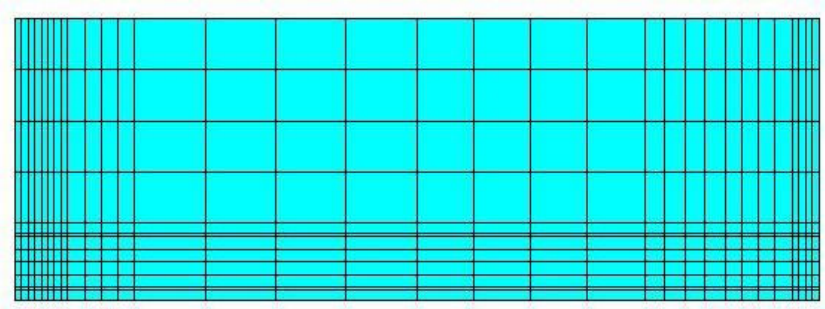


Fig.2.131. Initial mesh for the battery problem test

The test is based on the battery problem from the Sandia National Laboratory (described in Chapter 3.1.2). The sequential and parallel algorithms implemented in *hp2D* and *hp2Dpar* codes have been executed on the initial mesh presented in Figure 2.131. The initial mesh contains 512 initial mesh elements. Such a large number of initial mesh elements is expected to maintain a uniform load balancing for a large number of processors. The computational problem has been solved up to 0.1 % relative error of the solution in the energy norm. This has required 45 iterations of the self-adaptive *hp*-FEM algorithm. The parallel algorithm has been executed on 4, 8, 16 and 32 processors, on the LONGHORN linux cluster from Texas Advanced Computing Center.

For different numbers of processors we have executed the measurements of the total computational time (Figure 2.132), of efficiency (Figure 2.133)

$$Efficiency = \frac{T_1}{PT_P} \tag{2.82}$$

and of speedup (Figure 2.134)

$$Speedup = \frac{T_1}{T_P} \tag{2.83}$$

Here T_1 is the total sequential code time, T_P is the total parallel code time for P processors. Since the number of initial mesh elements was 512, there are many tasks assigned to a single processor (each super-task consists of many tasks).

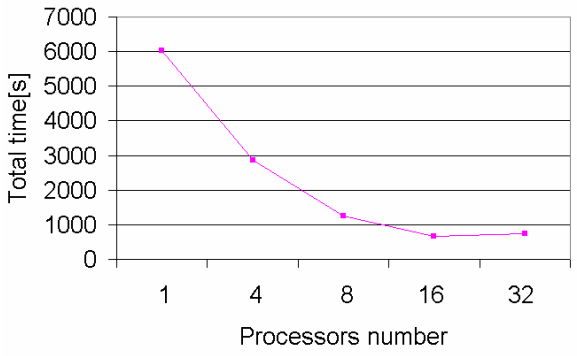


Fig.2.132. Total computational time for different numbers of processors, for the battery problem

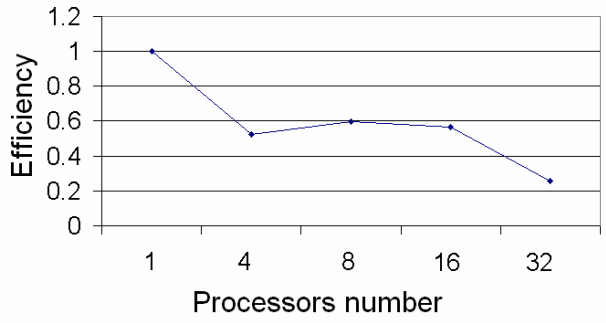


Fig.2.133. Efficiency for different numbers of processors, for the battery problem

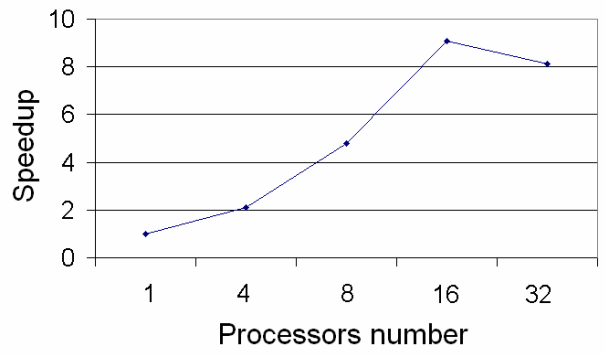


Fig.2.134. Speedup for different numbers of processors, for the battery problem

In order to investigate the loss of speedup for a large number of processors, we have performed the detailed measurements for 16 and 32 processors.

The execution time for each part of the algorithm during all iterations of the self-adaptive *hp*-FEM is presented in Figure 2.135. The total execution time varies from 23 seconds in the first iteration up to 124 seconds in the last one. The highest influence on the total execution time has the ``Fine mesh solution", as it was expected. This execution time is uniformly rising from 11 seconds in the first iteration, when the size of fine grid problem is 20,000 degrees of freedom, up to 85 seconds, when the size of fine grid problem is about 80,000 degrees of freedom. Another expensive part is the ``Selection of optimal refinements", where the optimal refinements for each finite element are determined by comparing the error decrease rates for all considered *hp* refinements. The algorithm requires a solution of several local systems of equations for each particular finite element, and its cost is rising with the number of degrees of freedom. The next most expensive parts are the "Coarse mesh solution" and the "Execution of *h* refinements". The less expensive parts are the "Execution of *p* refinements" and the "Global *hp* refinement".

The measurements of the serial version of the algorithm, presented in Figure 2.135 have been compared with similar measurements for the parallel version of the algorithm executed on 16 processors, presented in Figure 2.136.

The number of iterations, required in order to obtain the same 0.1 % accuracy as in the serial version, is lower and equal to 37. This can be explained by better accuracy of the parallel frontal solver algorithm, in comparison with the sequential solver algorithm.

We have compared the refinement histories for parallel and serial computations. The first difference appears in sixth iteration, as presented in Figure 2.137. The difference results from numerically different error estimations for the same finite element. In the parallel code, the error estimation for the element was equal to $0{,}11031 \cdot 10^{-7}$, while the estimation in the serial code was $0{,}14381 \cdot 10^{-7}$. The difference in the eighth digit results in a different *p*-strategy chosen by the element. The next relevant difference appears in the seventh iteration, as presented in Figure 2.138, and results again from numerically different error estimations for some finite elements. The error estimation is calculated as a difference between coarse and fine grid solutions.

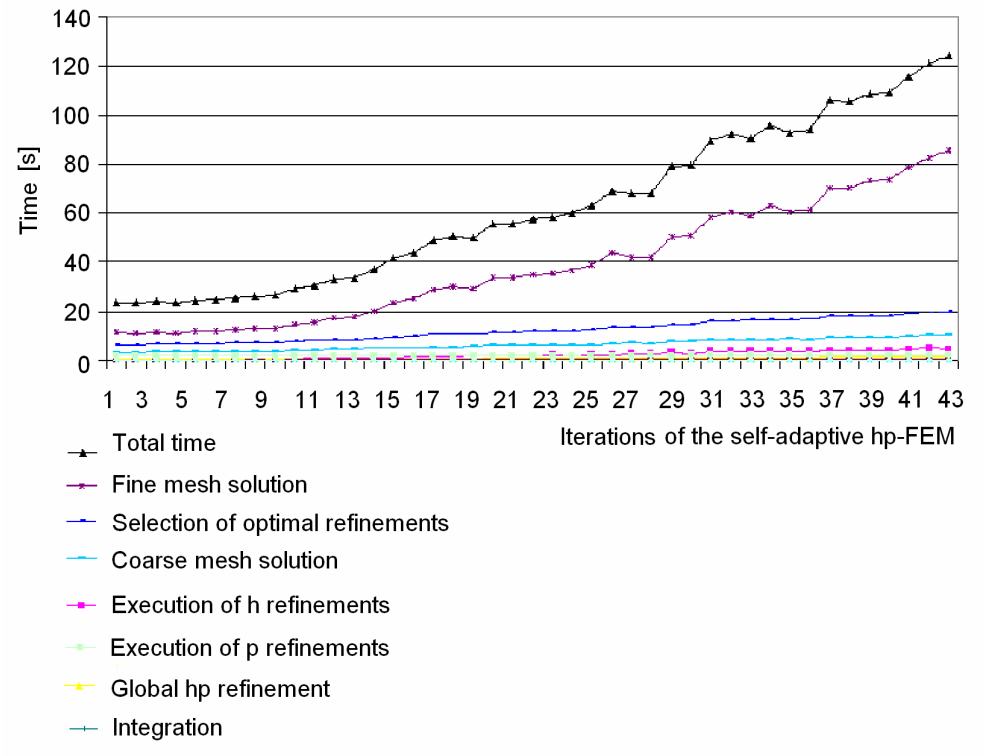


Fig.2.135. Measurements of execution times for particular parts of sequential algorithm, for the battery problem

The better accuracy of the parallel multiple front solver algorithm, in comparison with the serial front solver, leads to slightly better choices in the refinement algorithm. The number of differences grows with an increasing number of iterations and results in faster convergence of the numerical error within the parallel algorithm.

The total computational time for the parallel execution varies from about 4 seconds in the first iteration up to 18 seconds in the last one. Again, the most expensive part is the "Fine grid solution", for which the execution time varies from about 2 seconds up to about 9 seconds. The next most expensive components are the "Selection of optimal refinements" and the "Coarse mesh solution". For the parallel version of the algorithm, the "Integration" part is also essential. The mesh transformations, such as the "Execution of *h* and *p* refinements" and the „Global *hp* refinement" have almost no influence on the parallel algorithm.

We can make the following observations as a result of comparing the sequential and parallel algorithms. The parallel algorithm is about 8 times faster on 16 processors, thus the efficiency of the parallel algorithm on 16 processors is about 50 %. The fine mesh problem solution is 10 times faster on 16 processors, with 70 % efficiency. The selection of optimal refinements is about 8 times faster and delivers 50 % efficiency. Both, the coarse mesh solution and the integration are about 3 times faster, with 20 % efficiency only. The mesh transformations are not expensive. The *h* adaptations take no more than 4 seconds on a single processor and less than 1 second on 16 processors. Similarly, the *p* refinements take less than 2 seconds on a single processor and less than 1 second on 16 processors, and the global *hp* refinement (fine mesh generation) always takes less than 1 second.

Finally, a similar comparison has been performed for 32 processors, as shown in Figure 2.139. Surprisingly, in case of 32 processors, the overall shape of execution is similar to the shape for 16 processors.

The parallel algorithm is also about 8 times faster on 32 processors, thus the efficiency of the parallel algorithm on 16 processors decreases down to 25 %. The fine mesh problem solution is also 10 times faster on 32 processors, which results in 35 % efficiency only. The selection of the optimal refinements is also about 8 times faster; the coarse mesh solution and the integration are about 3 times faster. The mesh transformations are again not expensive, less than 1 second on 32 processors.

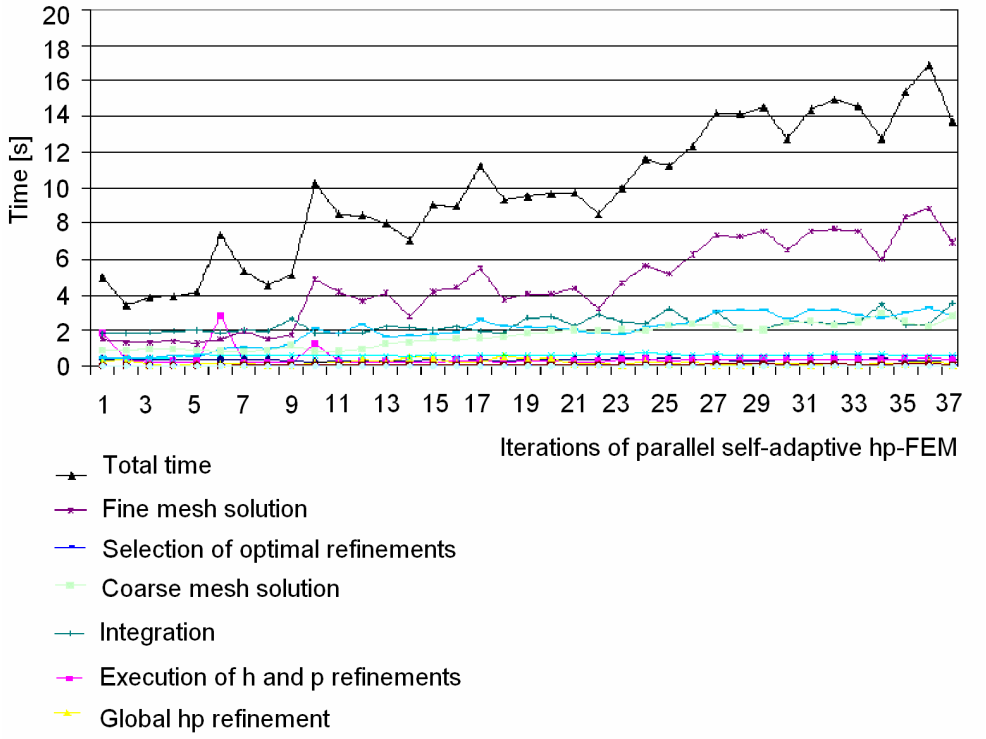


Fig.2.136. Measurements of maximum execution time for all processors
for particular parts of parallel algorithm, executed on 16 processors, for the battery problem

In order to further investigate the loss of efficiency for 32 processors, the following experiment has been performed. The computational cost of the elimination for a single finite element, with the basic operation defined as a single arithmetic operation is

$$\text{computational\_cost} = (p_h+1)^3 (p_v+1)^3 \tag{2.84}$$

since a number of degrees of freedom on an element with polynomial order of approximation p_h in the horizontal and p_v in vertical direction is $(p_h+1)(p_v+1)$, and the elimination is a cube of the number of degrees of freedom. The cost of the integration performed by a single computational task is also defined by (2.84).

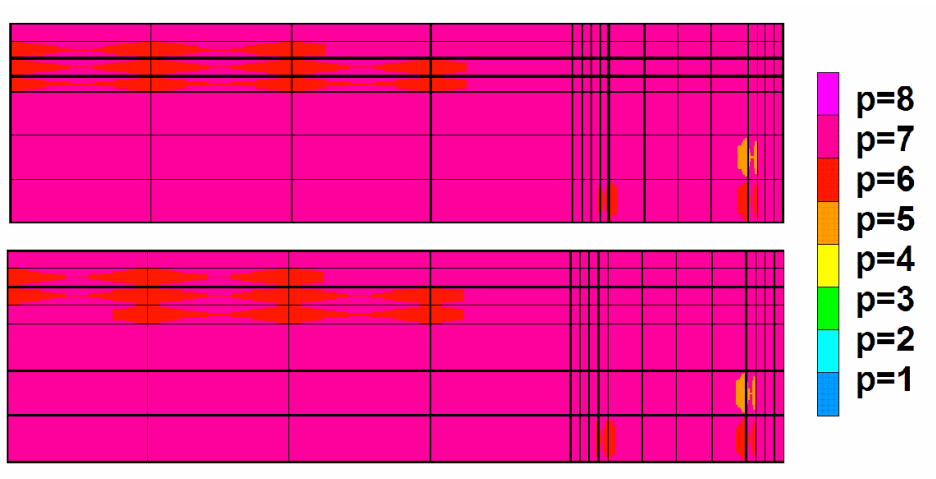


Fig.2.137. First difference in the mesh obtained in the sixth iteration by the parallel and the serial algorithms, for the battery problem. **Top panel:** serial code. **Bottom panel:** parallel code. Different colours denote different polynomial orders of approximation

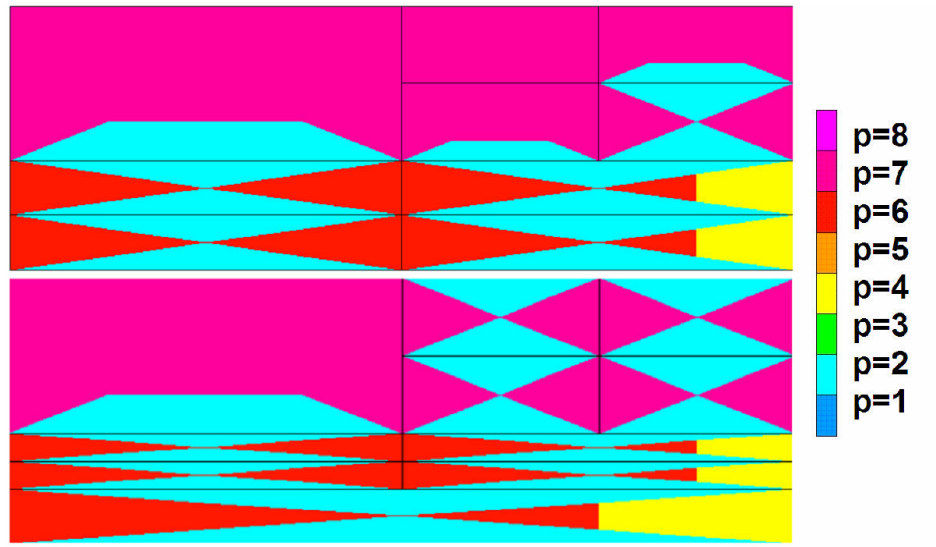


Fig.2.138. Another difference in the mesh obtained in the seventh iteration by the parallel and the serial algorithms , for the battery problem. **Top panel:** serial code. **Bottom panel:** parallel code. Different colours denote different polynomial orders of approximation

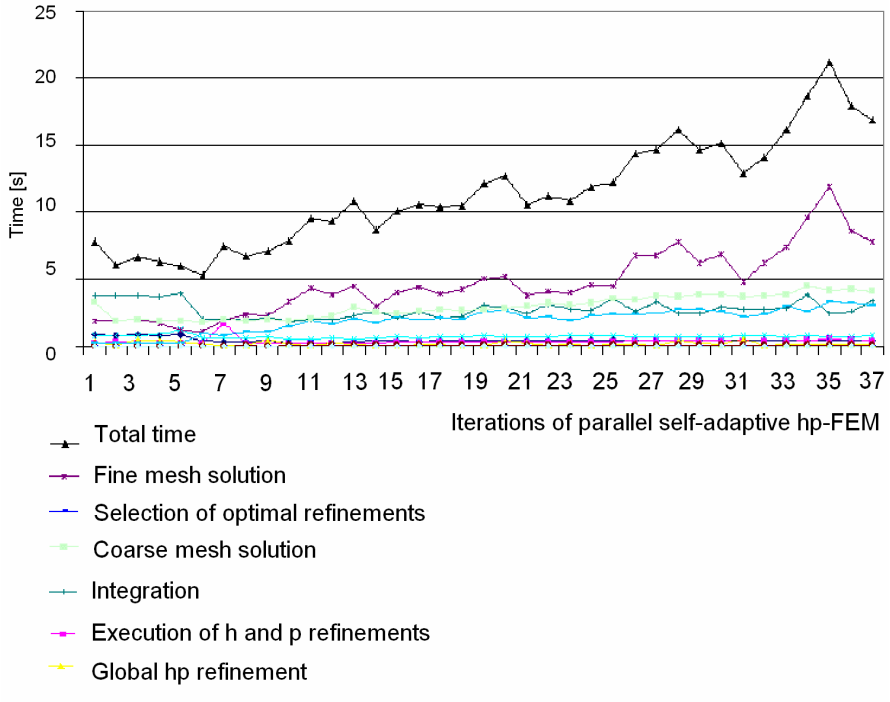


Fig.2.139. Measurements of the maximum execution times for all processors for particular parts of the parallel algorithm, executed on 32 processors, for the battery problem

In order to investigate the problem of a lack of significant differences between computational times for 16 and 32 processors, we have performed some measurements of the computational loads for all processors, during particular iterations. Figure 2.140 presents the measured distribution of computational load, for each of 32 processors, in particular iterations. The total computational load for each initial mesh element is estimated as a sum of loads from all active finite elements within the initial mesh element. Up to the sixth iteration, the computational load is uniformly distributed between all tasks, and the number of tasks per processor is almost constant. After the sixth iteration, the load begins to rise dramatically on about 14 processors, while other processors have zero load and no computational tasks assigned.

The reason for such behavior can be explained by the following example. Let us consider a case of 4 computational tasks corresponding to 4 initial mesh elements, with different polynomial orders of approximations. The left top element has both orders of approximation equal to 9, thus the computational cost for this element is 10^6. Both adjacent elements have one order of approximation equal to 9 (to fulfill the minimum rule), and the other order of approximation equal to 1. Thus, the computational cost for these elements is 10^3. Finally, the bottom right element has both orders of approximation equal to 1, which results in the computational cost of order 10^1. The optimal load balancing in case of 4 processors is that processor 1 has the first element with the highest load, and all other elements are assigned to processor 2 (see Figure 2.141). In other words, processors 3 and 4 have no computational tasks assigned.

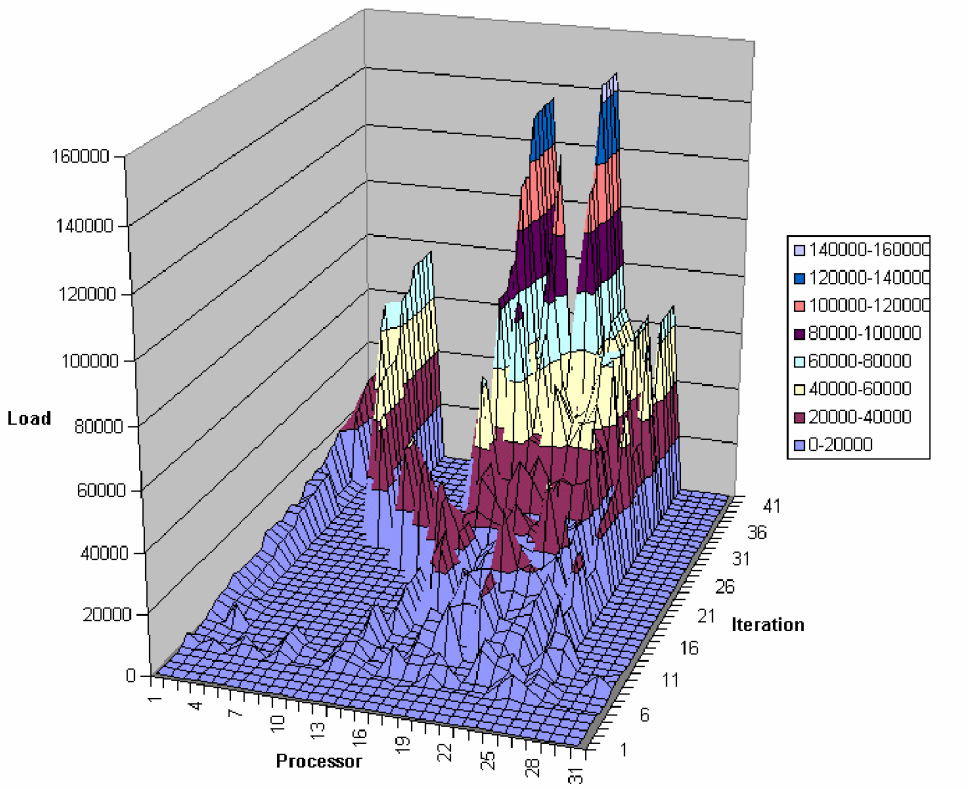


Fig.2.140. Load distribution for 32 processors during particular iterations of the self-adaptive *hp*-FEM, for the battery problem

In the Sandia battery problem there are about four areas of the computational mesh where the singularities are present, and most of the *hp*-refinements are required in the neighborhood of these areas. There are about sixteen initial mesh elements which completely cover these areas with singularities. After some initial steps, where global *hp*-refinements take place, the algorithm starts strong *hp*-refinements around the detected singularities, as presented in Figure 2.142. These refinements are performed exclusively within these sixteen initial mesh elements. The load for some of these elements is one order of magnitude higher than overall load for all other elements.

The following conclusions can be drawn from this experiment:

a) the most time-consuming step of the computations is the fine mesh solution, as it is presented in Figures 2.136 and 2.139,
b) the computational costs for the mesh transformation algorithms, including the mesh refinements and mesh partitioning, is one order of magnitude smaller than the computational cost for the solver. This implies that we need to investigate the development of a better parallel solver,
c) in order to solve the battery problem wih accuracy 0.1% of the relative error in the energy norm, it is necessary to perform 43 iterations of the serial algorithm, but

only 37 iterations of the parallel algorithm on 16 processors, and again 32 iterations of the parallel algorithm with 32 processors. Thus, we need to compare 43 iterations of the serial code with 37 iterations of the parallel code,

d) the scalability of the parallel code is limited by the number of singularities. This is because there are only 4 initial mesh elements covering an area with the strongest singularities. After some initial number of iterations, most of *hp* refinements are required in the neighborhood of these areas. However, this is a difficulty only in case of small problems with a small number of singularities.

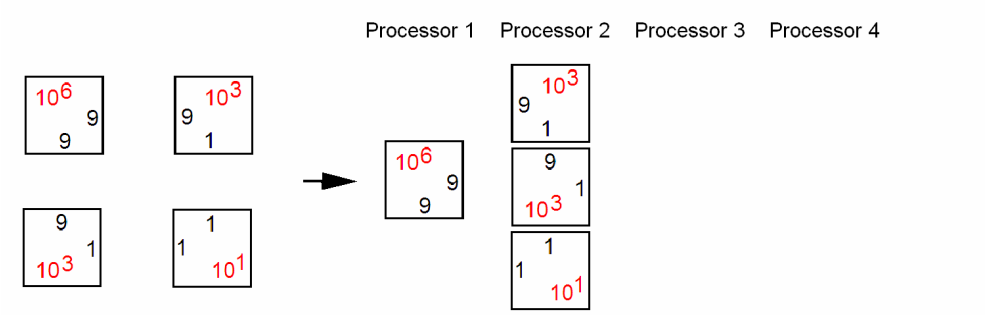


Fig.2.141. Load balance for four tasks with different polynomial orders of approximation, where the first task has high polynomial orders of approximation resulting in computational cost several orders of magnitude higher than other elements. The optimal load balance is such that the first task is assigned to the first processors, all other tasks are assigned to the second processor, and remaining processors are idle.

2.6.2. Scalability of the parallel self-adaptive *hp*-FEM algorithm in three dimensions, with the grain defined on the level of initial mesh elements and with multiple front parallel solver

A similar test has been executed in three dimensions, with the use of *hp3D* and *hp3Dpar* applications. The goals of this test are as follows:

a) to test the scalability of the parallel self-adaptive *hp*-FEM algorithm implemented in three dimensions on the level of super-tasks, partitioned with the grains defined on the level of tasks (on the level of sub-graphs corresponding to the initial mesh elements),

b) to test the scalability of the multiple front parallel solver applied to three-dimensional *hp* adaptive computations.

The problem of highly non-uniform load on finite elements with different polynomial orders of approximation is more visible in three dimensions. There are three polynomial orders of approximation for 3D elements, along *x, y* and *z* axis of the coordinate system. The computational cost of the elimination for a 3D element with the basic operation defined as a single arithmetic operation, can be estimated as

$$\text{computational\_cost} = (p_x+1)^3 (p_y+1)^3 (p_z+1)^3 \qquad (2.85)$$

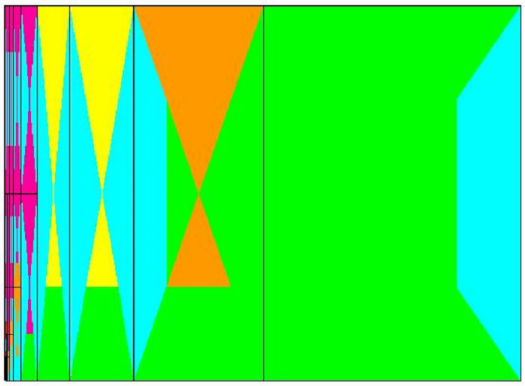


Fig.2.142. Highly *hp*-refined initial mesh element in the neighborhood of a strong singularity. Different colours denote different polynomial orders of approximation

The problem of non-uniform load will be discussed using three cases, described in detail in Chapters 3.2.1, 3.2.3 and 3.2.2:

1) the 3D Fichera model problem,
2) the Step-and-Flash Imprint Lithography (SFIL), the patterning process which uses photopolimerization to replicate microchip pattern from the template into the substrate. The goal of the *hp*-FEM simulation in this case is to compute volumetric shrinkage of the feature modeled by linear elasticity with thermal expansion coefficient,
3) the resistance heating of the Al-Si billet in steel die for tixoforming process. The goal of *hp*-FEM simulation is to compute heat distribution during the resistance heating process, modeled by the Poisson equation with Fourier boundary condition of the third type.

When the computational problem contains singularities related to either jumps in material data, jumps of prescribed boundary conditions, or complicated geometry, the generated *hp* meshes are irregular and may contain very small finite elements with high polynomial orders of approximation, especially in the areas close to these singularities.

In the first case, there is only one singularity in the center of the domain. The generated *hp* mesh contains a single finite element with interior node where the polynomial orders of approximation are set to 7 in all three directions as well as three finite elements with interior nodes where the polynomial orders of approximation are set to 6 in two directions and to 7 in the third direction. The load representing the computational cost of these elements for integration and elimination components is much higher than the load for all other elements (see Figure 2.143).

In the second case, there is one central finite element with the polynomial orders of approximation higher than the orders of other finite elements. The load for this element is higher than the load for all other finite elements, and it is equal to 5 in two directions and to 6 in the third direction (see Figure 2.144).

In the third case, there are many singularities related to jumps in material data. There are many finite elements with high polynomial orders of approximation and the load distribution is quite uniform (see Figure 2.145).

Let us focus now on the Fichera problem. The Fichera problem has been solved with accuracy 1% of the relative error in the energy norm. The total computational times, efficiency and speedup are illustrated in Figures 2.146-2.148. The computational time for different parts of the self-adaptive *hp*-FEM algorithm for particular iterations, for 4 and 8 processors are presented in Figures 2.149-2.150.

Moreover, the detailed measurements of the particular parts of the multiple front parallel solver algorithm have been performed. The measurements for 4 processors are presented in Figure 2.119, while the measurements for 8 processors are presented in Figure 2.120.

We can draw the following conclusions from the measurements performed for the Fichera problem:

e) the most time-consuming step of the computations is the fine mesh solution, as presented in Figures 2.149-2.150,
f) the most time-consuming step of the solver is the forward elimination, as illustrated in Figures 2.151-2.152, however the time necessary for the solution of the interface problem grows as a quadratic function with the iteration number,
g) if the degrees of freedom corresponding to the fine grid solution, obtained from the parallel solver, differ from those obtained using the serial version of the code by 10^{-8} or more, the resulting meshes selected by the mesh optimization algorithm start diverging from each other,
h) in order to solve the Fichera problem with the accuracy 1% of the relative error in the energy norm, it is necessary to perform 6 iterations of the serial algorithm, but only 5 iterations of the parallel algorithm on 4 processors, and again 6 iterations of the parallel algorithm code on 8 processors. Thus, we need to compare 6 iterations of the serial code with 5 iterations of the parallel code on 4 processors, and with 6 iterations of parallel code on 8 processors.

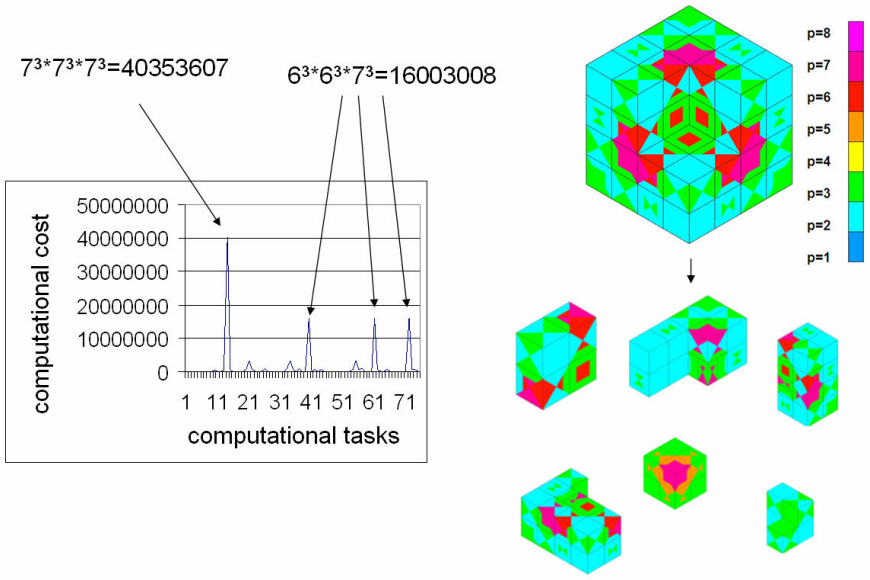


Fig.2.143. Load imbalance for computational tasks, for the Fichera model problem

The time of the fine grid solution for 8 processors is not well balanced, as presented in Figure 2.143. The reason for this is the number of initial mesh elements covering the singularities in the Fichera problem. In the Fichera problem, there are only 4 initial mesh elements covering an area with the strongest singularities. Most of *hp* refinements in the Fichera problem are required in the neighborhood of this area.

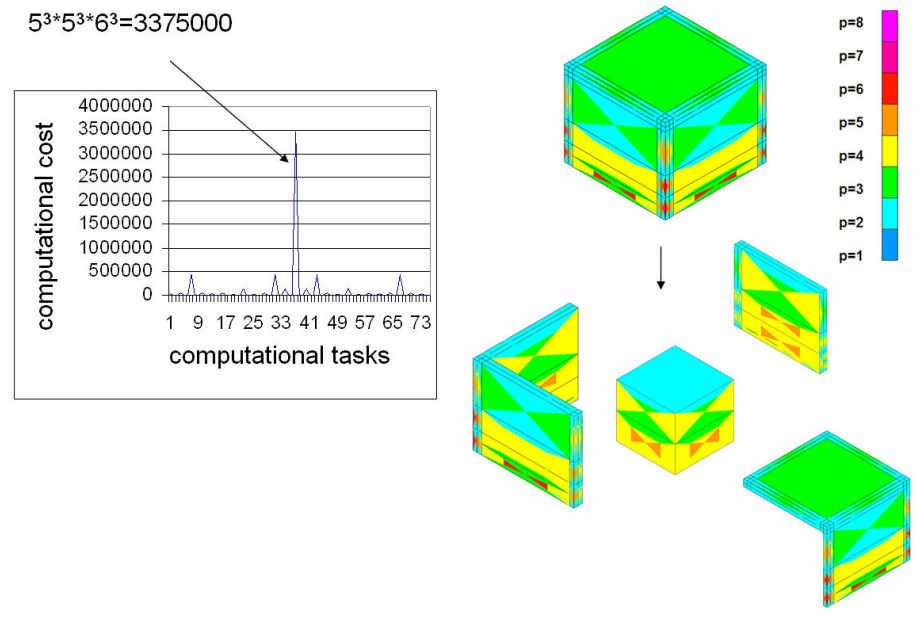


Fig.2.144. Load imbalance for computational tasks, for the SFIL problem

When the number of singularities in the problem is small, the load distribution for computational super-tasks is not uniform. This is related to a high contrast in the polynomial orders of approximation used in the computational tasks (initial mesh elements).

The detailed measurements for particular parts of the multiple front solver algorithm executed on the Fichera model problem are presented in Figures 2.144 and 2.145, for 4 and 8 processors respectively.

Table 2.4

The number of degrees of freedom for particular iterations of the 3D self-adaptive *hp*-FEM, for the Fichera model problem

Iteration	Number of degrees of freedom
1	705
2	1407
3	2621
4	4882
5	9093
6	16935

The numbers of degrees of freedom for particular iterations of the self-adaptive *hp*-FEM algorithm in this experiment are illustrated in Table 2.4.

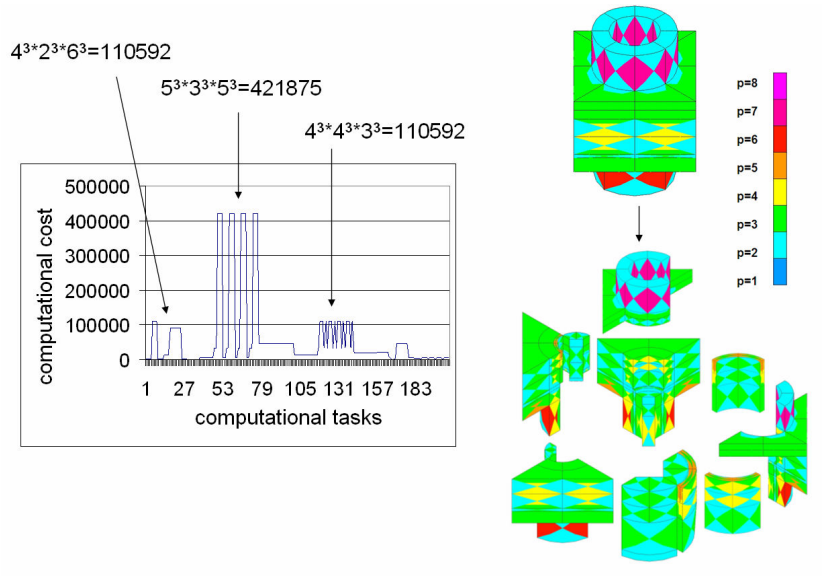


Fig.2.145. Load imbalance in computational tasks, for the resistance heating problem

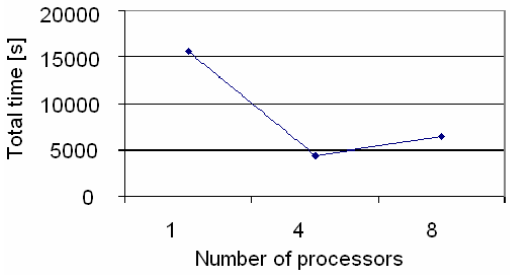


Fig.2.146. Total computational times for different numbers of processors, for the Fichera problem

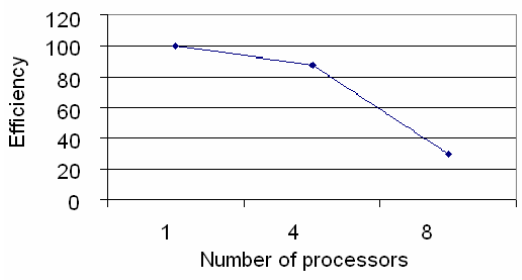


Fig.2.147. Efficiency for different numbers of processors, for the Fichera problem

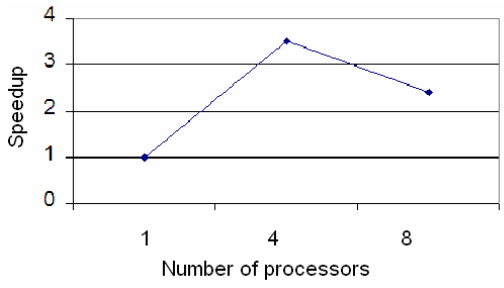


Fig.2.148. Speedup for different numbers of processors, for the Fichera problem

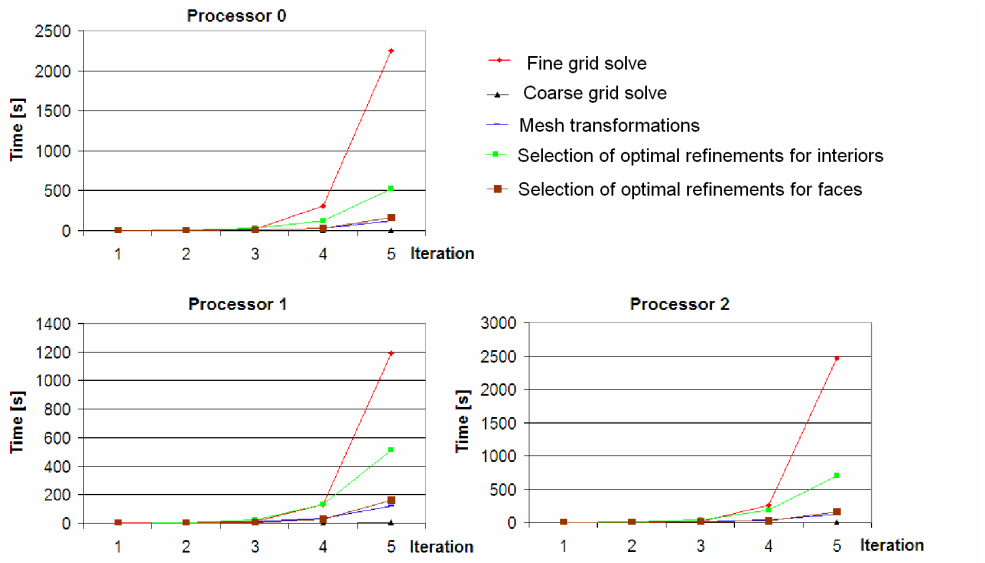


Fig.2.149. Measurements of execution times for particular parts of the parallel algorithm on 4 processors, for the Fichera problem. The last processor is responsible for the solution of interface problem

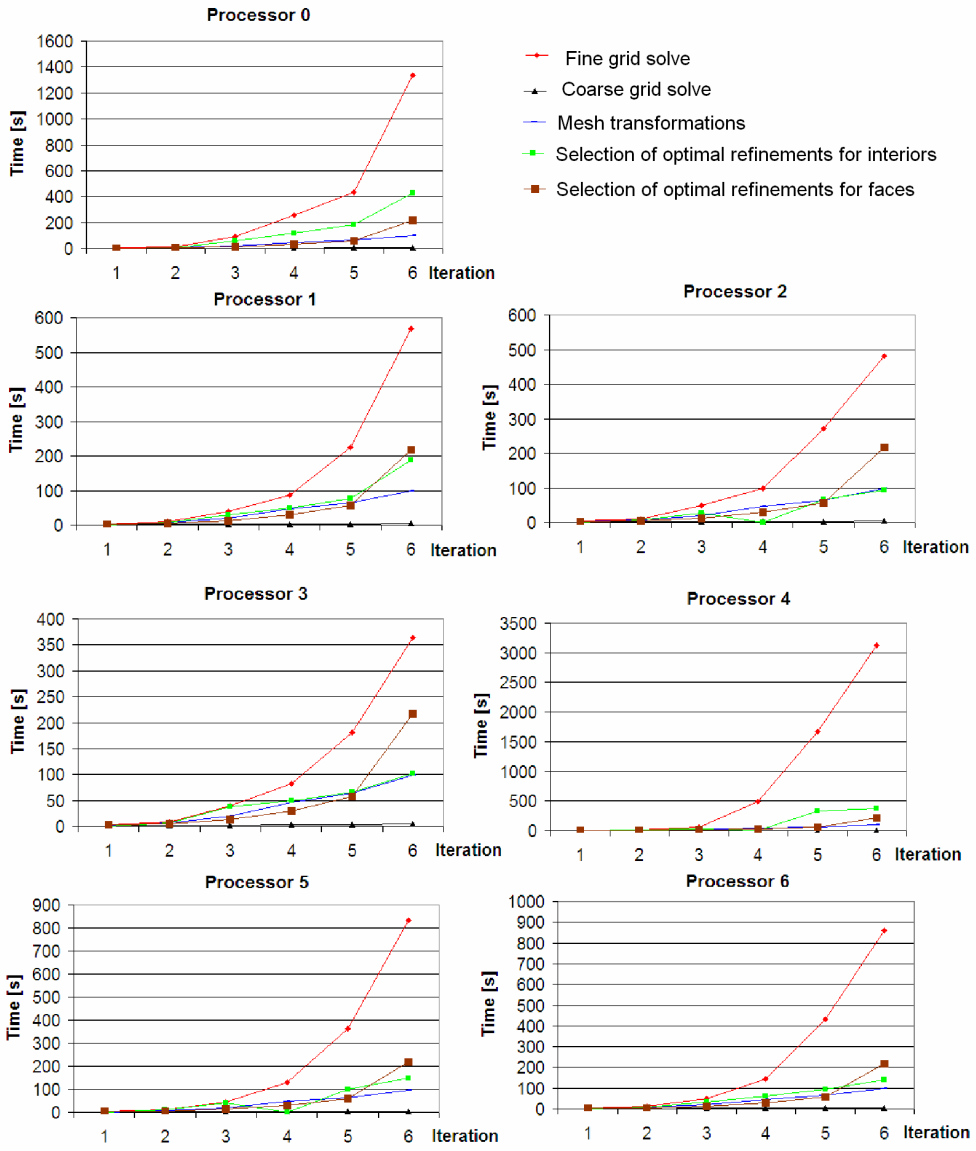


Fig.2.150. Measurements of execution times for particular parts of the parallel algorithm on 8 processors , for the Fichera problem. The last processor is responsible for the solution of interface problem

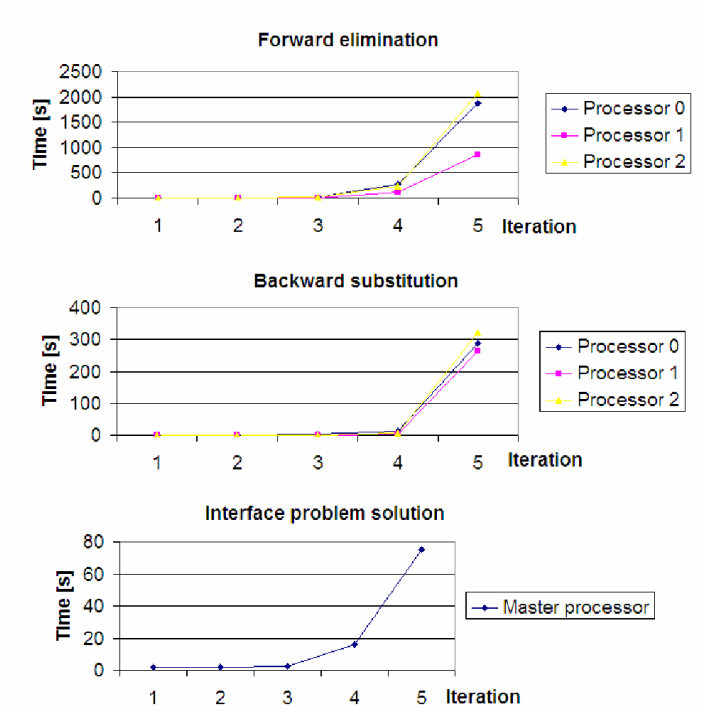


Fig.2.151. Execution times for the multiple front solver, executed on 4 processors for the fine mesh, for the Fichera problem

The following conclusions can be drawn from this experiment:

a) the multiple front solver algorithm eliminates the interior nodes of a sub-domain very slowly (the partial forward elimination on sub-domains, executed to compute the Schur complement contribution is very slow). This is because the entire sub-domain is aggregated at the same time, without constructing the multi-level elimination trees,
b) the time of the global interface problem solution grows as a quadratic function. This is because the entire interface problem is aggregated to a single global matrix.

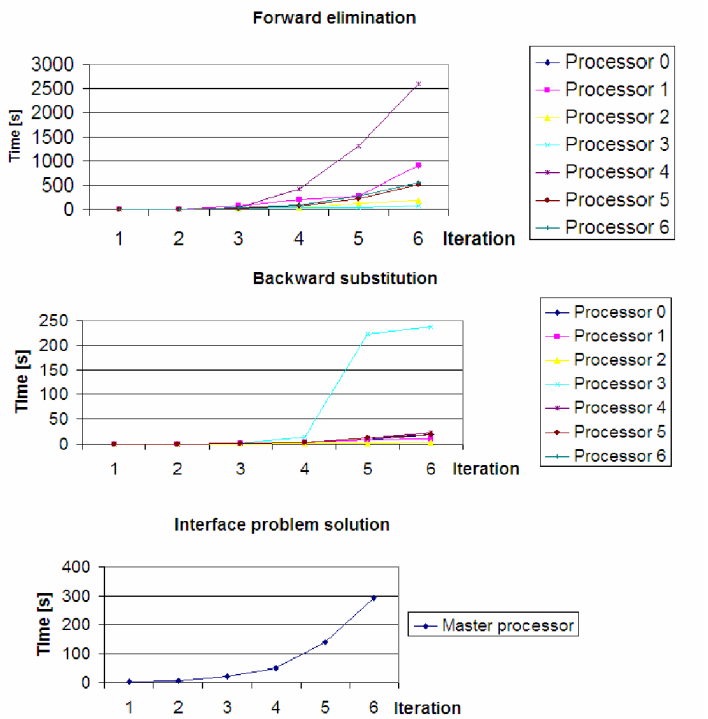


Fig.2.152. Execution times for the multiple front solver, executed on 8 processors for the fine mesh, for the Fichera problem

2.6.3. Scalability of the parallel self-adaptive *hp*-FEM algorithm in three dimensions, with the grain defined on the level of initial mesh elements and with multi-level parallel solver

In this section, we describe the third sequence of numerical experiments. The goal of these experiments is to test the scalability of a new multi-level parallel solver implemented in three dimensions and interfaced with the *hp3Dpar* code. For the technical details on the solver algorithm and its implementation , see Appendix D.

The new solver algorithm improves the way of solving the interface problem by the multiple front solver. The algorithm is still not as general as the parallel solver introduced in Section 2.1.4, however it consitutes the first step towards that solver.

The multi-level parallel solver employs the sequential version of the MUMPS solver for computing the Schur complements of degrees of freedom related to the interior of a single sub-domain, with respect to the degrees of freedom related to the interface. This is done by performing partial forward eliminations followed by partial backward substitutions (for more details, see Appendix D).

The experiment has been peformed with *hp3Dpar* application, on the 3D Direct Current (DC) borehole resistivity measurements simulation problem, described in detail in

Chapter 3.2.5. For those experiments we use the uniform *hp* mesh with 20,000 finite elements, with uniform polynomial orders of approximation p=3, and with 700,000 degrees of freedom, as presented in Figure 2.153.

The efficiency of this solver is documented in Figure 2.154. The low efficiency of the solver for a small number of processors can be explained as follows: the total time spent by sequential MUMPS solver on the forward elimination on the entire domain is about five times faster than the time needed for the partial forward elimination of the interior degrees of freedom, with respect to the interface degrees of freedom on a single sub-domain. The MUMPS is able to construct an efficient ordering of the degrees of freedom corresponding to the cylindrical shape of the domain. However, in case of four sub-domains, the construction of a good ordering is limited by the presence of interface degrees of freedom that must be always at the end of the list of sub-domain degrees of freedom. In order to notice any speedup for the parallel solver, the solver must be executed on at least four processors. The execution times for 16 processors, shown in Figure 2.155, are not uniform and depend strongly on the sub-domain local numbering of nodes for the MUMPS solver. The "Schur complement on level 2 (and 3)" stands for the second and third level of the elimination tree constructed for the global interface problem. The corresponding execution times for 4 and 8 processors are illustrated in Figures 2.156 and 2.157.

Another problem consists in solving the same equation for a non-uniform mesh with polynomial orders of approximation p=1,…,8 and with about 250 000 degrees of freedom (see Figure 2.158). The efficiency of the solver is presented in Figure 2.159. The maximum efficiency of 60% is reached for 8 processors, while the efficiency for 16 processors is worse.

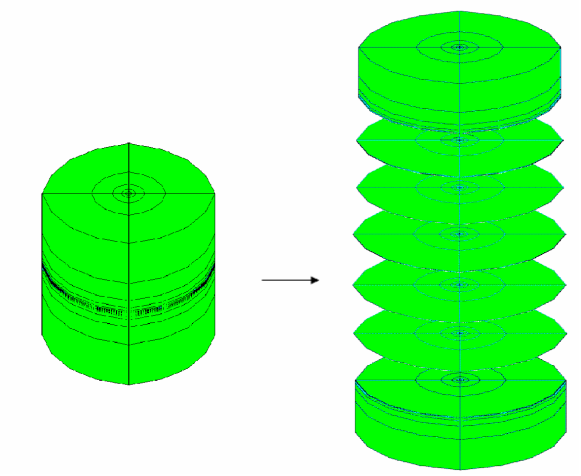


Fig.2.153. Uniform mesh of 20,000 finite elements of order p = 3, with about 700 000 degrees of freedom

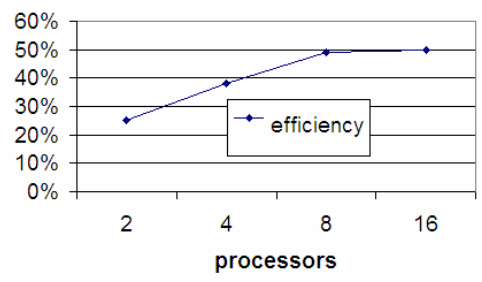


Fig.2.154. Efficiency of the parallel solver for the mesh from Figure 2.153

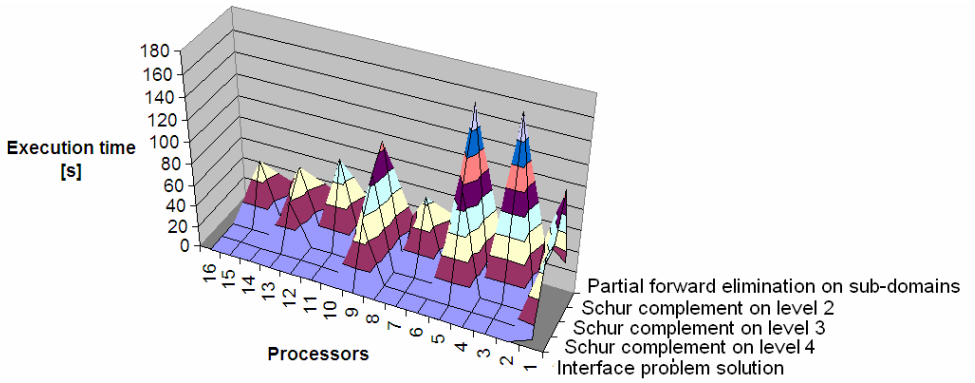


Fig.2.155. Total execution time for different parts of the solver algorithm, for 16 processors, for the mesh from Figure 2.153

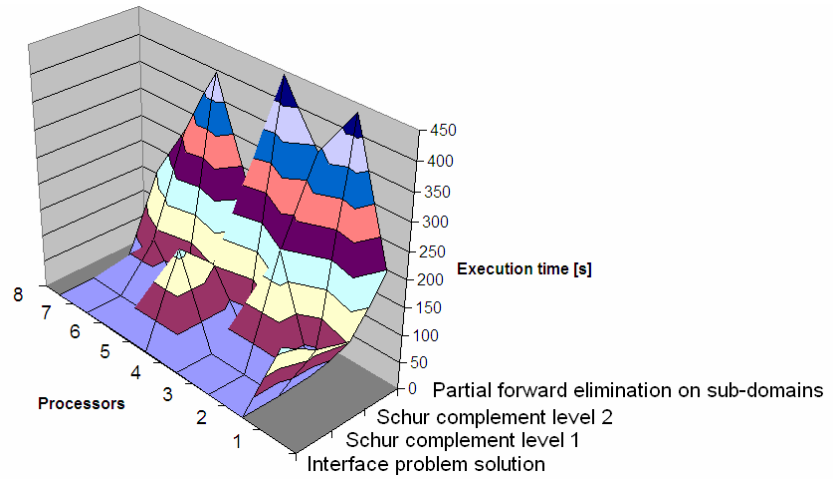


Fig.2.156. Total execution time for different parts of the solver algorithm, for 8 processors, for the mesh from Figure 2.153

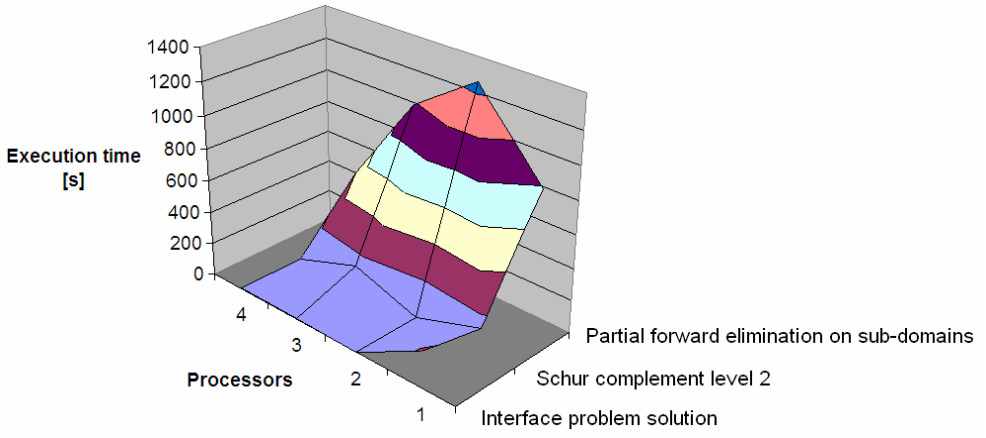


Fig.2.157. Total execution time for different parts of the solver algorithm, for 4 processors, for the mesh from Figure 2.153

The execution times for 4, 8 and 16 processors, presented in Figures 2.160-2.162 are much more uniform. This is because there are many degrees of freedom inside highly *hp*-refined elements. This is related to the fact that there are many highly refined finite elements inside each sub-domain, and most of the execution time is spent on eliminating the interior degrees of freedom from sub-domains.

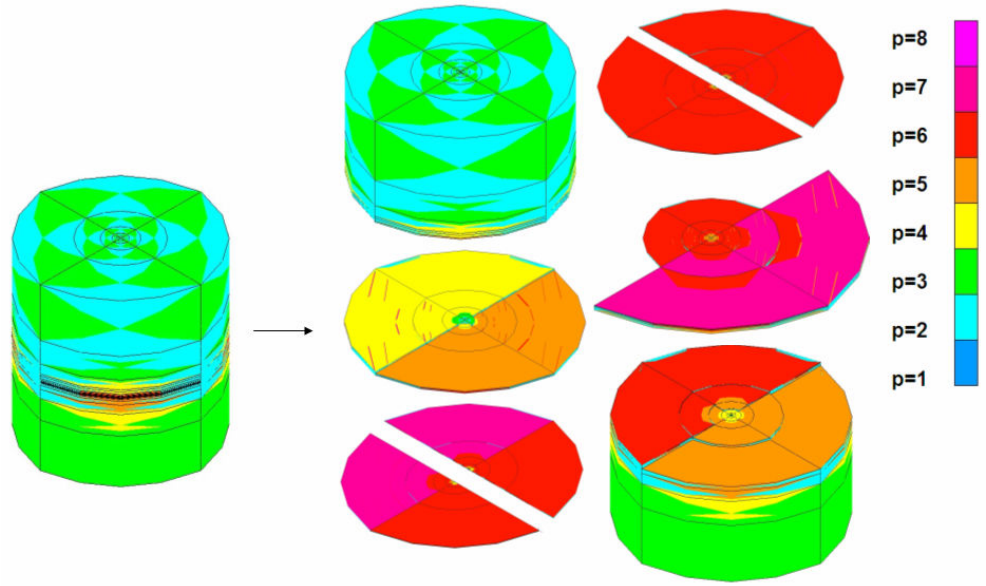


Fig.2.158. Non-uniform *hp* mesh, with p=1,…,8, with 250, 000 degrees of freedom, distributed into 8 sub-domains

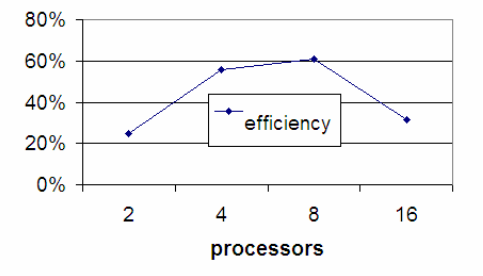


Fig.2.159. Efficiency of the parallel solver, for the mesh from Figure 2.158

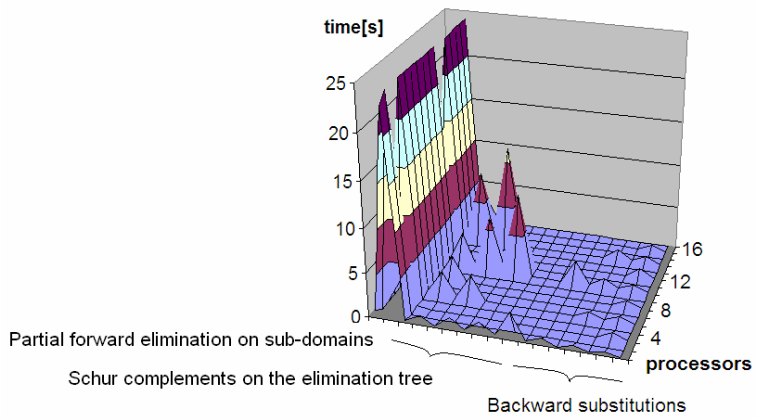


Fig.2.160. Total execution time for different parts of the solver algorithm, for 16 processors, for the mesh from Figure 2.158

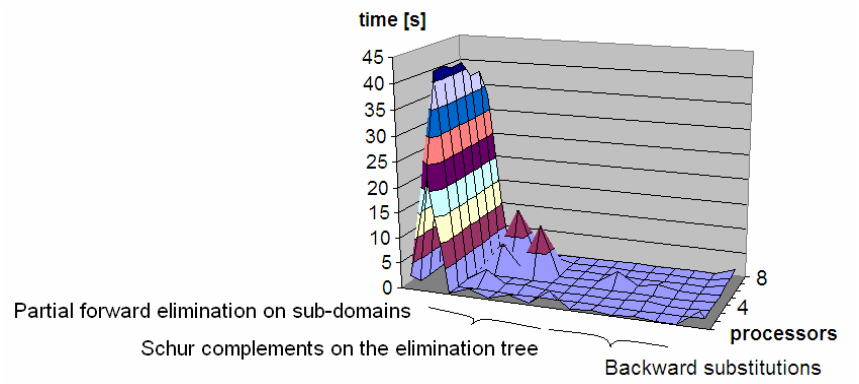


Fig.2.161. Total execution time for different parts of the solver algorithm, for 8 processors, for the mesh from Figure 2.158

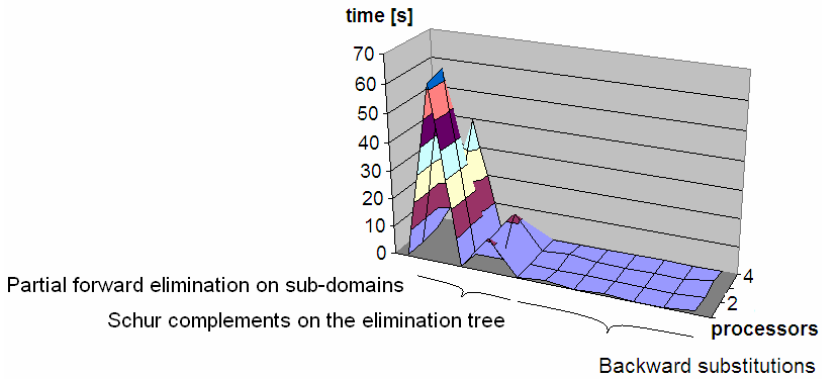


Fig.2.162. Total execution time for different parts of the solver algorithm, for 4 processors, for the mesh from Figure 2.158

The following conclusions can be drawn from this experiment:

a) the total execution time spent by MUMPS on eliminating the degrees of freedom related to the interiors of sub-domains is highly non-uniform. It seems that the MUMPS solver has problems with generating a proper ordering for large sub-domains with many finite elements and high polynomial orders of approximation. The optimal order of elimination should start with the elimination of the most expensive element interiors, through the element faces and edges. The computational cost of the elimination of element interiors is three orders of magnitude higher than of the elimination of element faces and edges,
b) in order to improve the scalability of the solver, it is necessary to provide a quasi-optimal order of elimination for the interiors of sub-domains. It can be done by creating a new parallel solver working according to the elimination pattern strictly provided by the graph grammar model introduced in Section 2.1.4.

2.6.4. Scalability of the parallel self-adaptive *hp*-FEM algorithm in two and a half dimensions, with the grain defined on the level of initial mesh elements and with multi-level multi-frontal direct sub-structuring parallel solver

The fourth sequence of numerical experiments concerns the multi-level multi-frontal direct sub-structuring parallel solver algorithm, created on the basis of the graph grammar model of the direct solver presented in Section 2.1.4. The solver will be called shortly the *parallel recursive solver*. For the technical details on the implementation of the parallel recursive solver algorithm, see Appendix D.

The parallel recursive solver has been tested on the 3D DC borehole resistivity measurement simulations problem, formulated on the 2D mesh in the non-orthogonal system of coordinates, with Fourier series expansions in the third, azimuthal direction (compare Chapter 3.2.4).

The parallel recursive solver algorithm has been tested on the following three *hp* meshes, presented in Figure 2.163 on the LONESTAR cluster from the Texas Advanced Computing Center (TACC):

1) the first mesh has 2304 active finite elements and uniform p=3 throughout the entire mesh,
2) the second mesh has 9216 active finite elements and uniform p=4,
3) the third mesh is the optimal mesh acquired by performing 10 iterations of the self-adaptive *hp*-FEM. The mesh is highly non-uniform with the polynomial orders of approximation varying from p=1,...,8.

There are 10 Fourier modes used in the azimuthal direction (see Chapter 3.2.4). Thus, the total number of the degrees of freedom related to a single node is equal to 10p, where p denotes the polynomial orders of approximation in the node.

There are 576 initial mesh elements on each mesh. The first mesh has been obtained by performing one global *hp* refinement, i.e. each initial mesh element has been broken into 4 son elements, and the polynomial orders of approximation have been uniformly raised by one (from p=2 to p=3). It implies that the depth of the refinement trees is equal to 2 on the first mesh. The second mesh has been obtained by performing two global *hp* refinements, so the depth of the refinement tree is equal to 4. The third non-uniform mesh has been obtained by performing multiple *h*, *p* or *hp* refinements selected by the self-adaptive *hp*-FEM algorithm. The number of degrees of freedom as well as the number of non-zero entries is presented in Table 2.5.

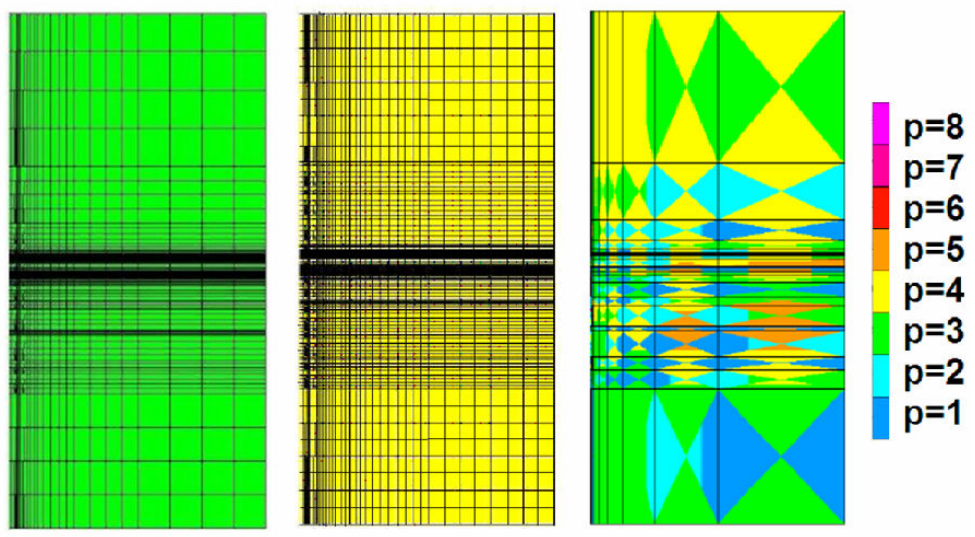


Fig.2.163. Three *hp* meshes used for testing the solver

The measurements presented in Figures 2.164, 2.166 and 2.168 describe the maximum time (on processors) spent for the sequential elimination on refinement trees and on sub-domains, as well as the total time spent on backward substitutions. The logarithmic scale is used to measure time in Figures 2.164 and 2.166. The Figures 2.165, 2.167, and 2.169 show

the maximum memory usage, where the maximum is taken from all nodes of the distributed elimination tree.

Table 2.5
Three computational meshes used in the numerical experiments

	First uniform p=3	Second uniform p=4	Third non-uniform p=1,…,8
Problem size N	315 555	1 482 570	51 290
Number of non-zero entries NZ	10 745 846	68 826 475	1 666 190

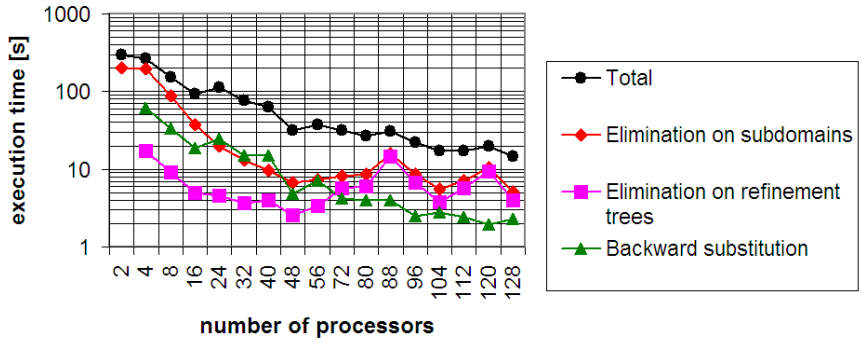


Fig.2.164. Execution times of the parallel recursive solver algorithm, measured on the first mesh

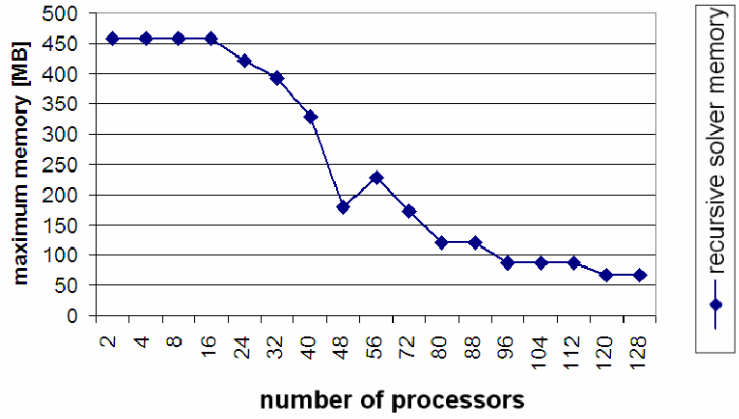


Fig.2.165. Maximum memory usage of the parallel recursive solver algorithm, measured on the first mesh

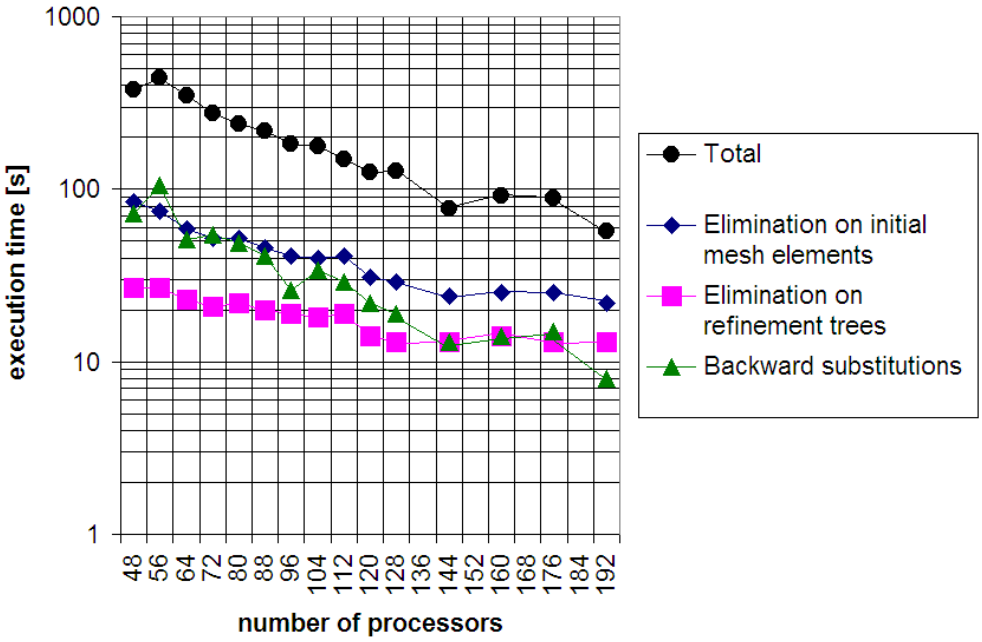


Fig.2.166. Execution times of the parallel recursive solver algorithm, measured on the second mesh

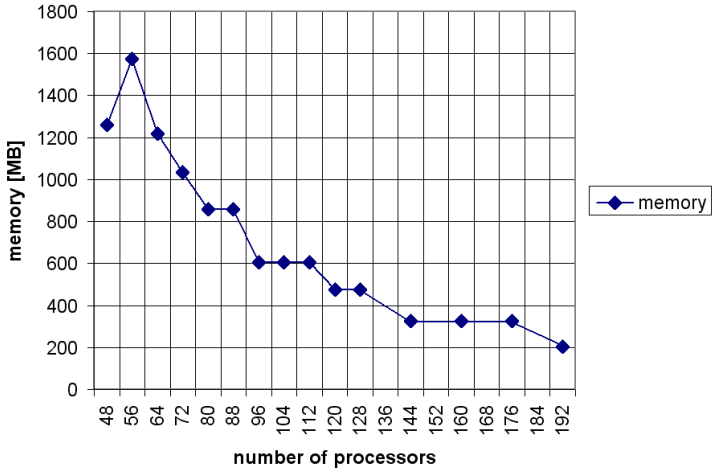


Fig.2.167. Maximum memory usage of the parallel recursive solver algorithm, measured on the second mesh

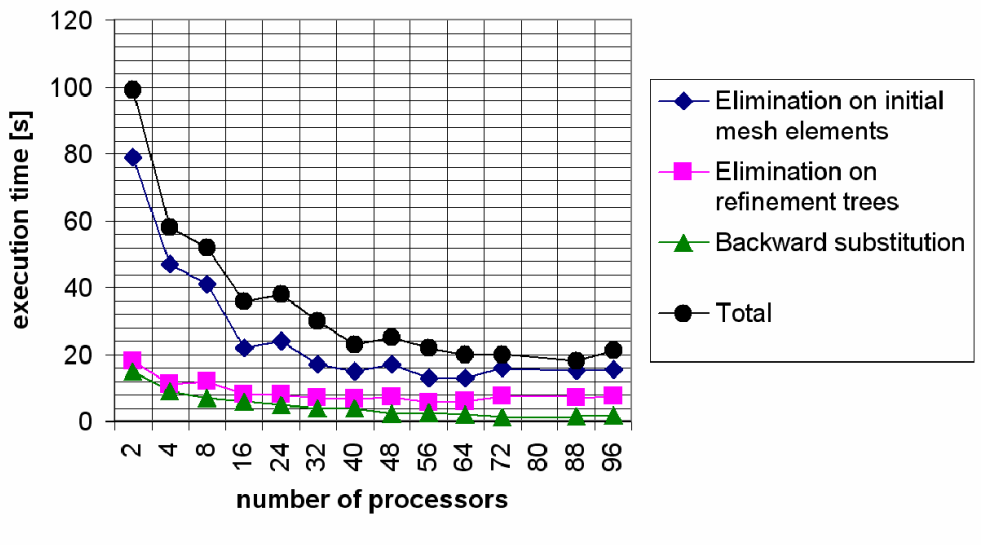


Fig.2.168. Execution times of the parallel recursive solver algorithm, measured on the third mesh

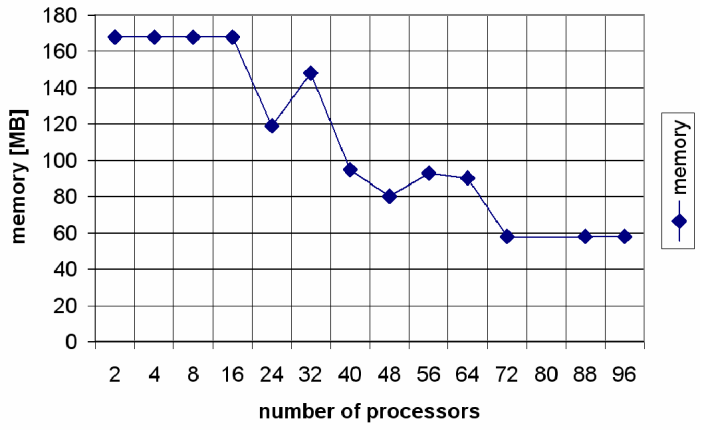


Fig.2.169. Maximum memory usage of the parallel recursive solver algorithm, measured on the third mesh

The following conclusions can be drawn from the presented measurements:

a) all considered meshes have been obtained from the initial mesh with $32 \cdot 18 = 576$ rectangular initial elements. All the meshes have been partitioned at the level of the initial mesh. The first and the second meshes have been uniformly *hp* refined, so each initial mesh element is uniformly loaded,

b) we achieve the maximum speedup of the parallel recursive solver algorithm and the minimum memory usage when the structure of the elimination tree is regular. If the sub-domains have a regular pattern (regular number of finite element layers), e.g.

as presented in Figure 2.171, the elimination tree also has a regular pattern: the depth of the elimination tree is uniform (all paths from the tree root down to each leaf have the same length). The performance of the solver algorithm is worse when the structure of the elimination tree is not uniform, e.g. there is the longest single path from the root of the elimination tree down to the deepest single leaf,

c) for the first mesh, the maximum speedup is achieved for 16 (576/16=36=6·6 layers) or 48 (576/48=12=4·3 layers) processors. Besides, the memory usage decreases rapidly for 48 or 96 (576/96=6=3·2 layers) processors.
For the second mesh, the maximum speedup is achieved for 144 (576/144=4=2·2 layers) or 192 (576/192=3=1·3 layers) processors, and the memory usage decreases for 96, 144 or 192 processors.
The third mesh is not uniformly *hp* refined and the mesh partition can be highly non-uniform, so the structure of the elimination tree can be also non-uniform. In this case, the maximum speedup or decrease of memory usage does not follow the above pattern,

d) the parallel recursive solver algorithm scales very well up to the maximum number of used processors, both in terms of the execution time and memory usage. The limitation of the solver scalability is the size of the maximal sequential part of the algorithm. This involves a recomputation of the partial LU factorizations on the longest path, from the root of the elimination tree, through the level of sub-domains, the level of initial mesh elements, down to the deepest leaf of the refinement tree. Such a limit has been probably reached for the third mesh, when using 72 processors.

The next numerical experiment concerns the measurements of the efficiency of the parallel solver. The solver has been again executed on the 3D DC borehole resistivity measurement simulations.

The employed computational mesh has been constructed from the 2D coarse mesh with 32·18 = 576 rectangular elements, by breaking each element into four sons and increasing the polynomial orders of approximation to 2 for every edge and every interior. The 3D problem has been reduced to 2D by using the Fourier series expansion in non-orthogonal system of coordinates (see Chapter 3.2.4 for more details). There are 10 Fourier modes employed at each node, and the total number of degrees of freedom is about 141, 000. The number of unknowns at each node is equal to the number of Fourier modes, the polynomial orders of approximation are equal to 2, so the comparison between theoretical and experimental efficiencies should be made as for p=20.

Table 2.6

Execution times of parallel recursive solver algorithm for an increasing number of processors

processors	1	2	4	8	12	16	20	24	28	32	40	48
time [s]	211	117	73	42	33	25	30	24	25	17	12	6.81
processors	56	64	72	80	88	96	104	112	120	128	144	160
time [s]	8.78	7	5.52	4.45	4.5	3.52	3.67	3.13	3.13	3.16	2.06	2.49
processors	176	192	208	224	240	256						
time [s]	2.41	1.72	1.85	2.02	2.12	2.26						

The execution times of parallel recursive solver algorithm for an increasing number of processors are presented in Table 2.6. The detailed efficiency measurements for an increasing number of processors are presented in Figure 2.170.

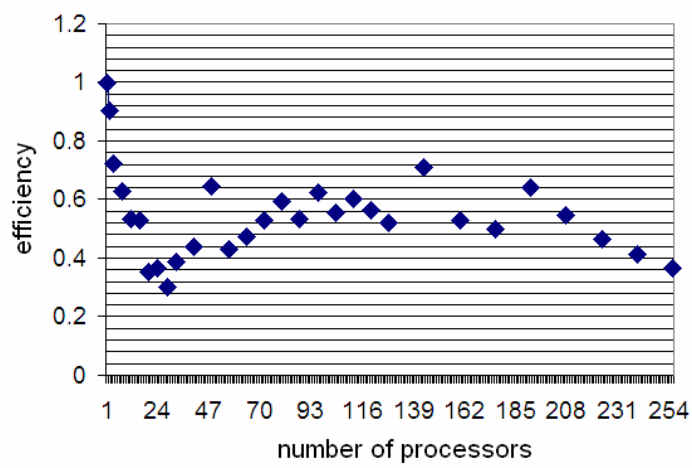


Fig.2.170. Efficiency of the parallel recursive solver algorithm measured on 141,000 degrees of freedom, for the mesh with 10 Fourier modes and uniform p=2

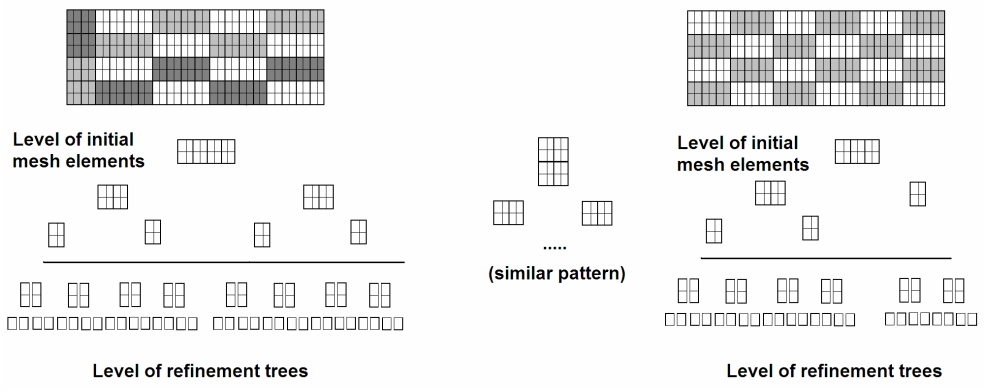


Fig.2.171. Structure of sub-domains for different numbers of processors

2.6.5. Comparison of the scalability of the multi-level multi-frontal direct sub-structuring parallel solver with the MUMPS parallel solver

This section presents a comparison of the parallel recursive solver with the MUMPS solver. The recursive solver has been compared with two versions of parallel MUMPS solver with METIS ordering:

1) the parallel MUMPS solver with distributed entries (the input matrix stored in the distributed manner, submitted from all processors in assembled format),
2) the direct sub-structuring method with sequential MUMPS solver used to compute the Schur complements on the sub-domains and with parallel MUMPS solver with distributed entries used to solve the interface problem.

The comparison has been performed on three meshes, described in detail at the beginning of this section. The measurements of the execution time and memory usage for the parallel recursive solver algorithm are illustrated in Figures 2.164 - 2.169. The corresponding measurements of the MUMPS solver with distributed entries are presented in Figures 2.172, 2.173, 2.176, 2.179, 2.180 and the corresponding measurements of the direct sub-structuring method with MUMPS solver are presented in Figures 2.174, 2.175, 2.177, 2.178 and 2.181.

The *Integration* in Figures 2.172, 2.176 and 2.179 means the integration of local matrices for *hp* finite elements. It is performed by the interface routine preparing an assembled list of non-zero entries for the MUMPS. In case of the MUMPS-based direct sub-structuring method presented in Figures 2.174, 2.177 and 2.181, the *Integration* stands for the integration for active finite elements, and the *Preparation* means transferring the Schur complement outputs into the lists of non-zero entries for the MUMPS parallel solver with distributed entries.

Note that the *Integration* for the MUMPS solver means the integration for active finite elements and it is performed in a loop when preparing an assembled list of non-zero entries in the interface to the MUMPS routine. In the parallel recursive solver algorithm, the integration for active finite elements is executed on the leaves of the elimination tree. These operations are included in the *Elimination on refinement trees*. The *Analysis* stage for the MUMPS involves the execution of the connectivity graph algorithm (e.g. METIS). This is not necessary in case of the parallel recursive solver algorithm, since the order of elimination is directly obtained from the mesh data structure (the binary tree constructed for the sub-domains and initial mesh elements, and the order of elimination on refinement trees follows the history of refinements stored in our data structure).

The *Factorization* for the MUMPS involves all *Elimination on initial mesh elements* and *Elimination on refinement trees*. There is no way to distinguish these two parts for the MUMPS solver. The *Backward substitution* for the recursive solver is equivalent to the *Solution* for the MUMPS.

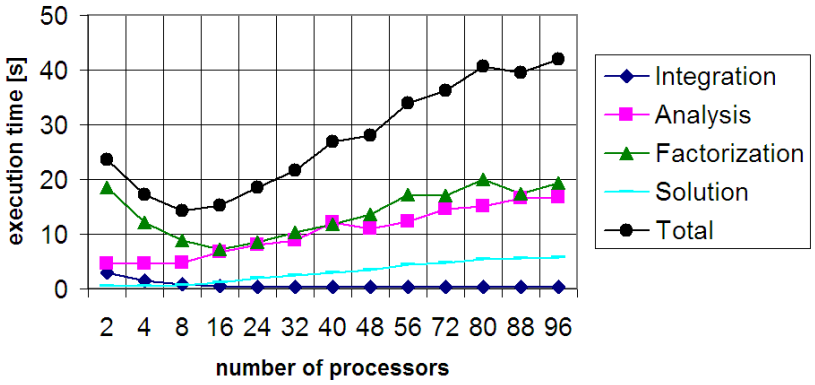


Fig.2.172. Execution time of the parallel MUMPS solver with distributed entries, measured on the first mesh

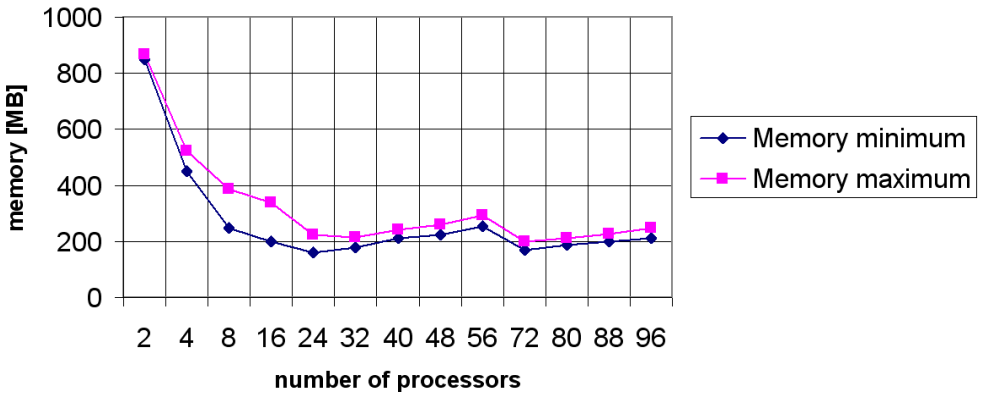


Fig.2.173. Minimum and maximum memory usage of the parallel MUMPS solver with distributed entries, measured on the first mesh

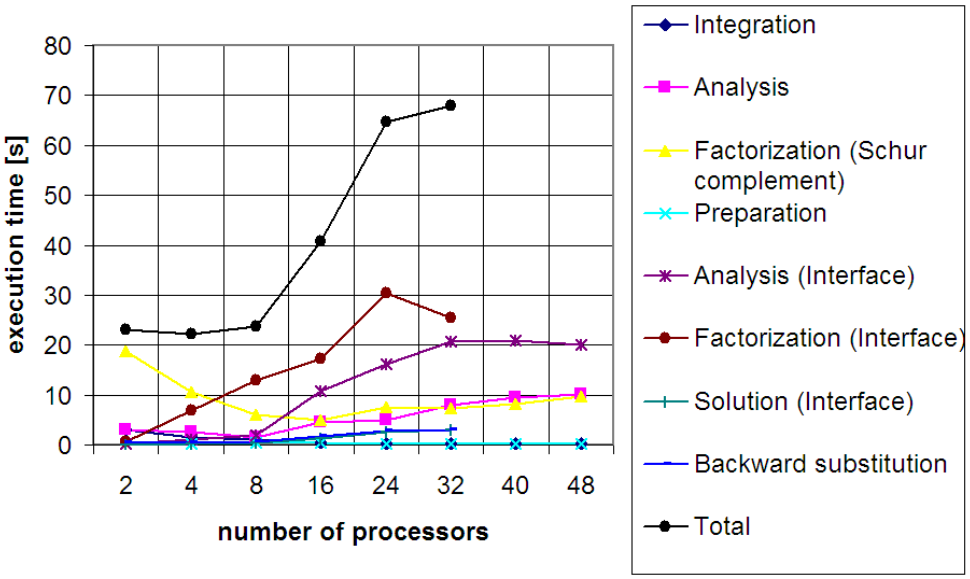


Fig.2.174. Execution times of particular parts of the MUMPS-based direct sub-structuring method, measured on the first mesh

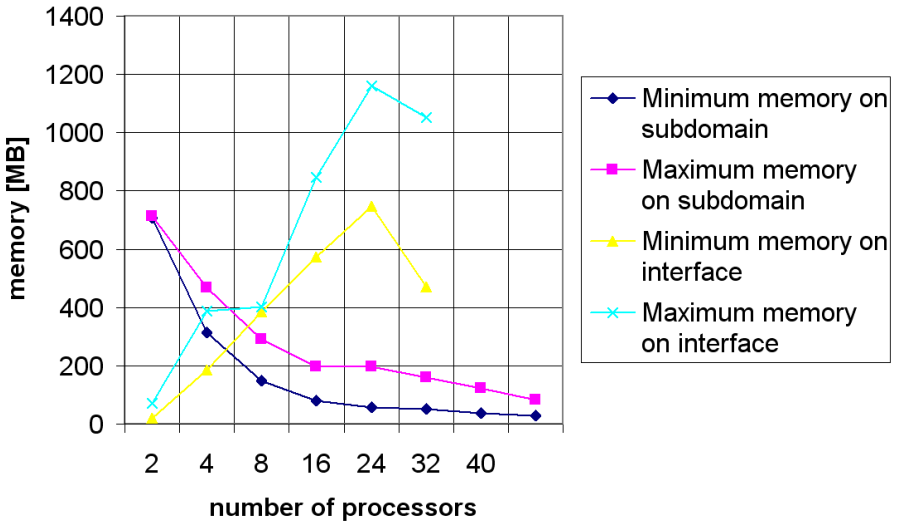


Fig.2.175. Minimum and maximum memory usage of the MUMPS-based direct sub-structuring method, measured on the first mesh

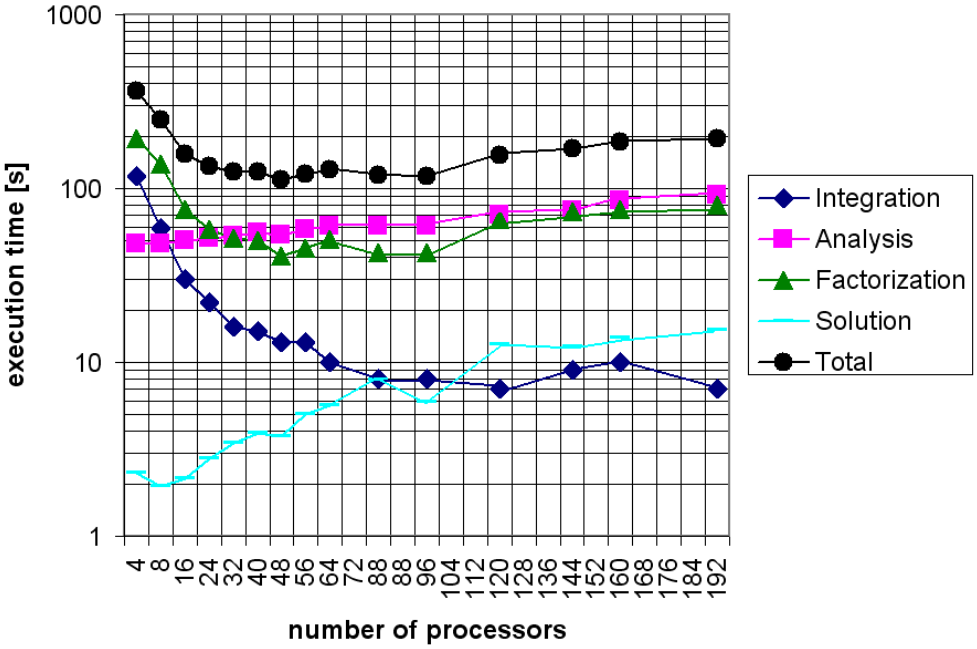


Fig.2.176. Execution time of the parallel MUMPS solver with distributed entries, measured on the second mesh

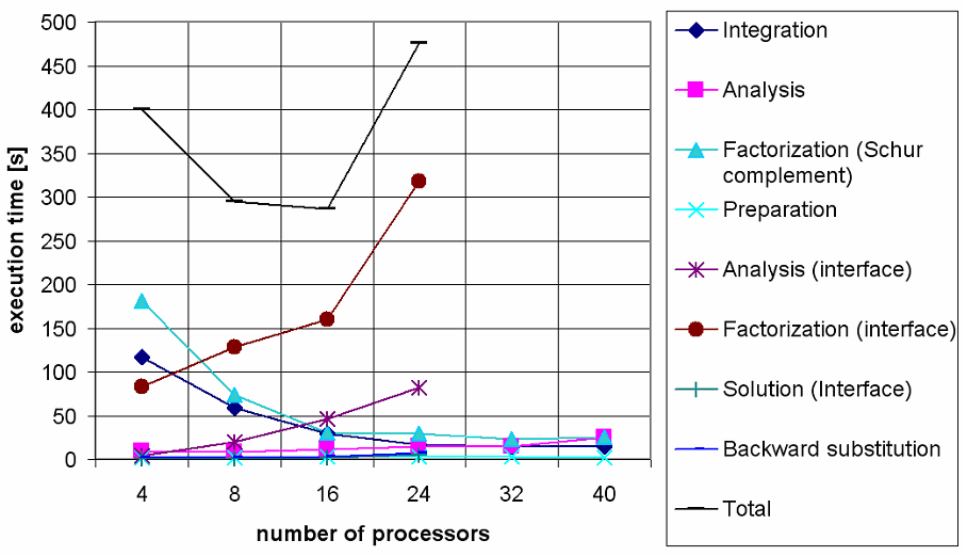


Fig.2.177. Execution times of particular parts of the MUMPS-based direct sub-structuring method, measured on the second mesh

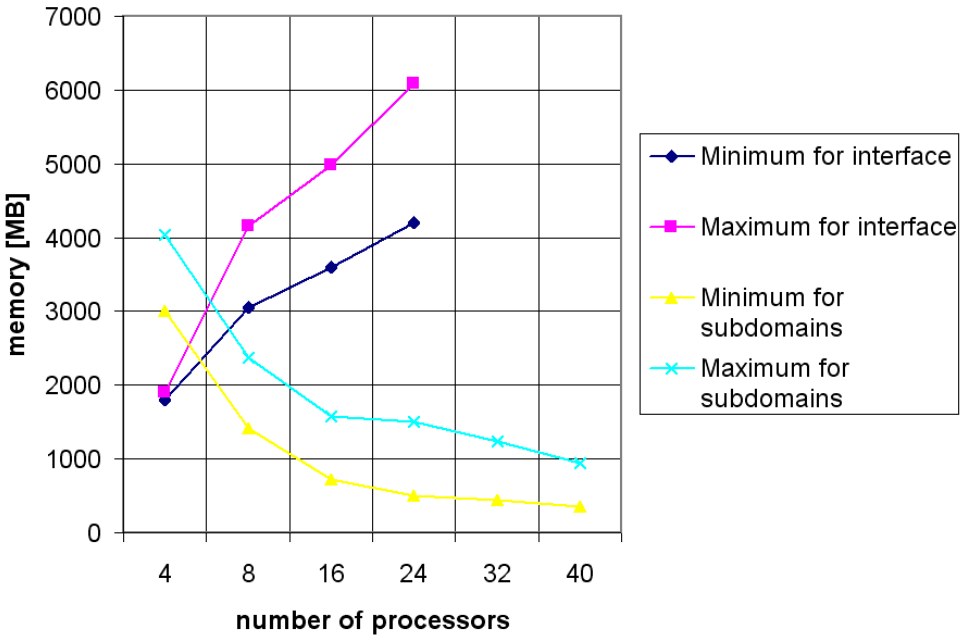


Fig.2.178. Minimum and maximum memory usage of the MUMPS-based direct sub-structuring method, measured on the second mesh

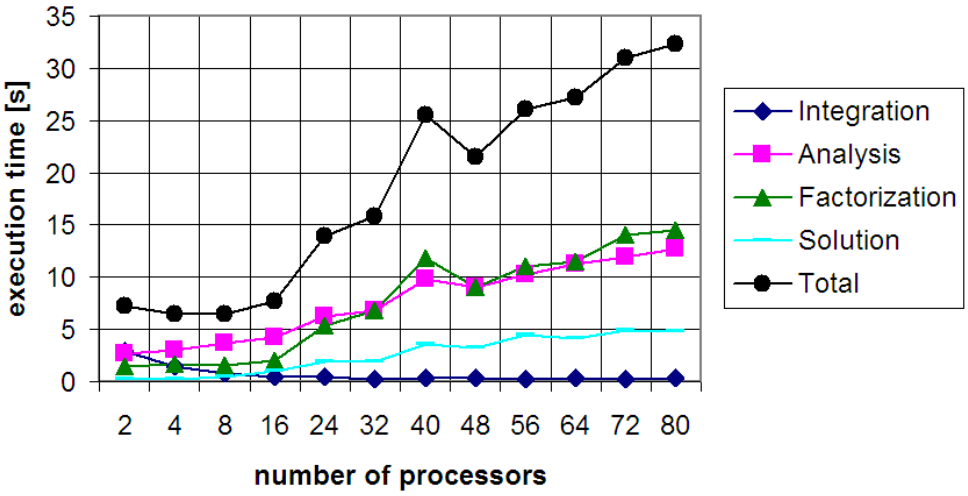


Fig.2.179. Execution time of the parallel MUMPS solver with distributed entries, measured on the third mesh

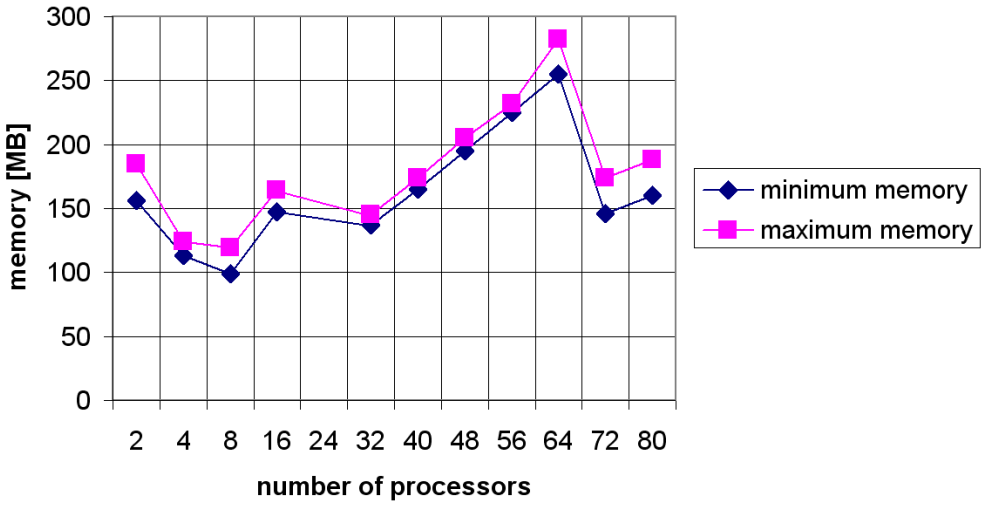


Fig.2.180. Minimum and maximum memory usage of the parallel MUMPS solver with distributed entries, measured on the third mesh

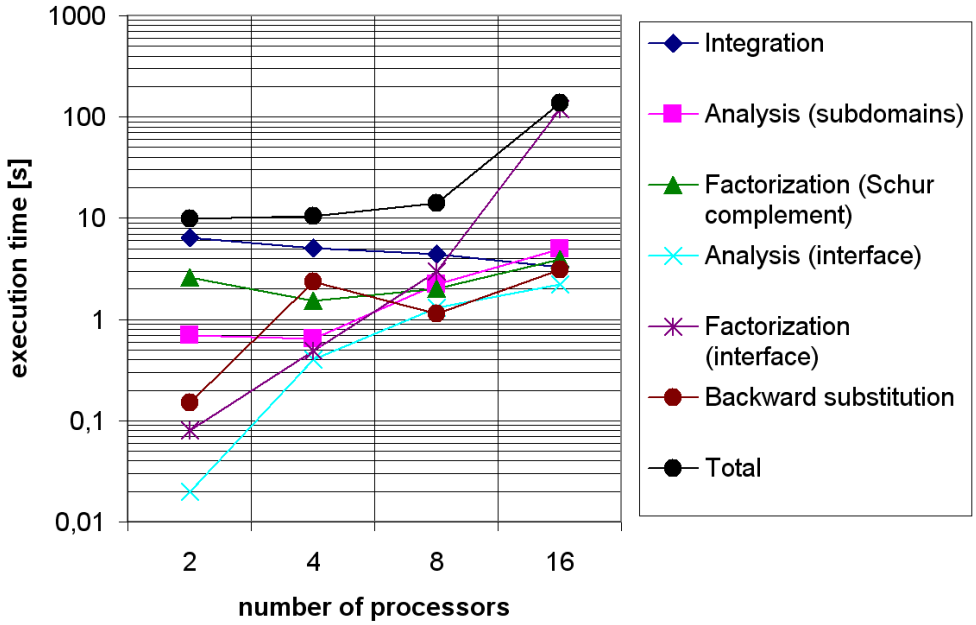


Fig.2.181. Execution times of particular parts of the MUMPS-based direct sub-structuring method, measured on the third mesh

We can draw the following conclusions about the efficiency of the parallel recursive solver algorithm:

a) we have introduced an efficient parallel direct solver for *hp*-FEM. It is based on the three-level elimination trees: sub-domains, initial mesh elements, and refinement trees. The solver uses the knowledge of the history of refinements to construct the third level elimination tree. The solver is modeled by a set of graph grammar productions presented in Chapters 2.1.2 and 2.1.4,
b) the comparison of the theoretical analysis with the experimental data shows that the efficiency of the implemented parallel recursive solver algorithm is worse for a low number of processors. This is because in the numerical experiment we have used the domain which has a shape not of square, but rather of a rectangle,
c) the efficiency of the paralell recursive solver algorithm is maximal for the number of processors that divides the number of finite elements of the computational mesh. There are 32*18 = 576 initial mesh elements. The maximum speedup for a large number of processors is reached on 144 or 192 processors. In case of 144 processors, there are 576/144 = 4 initial mesh elements per processor. In case of 192 processors, there are 576/192 = 3 initial mesh elements per processor. For both 4 and 3 initial mesh elements per processor, the elimination trees have a regular structure, presented in Figure 2.171, and the algorithm scales well. However, when the number of processors does not divide the number of finite elements, a structure of sub-domain may vary, the elimination trees have different sizes, which results in a loss of the solver scalability,
d) the efficiency of the parallel recursive solver algorithm is higher the computational problems with large number of degrees of freedom per node. The number of degrees of freedom per node is equal to the polynomial order of approximation multiplied by a number of equations. Thus, the solver becomes efficient if there are multiple equations in the problem (like multiple Fourier modes in the 3D DC borehole resistivity measurements simulation problem) or the polynomial order of approximation on the *hp* mesh is high. The self-adaptive *hp*-FEM increases the polynomial orders of approximation in consecutive iterations, so the efficiency of the solver grows when the mesh is more *p* refined.

The following conclusions can be drawn from the presented comparison of the parallel recursive solver with the MUMPS solver:

a) the parallel recursive solver is slower than the parallel MUMPS solvers, for a low number of processors, because of the following reasons:
   - the solver algorithms, which generate a local numbering of matrices at tree nodes and make decisions about the degrees of freedom that can be eliminated, are not optimized,
   - we assume that all matrices are non-symmetric to minimize the memory usage by performing the trick described in Appendix D, while the tested MUMPS parallel solvers work on symmetric matrices,
   - the backward substitution is accompanied by an additional full forward elimination, necessary to regenerate the structure of the LU factorization,
b) on the other hand, the parallel recursive solver scales very well, up to the maximum number of used processors, and becomes twice as fast as the MUMPS

solver for a large number of processors (57 seconds for the solver on the second mesh versus 112 seconds of the parallel MUMPS with distributed entries),

c) the MUMPS-based direct sub-structuring method usually runs out of memory for a large number of processors (40 or 32 for the first or the second mesh respectively - it requires more than 8096 MB of memory per processor),

d) all presented problems are relatively sparse, and all MUMPS-based parallel solvers reach the minimum execution time on 8 or 16 processors. However, the memory usage for all three types of MUMPS-based solver is large. The memory usage is usually stabilized for the parallel MUMPS with distributed entries, but the execution time increases. On the other hand, the memory usage of the parallel recursive solver is lower than for any MUMPS solver.

3. Applications

The parallel self-adaptive *hp*-FEM algorithm has been implemented in the *parhp2d* code (Paszyński, Kurtz, Demkowicz 2006). The two-dimensional code has been generalized to three dimensions in the *parhp3d* code (Paszyński, Demkowicz 2006). The technical details of both implementations are explained in Appendix C.

A number of two- and three-dimensional computational problems, presented in this chapter, have been solved by the implemented codes. Some of these problems are purely academic, whereas others are real-life applications to industry.

The numerical experiments presented in this chapter illustrate the power of the self-adaptive *hp*-FEM algorithm. Most of the presented problems are very challenging engineering problems, for which a highly accurate numerical solution is needed. The high accuracy solution on relatively small computational meshes may be achieved thanks to the exponential convergence of the self-adaptive *hp*-FEM algorithm. Such high accuracy solutions are either impossible to obtain at all with other numerical methods, or possible to obtain with classical methods, but on huge computational grids with millions of degrees of freedom.

First, Section 3.1.2 presents the battery problem from the Sandia National Laboratory. It is related to the minimization of energy loss due to internal heating of the battery. The high accuracy, less than 1% of the relative error in the energy norm is necessary. The self-adaptive *hp*-FEM is the only known algorithm providing such numerical solution in reasonable computational time (less than many hours).

Another challenging engineering problem, presented in Section 3.2.3, is the Step-and-Flash Imprint Lithography. The problem is connected with the simulation of the modern micro-cheap production technology. A high accuracy solution of this direct problem on computationally cheap meshes is required for the solution of the inverse problem related to the optimization of the production process.

Finally, in Sections 3.2.4 and 3.2.5 we describe the problems of 3D DC/AC borehole resistivity measurement simulations in deviated wells with steal casing. These are also challenging engineering problems that can be solved only by employing the parallel self-adaptive *hp*-FEM algorithm. The only existing solutions to these problems are those presented here, and it seems impossible to solve them by means of other known numerical methods (Pardo, Torres-Verdin, Paszyński 2008 and Pardo, Torres-Verdin, Nam, Paszyński, Calo 2008). The problem is formulated in Section 3.2.4 using the Fourier series expansions in non-orthogonal system of coordinates, and in Section 3.2.5 using fully three-dimensional formulation. These problems are of great interest to the oil industry.

All other computational problems are model academic problems used as benchmarks for the parallel self-adaptive *hp*-FEM algorithm. All of them have been used in numerical experiments performed to verify the parallel self-adaptive *hp*-FEM algorithms, to test their scalability, efficiency and speedup.

This chapter provides also the exact mathematical formulation of all these problems.

Fig.3.1. Different colours for different polynomial orders of approximations

The approximation spaces on the computational meshes presented in this chapter contain the polynomials of different orders. These orders are denoted by different colours, as presented in Figure 3.1.

3.1. Two-dimensional applications

3.1.1. L-shape domain model problem

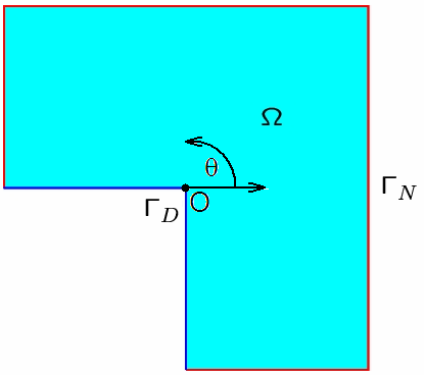


Fig.3.2. L-shape domain

The L-shape domain problem is a model academic problem formulated by Babuška in 1986, to test the convergence of the *p* and *hp* adaptive algorithms. The problem consists in solving the temperature distribution on the L-shape domain, presented in Figure 3.2, with fixed zero temperature in the internal part of the boundary, and the Neumann boundary condition prescribing the heat transfer on the external boundary. There is a single singularity in the central point of the domain (the gradient of temperature goes to infinity, compare Figure 3.3), so an accurate numerical solution requires a sequence of adaptations in the direction of the central point. The problem has been solved by the parallel *par2Dhp90* code (Paszyński, Kurtz, Demkowicz 2003, Paszyński, Kurtz, Demkowicz 2006).

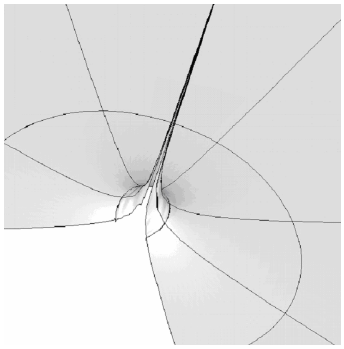


Fig.3.3. Detailed view at the $\|\nabla u\|$ at the central part of the mesh

3.1.1.1. Strong formulation

Find $u: R^2 \supset \Omega \ni x \to u(x) \in R$ the temperature distribution such that

$$\sum_{i=1}^{2} \frac{\partial^2 u}{\partial x_i^2} = 0 \text{ in } \Omega \tag{3.1}$$

with boundary conditions

$$u = 0 \text{ on } \Gamma_D \tag{3.2}$$

$$\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N \tag{3.3}$$

with n being the unit normal outward to $\partial\Omega$ vector, and

$$g(r,\theta) = r^{\frac{2}{3}} \sin^{\frac{2}{3}}\left(\theta + \frac{\Pi}{2}\right) \tag{3.4}$$

is defined in the radial system of coordinates with the origin point O presented in Figure 3.2. The formula (3.4) is actually the exact solution to the L-shape problem.

3.1.1.2. Weak formulation

Find $u \in V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{3.5}$$

$$b(u,v) = \int_{\Omega} \sum_{i=1}^{2} \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} dx \tag{3.6}$$

$$l(v) = \int_{\Gamma_N} g\, v\, dS \tag{3.7}$$

where

$$V = \left\{ v \in L^2(\Omega) : \int_{\Omega} \|v\|^2 + \|\nabla v\|^2 \, dx < \infty : \mathrm{tr}(v) = 0 \text{ on } \Gamma_D \right\} \tag{3.8}$$

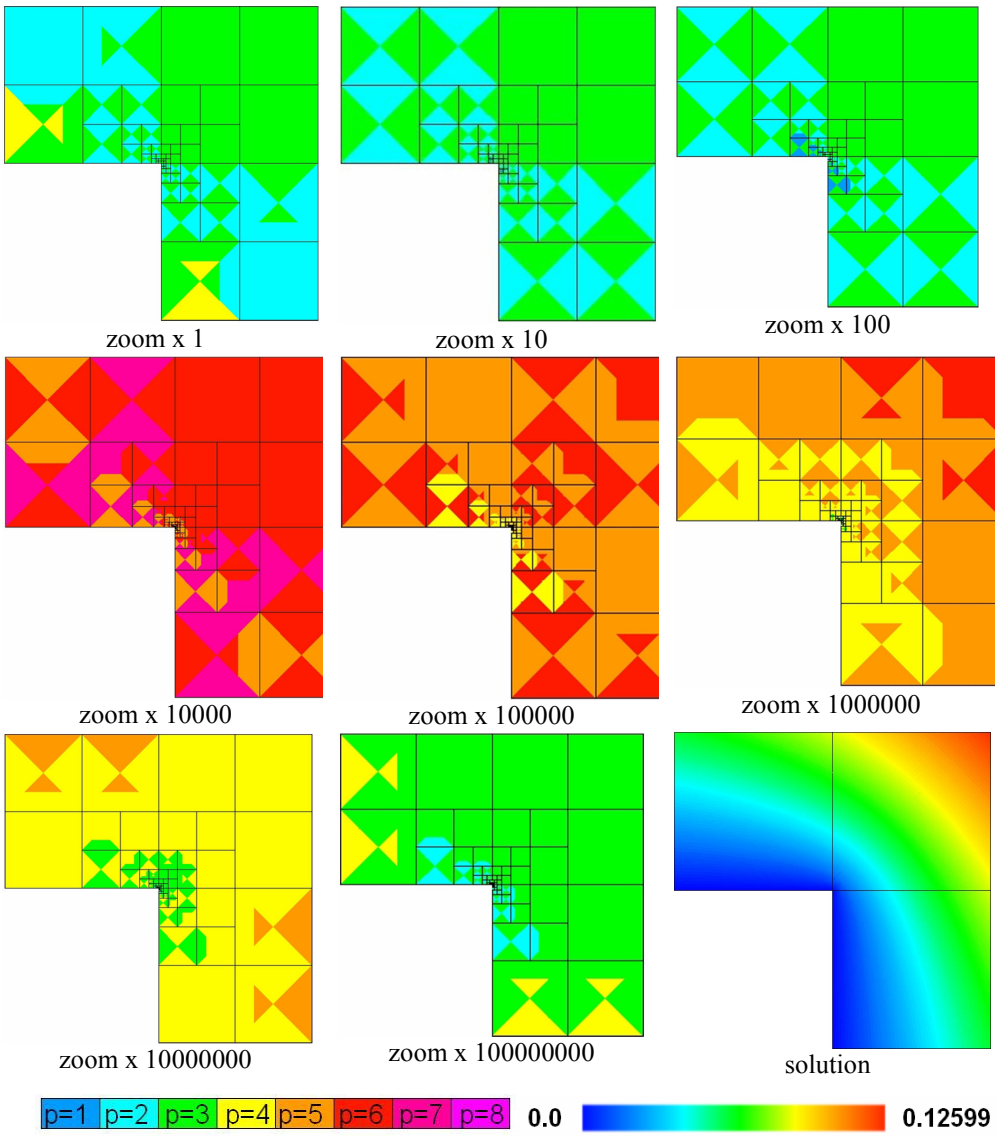


Fig.3.4. Sequence of meshes generated by the self-adaptive *hp*-FEM algorithm, for the Fichera problem. The final optimal mesh delivers a solution with relative error below 0,001%. The resulting temperature distribution is also presented here. The bottom panels present different colours denoting different polynomial orders of approximation and the scale for the temperature scalar field

3.1.1.3. Results

The self-adaptive *hp*-FEM generates a sequence of meshes delivering exponential convergence of the relative error in the energy norm, with respect to the number of degrees

of freedom. The sequence of meshes together with the solution is presented in Figure 3.4. The corresponding convergence curve has been discussed in Preface (compare Figure 1.5)

3.1.2. Battery problem

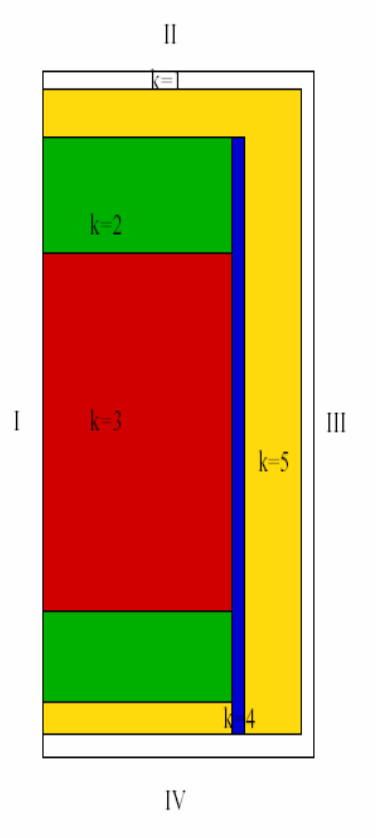


Fig.3.5. Domain for the battery problem

The battery problem comes from the Sandia National Laboratory, located in New Mexico, USA, the largest US government-owned scientific laboratory. The domain of the problem presented in Figure 3.5 contains five highly anisotropic materials. The problem consists in solving the Poisson equation, with an anisotropic heat transfer coefficient K depending on a kind of material and on the heat transfer direction. There are strong singularities in all areas where three different materials meet. The self-adaptive *hp*-FEM algorithm is the only way to solve this problem with accuracy less then 1% of the relative error in the energy norm. The optimal mesh generated after 43 iterations of the self-adaptive *hp*-FEM algorithm requires less than 3000 degrees of freedom. All other mesh adaptation techniques require *several millions* of degrees of freedom to achieve such 1%

relative error accuracy. Figure 3.7 shows the comparison of convergence histories of the self-adaptive *hp*-FEM algorithm with the *h* adaptive algorithm. The battery problem has been solved by *par2Dhp90* code (Paszyński, Kurtz, Demkowicz 2006).

3.1.2.1. Strong formulation

Find $u: R^2 \supset \Omega \ni x \to u(x) \in R$ the temperature distribution such that

$$-\sum_{i,j=1}^{2} \frac{\partial}{\partial x_i}\left(K_{ij} \frac{\partial u}{\partial x_j} \right) = f^{(k)} \text{ in } \Omega \tag{3.9}$$

with anisotropic heat transfer

$$K = K^{(k)} = \begin{bmatrix} K_{11}^{(k)} & 0 \\ 0 & K_{22}^{(k)} \end{bmatrix} \tag{3.10}$$

defined for particular materials $(k)=1,...,5$ as

Table 3.1
Heat transfer coefficients for different materials

Material (k)	1	2	3	4	5
${K_{11}}^{(k)}$	25	7	5	0.2	0.05
${K_{22}}^{(k)}$	25	0.8	0.0001	0.2	0.05

with heat source term defined for particular materials $(k)=1,...,5$ as

Table 3.2
Heat source values for different materials

Material (k)	1	2	3	4	5
$f^{(k)}$	0	1	1	0	0

$$\sum_{i=1}^{2} K_{ii} \frac{\partial u}{\partial x_i} n_i = g^{(k)} - \beta^{(k)} u \text{ on } \Gamma_k \tag{3.11}$$

where $g^{(k)}, \beta^{(k)}$ for particular parts of the boundary $k \in \{I, II, III, IV\}$ are given in Table 3.3.

Table 3.3
Boundary condition data for different materials

Boundary condition data	$\beta^{(k)}$	$g^{(k)}$
I	0.0	0.0
II	1.0	3.0
III	2.0	2.0
IV	3.0	1.0

3.1.2.2. Weak formulation

Find $u \in V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{3.12}$$

$$b(u,v) = \int_{\Omega} \sum_{i=1}^{2} K_{ii} \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} dS + \int_{\Gamma} \beta u v \, dS \tag{3.13}$$

$$l(v) = \int_{\Gamma} f v \, dS + \int_{\Gamma} g v \, dS \tag{3.14}$$

where

$$V = \left\{ v \in L^2(\Omega) : \int_{\Omega} \|v\|^2 + \|\nabla v\|^2 \, dx < \infty \right\} \tag{3.15}$$

3.1.2.3. Results

The self-adaptive *hp*-FEM generates a sequence of meshes delivering the exponential convergence, as presented in Figure 3.6. The resulting optimal mesh provides the solution with 0.1 % relative error accuracy (see Figures 3.8 and 3.9).

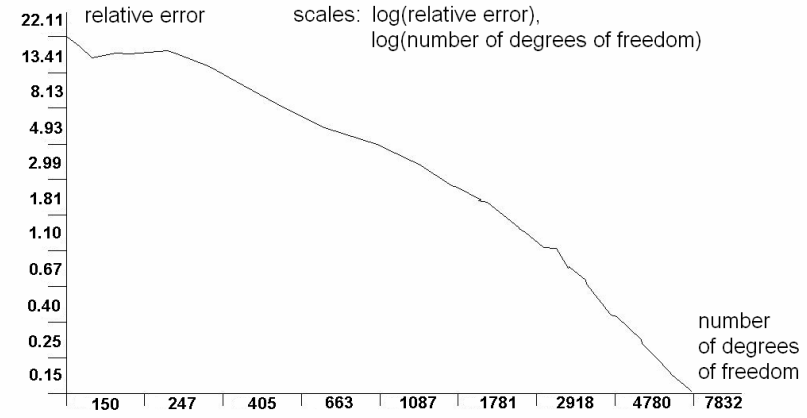


Fig.3.6. Convergence history of the self-adaptive *hp*-FEM algorithm for the battery problem

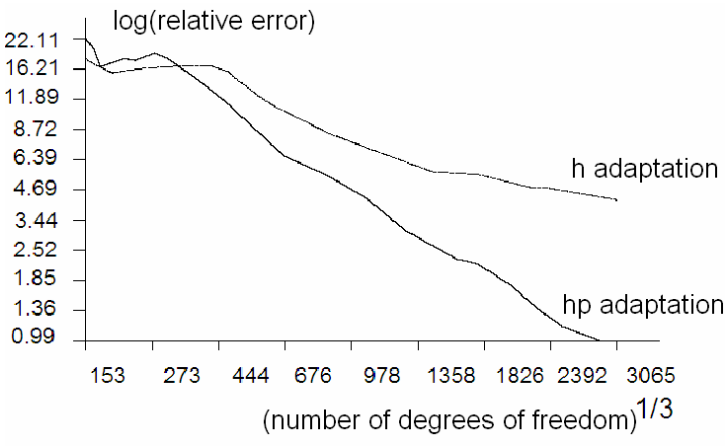


Fig.3.7. Comparison of convergence of *h* and *hp* adaptive algorithms

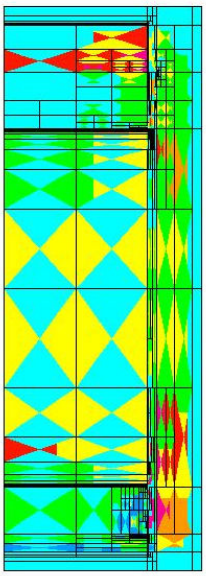


Fig.3.8. Optimal mesh generated by the self-adaptive *hp*-FEM.
Different colours denote different polynomial orders of approximations (compare Figure 3.1.)

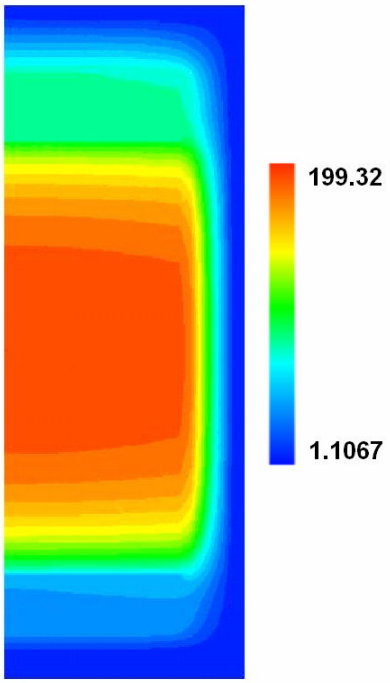


Fig.3.9. Solution with 0.1% relative error

It should be emphasized that the solution of the problem with 0.1% relative error in the energy norm can be achieved in practice *only* by using the self-adaptive *hp*-FEM algorithm. This is because the size of the computational mesh generated by other mesh adaptation algorithms providing the 0.1% accuracy of the solution is huge. Let us compare the history of convergence of the self-adaptive *hp*-FEM algorithm with the self-adaptive *h*-FEM algorithm (the fully automatic *h* adaptivity on a uniform p=2 meshes). All other mesh adaptation techniques based on the two-grid paradigm are worse than the self-adaptive *hp*-FEM and self-adaptive *h*-FEM algorithms, so it is enough to compare only those two algorithms. The lack of exponential convergence of the self-adaptive *h*-FEM algorithm, illustrated in Figure 3.7, implies that the computational mesh providing 0.1% accuracy will contain several millions of degrees of freedom, while the optimal mesh generated by the self-adaptive *hp*-FEM contains less than 3000 degrees of freedom only.

3.2. Three-dimensional applications

3.2.1. Fichera model problem

The Fichera problem is a model academic problem developed to test the convergence of the adaptive algorithms. It is a three-dimensional generalization of the L-shape domain problem described in Chapter 3.1.1. The Fichera model problem is defined for a cubic domain with 1/8 of the cube removed (see Figure 3.10). There is also a single singularity at the central point of the domain. The Neumann boundary conditions employ the solution g to the L-shape domain problem. The Fichera problem has been solved with *par3Dhp90* code (Paszyński, Demkowicz 2007).

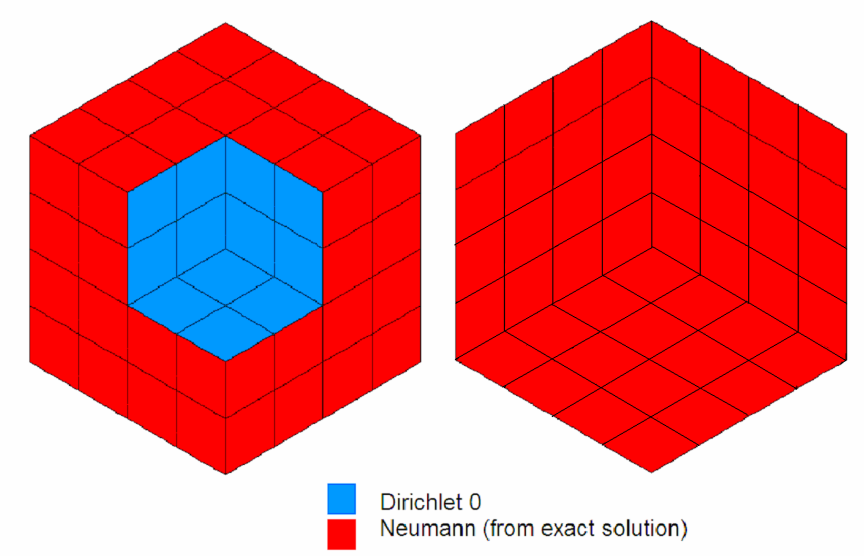


Fig.3.10. Domain for the Fichera problem

3.2.1.1. Strong formulation

Find $u: R^3 \supset \Omega \ni x \to u(x) \in R$ the temperature distribution such that

$$\sum_{i=1}^{3} \frac{\partial^2 u}{\partial x_i^2} = 0 \text{ in } \Omega \tag{3.16}$$

with boundary conditions

$$u = 0 \text{ on } \Gamma_D \tag{3.17}$$

$$\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N \tag{3.18}$$

with n being the unit normal outward to $\partial\Omega$ vector, and g is the exact solution (3.4).

3.2.1.2. Weak formulation

Find $u \in V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{3.19}$$

$$b(u,v) = \int_{\Omega} \sum_{i=1}^{3} \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} dx \tag{3.20}$$

$$l(v) = \int_{\Gamma_N} g\, v\, dS \tag{3.21}$$

where

$$V = \left\{ v \in L^2(\Omega) : \int_{\Omega} \|v\|^2 + \|\nabla v\|^2 \, dx < \infty : \mathrm{tr}(v) = 0 \text{ on } \Gamma_D \right\} \tag{3.22}$$

3.2.1.3. Results

The sequence of meshes generated by the self-adaptive *hp*-FEM together with the solution is presented in Figures 3.11 and 3.12. The corresponding exponential convergence curve is presented in Figure 3.13.

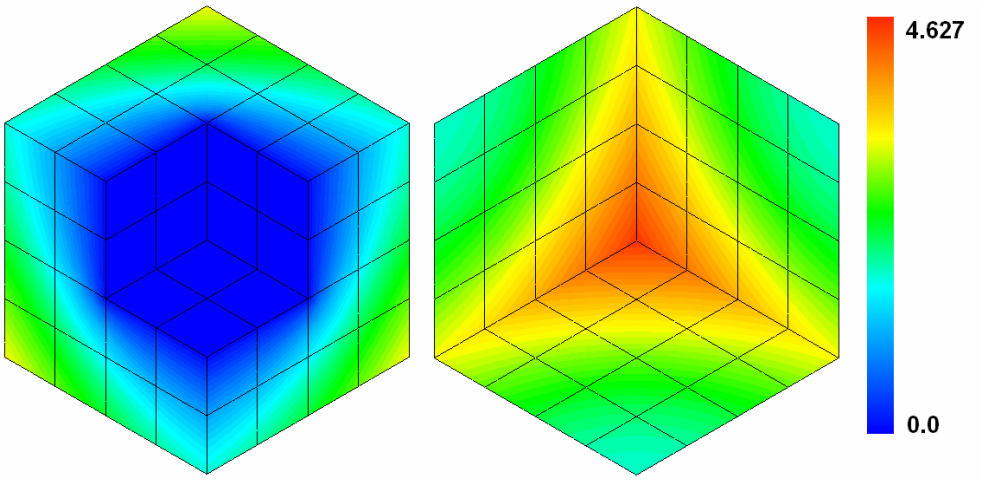


Fig.3.11. Solution of the Fichera problem with 1% relative error accuracy

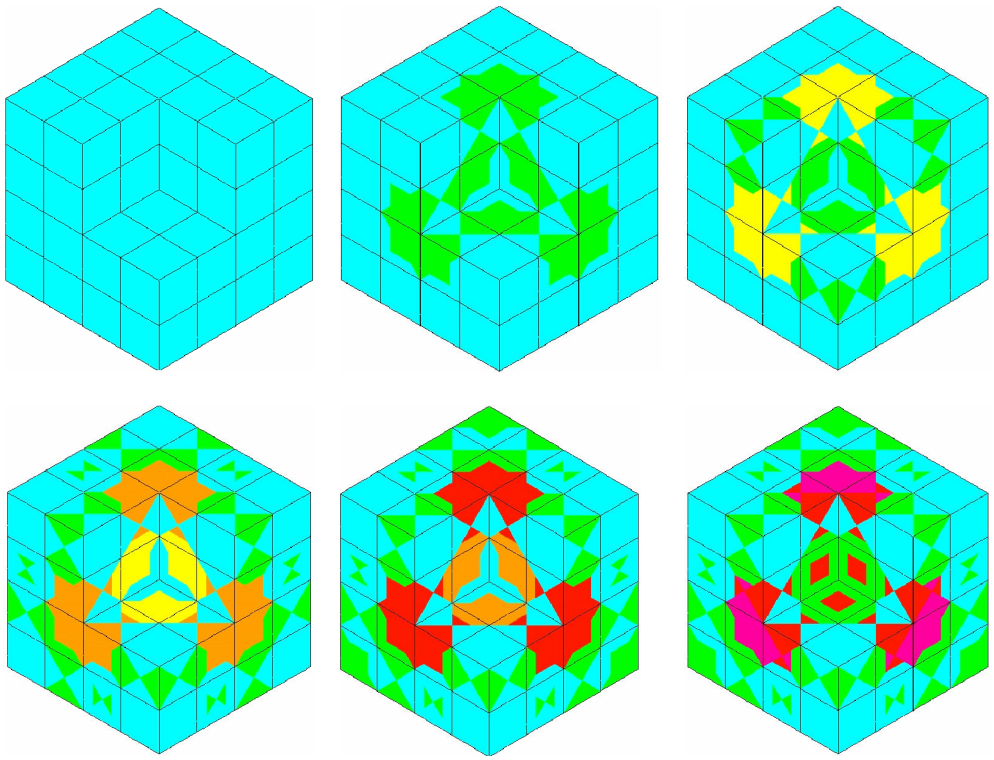


Fig.3.12. Sequence of meshes generated by the self-adaptive *hp*-FEM for the Fichera problem. Different colours denote different polynomial orders of approximations (compare Figure 3.1.)

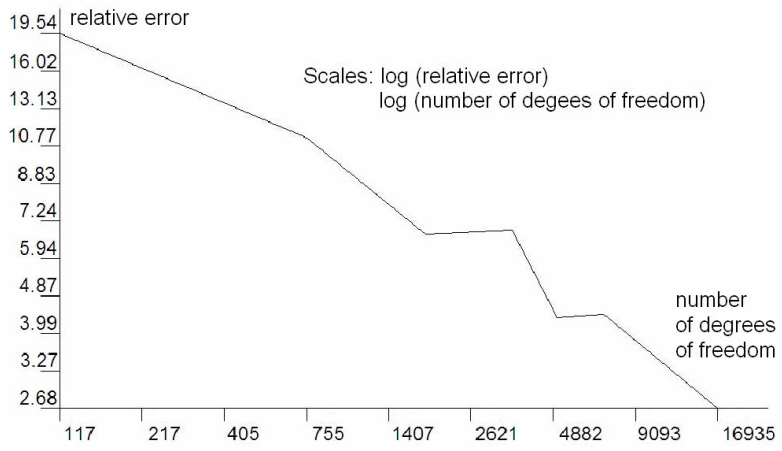


Fig.3.13. Convergence curve for the Fichera problem

3.2.2. Resistance heating of the Al-Si billet in steal die

In this section an industrial problem of the resistance heating of Al-Si billet in steel die for thixoforming process, formulated in Sołek 2004, is solved using the self-adaptive *hp*-FEM algorithm. It involves two interconnected problems: the electric current and the heating. In the original model, both of them were solved together. In the model presented here, we take into consideration only the heating problem. The heat generated as a result of electrical resistance is balanced with heat convection on model boundaries. This is a simplified approach providing that the heat is constant on each part of the domain.

The model geometry and boundary conditions are presented in Figure 3.14. Due to numerical needs, the model has been transformed into non-dimensional units. There are three main parts of the assembly: Al-Si billet, steel die and steel stamp.The interfaces between these parts are introduced as "artificial materials". The material properties for all real and artificial materials are presented in Table 3.4. The low thermal conductivity coefficients of "interface materials" inhibits heat flux between all parts of the assembly. The heat generated as a result of electrical resistance is introduced as constant heat generated in the entire model area. For more details, see Paszyński, Macioł 2007.

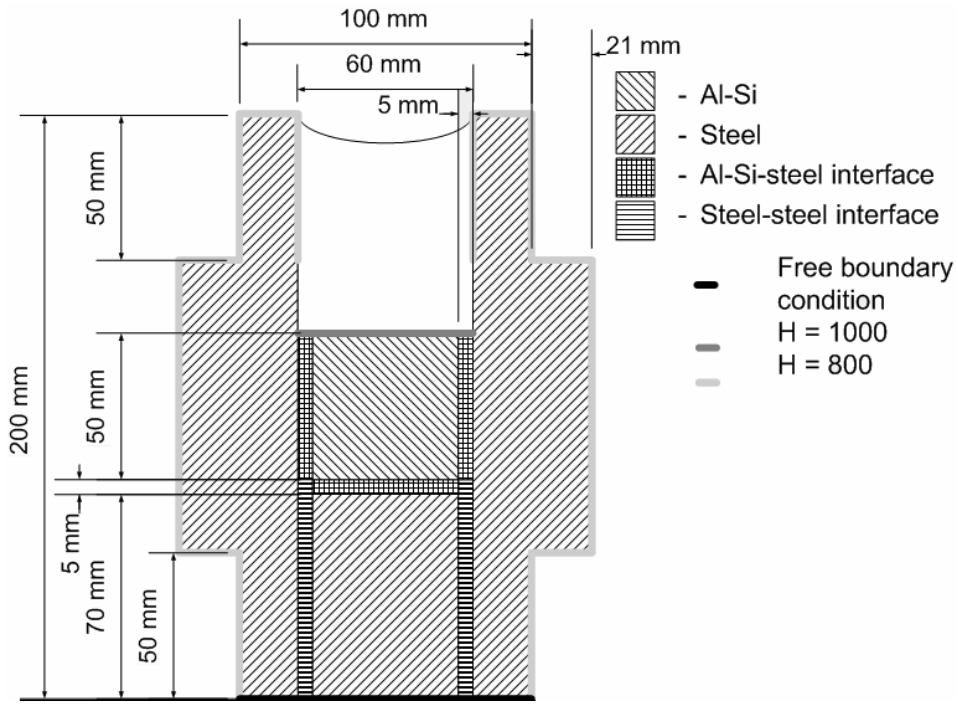


Fig.3.14. Computational domain for the resistance heating problem

Table 3.4
Material data for the resistance heating problem

Material	resistance R (generated heat $Q=iR^2$ [W/m^3])	thermal conductivity K	boundary convection H
	[Ohm]	[W/mK]	[W/m^{2}K]
Al-Si	1	160	1000
Steel	5	45	800
Al-Si interface	1	8	-
Steel-steel interface	1	5	-

3.2.2.1. Strong formulation

Find $u: R^3 \supset \Omega \ni x \to u(x) \in R$ the temperature distribution such that

$$-K\sum_{i=1}^{3}\frac{\partial^2 u}{\partial x_i^2} = Q \text{ in } \Omega \tag{3.23}$$

with boundary conditions

$$\frac{\partial u}{\partial n} = H\left(u_{env} - u\right) \text{ on } \Gamma_N \tag{3.24}$$

where n is the unit normal outward to $\partial\Omega$ vector, u_{env} is the ambient (enviromental) temperature, H is the boundary convection coefficient, defined on the Neumann boundary, denoted by light grey and grey colours in Figure 3.14, and free boundary condition (no boundary condition) on the other part of the boundary, denoted by black colour in Figure 3.14.

3.2.2.2. Weak formulation

Find $u \in V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{3.25}$$

$$b(u,v) = \int_{\Omega} K\sum_{i=1}^{3}\frac{\partial u}{\partial x_i}\frac{\partial v}{\partial x_i}dx + \int_{\Gamma} H\,u\,v\,dS \tag{3.26}$$

$$l(v) = \int_{\Gamma} f\,v\,dS + \int_{\Gamma} e_{env}\,v\,dS \tag{3.27}$$

where

$$V = \left\{ v \in L^2(\Omega): \int_{\Omega} \|v\|^2 + \|\nabla v\|^2\,dx < \infty \right\} \tag{3.28}$$

3.2.2.3. Results

The sequence of meshes and the solution are presented in Figures 3.15 and 3.16. The corresponding exponential convergence history is presented in Table 3.5. The result of this simulation is the temperature distribution. Figure 3.16 shows that the highest temperature occurs in the stamp. This effect is strongly undesired because Al-Si billet is heated mainly with the convection mechanism, and not with the resistance mechanism. The differences between thermal conduction coefficients and heat generation quantities result in considerable differences between temperatures in parts of the assembly. The temperature gradients also vary considerably.

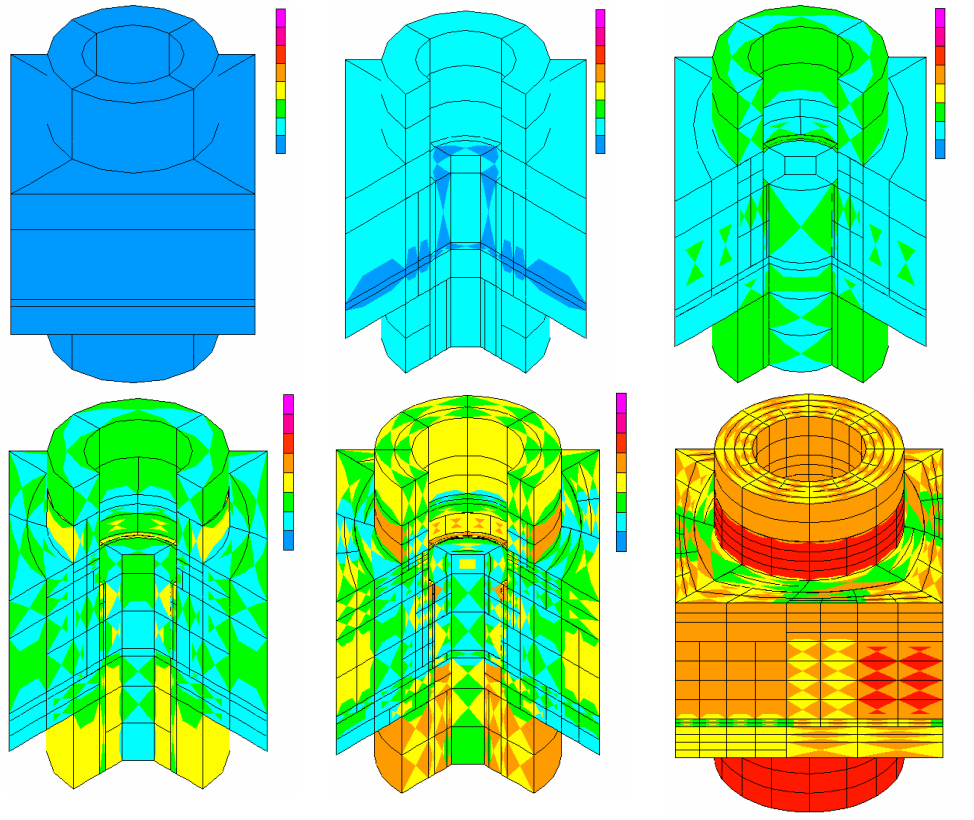


Fig.3.15. Sequence of meshes generated by self-adaptive *hp*-FEM for the resistance heating problem. Different colours denote different polynomial orders of approximations (compare Figure 3.1)

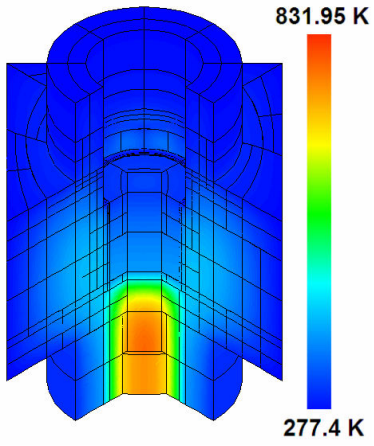


Fig.3.16. Solution of the resistance heating problem with 1% relative error accuracy

Table 3.5
Number of degrees of freedom and relative error for the generated sequence of meshes

Iteration	Number of degrees of freedom	Relative error in energy
1	116	58.43%
2	753	5.05%
3	2353	3.56%
4	6477	3.35%
5	32034	0.98%

3.2.3. Step-and-Flash Imprint Lithography simulations

Step and flash imprint lithography (SFIL) is a patterning process utilizing photopolymerization to replicate the topography of a template onto a substrate (Bailey, Colburn, Choi, Grot, Ekerdt, Sreenivasan, Willson, 2002, Colburn, Suez, Choi, Meissi, Bailey, Sreenivasan, Ekerdt, Willson, 2001, Burns, Johnson, Schmid, Kim, Dickey, Meiring, Burns, Stacey, Willson, 2004).

The SFIL process can be described in the following six steps, as it is illustrated in Figure 3.17.

1) *dispense*. The SFIL process employs a template / substrate alignment scheme to bring a rigid template and substrate into parallelism, trapping the etch barrier in the relief structure of the template,
2) *imprint*. The gap is closed until the force that ensures a thin base layer is reached,
3) *exposure*. The template is then illuminated through the backside to cure etch barrier,
4) *separate*. The template is withdrawn, leaving low-aspect ratio, high resolution features in the etch barrier,

5) *breakthrough etch*. The residual etch barrier (base layer) is etched away with a short halogen plasma etch,
6) *transfer etch*. The pattern is transferred into the transfer layer with an anisotropic oxygen reactive ion etch, creating high-aspect ratio, high resolution features in the organic transfer layer.

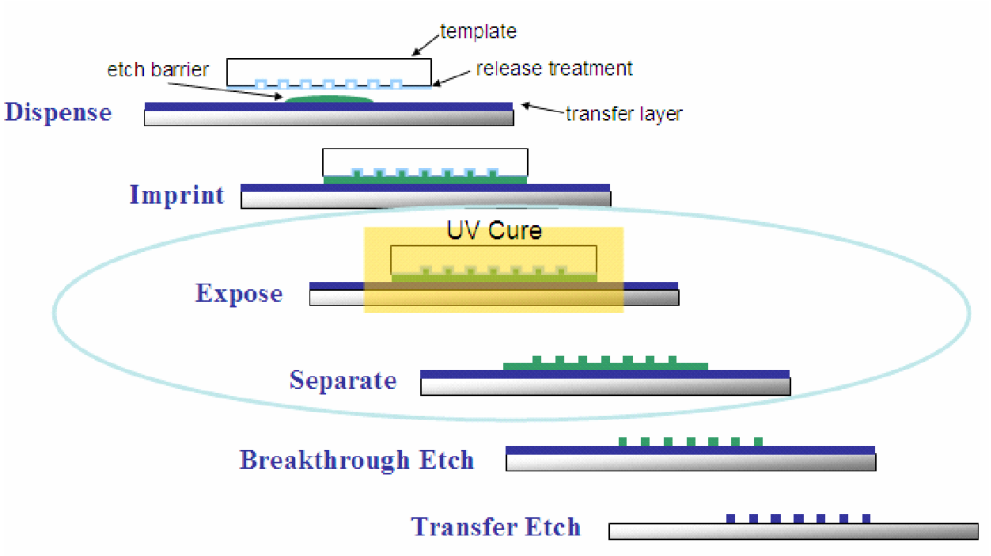


Fig.3.17. Step-and-Flash Imprint Lithography process

The major processing steps of SFIL include: depositing a low viscosity, silicon containing, photocurable etch barrier on to a substrate; bringing the template into contact with the etch barrier; curing the etch barrier solution through UV exposure; releasing the template, while leaving high-resolution features behind; a short, halogen break-through etch; and finally an anisotropic oxygen reactive ion etch to yield high aspect ratio, high resolution features.

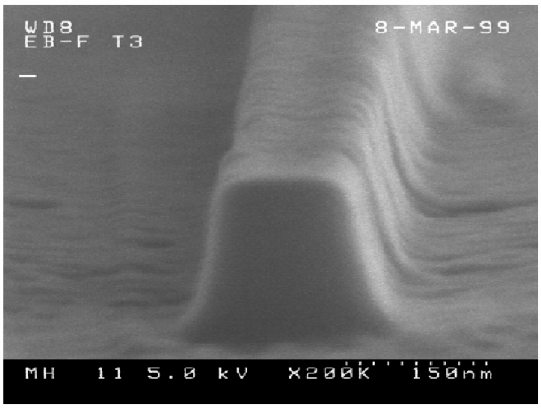


Fig.3.18. Shrinkage of the feature after removal of the template
(picture obtained from Prof. Grant C. Willson from The University of Texas at Austin)

Photopolymerization, however, is often accompanied by densification. The interaction potential between photopolymer precursors undergoing free radical polymerization changes from van der Waals' to covalent. The average distance between molecules decreases and causes volumetric contraction. Densification of the SFIL photopolymer (the etch barrier) may affect both the cross sectional shape of the feature and the placement of relief patterns. The exemplary shrinkage of the feature measured after removing the template is presented in Figure 3.18.

The linear elasticity model with thermal expansion coefficient is used to verify the material response of polymerized networks in cured etch-barrier layers that are formed during the third step of the SFIL process. The problem has been solved on the 3D cubic shape domain, presented in Figure 3.19.

For more details on the problem formulation, see Paszyński, Romkes, Collister, Meiring, Demkowicz, Willson, 2005.

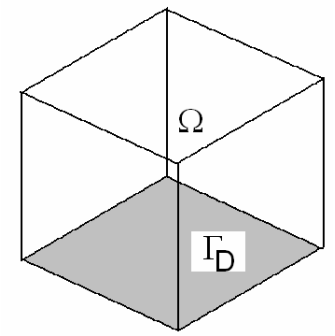


Fig.3.19. Domain for the SFIL problem

3.2.3.1. Strong formulation

Find $u: R^3 \supset \Omega \ni x \to u(x) = (u_1(x), u_2(x), u_3(x)) \in R^3$ the displacement vector field such that

$$\sum_{j=1}^{3} \frac{\partial \sigma_{ij}}{\partial x_j} = 0 \text{ in } \Omega \tag{3.29}$$

where

$$\sigma_{ij} = 2\mu\varepsilon_{ij} + \lambda\delta_{ij}\sum_{i=1}^{3}\varepsilon_{kk} - \alpha\delta_{ij}(T - T_o) \tag{3.30}$$

where

$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{3.31}$$

is the Green tensor, δ_{ij} is the Kronecker delta, μ and λ are Lame coefficients, defined in terms of the Young modulus E and Poisson ratio ν

$$\mu = \frac{E}{2(1+\nu)} \tag{3.32}$$

$$\lambda = \frac{\nu E}{1+\nu} \tag{3.33}$$

The α is the thermal expansion coefficient, defined as the volumetric shrinkage of the etch barrier for the 1 degree increase of the temperature $\Delta T = (T - T_0) = 1$

$$\alpha = \frac{\Delta V}{V\Delta T} < 0 \tag{3.34}$$

The values of the Young modulus, Poisson ratio and the thermal expansion coefficient are provided by the experiments (Colburn, 2001) and by the inverse analysis (Paszyński, Barabasz, Schaefer, 2007). The values used in the numerical simulation are listed in Table 3.6.

Table 3.6
Values of the Young modulus and Poisson ratio obtained by measurements, and the value of the thermal expansion coefficient obtained by inverse analysis

Parameter	E	ν	α
Value	1 [GPa]	0.3	-0.06115

The boundary conditions for the SFIL problem are

$$u = 0 \text{ on } \Gamma_D \tag{3.35}$$

with free boundary condition (no boundary condition) on the external boundary.

3.2.3.2. Weak formulation

Find $u \in V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{3.36}$$

$$b(u,v) = \int_{\Omega} \sum_{i,j,k,l=1}^{3} E_{ijkl} \frac{\partial u_k}{\partial x_l} \frac{\partial v_i}{\partial x_j} dx \tag{3.37}$$

$$l(v) = \alpha \Delta T \int_{\Gamma_N} \sum_{i=1}^{3} \frac{\partial v_i}{\partial x_i} dS \tag{3.38}$$

where

$$E_{ijkl} = \mu(\delta_{ik}\,\delta_{jl} + \delta_{il}\,\delta_{jk}) + \lambda\,\delta_{ij}\,\delta_{kl} \tag{3.39}$$

and

$$V = \left\{ v \in L^2(\Omega) : \int_{\Omega} \|v\|^2 + \|\nabla v\|^2 \, dx < \infty : \mathrm{tr}(v) = 0 \text{ on } \Gamma_D \right\} \tag{3.40}$$

3.2.3.3. Results

The convergence of the self-adaptive *hp*-FEM for this problem is also exponential. The sequence of meshes and the solution are presented in Figures 3.20 and 3.21. The corresponding convergence curve is presented in Figure 3.22.

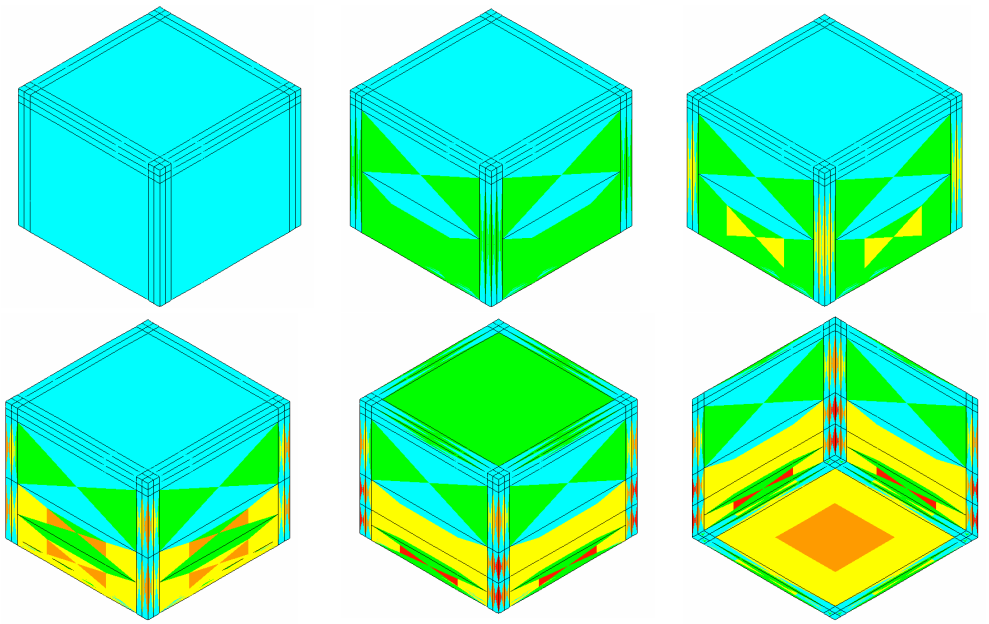


Fig.3.20. Sequence of meshes generated by the self-adaptive *hp*-FEM for the SFIL problem, with the front and bottom of the final mesh. Different colours denote different polynomial orders of approximations presented in Figure 3.1

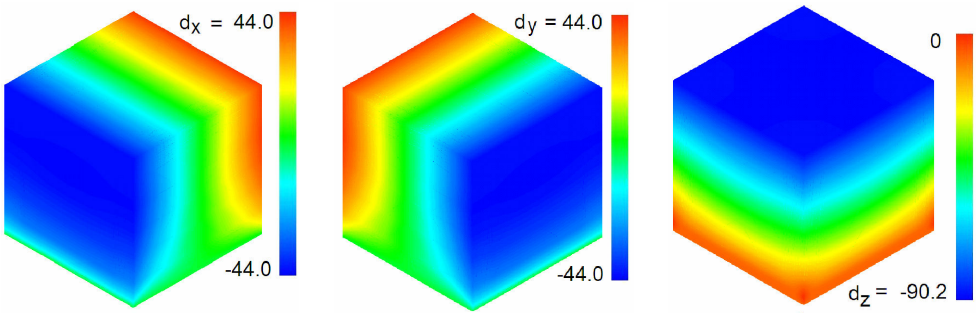


Fig.3.21. Three components of the displacement field (in nanometers) for the solution obtained on the final mesh

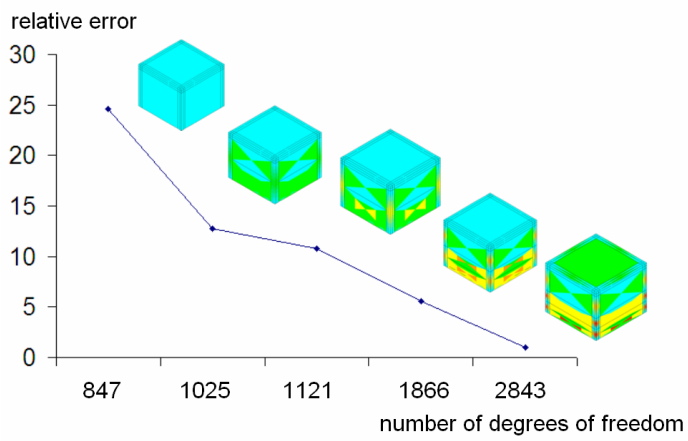


Fig.3.22. History of convergence for the SFIL problem

Note that the optimal mesh providing a high accuracy solution generated by the self-adaptive *hp*-FEM algorithm contains only 2843 degrees of freedom. This is critical for the solution of the inverse problem, which requires multiple direct problem solutions for different values of the model parameters, such as the Young modulus and Poisson ratio. This has been discussed also in Paszyński, Barabasz, Schaefer 2007.

3.2.4. 3D DC/AC borehole resistivity measurement simulations in deviated wells with non-orthogonal system of coordinates

In this section we discuss the problem of 3D direct and alternate current (DC/AC) borehole resistivity measurement simulations in deviated well. The problem comes from the Joint Research Consortium on the Formation Evaluation, the University of Texas in Austin.

The expression "simulation of measurements" is widely used within the geophysical community. A quantity of interest, in this case the voltage, is measured at a receiver electrode located in the logging instrument. This section presents computer simulations of how those measurements are obtained.

Nowadays, the logging instruments are equipped with several transmitter and receiver electrodes. These instruments move along the axis of the borehole and measure the voltage induced at the receiver electrodes at different positions. The voltage measured at the receivers is expected to be proportional to the electrical conductivity of the nearby formation. Thus, the logging instruments are used to estimate the properties (in this case, the electrical conductivity) of the sub-surface material, with the ultimate objective of describing hydrocarbon (oil and gas) bearing formations.

In this chapter the behavior of a resistivity logging instrument is examined by performing its computer-based simulations for the borehole environment. The resulting simulator is intended to be used in future as a core part of the inverse problem infrastructure. The inverse infrastructure will give the possibility to determine the unknown conductivities of formation layers, on the basis of actual measurements recorded by logging instruments.

The resistivity logging tool with receiver and transmitter electrodes is moving along the trajectory of the well. The electromagnetic waves generated by the transmitter electrode are reflected from formation layers and recorded by the receiver electrodes. The oil industry is particularly interested in the 3D simulations of resistivity measurements in deviated wells, where the angle between the borehole and the formation layers is not equal to 90 degrees ($\theta_0 \neq 90$).

A fast numerical high accuracy solution of the direct problem is necessary to solve the inverse problem which consists in localizing the oil formation in the ground. The solution of this problem is of great interest to the oil industry. It is possible to find a high accuracy solution of the direct problem only by using the parallel self-adaptive *hp*-FEM algorithm presented in this dissertation.

3.2.4.1. Strong formulation

Find $u: R^3 \supset \Omega \ni x \to u(x) \in R$ the electrostatic scalar potential such that

$$-\sum_{i=1}^{3} \sigma \frac{\partial^2 u}{\partial x_i^2} = \nabla \circ J \text{ in } \Omega \tag{3.41}$$

(conductive media equation) where $\nabla \circ J$ is the load (divergence of the impressed current, Pardo, Demkowicz, Torres-Verdin, Paszyński 2006) and σ represents the conductivity of the media.

3.2.4.2. Weak formulation

To solve the above 3D problem (3.41), we introduce the new quasi-cylindrical non-orthogonal system of coordinates shown in Figure 3.23. This chapter presents a summary of the derivation presented in detail in Pardo, Calo, Torres-Verdin, Nam 2007 and Pardo, Torres-Verdin, Nam, Paszyński, Calo 2008. The notation from these papers is also used here. The variational formulation with respect to the electric potential u in new system of coordinates can be expressed in the following way:

Find $u \in V$ such that:

$$\int_{\Omega} \sum_{n=1}^{3} \frac{\partial u}{\partial \zeta_n} \hat{\sigma} \frac{\partial v}{\partial \zeta_n} d\zeta = \int_{\Omega} v \nabla \circ \hat{J} \, d\zeta \quad \forall v \in V \tag{3.42}$$

where

$$V = \left\{ v \in L^2(\Omega) : \int_{\Omega} \|v\|^2 + \|\nabla v\|^2 dx < \infty : \mathrm{tr}(v) = 0 \text{ on } \Gamma_D \right\} \tag{3.43}$$

The electric conductivity of media in new system of coordinates is equal to $\hat{\sigma} := Jac^{-1} \sigma Jac^{-1^T} |Jac|$, and $\nabla \circ \hat{J} := \nabla \circ J |Jac|$ with Jac being the Jacobian matrix of the change of coordinates with respect to the Cartesian reference system of coordinates (x_1, x_2, x_3), namely,

$$Jac = \frac{\partial(x_1, x_2, x_3)}{\partial(\zeta_1, \zeta_2, \zeta_3)} \tag{3.44}$$

Then, we take a Fourier series expansion of the solution, material and ζ_2 direction

$$u(\zeta_1, \zeta_2, \zeta_3) = \sum_{l=-\infty}^{l=+\infty} u_l(\zeta_1, \zeta_3) e^{jl\zeta_2} \tag{3.45}$$

$$\sigma(\zeta_1, \zeta_2, \zeta_3) = \sum_{m=-\infty}^{m=+\infty} \sigma_m(\zeta_1, \zeta_3) e^{jm\zeta_2} \tag{3.46}$$

$$\nabla \circ J(\zeta_1, \zeta_2, \zeta_3) = \sum_{l=-\infty}^{l=+\infty} \nabla \circ J_l(\zeta_1, \zeta_3) e^{jl\zeta_2} \tag{3.47}$$

where

$$u_l = \frac{1}{2\Pi} \int_0^{2\Pi} u e^{-jl\zeta_2} d\zeta_2, \quad \sigma_m = \frac{1}{2\Pi} \int_0^{2\Pi} \sigma e^{-jm\zeta_2} d\zeta_2 \quad \text{and} \quad \nabla \circ J_l = \frac{1}{2\Pi} \int_0^{2\Pi} f e^{-jl\zeta_2} d\zeta_2$$

and j is the imaginary unit. We introduce symbol F_l such that applied to a scalar function u it produces the l^{th} Fourier modal coefficient u_l, and when applied to a vector or matrix, it produces a vector or matrix of the components being l^{th} Fourier modal coefficients of the original vector or matrix components.

Using the Fourier series expansions we get the following variational formulation:

Find $F_l(u) \in V$ such that:

$$\int_{\Omega} \sum_{l,m=-\infty}^{+\infty} F_l\left(\frac{\partial u}{\partial \xi}\right) F_m(\hat{\sigma}) \frac{\partial v}{\partial \xi} e^{j(l+m)\zeta_2} \, d\boldsymbol{\zeta} = \\ = \int_{\Omega} v F_l\left(\hat{f}\right) e^{jl\zeta_2} \, d\boldsymbol{\zeta} \quad \forall v \in V \tag{3.48}$$

The summation is applied with respect to $-\infty \le l, m \le \infty$. We select a mono-modal test function $v = v_k \, e^{jk\zeta_2}$.

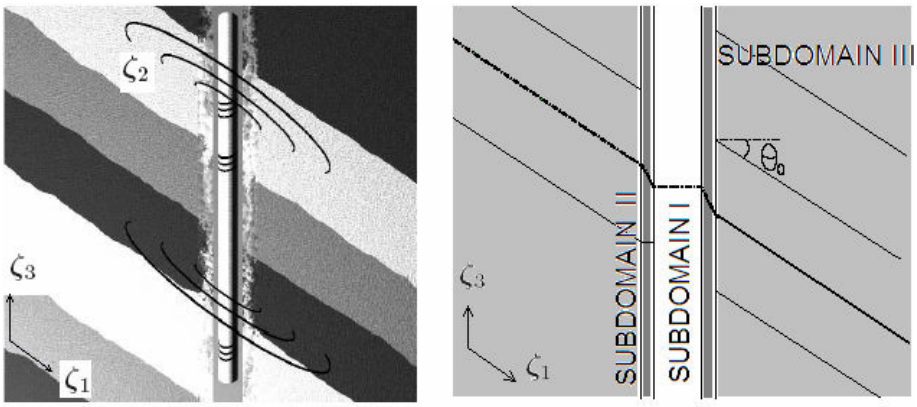


Fig.3.23. Three non-orthogonal systems of coordinates in the borehole and formation layers

Thanks to the orthogonality of the Fourier modes in $L^2(\Omega)$ the variational problem (3.48) reduces to:

Find $F_l(u) \in V$ such that:

$$\int_{\Omega_{2D}} \sum_{n=k-2}^{n=k+2} F_l\left(\frac{\partial u}{\partial \xi}\right) F_{k-l}(\hat{\sigma}) F_l\left(\frac{\partial v}{\partial \xi}\right) d\zeta_1 \, d\zeta_3 = \\ = \int_{\Omega_{2D}} F_k(v) F_k\left(\hat{f}\right) d\zeta_1 \, d\zeta_3 \quad \forall F_k(v) \in V \tag{3.49}$$

since five Fourier modes are enough to represent exactly the new material coefficients. For more details, see Pardo, Calo, Torres-Verdin, Nam 2007.

In a similar (however more algebraically complicated) manner the variational formulation for the AC problem can be derived, which is presented in Pardo, Torres-Verdin, Nam, Paszyński, Calo 2008.

3.2.4.3. Results

The simulations described in this chapter provide a highly accurate value of the potential of the electromagnetic field for different position of the transmitter and receiver electrodes.

In order to achieve this goal, we must generate highly non-uniform meshes with high polynomial order of approximation in the entire domain. An exemplary mesh providing such a high accuracy solution for a single position of receiver and transmitter electrodes is presented in Figure 3.24.

The main goal of the simulation is to generate the so-called resistivity logging curves which are of great interest to the oil industry. The curves are obtained by solving either DC or AC formulation, for many positions of the receiver and transmitter antenna, for different dip angles. The simulation reflects the process of resistivity measurements by the tool shifted along the borehole.

Figure 3.25 presents the solution of an exemplary 3D AC problem, for the resistivities of formation layers introduced on the first panel of this figure. The logging curves presented on the second and third panel in Figure 3.25 become a solution of this problem. The simulation is performed for axially-symmetric and for 30 and 60 degrees deviated well. We take into consideration real and complex components of the solution.

Several other challenging 3D problems from the Joint Research Consortium on Formation Evaluation have been solved by employing the Fourier series expansion in the non-orthogonal system of coordinates, as described in Paszyński, Pardo, Torres-Verdin, Demkowicz, Calo, 2007, Paszyński, Pardo, Torres-Verdín, 2007, Pardo, Torres-Verdin, Nam, Paszyński, Calo, 2008, Pardo, Calo, Torres-Verdin, Nam, 2008.

Each of these problems is related to different configurations of the tool and formation layers. The parallel self-adaptive *hp*-FEM is the only known numerical method providing a high accuracy solution in reasonable time (less than few minutes).

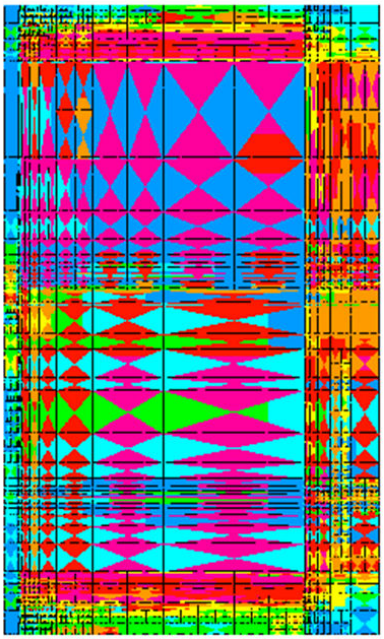


Fig.3.24. Optimal mesh generated by the self-adaptive *hp*-FEM code

3.2.5. 3D DC borehole resistivity measurement simulations with through-casing instruments in deviated well

The problem of simulating 3D direct current (DC) borehole resistivity measurements with through-casing instruments for the assessment of rock formation properties comes from the Joint Research Consortium on Formation Evaluation. This problem is the most challenging problem coming from the Consortium. It has been solved by using fully 3D formulation.

In order to avoid the mechanical collapse of wells in oil fields, the use of steel casing has been a common technique throughout the last decades. However, the casing of a well with a steel pipe has some undesired effects. For instance, the assessment of electrical properties of the rock formation becomes more challenging, since the casing highly attenuates the electromagnetic fields. As a matter of fact, it was not until the last decade when it was possible for the first time to obtain meaningful data from electromagnetic logging measurements. As this data becomes available, there is an increasing need to perform computer-aided simulations of resistivity measurements through casing to assess subsurface rock formation properties.

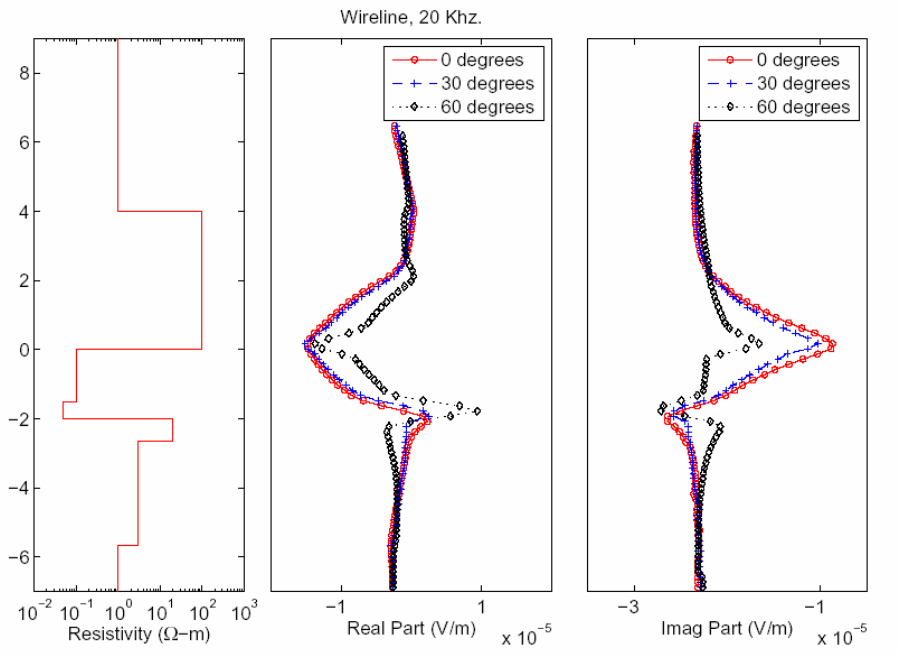


Fig.3.25. Exemplary logging curve obtained from the AC simulation

The main challenge for simulating through-casing measurements is the large conductivity contrast between the rock formation and the casing. This requires the use of large computational domains and accurate numerical methods to deal with high contrasts on

the materials (up to eleven orders of magnitude), strong singularities and large dynamic ranges (up to fourteen orders of magnitude).

The simulations obtained from deviated cased wells (when the borehole is not perpendicular to the rock formation layers) are of special interest to the oil industry.

For the analysis of this problem we take into consideration one current (emitter) and three voltage (collector) electrodes that are used to measure the second difference of the electric potential along the well trajectory.

The problem geometry can be described by using cylindrical coordinates (ρ,φ,z). The following sources, receivers, and materials are used in Figure 3.26:

a) four (one current and three voltage) 2 × 5-cm ring electrodes located 8 cm from the axis of symmetry and moving along the vertical direction (z axis). Voltage (collector) electrodes are located 100, 125, and 150 cm above the current (emitter) electrode, respectively,

b) borehole: a cylinder Ω_A of radius 10 cm surrounding the axis of symmetry $\Omega_A = \{(x,\varphi,z): \rho \le 10\,\text{cm}\}$ with resistivity $R = 0.1\,\Omega\cdot m$,

c) casing: a pipe (cylindrical shell) Ω_B of thickness 1.27 cm surrounding the axis of symmetry $\Omega_B = \{(x,\varphi,z): 10\,\text{cm} \le \rho \le 11.27\,\text{cm}\}$, with resistivity $R = 10^{-5}\,\Omega\cdot m$,

d) formation material 1: a subdomain Ω_C defined by $\Omega_C = \{(x,\varphi,z): \rho > 11.27\,\text{cm}, 0\,\text{cm} \le z \le 100\,\text{cm}\}$ with resistivity $R = 10^4\,\Omega\cdot m$,

e) formation material 2: a subdomain Ω_D defined by $\Omega_D = \{(x,\varphi,z): \rho > 11.27\,\text{cm}, -50\,\text{cm} \le z \le 0\,\text{cm}\}$ with resistivity $R = 0.01\,\Omega\cdot m$,

f) formation material 3: a subdomain Ω_E defined by $\Omega_E = \{(x,\varphi,z): \rho > 11.27\,\text{cm or } z > 100\,\text{cm}\}$ with resistivity $R = 5\,\Omega\cdot m$.

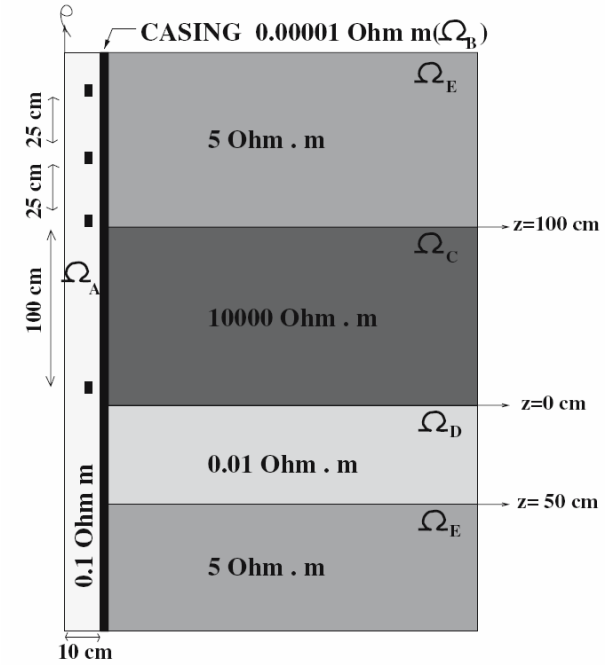


Fig.3.26. Two-dimensional cross-section of the 3D DC borehole resistivity measurements with through-casing instruments

The self-adaptive *hp*-FEM managed to simulate for the first time the 3D DC borehole resistivity measurements with through-casing instruments in deviated well.

3.2.5.1. Strong formulation

Find $u: R^3 \supset \Omega \ni x \to u(x) \in R$ the electrostatic scalar potential such that

$$-\sum_{i=1}^{3} \sigma \frac{\partial^2 u}{\partial x_i^2} = \sum_{i=1}^{3} \frac{\partial J}{\partial x_i} \text{ in } \Omega \tag{3.50}$$

where J denotes a prescribed, impressed current source, σ is the conductivity, and the electrostatic scalar potential u is related to the electric field E by

$$E = -\nabla u \tag{3.51}$$

The boundary conditions are defined as

$$u = 0 \text{ on } \Gamma_D \tag{3.52}$$

$$\sigma \frac{\partial u}{\partial n} = g \text{ on } \Gamma_N \tag{3.53}$$

where g is the prescribed flux on Γ_N, with n being the unit normal outward to $\partial\Omega$ vector.

3.2.5.2. Weak formulation

Find $u \in V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{3.54}$$

$$b(u,v) = \int_{\Omega} \sum_{i=1}^{3} \sigma \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} dx \tag{3.55}$$

$$l(v) = \int_{\Omega} \sum_{i=1}^{3} \frac{\partial J}{\partial x_i} v \, dS + \int_{\Gamma_N} g \, v \, dS \tag{3.56}$$

where

$$V = \left\{ v \in L^2(\Omega) : \int_{\Omega} \|v\|^2 + \|\nabla v\|^2 \, dx < \infty : \mathrm{tr}(v) = 0 \text{ on } \Gamma_D \right\} \tag{3.57}$$

3.2.5.3. Results

Figures 3.27 and 3.28 present exemplary computational *hp* meshes generated by the self-adaptive *hp*-FEM algorithm for the problem of through-casing resistivity instruments, as well as the solution for a single logging position.

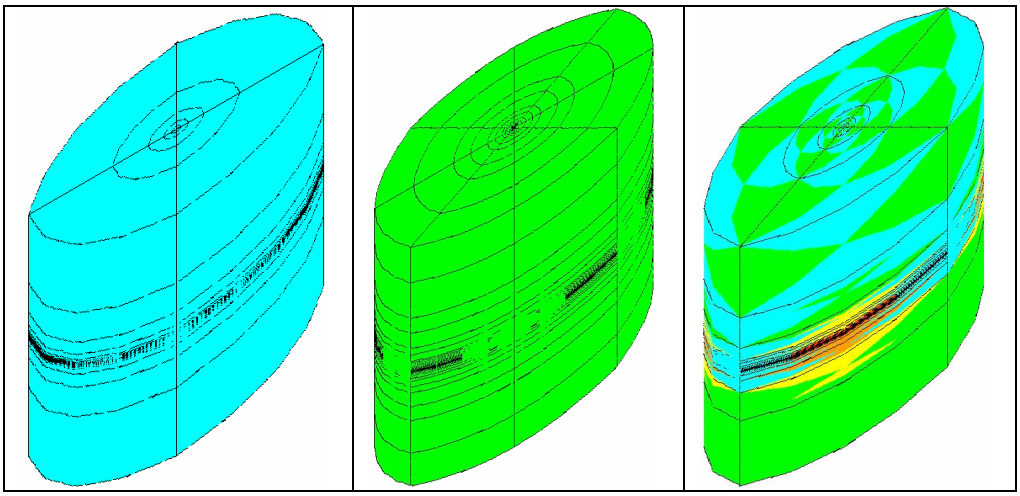


Fig.3.27. Coarse and fine meshes from the first iteration of the self-adaptive *hp*-FEM algorithm, and the exemplary highly *hp* refined mesh, for the deviated well case

For more details on the problem formulation and numerical results, see Pardo, Torres-Verdin, Paszyński 2008.

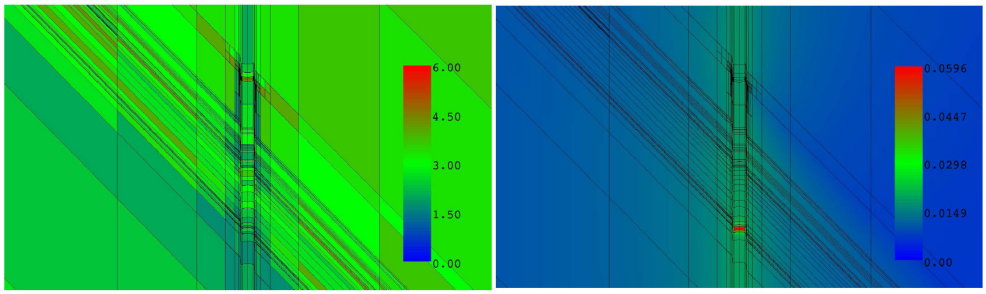


Fig.3.28. Simulation of through-casing resistivity measurements in 60 degrees deviated well. Logging instrument incorporates one transmitter and three receiver antennas. Left panel: Cross-section of the final hp-grid. Different colours indicate different polynomial orders of approximation, from 1 (linear polynomials) up to 6 (polynomials of degree 6). Right panel: Final solution (scalar potential)

The main goal of the simulation is to generate the so-called resistivity logging curves, which are of great interest to the oil industry. These curves are obtained by solving variational problem (3.54-3.57) for different positions of the receiver and transmitter antenna, for different dip angles. An exemplary logging curve for 60 degrees deviated well is presented in Figure 3.29 and the corresponding resistivities of formation layers are presented in Figure 3.26.

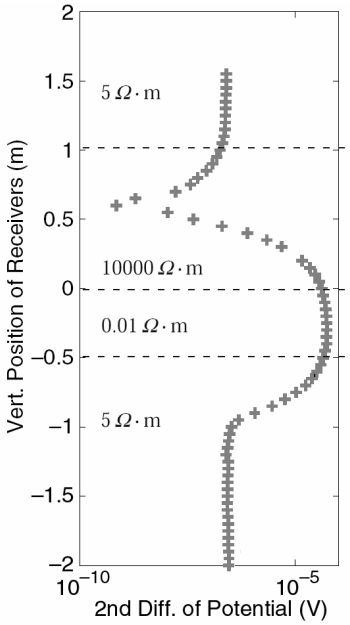


Fig.3.29. Exemplary logging curve from the simulation of through-casing resistivity measurements for 60 degrees deviated well

Several other challenging 3D problems from the Joint Research Consortium on Formation Evaluation have been also solved, as described in Paszyński, Demkowicz, Pardo, 2005, Pardo, Calo, Torres-Verdin, Paszyński, Nam, 2007, Pardo, Demkowicz, Torres-Verdin, Paszyński, 2006, Pardo, Demkowicz, Torres-Verdin, Paszyński, 2007a, Pardo, Demkowicz, Torres-Verdin, Paszyński, 2007b, Pardo, Paszyński, Demkowicz, Torres-Verdin, 2005a, Pardo, Paszyński, Demkowicz, Torres-Verdin, 2005b, Pardo, Paszyński, Torres-Verdin, Demkowicz, 2006a, Pardo, Paszyński, Torres-Verdin, Demkowicz, 2006b, Pardo, Paszyński, Torres-Verdin, Demkowicz 2007, Pardo, Torres-Verdin, Paszynski, 2008, Pardo, Torres-Verdin, Paszynski, Michler, Demkowicz, 2006, Paszyński, Pardo, Demkowicz, Torres-Verdin, 2006, Paszyński, Pardo, Torres-Verdin, 2007a, Paszyński, Pardo, Torres-Verdin, 2007b, Paszyński, Pardo, Torres-Verdín, 2007c, Paszyński, Pardo, Torres-Verdin, Demkowicz, 2006, Paszyński, Pardo, Demkowicz, Torres-Verdin, 2007.

All those problems are especially important for the oil industry. Each of them is related to different configurations of the tool and formation layers. The parallel self-adaptive *hp*-FEM is the only known numerical method providing a high accuracy solution in reasonable time (less than few minutes).

1. Conclusions and future work

4.1. Summary of obtained results

The material presented in this work allows us to develop a formal mathematical model of a wide class of adaptive mesh-based algorithms.. The computational mesh has been presented as a composite programmable graph (CP-graph) and the mesh transformations performed by the adaptive algorithms have been represented by the graph grammar productions. We have introduced the control diagrams to establish the partial order of executions of graph grammar productions (graph transformations).

The graph grammar-based formalism has been employed in Chapter 2.1 to describe the self-adaptive *hp* Finite Element Method (*hp*-FEM) algorithm, one of the most complicated mesh-based adaptive algorithms. We have introduced the formal graph grammar-based definition of all parts of the self-adaptive *hp*-FEM algorithm. It includes the following algorithms:

a) the algorithm generating the initial mesh,
b) the coarse mesh direct solver algorithm,
c) the algorithm transforming the coarse mesh into the fine mesh,
d) the algorithm of the fine mesh direct solver,
e) the algorithm for the selection of optimal mesh refinements,
f) the algorithms executing a required *h* or *p* refinement and enforcing the mesh regularity rules.

The developed formalism has allowed us to examine the concurrency of the above algorithms. For that reason, the model of concurrency has been developed in Chapter 2.1. It is based on the atomic computational task defined as a single graph grammar production, assumed to be undividable sequential operation. The interdependences among the graph grammar productions (atomic tasks) were analysed and summarized in the control diagrams. The control diagrams of the graph grammar model set the partial order of execution and identify graph grammar productions that can be executed concurrently. The model of concurrency for the atomic tasks assumes that the graph representation of the mesh is fully available for every task.

Several atomic tasks have been agglomerated to the so-called task, representing the operations performed on the level of sub-graphs corresponding to a single initial mesh element. The agglomeration has been performed under the assumption that each task has its own local memory, and stores its local sub-graph representing a single initial mesh element.

The tasks constitute the grain for the mesh partitioning and load balancing algorithms. We have defined the communication channels between tasks and the self-adaptive algorithm has been redefined in Section 2.2 on the level of sub-graphs representing single initial mesh elements. The new control diagrams have been created, defining the partial order of execution of atomic tasks from the point of view of tasks, with some additional inter-task communication added. The tasks defined on the level of sub-graphs assigned to the initial mesh elements constitute the grain for the load balancing and mesh partitioning algorithms, defined in Section 2.3. Thus, the CP-graph representation of the mesh has been partitioned on the level of sub-graphs representing the initial mesh elements and we have introduced a new set of graph grammar productions responsible for partitioning and merging the graph representation of the mesh into several sub-domains.

Next, several tasks – grains have been agglomerated to the so-called super-tasks. The agglomeration has been performed under the assumption that each super-task has its own local memory, and stores its local sub-graph representing a single sub-domain with multiple initial mesh elements. The super-tasks model is suitable for the distributed memory architectures, where the number of processors is usually smaller than the number of tasks. At this point, it is assumed that each super-task is assigned to a single processor. Thus, each super task has its own global identifier, the processor number. At the end of each iteration of the self-adaptive *hp*-FEM algorithm, the partition into several super-tasks has been updated by executing the load balancing algorithm. Again, it has been done using the grain defined as the tasks. The input for the load balancing algorithm is the estimation of the computational cost for all tasks. The output from the algorithms is a new redistribution of tasks. The tasks have been again agglomerated into super-tasks.

In Section 2.4 the self-adaptive *hp*-FEM algorithm has been expressed on the level of super-tasks. The control diagrams have been updated to define the atomic tasks executed by each super-task, with some necessary inter-super-task communication added.

In Section 2.5 we have discussed the computational and communication complexities of some selected parts of the self-adaptive algorithm and, consequently, several numerical experiments were performed in Section 2.6. From the theoretical analysis and numerical experiments it results that the scalability of the parallel self-adaptive *hp*-FEM algorithm partitioned on the level of tasks – grains related to the initial mesh elements is quite good. Several parallel direct solvers have been interfaced then with the algorithm and tested on the LONGHORN and LONESTAR linux clusters from Texas Advanced Computing Center. It has been proven that the best scalability of the direct solver is achieved for the parallel solver implemented according to the presented graph grammar-based model.

The numerical experiments indicate that the load balancing and mesh repartitioning algorithms performed on the level of initial mesh elements balance well the computations if the computational problem contains many singularities. If a problem is small and contains a small number of local singularities, the mesh adaptation usually occurs in in the closest neighborhood of those singularities, and there are only a limited number of undividable tasks – related to the sub-graphs assigned to the initial mesh elements covering these singularities. But this is not a problem, since small problems with small number of singularities do not require massive parallel computations. The parallel self-adaptive *hp*-FEM algorithm scales well for large problems with large number of singularities, which has been documented by the performed numerical experiments.

The two- and three-dimensional self-adaptive *hp*-FEM algorithms have been already implemented in the *hp2Dpar* and *hp3Dpar* codes. Many challenging engineering problems, including the applications to material science, geo-science and remote sensing, nano-lithography (micro-cheap production process) and wave propagation problems have been solved by the implemented algorithms. The numerical experiments discussed in Chapter 3 present the power of the self-adaptive *hp*-FEM algorithm. All the described problems require high accuracy of the numerical solution. Up till now, it has been impossible to achieve such solution using other numerical methods. The self-adaptive *hp*-FEM, thanks to the exponential convergence of the numerical error with respect to the mesh size, is able to solve these problems with high numerical accuracy on relatively small meshes. Other numerical methods either require grids with millions of degrees of freedom, or even are not able to solve these problems at all.

4.2. Significance of obtained results

Up till now, neither a comprehensive formal model of the adaptive mesh-based algorithms, nor a universal methodology for the parallelization of all components of these algorithms has been created. The formalism proposed in this dissertation enables a complete description of concurrent versions of the class of adaptive mesh-based algorithms. The adaptive mesh-based algorithms are very complex, they consist of multiple inter-dependent sub-algorithms, each of them requiring a different parallelization strategy to be considered.

The graph grammar model allows for a comprehensive description of all aspects of the adaptive algorithms, including the generation of initial mesh, the direct solver algorithm, the mesh transformations including *h* and *p* adaptivity, as well as the enforcement of the mesh regularity rules (the 1-irregularity rule and the minimum rule).

For the first time, a formal analysis of the parallelization of the adaptive mesh-based algorithms has been performed. As a result, we have managed to create the model of concurrency, by means of relating the atomic tasks with graph grammar productions and of denoting partial ordering of these productions. This has allowed us to examine the potential concurrency in all parts of the adaptive algorithms. The created model expresses for the first time the concurrent version of the self-adaptive *hp*-FEM algorithm intended for the shared memory architecture.

The atomic tasks have been agglomerated into tasks, in order to partition the graph representation of the mesh into several sub-graphs, related to the initial mesh elements. Each task is supposed to use a local memory, where it stores its sub-graphs, representing a single initial mesh element. The self-adaptive *hp*-FEM algorithm has been expressed on the level of tasks by introducing several control diagrams defining the partial order of execution of atomic tasks by a single task, with some necessary additional minimum inter-task communication. The tasks have become later the grain for the load balancing and mesh repartitioning algorithms. Thus, several tasks have been again agglomerated into super-tasks, representing the partition of the graph representation of the mesh into multiple sub-graphs, assigned to the sub-domains with multiple initial mesh elements. The load balancing / mesh repartitioning algorithms have updated the partition of graph representation of the mesh into new super-tasks after each iteration of the self-adaptive *hp*-

FEM algorithm. The control diagrams have been finally updated, to express the algorithm executed on the level of super-tasks. The resulting model enables an efficient implementation of parallel self-adaptive *hp*-FEM algorithm for the distributed and hybrid memory architectures.

The adaptive *hp*-FEM algorithms require efficient sequential and parallel solvers. In general, the solvers can be classified as iterative or direct. The iterative solvers are more efficient than the direct solvers, provided that the considered system of equations is well-conditioned. A complex, dynamically changing structure of the computational mesh and non-uniform polynomial orders of approximation result in problems with convergence of iterative solvers (since the resulting system of equations is not well-conditioned). The only existing iterative solvers used by the adaptive algorithms are based on multi-grid paradigm (usually two-grid paradigm). The solution for a much larger fine mesh is obtained by several solutions of the computational problem projected onto a smaller coarse mesh. The coarse mesh solution must be obtained by the direct solver. Thus, there is a need for efficient sequential and parallel direct solvers, even if iterative multi-grid solvers are applied.

Up till now, there has been no direct solver incorporated to the structure of adaptive *hp* meshes. We have managed to create a new sequential and parallel direct solver algorithm using the structure of *hp* mesh to solve the problem in an efficient way. The solver employs the multi-level structure of the elimination tree created out of the graph representation of the *hp* mesh. The multi-level structure allows us to separate the level of refinements from the level of initial mesh elements and the level of sub-domains used by the domain decomposition paradigm. The implemented parallel version of the solver algorithm has been compared to the MUMPS solver, known as one of the best available parallel direct solver. The developed parallel solver has outperformed the MUMPS solver both in terms of the execution time and memory usage. Moreover, the proposed solver allows for a reutilization of partial LU factorizations distributed within the nodes of the elimination tree. This reduces the computational complexity of the solver in a significant way, since the partial LU factorization has to be recomputed only on the refined parts of the mesh.

On the basis of the developed formalism, the self-adaptive *hp*-FEM algorithm has been designed and implemented in both two and three dimensions. Nowadays it is the only existing fully automatic *hp*-adaptive FEM software system in the world. The project proves to be successful thanks to the developed graph grammar model, a thorough theoretical analysis of the computational and communication complexities of the parallel algorithms, and the resulting efficient design and implementation of pioneer algorithms. The two- and three-dimensional software systems allow us to solve several challenging engineering problems.

Let us mention two examples of such challenging problems. The two-dimensional battery problem from the Sandia National Laboratories is very specific because of strong material contrast with anisotropic properties. Contrary to other known methods, the self-adaptive *hp*-FEM provides the solution with accuracy less than 1% relative error in the energy norm, for a mesh with less than 3000 degrees of freedom. Other solutions require *several millions* of degrees of freedom to solve the problem with the same accuracy. The simulation performed for the three-dimensional direct current (DC) borehole resistivity measurements acquired in steel-cased deviated wells for the assessment of rock formation properties are the only existing numerical simulations of this process. This is because of a

high contrast of material data in the model – resistivities of formation layers and steel casing differs by six orders of magnitude – and it is possible to achieve the required high accuracy of the solution *only* by using the self-adaptive *hp*-FEM algorithm.

4.3. Current and future work

The sequential and parallel self-adaptive *hp*-FEM algorithms have been designed for elliptic boundary-value problems, in the form presented in Appendix B (B.1-B.8). The extension of the self-adaptive *hp*-FEM code for the Stokes problem is currently under development (Matuszyk, Paszyński, 2007a, Matuszyk, Paszyński, 2007b, Matuszyk, Paszyński, 2007c, Matuszyk, Paszyński, 2008). There are two main challenges that have to be faced during the extension process:

1) the need of approximation of two continuous fields: vector velocity field $u:R^2 \supset \Omega \ni x \to u(x) \in R$, and scalar pressure field $u:R \supset \Omega \ni x \to p(x) \in R$,
2) the assumption of minimal changes in the code implies the usage of the equal-order approximation Q^p / Q^p for the velocity and pressure. This implies a necessity of using a stabilization technique to overcome the limitations imposed on FE spaces by the famous Ladyzhenskaya-Babuška-Brezzi (LBB) condition (Brezzi, Fortin 1991). A good example of such method is the Hughes-Franca stabilization method (Hughes, Franca, Balestra, 1986, Hughes, Franca, 1987).

Both the sequential and parallel self-adaptive *hp*-FEM algorithms are limited to the stationary problems. The extension of the self-adaptive *hp*-FEM algorithm to the non-stationary problems is currently under development Matuszyk, Paszyński, 2007c. Again, there are some challenges that we have to deal with during the extension process:

a) the non-stationary problems require the computational meshes from the current and previous time step to be stored in memory. During the adaptive computations, the meshes are usually non-conforming, since they are obtained by a sequence of *h* and *p* refinements from a prescribed initial mesh. The solution from the previous time step is applied to obtain the current time step solution. Thus, we need to implement a suitable projection between the current and the previous time steps,
b) the nature of the solution changes from one time step to the other. The optimal *hp* mesh for the previous time step may be no longer optimal for the next time step. Some refinements may be no longer necessary. Thus, the mesh unrefinement algorithm must be designed in order to optimize the size of computational meshes.

The presented CP-graph grammar model is limited to the two-dimensional rectangular finite elements (Paszyński, Paszyńska 2007). The extension of the graph grammar model to the triangular finite elements is under development. The graph grammar for the triangular elements can be designed using very similar graph transformations, as presented in (Paszyńska, Paszyński, Grabska 2008).

The presented graph grammar model is limited to the two-dimensional meshes. The two-dimensional *hp2Dpar* code has been created by designing the UML model classes and

objects related to CP-graph vertices with classes and objects. The three-dimensional code *hp3Dpar* has been designed by the generalization of the UML diagrams to three dimensions. The parallel algorithms for the three-dimensional code have been obtained by the generalization of the two-dimensional algorithms, expressed by graph grammar transformations, to the three dimensions. However, there is still a need for a formal model of the three-dimensional computations. This is a challenging task, since the three-dimensional graph grammar model seems to be very complicated.

The current version of the multi-level multi-frontal direct-substructuring parallel solver, defined in Appendix D, does not include the reutilization of partial LU factorizations, introduced in Section 2.4. The future work will involve the development of a new version of the solver, efficiently employing the reutilization techniques. Some preliminary work on the theoretical analysis of the reutilization algorithm has been already presented in (Paszyński, Schaefer 2008).

The multi-level multi-frontal direct-substructuring parallel solver can be used as the coarse mesh solver for the parallel version of the two-grid solver, designed for *hp*-FEM. The sequential version of the two-grid solver has been developed by David Pardo (Pardo, 2004). The solver uses the two-grid paradigm. The fine mesh problem solution is achieved by means of several solutions of the problem projected onto the coarse mesh. The main technical challenge for the parallel two-grid solver algorithm is to maintain patches of finite elements, which can be much wider than a single initial mesh element. One solution is to employ the domain decomposition paradigm with overlapping sub-domains. Another solution is to develop the iterative solver on the shared data structure. The preliminary results for the second approach have been described in Pardo, Nam, Torres-Verdin, Paszyński 2008.

The inverse parametric problems belong to the group of the heaviest computational tasks. Their solution require a sequence of direct problem solutions, e.g. obtained by using the self-adaptive *hp*-FEM, thus the accuracy of the inverse problem solution is limited by the accuracy of the direct problem solution. Using the maximum accuracy for the direct problem solution by each iteration of the inverse solver leads to unnecessary computational costs. A better strategy is to balance dynamically the accuracy of both iterations.

The relation between the error of optimization method, defined as the incorrectness of objective function value, and the *hp*-FEM solution error has been already developed (Paszyński, Barabasz, Schaefer, 2007). The method was tested with the Hook-Jeeves algorithm used for the inverse problem, for the identification of the SFIL process parameters (Paszyński, Szeliga, Barabasz, 2007). The theoretical foundations for the Hierarchical Genetic Search (HGS) inverse algorithm coupled with *hp*-FEM have been also already developed (Schaefer, Barabasz, Paszyński, 2007, Schaefer, Barabasz, Paszyński, 2008). The future work will involve a further theoretical analysis of this method as well as its successful implementation.

The self-adaptive *hp* FEM starts from an arbitrary initial mesh, selected by the user. It generates a sequence of meshes delivering convergence of the accuracy of the numerical solution with respect to the mesh size. The sequence of meshes is obtained by performing multiple *h, p* or *hp* refinements. The *h* refinement consists in breaking selected finite element into smaller son elements, and the *p* refinement consists in adjusting polynomial order of approximation on selected element edges and interiors. The ration of convergence of the solution on a sequence of meshes depends on the quality of the selected initial mesh.

The best convergence is obtained when the initial mesh fits the material data. The design of such optimal initial mesh by hand is very difficult, and sometimes even impossible, e.g. when the material data comes from some stochastic procedure.

The foundations of the genetic algorithm for the optimal selection of the initial mesh have been already presented (Paszyńska, Paszyński 2007, Paszyńska, Paszyński, 2008), and the future work will focus on a further development of the algorithm.

The agent-oriented programming paradigm seems to be very promising for developing efficient applications in the distributed environment, with multiple computational nodes and with dynamically changing topology of the network. The future work will involve the design and development of the agent-oriented version of the self-adaptive *hp*-FEM algorithm. Some preliminary work on this problem has been already discussed in (Paszyński, 2006, Paszyński, 2007b).

References

Amestoy P. R., Duff I. S., L'Excellent J.-Y., 2000: *Multifrontal parallel distributed symmetric and unsymmetric solvers*. Computer Methods in Applied Mechanics and Engineering 184, 501-520

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

Amestoy P. R., Guermouche A., L'Excellent J.-Y., Pralet S., 2006: Hybrid scheduling for the parallel solution of linear systems. Parallel Computing, 32, 2, 136-156

Babuška I., Guo B., 1986a: *The hp-version of the finite element method, Part I: The basic approximation results*. Computational Mechanics, 1, 21-41

Babuška I., Guo B., 1986b: *The hp-version of the finite element method, Part II: General results and applications*. Computational. Mechanics, 1, 203-220

Bailey T. C., Colburn M. E., Choi B. J., Grot A., Ekerdt J. G., Sreenivasan S. V., Willson C. G., 2002: *Step and Flash Imprint Lithography: A Low-Pressure, Room Temperature Nanoimprint Patterning Process. Alternative Lithography. Unleashing the Potentials of Nanotechnology*. C. Sotomayor Torres, Editor, Elsevier

Bajer A., Rachowicz A., Walsh T., Demkowicz T., 2001: *A Two-Grid Parallel Solver for Time Harmonic Maxwell's Equations and hp Meshes*. Proceedings of Second European Conference on Computational Mechanics, Kraków, June 25-29

Banaś K., 2003: *A Model for Parallel Adaptive Finite Element Software*. Proceedings of 15th International Conference on Domain Decomposition Methods, Freie Universitat Berlin, July 21-25

Banaś K., Płażek J., 1997: *Parallel Iterative Solvers for the Finite Element Method*. 13th international conference on Computer Methods in Mechanics, Vol. I, 115-120

Bauer A. C., Patra A. K., 2004: *Robust and Efficient Domain Decomposition Preconditioners for Adaptive hp Finite Element Approximations of Linear Elasticity with and without Discontinuous Coefficients*. International Journal of Numerical Methods in Engineering, 59, 3, 337-364

Bastian P., Birken K., Johannsen K., Lang S., Neuss N., Rentz-Reichart H., 1997: *UG - a flexible software toolbox for solving partial differential equations*. Computing and Visualization in Science, 1, 1, 27-40

Bastian P., 1998: *Load Balancing For Adaptive Multigrid Methods*. SIAM Journal on Scientific Computing, 19, 4, 1303-1321

Beal M. W., Shephard M. S., 1997: *A General Topology-Based Mesh Data Structure*, International Journal for Numerical Methods in Engineering, 40, 1573-1596

Booch G., Rumbaugh J., Jacobson I., 1998: *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 1st edition

Brezzi F., Fortin M., 1991: *Mixed and Hybrid Finite Element Methods*. Springer-Verlag

Burns R. L., Johnson S. C., Schmid G. M., Kim E. K.,. D. Dickey M. D., Meiring J., Burns S. D., Stacey N. A., Willson C. G., 2004: *Mesoscale modeling for SFIL simulating polymerization kinetics and densification*. Proceeding of SPIE

CHAMPION Cluster Users' Manual
http://www.tacc.utexas.edu/services/userguides/champion

Clos C., 1984: *A Study of Non-Blocking Switch Networks, Interconnection Networks for Parallel and Distributed Processing*. Chuan-lin Wu and Tse-yun Feng (ed.), IEEE Computer Society Press, 126-144

Colburn M. E., 2001: *Step and Flash Imprint Lithograpy: A Low Pressure, Room Temperature Nonoimprint Lithography*. PhD. Thesis, The University of Texas in Austin

Colburn M. E., Suez I., Choi B. J., Meissi M., Bailey T., Sreenivasan S. V., Ekerdt J. E., Willson C. G., 2001: *Characterization and modeling of volumetric and mechanical properties for SFIL photopolymers.* Journal of Vacuum Science and Technology, B 19(6)

Demkowicz L., 2004: *Projection-Based Interpolation*, ICES Report 04-03

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

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

Demkowicz L., Rachowicz W., Devloo Ph., 2001: *A Fully Automatic hp-Adaptivity*. Journal of Scientific Computing, 17, 1-3, 127-155

Devine K. D., Flaherty J. E., 1996: *Parallel Adaptive hp-Refinement Techniques for Conservation Laws*. Applied Numerical Mathematics, 20, 367-386

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

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

Edwards H. C., 1997: *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its application to Least Squares C-infinity Collocation*. PhD. Dissertation, The University of Texas at Austin

Edwards H. C., 2002: *SIERRA Framework Version 3: Core Services Theory and Design.* SAND2002-3616, Albuquerque, NM: Sandia National Laboratories

Edwards H. C., Stewart J. R., 2001: *SIERRA, A Software Environment for Developing Complex Multiphysics Applications*. Computational Fluid and Solid Mechanics, Proc. First MIT Conf. Cambridge MA

Edwards H. C., Stewart J. R., Zepper J. D., 2002: *Mathematical Abstractions of the SIERRA Computational Mechanics Framework*. Proceedings of the Fifth World Congress on Computational Mechanics, Vienna Austria

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

Foster I., *Desiging and Building Parallel Programs*, http://www-unix.mcs.aml.gov/dbpp

Geng P., Oden T. J., van de Geijn R. A., 2006: *A Parallel Multifrontal Algorithm and Its Implementation*. Computer Methods in Applied Mechanics and Engineering 149, 289-301

Giraud L., Marocco A., Rioual J.-C., 2005: *Iterative versus direct parallel substructuring methods in semiconductor device modeling*. Numerical Linear Algebra with Applications, 12, 1, 33-55

Grabska E., 1993a: *Theoretical Concepts of Graphical Modeling. Part One: Realization of CP-Graphs*. Machine Graphics and Vision 2, 1, 3-38

Grabska E., 1993b: *Theoretical Concepts of Graphical Modeling. Part Two: CP-Graph Grammars and Languages*. Machine Graphics and Vision 2, 2, 149-178

Grabska E., Hliniak G., 1993: *Structural Aspects of CP-Graph Languages. Schedae Informaticae.* 5, 81-100

Hughes T. J. R., Franca L. P., 1987: *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*. Computer Methods in Applied Mechanics and Engineering, 65, 85-96

Hughes T. J. R., Franca L. P., Balestra M., 1986: *A New FEM for Computational Fluid Dynamics: V. Circumventing The Babuška-Brezzi Condition: A Stable Petrov-Galerkin Formulation of The Stokes Problem Accomodating Equal-Order Interpolations*. Computer Methods in Applied Mechanics and Engineering, 59, 85-99

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

Karypis G., Kumar V., 1999: *A fast and high quality multilevel scheme for partitioning irregular graphs*. SIAM Journal on Scientific Computing, 20, 1, 359 – 392

Khaira M. S., Miller G. L., Sheffler T. J., 1992: *Nested Dissection: A survey and comparison of various nested dissection algorithms*, CMU-CS-92-106R, Computer Science Department, Carnegie Mellon University

Laszloffy A., Long J., Patra A. K., 2000: *Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive hp finite element simulations*. Parallel Computing, 26, 1765-1788

LONESTAR Cluster Users' Manual
http://www.tacc.utexas.edu/services/userguides/lonestar

Matuszyk P., Paszyński M., 2007: *Fully Automatic hp Finie Element Method for Stokes Problem in Two Dimensions*. CMM-2007 : 17th international conference on Computer Methods in Mechanics, Łódź–Spała, Poland, June 19–22

Matuszyk P., Paszyński M., 2007b: *Extensions of the 2D automatic hp-adaptive FEM for Stokes and non-stationary heat transfer problems*. 9th US National Congress on Computational Mechanics, San Francisco, July 16-21

Matuszyk P., Paszyński M., 2007c: *Fully automatic 2D hp-adaptive Finite Element Method for Non-stationary Heat Transfer Problem*s. COMPLAS IX : COMputational PLASticity : fundamentals and applications, Pt. 2, eds. Eugenio Oñate, Roger Owen, Benjamín Suárez, Barcelona, Spain, September 5–7

Matuszyk P., Paszyński M., 2008: *A Fully Automatic hp Finite Element Method for Stokes Problem in Two Dimensions*, Computer Methods in Applied Mechanics and Engineering, 197, 51-52, pp. 4549-4558

MUMPS: *A MUltifrontal Massively Parallel sparse direct Solver*.
http://www.enseeiht.fr/lima/apo/MUMPS

Pardo D., 2004: *Integration of hp-Adaptivity with a Two-Grid Solver*. PhD. Dissertation, The University of Texas at Austin

Pardo D., Calo V. M., Torres-Verdin C., Nam M. J., 2008: *Fourier Series Expansion in a Non-Orthogonal System of Coordinates for Simulation of 3D DC Borehole Resistivity Measurements*. Computer Methods in Applied Mechanics and Engineering, 197, 1-3, 1906-1925

Pardo D., Calo V., Torres-Verdin C., Paszyński M., Nam M. J., 2007: *Self-adaptive hp finite-element simulation of multi-component induction measurements acquired in dipping, invaded and anisotropic formations*. Seventh Annual Report of Joint Industry Research Consortium on Formation Evaluation, The University of Texas at Austin, August 16-17

Pardo D., Demkowicz L., Torres-Verdin C., Paszyński M., 2006: *Simulation of Resistivity Logging-While-Drilling (LWD) Measurements Using a Self-Adaptive Goal-Oriented hp-Finite Element Method.* SIAM Journal on Applied Mathematics, 66, 2085-2106

Pardo D., Demkowicz L., Torres-Verdin C., Paszyński M., 2007a: *A Three-Dimensional Self-Adaptive, Goal-Oriented hp-Finite Element Method with a Multigrid Solver*, *Applications to Electromagnetics.* SIAM Conference in Computational Science and Engineering, Costa Mesa, CA, USA, February 19-23

Pardo D., Demkowicz L., Torres-Verdin C., Paszyński M., 2007b: *A Goal Oriented hp-Adaptive Finite Element Strategy with Electromagnetic Applications. Part II: Electrodynamics*. Computer Methods in Applied Mechanics and Engineering, special issue in honor of Prof. Ivo Babuška, 196, 3585-3597

Pardo D., Nam M. J., Torres-Verdin C., Paszyński M., 2008: *A Parallel, Fourier Finite Element Formulation with an Iterative Solver for the Simulation of 3D LWD measurements Acquired in Deviated Wells*. PIERS Online, 4, 5, 551-555

Pardo D., Paszyński M., Demkowicz L., Torres-Verdin C., 2005a: *Self-Adaptive Goal-Oriented hp-Finite Element Simulations of (1) Axisymmetric Borehole Accoustics and (2) 3D Resistivity Logging Instruments.* Fifth Annual Report of Joint Industry Research Consortium on Formation Evaluation, The University of Texas at Austin, August 17

Pardo D., Paszyński M., Demkowicz L.,. Torres-Verdin C., 2005b: *Parallel Self-Adaptive Goal-Oriented hp-Finite Element Simulations of 3D Resistivity Logging Instruments*. Fifth Annual Report of Joint Industry Research Consortium on Formation Evaluation, The University of Texas at Austin, August 17

Pardo D., Paszyński M., Torres-Verdin C., Demkowicz L., 2006a: *Numerical Simulations of 3D DC Borehole Resistivity Measurements using an hp-Adaptive Goal-Oriented Finite Element Formulation*, Sixth Annual Report of Joint Industry Research Consortium on Formation Evaluation, The University of Texas at Austin, August 16-18

Pardo D., Paszyński M., Torres-Verdin C., Demkowicz L., 2006b: *High Accuracy Simulations of 2D and 3D Resistivity Logging Instruments using a Self-Adaptive Goal-Oriented hp-FEM*. MAFELAP 2006 Conference on the Mathematics of Finite Elements and Applications*,* Brunel Institute of Computational Mathematics, Brunel University, Uxbridge, England, June 13-16

Pardo D., Paszyński M.,. Torres-Verdin C., Demkowicz L., 2007: *Simulation of 3D DC Borehole Resistivity Measurements with a Goal-Oriented hp Finite-Element Method. Part I: Laterolog and LWD*. Journal of the Serbian Society for Computational Mechanics, 1, 62-73

Pardo D., Torres-Verdin C., Paszynski M., Michler C.,. Demkowicz L., 2006: *A 2D and 3D hp-Finite Element Method for Simulation of Through Casing Resistivity Logging Instruments.* Proceedings to the IEEE International Symposium on Antennas and Propagation, Albuquerque, NM, USA, July 10-15

Pardo D., Torres-Verdin C., Nam M. J., Paszyński M.,. Calo V., 2008: *Fourier Series Expansion in a Non-Orthogonal System of Coordinates for the Simulation of 3D Alternating Current Borehole Resistivity Measurements*. Computer Methods in Applied Mechanics and Engineering, 197, 45-48

Pardo D., Torres-Verdin C., Paszyński M., 2008: *Simulations of 3D DC Borehole Resistivity Measurements with a Goal-Oriented hp-Finite-Element Method. Part II: Through Casing Resistivity Instruments*. Computational Geosciences, 12, 1, 83-89

Paszyńska A., Paszyński M., 2007: *An Application of Hierarchical Chromosome Based Genetic Algorithm to the Optmization of Platform Shape*. Evolutionary computation and global optimization, Będlewo, Poland, June 11–13

Paszyńska A., Paszyński M., 2008: *An application of hierarchical chromosome based genetic algorithm for finding optimal initial mesh for the self-adaptive hp FEM calculations*. ICMAM 2008 : European workshop on Intelligent Computational Methods and Applied Mathematics, Kraków, Poland, March 28-31

Paszyńska A,, Paszyński M., Grabska E., 2008: *Graph Transformations for Modeling hp-Adaptive Finite Element Method with Triangular Elements*. M. Bubak et al. (Eds.): ICCS 2008, Part III, Lecture Notes In Computer Science 5103, 604–614

Paszyński M., 2006: *The aplication of agents to parallel mesh refinements in domain decomposition based parallel fully automatic hp adaptive finite element codes*. Lectures Notes in Computer Science 3993/2006, 751-758

Paszyński M., 2007a: *Performace of Multi Level Parallel Direct Solver for hp Finite Element Method*. Lecture Notes in Computer Science 4967, 1303-1312

Paszyński M., 2007b: *Agents based hierarchical parallelization of complex algorithms on the example of hp Finite Element Method*, Y. Shi et al. (Eds.): ICCS 2007, Part II, Lecture Notes in Computer Science 4488, 912–919

Paszyński M., 2007c: *Parallelization Strategy for Self-Adaptive PDE Solvers*. submitted to Fundamenta Informaticae

Paszyński M., Barabasz B., Schaefer R., 2007: *Efficient Adaptive Strategy for Solving Inverse Problems*, Y. Shi et al. (Eds.): ICCS 2007, Part I, Lecture Notes In Computer Science 4487, 342–349

Paszyński M., Demkowicz L., 2006: *Parallel Fully Automatic hp-Adaptive 3D Finite Element Package,* Engineering with Computers, 22, 3-4, 255-276

Paszyński M., Demkowicz L., Pardo D., 2005: *Verification of goal-oriented HP-adaptivity,* Computers and Mathematics with Applications, 50, 1395–1404

Paszyński M., Kurtz J., Demkowicz L., 2003: *Parallel Fully Automatic hp-Adaptive Codes for Acoustics and Electromagnetics*. Proceedings of Super-Computing 03, Phoenix, Arizona

Paszyński M., Kurtz J., Demkowicz L., 2006: *Parallel Fully Automatic hp-Adaptive 2D Finite Element Package*. Computer Methods in Applied Mechanics and Engineering, 195, 7-8, 25, 711-741

Paszyński M., Macioł P., 2006: *Application of Fully Automatic 3D hp Adaptive Code to Orthotropic Heat Transfer in Structurally Graded Materials*. Journal of Materials Processing Technology, 177, 1-3, 68-71

Paszyński M., Pardo D., Demkowicz L., Torres-Verdin C., 2006: *Parallel hp-Finite Element Simulations of 3D Resistivity Logging Instruments*. 13th ISPE International Conference on Concurrent Engineering: Research and Applications, Antibes, France, September 2006, in Leading the Web in Concurrent Engineering: Next Generation Concurrent Engineering, P. Ghodous et al. (Eds.) IOS Press, 635-642

Paszyński M., Pardo D., Demkowicz L., Torres-Verdin C., 2007: *An algorithm for transferring 2D arbitrary hp-refined finite element axially symmetric meshes to three dimensions*, XIV Conference in Computer Methods in Material Science, Zakopane, Poland

Paszyński M., Pardo D., Torres-Verdin C., 2007a: *Fast numerical simulation of 3D DC/AC borehole resistivity measurements with a parallel hp-adaptive and goaloriented finite-element formulation*, Seventh Annual Report of Joint Industry Research Consortium on Formation Evaluation, The University of Texas at Austin, August 16-17

Paszyński M., Pardo D., Torres-Verdín C., 2007b: *Simulation of 3D Resistivity Logging Measurements with a Parallel Implementation of 2D hp-Adaptive Goal-Oriented Finite Element Method*, International Conference of Numerical Analysis and Applied Mathematics, Corfu, Greece, September 16-20

Paszyński M., Pardo D., Torres-Verdin C., 2007c: *A nested dissection parallel direct solver for simulations of 3D DC/AC resistivity measurements* 9th U.S. National Congress on Computational Mechanics, San Francisco , July 23-26

Paszyński M., Pardo D., Torres-Verdin C., Demkowicz L., 2006: *Fast Numerical Simulations of 3D DC Borehole Resistivity Measurements with a Parallel Self-Adaptive Goal-Oriented Finite Element Formulation*. Sixth Annual Report of Joint Industry Research Consortium on Formation Evaluation, The University of Texas at Austin, August 16-18

Paszyński M., Pardo D., Torres-Verdin C., Demkowicz L., Calo V., 2007: *A Multi-Level Direct Substructuring Multi-Frontal Parallel Direct Solver for hp-Finite Element Method*. ICES-Report 07-33, The University of Texas in Austin, submitted to Parallel Computing

Paszyński M., Pardo D., Torres-Verdin C., Matuszyk P., 2007: *Efficient sequential and parallel solvers for hp Finite Element Method*, APCOM'07 – EPMESC XI : third Asian-Pacific Congress on Computational Mechanics in conjunction with eleventh international conference on Enhancement and Promotion of Computational Methods in Engineering and Science : December 3–6

Paszyński M., Paszyńska A., 2007: *Graph transformations for modeling parallel hp-adaptive FEM computations*, Lecture Notes in Computer Science, 4967, 1313-1322

Paszyński M., Romkes A., Collister E., Meiring J., Demkowicz L., Willson C. G., 2005: *On the Modeling of Step-and-Flash Imprint Lithography using Molecular Statics Models.* ICES Report 05-38, The University of Texas in Austin

Paszyński M., Schaefer R., 2008: *Reutilization of Partial LU Factorizations forSelf-adaptive hp Finite Element Method solver*. M. Bubak et al. (Eds.): ICCS 2008, Part I, Lecture Notes in Computer Science 5101, 965–974

Paszyński M., Szeliga D., Barabasz B., 2007: *An Algorithm for Relating Convergence Ratios of Inverse and Direct Problems Solutions by Means of the Self-Adaptive hp Finite Element Method*. CMM-2007 : 17th international conference on Computer Methods in Mechanics, Łódź–Spała, Poland, June 19–22

Patra A. K., 1999: *Parallel HP Adaptive Finite Element Analysis for Viscous Incompressible Fluid Problems*. PhD. Dissertation, University of Texas at Austin

Pike G., Semenzato L., Colella P., Hilfinger P. N., 1999: *Parallel 3D Adaptive Mesh Refinement in Titanium*. Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, USA, March

Płażek J., 1999: *Implementation Issues of Computational Fluid Dynamics Algorithms on Parallel Computers*. Lecture Notes in Computer Science, 1697, 349-355

Płażek J., 2000: *Scalable CFD Computations Using Message-Passing and Distributed Shared Memory Algorithms*. Lecture Notes in Computer Science, 1908, 282-288

Płażek J., Banaś K., Kitowski J., 2001: *Comparison of Message Passing and Shared Memory Implementations of the GMRES method on MIMD computers*. Scientific Programming, 9, 195-209

Płażek J., Banaś K., Kitowski J., Boryczko K., 1997: *Exploiting two-level parallelism in FEM applications*. Proceedings of the International Conference on High Performance Computing and Networking, Vol. 1225, Lecture Notes in Computer Science, 878-880

Rachowicz W., Pardo D., Demkowicz L., *Fully Automatic hp-Adaptivity in Three Dimensions*. Computer Methods in Applied Mechanics and Engineering (J.H. Argyris Memorial Issue), 195, 37-40, 4816-4842

Remacle J. F., Xiangrong Li, Shephard M.S., Flaherty J.E., 2000: *Anisotropic Adaptive Simulations of Transient Flows using Discontinuous Galerkin Methods*. Internation Journal of Numerical Methods in Engineering, 00, 1-6

Sagan H., 1994: *Space Filling Curve*. Springer, Berlin

Schaefer R., Barabasz B., Paszyński M., 2007: *Twin Adaptive Strategy for Solving Inverse Problems*. Evolutionary computation and global optimization, Będlewo, Poland, June 11–13

Schaefer R., Barabasz B., Paszyński M., 2008: *Asymptotic guarantee of success of the Hp-HGS strategy*. Evolutionary computation and global optimization, Szymbark, Poland, June 02 - 04

Schwab Ch., 1998: *P and hp Finite Element Methods*. Oxford University Press

Scott J. A., 2003: *Parallel Frontal Solvers for Large Sparse Linear Systems*. ACM Transactions on Mathematical Software, 29, 4, 395-417

Smith B. F., Bjőrstad P., Gropp W., 1996: *Domain Decomposition, Parallel Multi-Level Methods for Elliptic Partial Differential Equations* Cambridge University Press, New York, 1st ed.

Sołek W., 2004: *Numeryczna Symulacja Tiksotropowego Wypełniania Formy Stopami Glinu i Magnezu*. PhD. Dissertation, AGH University of Science and Technology (in Polish)

Stewart J. R., Edwards H. C., 2002: *SIERRA Framework Version 3: h-Adaptivity Design and Use*. SAND2002-4016, Albuquerque, NM: Sandia National Laboratories

Walsh T., Demkowicz L., 1999: *A Parallel Multifrontal Solver for hp-Adaptive Finite Elements*. TICAM Report 99-01

Wheat S., 1992: *A fine grained data migration approach to application load balancing on MP MIMP machines*. PhD. Dissertation, University of New Mexico, Albuquerque

Woodward P., Colella P., 1984: *The numerical simulation of two dimensional fluid flow with strong shocks*. Journal of Computational Physics, 54, 115-173

Zienkiewicz O. C., 1967: *Finite Element Method*, John Wiley, New York

ZOLTAN: *Data-Management Services for Parallel Applications*, http://www.cs.sandia.gov/Zoltan

Appendix A – *hp* Finite Element

There are two ways to define 1D *hp* reference finite element. The classical way is to use the Ciarlet definition (Ciarlet 1978) whereas a more familiar way for *hp* people is to use the definition introduced by Demkowicz 2007.

Definition A.1. 1D reference *hp* finite element (Demkowicz 2007)
The reference 1D *hp* finite element is a triple

$$\left(\hat{K}, X\left(\hat{K}\right), \Pi_p\right) \tag{A.1}$$

defined by the following four-step process:

1) geometry,

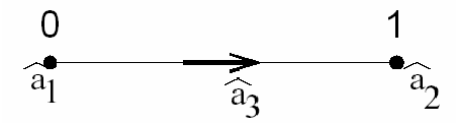


Fig.A_1. Nodes of the 1D *hp* finite element

2) selection of nodes – there are two vertex nodes $\hat{a}_1, \hat{a}_2$ and one edge node $\hat{a}_3$ selected, presented in Figure A_1,
3) definition of element space shape functions

$$X\left(\hat{K}\right) = \text{span}\left\{\hat{\chi}_j \in P^p\left(\hat{K}\right), j = 1,\ldots, p+1\right\} \tag{A.2}$$

where $P^p\left(\hat{K}\right)$ are polynomials of order p over $\hat{K} = [0,1]$, and

$$\hat{\chi}_1(\xi) = 1 - \xi \tag{A.3}$$

$$\hat{\chi}_2(\xi) = \xi \tag{A.4}$$

$$\hat{\chi}_3(\xi) = (1-\xi)\xi \tag{A.5}$$

$$\hat{\chi}_l(\xi) = (1-\xi)\xi(2\xi-1)^{l-3} \text{ for } l=4,\ldots,p+1 \tag{A.6}$$

4) definition of the projection-based interpolation operator

$$\Pi_p : H^1\left(\hat{K}\right) \to X\left(\hat{K}\right) \tag{A.7}$$

Given a function $u \in H^1\left(\hat{K}\right)$, its projection-based interpolant $\Pi_p u \in X\left(\hat{K}\right)$ is defined by requesting the following conditions

$$\Pi_p u(0) = u(0) \tag{A.8}$$

$$\Pi_p u(1) = u(1) \tag{A.9}$$

$$\left| \left(\Pi_p u\right)' - u' \right|_{H^1(0,1)} \to \min \tag{A.10}$$

where $\left| \left(\Pi_p u\right)' - u' \right|_{H^1(0,1)} = \int_0^1 \left(\left(\Pi_p u\right)' - u' \right)^2 d\xi$ is the H^1 semi-norm.

Remark A.1.

The minimization problem (A.10) is equivalent to:

Find $\Pi_p u \in X\left(\hat{K}\right)$ such that

$$\Pi_p u(0) = u(0) \tag{A.11}$$

$$\Pi_p u(1) = u(1) \tag{A.12}$$

$$\int_0^1 \left(\left(\Pi_p u\right)' - u' \right) v' d\xi = 0 \quad \forall v \in X\left(\hat{K}\right) \tag{A.13}$$

which in turn is equivalent to solving the following system of equations

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3,p+1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{p+1,1} & a_{p+1,2} & a_{p+1,3} & \cdots & a_{p+1,p+1} \end{bmatrix} \begin{Bmatrix} u_1^p \\ u_2^p \\ u_3^p \\ \vdots \\ u_{p+1}^p \end{Bmatrix} = \begin{Bmatrix} u(0) \\ u(0) \\ b_3 \\ \vdots \\ b_{p+1} \end{Bmatrix} \tag{A.14}$$

where

$$\Pi^p u = \sum_{i=1}^{p+1} u_i^p \chi_i \tag{A.15}$$

$$a_{ij} = \int_0^1 \chi_i' \chi_j' d\xi \tag{A.16}$$

$$b_i = \int_0^1 u' \chi_i' d\xi \tag{A.17}$$

Definition A.2. 1D reference *hp* finite element (based on Ciarlet 1978)

The reference 1D *hp* finite element is a triple

$$\left(\hat{K}, V^*\left(\hat{K}\right), X\left(\hat{K}\right)\right) \tag{A.18}$$

defined by the following four-step process:

1) geometry

$$\hat{K} = [0,1] \tag{A.19}$$

2) selection of nodes – there are two vertex nodes $\hat{a}_1, \hat{a}_2$ and one edge node $\hat{a}_3$ selected, presented in Figure A_1,

3) definition of space of degrees of freedom

$$V^*\left(\hat{K}\right) \tag{A.19}$$

where $V^*\left(\hat{K}\right)$ is the dual space to a functional space $V\left(\hat{K}\right)$, and $\{\psi_i\}_{i=1}^{p+1}$ is the basis of $V^*\left(\hat{K}\right)$.

There is one degree of freedom related to each vertex node, and p-1 degrees of freedom related to edge node.

$$\psi_1 : V\left(\hat{K}\right) \ni f \to f(0) \in R \tag{A.20}$$

$$\psi_2 : V\left(\hat{K}\right) \ni f \to f(1) \in R \tag{A.21}$$

and e.g. for p=2

$$\psi_3 : V\left(\hat{K}\right) \ni f \to 3\int_{\hat{K}} f'(\xi)(1-2\xi)d\xi \in R \tag{A.22}$$

4) construction of approximation space $X\left(\hat{K}\right) \subset V\left(\hat{K}\right)$

The elements of the basis of the approximation space $X\left(\hat{K}\right)$ are called the *shape functions* and are selected as the dual basis

$$\psi_i\left(\chi_j\right) = \delta_{ij} \tag{A.23}$$

Remark A.2

Let $\chi_j, j = 1,...,3$ denote shape functions defined in Definition A.1 in (A.3-5) for p=2, let $\psi_j : H^1(0,1) \to P^2(0,1)$ defined in Definition A.2 in (A.19-21). Then $\psi_i\left(\chi_j\right) = \delta_{ij}$.

Remark A.3 (Demkowicz 2007)

Let $\Pi_p : H^1(0,1) \to P^p\left(\hat{K}\right)$ be the projection-based interpolation operator. Let χ_1 and χ_2 denote the linear shape functions, and let $\chi_j, j = 3,..., p+1$ denote any basis for the space P_{-1}^p of polynomials of order less or equal p, vanishing at endpoints. Then, there is a unique set of linear and continuous functionals $\psi_j : H^1(0,1) \to P^p(0,1)$ such that

$$\Pi_p u = \sum_{j-1}^{p+1} \psi_j(u)\chi_j \tag{A.24}$$

Definition A.4. 1D *hp* finite element (Demkowicz 2007)

A 1D *hp* finite element is a triple

$$\left(K, X(K), \Pi_p\right) \tag{A.25}$$

defined by the following four-step process:

1) geometry

$$K = [x_l, x_r] \tag{A.26}$$

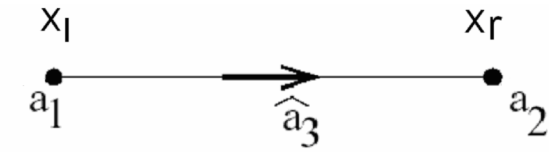


Fig.A_2. Nodes of the 1D *hp* finite element

2) selection of nodes – there are two vertex nodes a_1, a_2 and one edge node a_3 selected, presented in Figure A_2,
3) definition of element space shape functions $X(K)$

$$X(K) = \left\{ \chi = \hat{\chi} \circ x_K^{-1}, \hat{\chi} \in X(\hat{K}) \right\} \tag{A.27}$$

where $x_K : \hat{K} \to K$ is the map of the reference element $\hat{K} = [0,1]$ into an arbitrary element K

$$\hat{K} \ni \xi \to x_K(\xi) = x_l + (x_r - x_l) = x \in K \tag{A.28}$$

4) definition of the projection-based interpolation operator

$$\Pi_p : H^1(\hat{K}) \to X(\hat{K}) \tag{A.29}$$

defined by (A.8-10).

Remark A.4.
Note that the definition of the projection-based interpolation operator does not depend on the element K.

Definition A.5. 2D reference *hp* finite element (Demkowicz 2007)
The reference 2D *hp* finite element is a triple

$$(\hat{K}, X(\hat{K}), \Pi_p) \tag{A.30}$$

defined by the following four-step process:

1) geometry

$$\hat{K} = [0,1]^2 \tag{A.31}$$

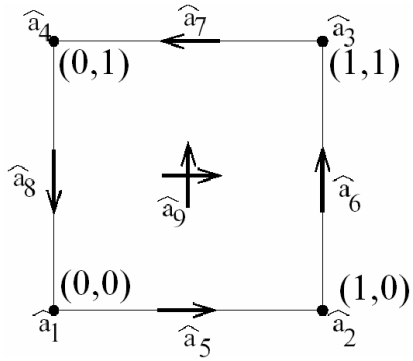


Fig.A.3 The 2D reference rectangular *hp* finite element

2) selection of nodes – there are four vertex nodes $\hat{a}_1, \hat{a}_2, \hat{a}_3, \hat{a}_4$, four edge nodes $\hat{a}_5, \hat{a}_6, \hat{a}_7, \hat{a}_8$ and one edge node $\hat{a}_9$ selected, presented in Figure A.3,

3) definition of element space shape functions $X(\hat{K})$

$$X(\hat{K}) = \text{span}\left\{\hat{\phi}_j \in Q^{(p_h, p_v)}(\hat{K}), j = 1,...,(p_h+1)(p_v+1)\right\} \tag{A.32}$$

where $Q^{(p_h, p_v)}$ are polynomials that are of order p_h with respect to ξ_1, and of order p_v with respect to ξ_2 over $\hat{K} = (0,1)^2$. Each element edge may have a different order of approximation $p_i, i = 1,...,4$. This allows us to match elements of different orders to one mesh. We assume that

$$p_1, p_3 \le p_h \text{ and } p_2, p_4 \le p_v \tag{A.33}$$

where the vertex shape functions are defined as

$$\hat{\phi}_1(\xi_1, \xi_2) = \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2) \tag{A.34}$$

$$\hat{\phi}_2(\xi_1, \xi_2) = \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2) \tag{A.35}$$

$$\hat{\phi}_3(\xi_1, \xi_2) = \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2) \tag{A.36}$$

$$\hat{\phi}_4(\xi_1, \xi_2) = \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2) \tag{A.37}$$

the edge shape functions are defined as

$$\hat{\phi}_{5,j}(\xi_1, \xi_2) = \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2) \quad j = 1,..., p_1 - 1 \tag{A.38}$$

$$\hat{\phi}_{6,j}(\xi_1, \xi_2) = \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad j = 1,..., p_2 - 1 \tag{A.39}$$

$$\hat{\phi}_{7,j}(\xi_1, \xi_2) = \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_2(\xi_2) \quad j = 1,..., p_3 - 1 \tag{A.40}$$

$$\hat{\phi}_{8,j}(\xi_1, \xi_2) = \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad j = 1,..., p_4 - 1 \tag{A.41}$$

and, finally, the interior shape functions are defined as

$$\hat{\phi}_{9,i,j}(\xi_1, \xi_2) = \hat{\chi}_{2+i}(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad i = 1,..., p_h - 1,\ j = 1,..., p_v - 1 \tag{A.42}$$

4) definition of the projection-based interpolation operator

$$\Pi_p : H^1(0,1) \to X(\hat{K}) \tag{A.43}$$

Given a function $u \in H^1(\hat{K})$, its projection-based interpolant $\Pi_p u \in X(\hat{K})$ is defined by requesting the following conditions

$$\Pi_p u(0,0) = u(0,0) \tag{A.44}$$

$$\Pi_p u(1,0) = u(1,0) \tag{A.45}$$

$$\Pi_p u(0,1) = u(0,1) \tag{A.46}$$

$$\Pi_p u(1,1) = u(1,1) \tag{A.47}$$

$$\left|(\Pi_p u)' - u'\right|_{H^1(e)} \to \min \text{ for each edge } e \text{ of the element } \hat{K} \tag{A.48}$$

where $\left|(\Pi_p u)' - u'\right|_{H^1(e)} = \int_e \left((\Pi_p u)' - u'\right)^2 dS$ is the first order seminorm.

$$\left| \left(\Pi_p u\right)' - u' \right|_{H^1(\hat{K})} \to \min \tag{A.49}$$

where $\left| \left(\Pi_p u\right)' - u' \right|_{H^1(\hat{K})} = \int_{\hat{K}} \left(\left(\Pi_p u\right)' - u' \right)^2 d\xi_1 \, d\xi_2$ is the first order seminorm.

Definition A.6. 2D *hp* finite element (Demkowicz 2007)
A 2D *hp* finite element is a triple

$$\left(K, X(K), \Pi_p\right) \tag{A.50}$$

defined by the following four-step process:

1) geometry K

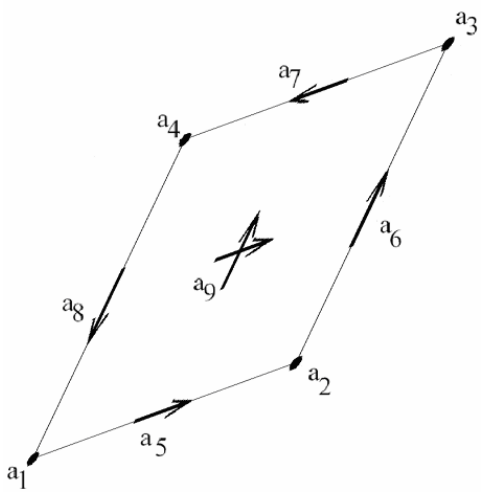


Fig.A.4 A 2D *hp* finite element

2) selection of nodes – there are four vertex nodes a_1, a_2, a_3, a_4, four edge nodes a_5, a_6, a_7, a_8 and one edge node a_9 selected, presented in Figure A.4,
3) definition of element space of shape functions $X(K)$

$$X(K) = \left\{ \chi = \hat{\chi} \circ x_K^{-1}, \hat{\chi} \in X(\hat{K}) \right\} \tag{A.51}$$

where $x_K : \hat{K} \to K$ is the map of the reference element $\hat{K} = [0,1]^2$ into an arbitrary element K

$$\hat{K} \ni \xi \to x_K(\xi) = x \in K \tag{A.52}$$

4) definition of the projection-based interpolation operator

$$\Pi_p : H^1(\hat{K}) \to X(\hat{K}) \tag{A.53}$$

defined by (A.43-47).

Appendix B – Self-adaptive *hp*-FEM

In this Appendix we introduce the mathematical details of the self-adaptive *hp*-FEM algorithm for 2D elliptic boundary value problem.

The algorithm, originally introduced by Demkowicz 2006 can be summarized in the following eight steps:

1) **generate an initial mesh**,
2) **solve the coarse mesh problem**,
3) **generate the fine mesh**,
4) **solve the fine mesh problem**,
5) **select the optimal refinement**,
6) **execute all required *h* refinements,**
7) **execute all required *p* refinements,**
8) **if** the maximum relative error of the solution is greater than the required accuracy, **then go to Step 2**. The new optimal mesh becomes the coarse mesh for the next iteration.

Definition B.1. Strong formulation of 2D elliptic boundary-value problem

Find $u: R^2 \supset \Omega \ni x \to u(x) \in R$ such that

$$-\sum_{i=1}^{2} \frac{\partial}{\partial x_i} \left(\sum_{j=1}^{2} a_{ij}(\mathbf{x}) \frac{\partial u}{\partial x_j} \right) + \sum_{j=1}^{2} b_j(\mathbf{x}) \frac{\partial u}{\partial x_j} + c(\mathbf{x}) u = f(\mathbf{x}) \text{ in } \Omega \tag{B.1}$$

$$u = u_D(\mathbf{x}) \text{ on } \Gamma_D \tag{B.2}$$

$$\sum_{i=1}^{2} \sum_{j=1}^{2} a_{ij}(\mathbf{x}) \frac{\partial u}{\partial x_j} n_i = g(\mathbf{x}) \text{ on } \Gamma_N \tag{B.3}$$

where

$$a_{ij}, b_j, c: R^2 \supset \Omega \ni \mathbf{x} \to a_{ij}(\mathbf{x}), b_j(\mathbf{x}), c(\mathbf{x}) \in R \tag{B.4}$$

are given coefficients of the differential operator (called "material data" from engineering point of view), and

$$u_D: R^2 \supset \Gamma_D \ni \mathbf{x} \to u_D(\mathbf{x}) \in R \tag{B.5}$$

$$g: R^2 \supset \Gamma_N \cup \Gamma_C \ni \mathbf{x} \to g(\mathbf{x}) \in R \tag{B.6}$$

$$\beta: R^2 \supset \Gamma_C \ni \mathbf{x} \to \beta(\mathbf{x}) \in R \tag{B.7}$$

(called "load data") are given, defined on the corresponding parts of the boundary $\partial\Omega = \Gamma = \Gamma_D \cup \Gamma_N \cup \Gamma_C$, presented in Figure B.1.

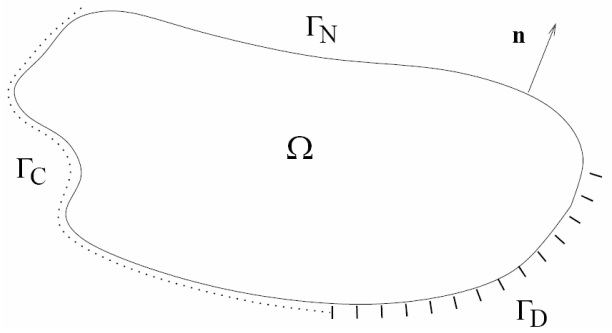


Fig.B.1. Computational domain Ω
with Dirichlet, Neumann and Cauchy boundaries $\partial\Omega = \Gamma = \Gamma_D \cup \Gamma_N \cup \Gamma_C$

Definition B.2. Variational (weak) formulation of 2D elliptic boundary-value problem
Find $u \in \widetilde{u}_D + V$ such that

$$b(u,v) = l(v) \quad \forall v \in V \tag{B.8}$$

where

$$b(u,v) = \int_\Omega \left(\sum_{i=1}^{2}\sum_{j=1}^{2} a_{ij} \frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_j} + \sum_{j=1}^{2} b_j \frac{\partial u}{\partial x_j} v + c\,u\,v \right) d\mathbf{x} + \int_{\Gamma_C} \beta\,u\,v\,ds \tag{B.9}$$

$$l(u,v) = \int_\Omega f\,v\,ds + \int_{\Gamma_C} g\,v\,ds \tag{B.10}$$

where $\widetilde{u}_D$ is a lift of the Dirichlet boundary condition

$$\mathrm{tr}(\widetilde{u}_D) = u_D \text{ on } \Gamma_D \tag{B.11}$$

and the $\widetilde{u}_D + V$ is an affine space obtained by adding the lift to test functions

$$\widetilde{u}_D + V = \{\widetilde{u}_D + v : v \in V\} \tag{B.12}$$

with

$$V = \left\{ v \in L^2(\Omega) : \int_\Omega \|v\|^2 + \|\nabla v\|^2 \, dx < \infty : \mathrm{tr}(v) = 0 \text{ on } \Gamma_D \right\} \tag{B.13}$$

Remark B.1
Any solution of the variational (weak) problem (B.8-13) is also a solution of the strong formulation (B.1-4).

Definition B.6. Coarse and fine meshes

The initial coarse mesh is an arbitrary mesh obtained by partitioning the domain Ω into a finite set $\left(K, X(K), \Pi_p\right) \in T_{hp}$ of *hp* finite elements

$$\bigcup_{K \in T_h} K = \Omega, \quad meas K_i \cap K_j = \begin{cases} 0 & i \neq j \\ K_i & i = j \end{cases} \tag{B.14}$$

and by selecting arbitrary polynomial orders of approximation on finite elements (under the minimum rule (A.32) constrained).

The fine mesh is obtained by breaking each finite element from the coarse mesh $\left(K, X(K), \Pi_p\right) \in T_{h/2, p+1}$ into four (in 2D) elements and increasing polynomial order of approximation by one. This can be expressed by the following conditions

$$\begin{aligned} &\forall \left(K, X(K), \Pi_p\right) \in T_{h/2, p+1} \\ &\exists K_1, K_2, K_3, K_4 \in T_{hp}\big|_K : K = K_1 \cup K_2 \cup K_3 \cup K_4, \\ &X(K_1), X(K_2), X(K_3), X(K_4) \in T_{hp}\big|_{X(K)}, \dim X(K_i) = \dim(K) + 5 \end{aligned} \tag{B.15}$$

where $T_{hp}\big|_K$ and $T_{hp}\big|_{X(K)}$ denote the projections onto the first and second component of $\left(K, X(K), \Pi_p\right) \in T_{h/2, p+1}$, respectively. Five new shape functions over $X(K_i)$ result from increasing the polynomial order of approximation on four element edges and one interior (adding four new shape functions on element edges and one new shape function on element interior).

Definition B.6. Coarse and fine mesh approximation spaces

The coarse mesh approximation space is defined as

$$V_{hp} = \text{span}\{ e_{hp}^j : \forall K \in T_{hp}\big|_K, \forall \phi_k \in X(K), \exists! e_{hp}^i : e_{hp}^i\big|_K = \phi_k \} \tag{B.16}$$

where e_{hp}^i is a global shape function (element of basis of V_{hp}), ϕ_k is a local shape function, $(k, K) \to i(k, K)$ is the function assigning global number $i(k, K)$ of degrees of freedom to the local shape function k from element K.

The fine mesh approximation space is defined in an analogous way as

$$\begin{aligned} &V_{h/2, p+1} = \\ &= \text{span}\{ e_{h/2, p+1}^j : \forall K \in T_{h/2, p+1}\Big|_K, \forall \phi_k \in X(K), \exists! e_{h/2, p+1}^i : e_{h/2, p+1}^i\Big|_K = \phi_k \} \end{aligned} \tag{B.17}$$

where $e_{h/2, p+1}^i$ is a global shape function (element of basis of $V_{h/2, p+1}$) ϕ_k is a local shape function, $(k, K) \to i(k, K)$ is the function assigning global number $i(k, K)$ of degrees of freedom to the local shape function k from element K.

Definition B.7. Coarse mesh problem

Find $\left\{u^{\,i}_{\,hp}\right\}_{i=1}^{N_{hp}}$ coefficients (called *degrees of freedom*) of the approximate solution $V \supset V_{hp} \ni u_{hp} = \sum_{i=1}^{N_{hp}} u^{\,i}_{\,hp}\, e^{\,i}_{\,hp}$ such that

$$\sum_{m=1}^{N_{hp}} b\left(e^m_{hp}, e^n_{hp}\right) = l\left(e^n_{hp}\right) \quad n = 1,\ldots,N_{hp} \tag{B.18}$$

$$b\left(e^m_{hp}, e^n_{hp}\right) = \int_\Omega \left(\sum_{i=1}^{2}\sum_{j=1}^{2} a_{ij} \frac{\partial e^m_{hp}}{\partial x_j}\frac{\partial e^n_{hp}}{\partial x_j} + \sum_{j=1}^{2} b_j \frac{\partial e^m_{hp}}{\partial x_j} e^n_{hp} + c\, e^m_{hp}\, e^n_{hp} \right) d\,\mathbf{x} + \tag{B.19}$$

$$+ \int_{\Gamma_C} \beta\, e^m_{hp}\, e^n_{hp}\, d\,s$$

$$l\left(e^n_{hp}\right) = \int_\Omega f\, e^n_{hp}\, d\,s + \int_{\Gamma_C \cup \Gamma_N} g\, e^n_{hp}\, d\,s \tag{B.20}$$

The approximation space $V_{hp} \subset V$ with basis $\left\{e^{\,i}_{\,hp}\right\}_{i=1}^{N_{hp}}$ of continuous functions is constructed by joining the corresponding element local shape functions from the coarse mesh.

Definition B.8. Fine mesh problem

Find $\left\{u^{\,i}_{\,h/2,p+1}\right\}_{i=1}^{N_{h/2,p+1}}$ coefficients (called *degrees of freedom*) of the approximate solution $V \supset V_{h/2,p+1} \ni u_{h/2,p+1} = \sum_{i=1}^{N_{h/2,p+1}} u^{\,i}_{\,h/2,p+1}\, e^{\,i}_{\,h/2,p+1}$ such that

$$\sum_{m=1}^{N_{h/2,p+1}} u^m_{h/2,p+1}\, b\left(e^m_{h/2,p+1}, e^n_{h/2,p+1}\right) = l\left(e^n_{h/2,p+1}\right) \quad n = 1,\ldots,N_{h/2,p+1} \tag{B.21}$$

$$b\left(e^m_{h/2,p+1}, e^n_{h/2,p+1}\right) = \int_\Omega \left(\sum_{i=1}^{2}\sum_{j=1}^{2} a_{ij} \frac{\partial e^m_{h/2,p+1}}{\partial x_j}\frac{\partial e^n_{h/2,p+1}}{\partial x_j} + \sum_{j=1}^{2} b_j \frac{\partial e^m_{h/2,p+1}}{\partial x_j} e^n_{h/2,p+1} + c\, e^m_{h/2,p+1}\, e^n_{h/2,p+1} \right) d\,\mathbf{x} + \tag{B.22}$$

$$+ \int_{\Gamma_C} \beta\, e^m_{h/2,p+1}\, e^n_{h/2,p+1}\, d\,s$$

$$l\left(e^n_{h/2,p+1}\right) = \int_\Omega f\, e^n_{h/2,p+1}\, d\,\mathbf{x} + \int_{\Gamma_C \cup \Gamma_N} g\, e^n_{h/2,p+1}\, d\,s \tag{B.23}$$

The approximation space $V_{h/2,p+1} \subset V$ with basis $\left\{e^{\,i}_{\,h/2,p+1}\right\}_{i=1}^{N_{h/2,p+1}}$ of continuous functions is constructed by joining the corresponding element local shape functions from the fine mesh.

Remark B.2
The solution of either coarse or fine mesh problem consists in generating the system of linear equations and solving the generated system by employing a direct solver (e.g. multi-frontal solver).

The generation of the system of linear equations can be obtained by executing the following algorithm

$B=0$, $L=0$
Loop with respect to elements K
  Loop with respect to element local shape functions ϕ_{k_1}
    $L(i(k_1,K))+=l(\phi_{k_1},\phi_{k_2})$
    Loop with respect to element local shape functions ϕ_{k_2}
      $B(i(k_1,K),i(k_2,K))+=b(\phi_{k_1},\phi_{k_2})$

where $(k_1,K)\to i(k_1,K)$ is the function assigning global number $i(k_1,K)$ of degrees of freedom to the local shape function k_1 from element K, and B is the so-called *global stiffness matrix*, and L is the so-called *global load vector*.

Remark B.2
The integrals involved in computing $b(\phi_{k_1},\phi_{k_2})$ and $l(\phi_{k_1},\phi_{k_2})$ are computed for the reference element $\hat{K}$ after performing the following change of variables.

$$b(\phi_{k_1},\phi_{k_2})=\int_K\left(\sum_{i=1}^{2}\sum_{j=1}^{2}a_{ij}(\mathbf{x})\frac{\partial\phi_{k_1}(\mathbf{x})}{\partial x_j}\frac{\partial\phi_{k_2}(\mathbf{x})}{\partial x_j}+\sum_{j=1}^{2}b_j(\mathbf{x})\frac{\partial\phi_{k_1}(\mathbf{x})}{\partial x_j}\phi_{k_2}(\mathbf{x})+c(\mathbf{x})\phi_{k_1}(\mathbf{x})\phi_{k_2}(\mathbf{x})\right)d\mathbf{x}+$$
$$+\int_{\Gamma_C\cap\hat{K}}\beta(\mathbf{x})\phi_{k_1}(\mathbf{x})\phi_{k_2}(\mathbf{x})|\,\mathrm{Jac}(x_K)|\,ds=$$
$$\int_{\hat{K}}\sum_{i=1}^{2}\sum_{j=1}^{2}a_{ij}(x_K(\xi))\frac{\partial\phi_{k_1}(x_K(\xi))}{\partial\xi_n}\frac{\partial\phi_{k_2}(x_K(\xi))}{\partial\xi_m}\frac{\partial\xi_m}{\partial x_i}\frac{\partial\xi_n}{\partial x_j}|\,Jac(x_K)|d\mathbf{x}+$$
$$\int_{\hat{K}}\left(\sum_{j=1}^{2}b_j(x_K(\xi))\frac{\partial\phi_{k_1}(x_K(\xi))}{\partial\xi_n}\phi_{k_2}(x_K(\xi))\frac{\partial\xi_n}{\partial x_j}+c(x_K(\xi))\phi_{k_1}(x_K(\xi))\phi_{k_2}(x_K(\xi))\right)|\,Jac(x_K)|d\mathbf{x}+$$
$$\int_{\Gamma_C\cap K}\beta(x_K(\xi))\phi_{k_1}(x_K(\xi))\phi_{k_2}(x_K(\xi))ds \tag{B.24}$$

$$l(\phi_{k_1})=\int_K f(\mathbf{x})\phi_{k_1}(\mathbf{x})d\mathbf{x}+\int_{(\Gamma_C\cup\Gamma_N)\cap K}g(\mathbf{x})\phi_{k_1}(\mathbf{x})ds=$$
$$=\int_{\hat{K}}f(x_K(\xi))\phi_{k_1}(x_K(\xi))|\,\mathrm{Jac}(x_K)|\,d\xi+\int_{(\Gamma_C\cup\Gamma_N)\cap\hat{K}}g(x_K(\xi))\phi_{k_1}(x_K(\xi))|\,\mathrm{Jac}(x_K)|\,ds \tag{B.25}$$

Here, $x_K : \hat{K} \to K$ is the map (A.51). Thus, the geometry of an arbitrary element K is coded in the Jacobian

$$Jac\left(x_K\right) = \left|\frac{d\,\mathbf{x}}{d\,\boldsymbol{\xi}}\right| = \left|\begin{bmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_1}{\partial \xi_2} \\ \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_2} \end{bmatrix}\right| \tag{B.26}$$

Note that partial derivatives involved in (B.23-24) can be obtained by computing the inverse of $\frac{d\,\mathbf{x}}{d\,\boldsymbol{\xi}}$ matrix

$$\begin{bmatrix} \frac{\partial \xi_1}{\partial x_1} & \frac{\partial \xi_1}{\partial x_2} \\ \frac{\partial \xi_2}{\partial x_1} & \frac{\partial \xi_2}{\partial x_2} \end{bmatrix} = \frac{d\,\boldsymbol{\xi}}{d\,\mathbf{x}} = \frac{d\,\mathbf{x}}{d\,\boldsymbol{\xi}}^{-1} = \begin{bmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_1}{\partial \xi_2} \\ \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_2} \end{bmatrix}^{-1} \tag{B.27}$$

Definition B.9. Projection-based interpolant

Let $V_{hp} \subset V_{h/2,p+1} \subset V$ be the coarse and fine mesh approximation spaces. Let V_w be any intermediate approximation space such that $V_{hp} \subset V_w \subset V_{h/2,p+1}$. Let $K \in T_{hp}$ be an hp finite element from the coarse mesh. Let $u_{hp} \in V_{hp}$ and $u_{h/2,p+1} \in V_{h/2,p+1}$ be the coarse and fine mesh solution, respectively.

The projection-based interpolant w of $u_{h/2,p+1} \in V_{h/2,p+1}$ into V_w over element K, namely $w\big|_K \in V_w\big|_K$, is obtained from the following three steps procedure:

1) interpolation at vertices

$$w(a_i) = u_{h/2,p+1}(a_i) \quad i = 1,...,4 \tag{B.28}$$

where a_i are vertex nodes of element K,

2) projection on element edges

$$\left| w' - u'_{h/2,+1} \right|_{H^1(e)} \to \min \text{ for each edge } e \text{ of the element } K \tag{B.29}$$

where $\left| w' - u'_{h/2,p+1} \right|_{H^1(e)} = \int_e \left(w' - u'_{h/2,p+1} \right)^2 dS$ is the first order semi-norm.

3) projection on element interior

$$\left| w' - w'_{h/2,p+1} \right|_{H^1(K)} \to \min \tag{B.30}$$

where $\left| w' - u'_{h/2,p+1} \right|_{H^1(K)} = \int_K \left(w' - u'_{h/2,p+1} \right)^2 d\xi_1\, d\xi_2$ is the first order semi-norm.

Definition B.10. Optimal approximation space for an element

Let $V_{hp} \subset V_{h/2,p+1} \subset V$ be the coarse and fine mesh approximation spaces. Let T_{hp} represent the coarse mesh elements. Let $u_{hp} \in V_{hp}$ and $u_{h/2,p+1} \in V_{h/2,p+1}$ be the coarse and fine mesh problem solutions, respectively.

The approximation space $V_{opt}{}^{K}$ is called the optimal approximation space for an element $K \in T_{hp}$, if the projection-based interpolant w_{opt} of $u_{h/2,p+1} \in V_{h/2,p+1}$ into $V_{opt}{}^{K}$ for element K realizes the following minimum

$$\frac{\left|u_{h/2,p+1}-u_{hp}\right|_{H^1(K)}-\left|u_{h/2,p+1}-w_{opt}\right|_{H^1(K)}}{\Delta\,\mathrm{nrdof}\left(\mathrm{V_{hp}},\mathrm{V_{opt}}^{K},K\right)}= \tag{B.31}$$

$$=\min_{V_{hp}\subseteq V_w\subseteq V_{h/2,p+1}}\frac{\left|u_{h/2,p+1}-u_{hp}\right|_{H^1(K)}-\left|u_{h/2,p+1}-w\right|_{H^1(K)}}{\Delta\,\mathrm{nrdof}\left(\mathrm{V_{hp}},\mathrm{V_{opt}}^{K},K\right)}$$

where w is the projection-based interpolant of $u_{h/2,p+1} \in V_{h/2,p+1}$ into V_w for element K, and $\Delta\,\mathrm{nrdof}(\mathrm{V},\mathrm{X},\mathrm{K}) = \dim V\big|_K - \dim X\big|_K$.

Remark B.3

The simplest algorithm finding the optimal approximation space on the coarse mesh T_{hp} is the following

```
Loop through coarse mesh elements K ∈ T_hp
  Loop through approximation spaces V_opt^K for element K
    rate_min = ∞
    Compute the projection-based interpolant w|_K of u_{h/2,p+1}|_K
    into V_opt^K
    Compute the error decrease rate
    rate(w) = ( |u_{h/2,p+1} − u_{h,p}|_{H1(K)} − |u_{h/2,p+1} − w|_{H1(K)} ) / Δ nrdof( V_hp, V_opt^K, K )
    If rate(w) < rate_min then rate_min = rate(w)
    V_opt^K corresponding to rate_min is the optimal approximation
    space for K
```

Remark B.5

The construction of a new global optimal approximation space V_{opt} consists in executing the mesh refinements corresponding to the upgrade of the element coarse mesh approximation space $V_{hp}\big|_K$ into the selected element optimal approximation space $V_{opt}{}^K$.

Appendix C – Technical details on implementation

Appendix C includes some technical details on the implementation of the parallel self-adaptive *hp*-FEM algorithms.

Sections C.1 and C.2 explain how the implementation of parallel, two and three-dimensional fully automatic *hp*-adaptive Finite Element Method codes *par2Dhp90* and *par3Dhp90* are related to the CP-graph grammar model presented in the dissertation. The parallel codes are the extensions of the sequential codes *2Dhp90* (Demkowicz 2006) and *3Dhp90* (Demkowicz, Kurtz, Pardo, Paszyński, Rachowicz, Zdunek 2007), which implement the self-adaptive *hp*-FEM algorithm. The codes have been written in Fortran 90 with Message Passing Interface (MPI) and the load balancing is done through an interface with the Zoltan library. The selection of Fortran 90 language is motivated by two facts. First, the parallel codes are the extensions of the serial codes already implemented in Fortran 90. Second, the Fortran implementation provides extremely efficient numerical computations, as it is a low-level programming language.

We use here the Unified Modeling Language (UML) introduced by Booch, Rumbaugh, Jacobson, 1998.

C.1 Parallel two-dimensional self-adaptive *hp*-FE code *par2Dhp90*

In this section, the correspondence between the parallel two-dimensional self-adaptive *hp*-Finite Element Method code *par2Dhp90* and the CP-graph grammar model is discussed. The parallel code does not support directly the CP-graph grammar model for mesh transformations occurring during the self-adaptive *hp*-FEM algorithm, however, the data structure and mesh transformations follow exactly the patterns expressed by the CP-graph grammar.

In the next part of the section, the `element`, `node` and `vertex` classes will be introduced and related to CP-graph vertices. In the *par2Dhp90* applications, the graph vertices are identified by `element`, `node` and `vertex` classes. The class declarations are introduced below.

Class:
Element
Attributes:
character(5) : type
'quadr', 'trian' the information about type of the element (currently, the code supports only quadrilateral elements)
integer(4) : neighbors
pointers to 4 adjacent elements stored in ELEMS table
integer(4) : vertices
pointers to 4 vertices stored in VERTS table
integer(5) : nodes
pointers to 4 edge nodes and 1 middle node stored in NODES table
integer(4) : bcond
boundary conditions for 4 element edges
(0 non / 1 Drichlet / 2 Cauchy)
integer : ghost
flag indicating if the element is a ghost element
(0 no / 1 yes)

Class:
Vertex
Attributes:
double(2) : geom_coord
geometrical coordinates of the node
integer : father
pointer to father node in NODES table
integer : father_iel
pointer to father initial mesh element in ELEMS table
(if any)
integer : sbs
0 if the vertex is not located on the interface, non zero if located on the interface
(if a vertex belongs to more then one sub-domain)
= index of interface node, used by parallel solver

Class:
Node
Attributes:
character(4) : type
type of the node:
'medg' for edge node,
'mdlq' for middle node
integer : order
order of approximation

```
integer : father
      pointer to father node in NODES table
integer : ref_kind
      flag coding refinement type of the node
integer, dimension(:) : vertex_sons
      dynamically allocated table storing pointers
      to all son vertices for broken edge or interior nodes
integer, dimension(:) : edge_sons
      dynamically allocated table storing pointers
      to all edge son nodes for broken edge or interior nodes
integer, dimension(:) : interior_sons
      dynamically allocated table storing pointers
      to all interior son nodes for broken interior nodes
double, dimension(:,:) : geom_coord
      geometrical degrees of freedom used
      to express geometry of curvilinear  edges,
      expressed as a combination of node shape functions
double, dimension(:,:) : zdofs
      degrees of freedom
      (coefficients of local shape functions)
      used for local approximation of the solution
integer : sbs
      0 if the vertex is not located on the interface,
      non zero if located on the interface
      (if  a  vertex  belongs  to  more  then  one  sub-domain)
      = index of interface node, used by parallel solver
integer : kref
      required refinement for the node,
      used during the virtual refinements
```

The classes are stored in the following `ELEMS`, `VERTS` and `NODES` collections

Collections of objects:

```
type(Element), pointer, dimension(:) :: ELEMS
      dynamically allocated table of Element class objects
type(Node), pointer, dimension(:) :: NODES
      dynamically allocated table of Node class objects
type(Vertex), pointer , dimension(:) :: VERTS
      dynamically allocated table of Vertex class objects
```

Let us focus on an exemplary CP-graph from Figure 2.38, representing two initial mesh elements, each of them broken into 4 son elements. The initial mesh elements, represented by **iel** graph vertices, correspond to `Element` class objects. In other words, the `Element` class objects are created for initial mesh elements only. Each initial mesh element may have up to four adjacent initial mesh elements. In the presented CP-graph example, each initial mesh element has one neighbor. In the *hp2d* code, the pointers to neighbors (actually indices of neighbors in `ELEMS` collection) are stored in `bcond` array.

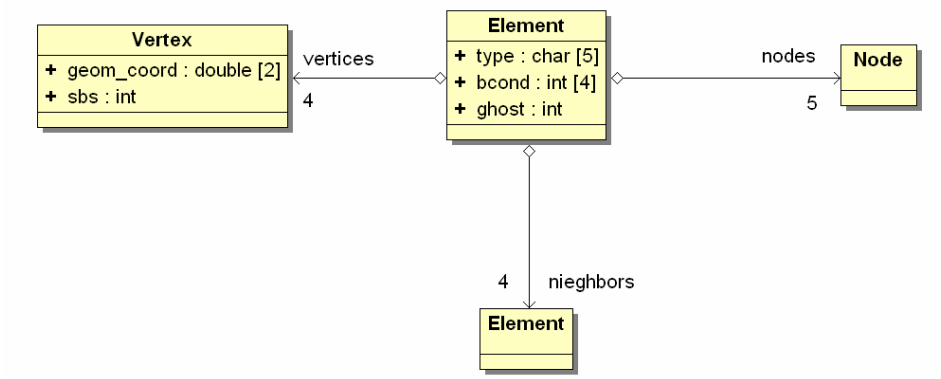


Fig.C.1. Relations between `Element`, `Node` and `Vertex`

Let us discuss the relations between `Element`, `Node` and `Vertex` classes presented in Figure C.1. The graph transformation introduced in Figure 2.4 generates a structure of each initial mesh element. Each element consists of four vertices, four edges and one interior. The element vertices represented by **v** graph vertices, correspond to `Vertex` class objects. Each `Element` class object aggregates the `vertices` list of four `Vertex` objects, so **iel** graph vertex is connected to four graph vertices labeled with **v**. The element edges are represented by **F** graph vertices, as presented in Figure 2.3 (broken edges are represented by graph vertices labeled with **e**). These graph vertices are related to `Node` class objects with `type='medg'`. The element interiors are represented by capital **I** graph vertices, as presented in Figure 2.3 (broken interiors are represented by graph vertices labeled with small **i**). These graph vertices are related to `Node` class object with `type='mdle'`. Each `Element` class object aggregates the `nodes` list of four `Node` objects of `'medg'` type and one `Node` object of `'mdle'` type, so **iel** graph vertex is connected to four graph vertices labeled with **v**. The links from element vertices, edges and interiors to father initial mesh element are stored in `father_iel` attribute.

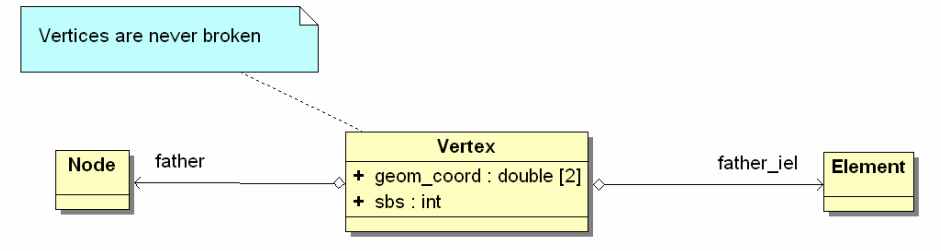


Fig.C.2. Links stored by `Vertex` class object

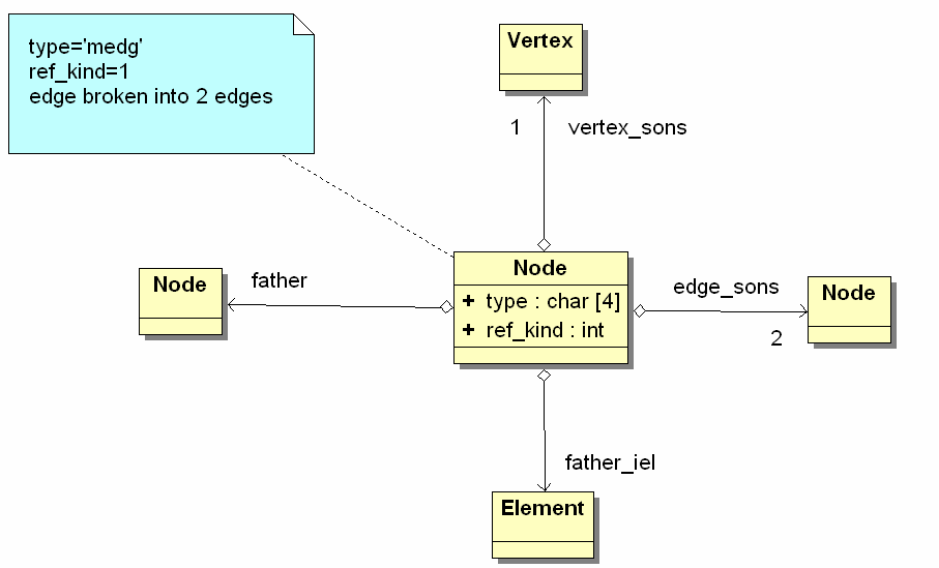


Fig.C.3. Relations between `Element`, `Node` and `Vertex` class objects for broken element edge

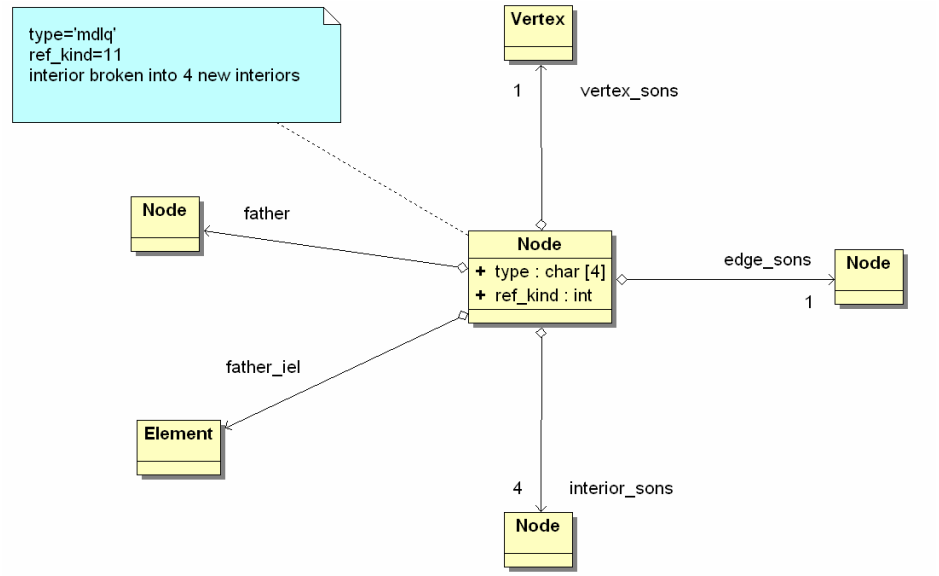


Fig.C.4. Relations between `Element`, `Node` and `Vertex` class objects for element interior broken in horizontal and vertical directions

The element vertices are never broken. However, `Vertex` class objects are created as a result of breaking element edges or interiors. Thus, `Vertex` class object created as a result of mesh refinements keeps `father` link to father `Node` objects, while initial mesh

elements vertices keep `father_iel` link to the initial mesh element, which is illustrated in Figure C.2.

When an element edge is broken, one new graph vertex representing element vertex and two new graph vertices representing element edges are created (compare Figure 2.27). These newly created graph vertices are again represented by one `Vertex` class object and two `Node` class objects, with `type='medg'`. The `Node` class object with `type='medg'` representing element edge aggregates a list of `vertex_sons` and `edge_sons`. When the edge is broken, the `ref_kind` attribute of the `Node` class object is set to 1, and the references to newly created `Vertex` class and `Node` class objects are stored on these lists. This is illustrated in Figure C.3.

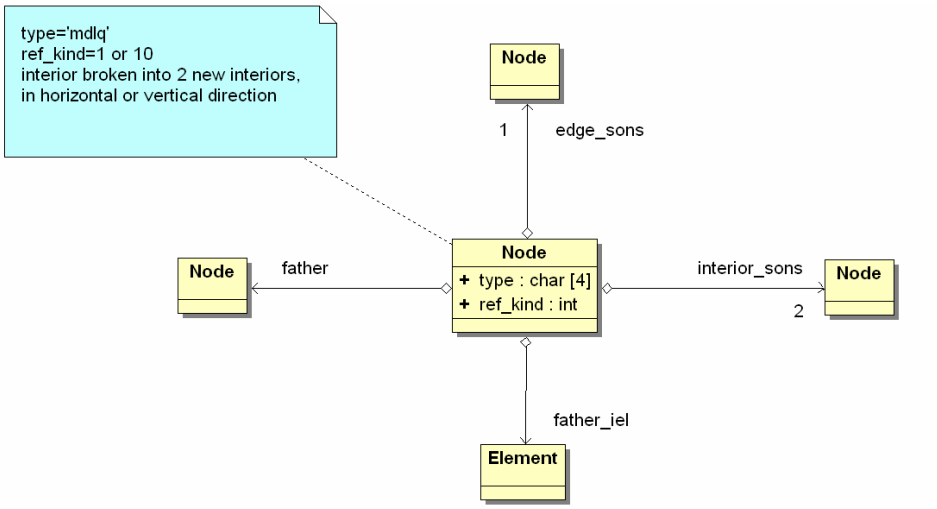


Fig.C.5. Relations between `Element, Node` and `Vertex` class objects for element interior broken in one direction

When an element interior is broken in both, horizontal and vertical directions, one new graph vertex representing element vertex, four new graph vertices representing element edges, and four new graph vertices representing element interiors are created (compare Figure 2.26). These newly created graph vertices are again represented by one `Vertex` class object, four `Node` class objects with `type='medg'`, and four `Node` class objects with `type='mdle'`. The `Node` class object with `type='mdle'` representing element interior aggregates a list of `vertex_sons, edge_sons` and `interior_sons`. When the interior is broken in both directions, the `ref_kind` attribute of the `Node` class object is set to 11, and the references to newly created `Vertex` class and `Node` class objects are stored on these lists. This is illustrated in Figure C.4.

The *hp2dpar* application supports also anisotropic mesh refinements, thus, the element interior can be broken in one direction. When an element interior is broken in one direction, one new graph vertex representing element edge, and two new graph vertices representing element interiors are created. These newly created graph vertices are again represented by one `Node` class objects with `type='medg'`, and two `Node` class objects with

`type='mdle'`. The `Node` class object with `type='mdle'` representing element interior aggregates a list of `vertex_sons`, `edge_sons` and `interior_sons`. When the interior is broken in one direction, the `ref_kind` attribute of the `Node` class object is set to 1 or 10, depending on the direction, and the references to newly created `Node` class objects are stored on these lists. This is illustrated in Figure C.5.

The mesh transformations algorithms implemented in *par2Dhp90* correspond to graph transformations defined by the graph grammar. For more technical details on the two-dimensional implementation *parhp2D90*, see Paszyński, Kurtz, Demkowicz 2006.

C.2 Parallel three-dimensional self-adaptive *hp*-FE code *par3Dhp90*

In this section, we discuss the correspondence between the parallel three-dimensional self-adaptive *hp*-Finite Element Method code *par3Dhp90* and the CP-graph grammar model. The graph grammar model has been designed to support two-dimensional mesh transformations. However, it is possible to extend the graph grammar for three dimensions. The three-dimensional code *par3Dhp90* employs the extension of the two-dimensional data structure designed for the two-dimensional code *par2Dhp90*.

In the following part of this appendix, the *element, node* and *vertex* classes are extended to support three-dimensional mesh computations. The extended class declarations are introduced below.

```
Class:
Element
Attributes:
character(5) : type
      'prism','brick','tetra' the information about
      type of the element
      (currently, the code supports only brick elements)
integer(6) : neighbors
      pointers to 6 neighboring elements stored
      in ELEMS table
integer(8) : vertices
      pointers to 8 vertices stored in VERTS table
integer(19) : nodes
      pointers to 12 edge nodes,
      6 face nodes and 1 middle node stored in NODES table
integer(6) : bcond
      boundary conditions for 6 element faces
      (0 non / 1 Drichlet / 2 Cauchy)
integer : ghost
      flag indicating if the element is a ghost element
      (0 no / 1 yes)
```

Class:
Vertex
Attributes:
double(3) : geom_coord
geometrical coordinates of the node
integer : father
pointer to father node in NODES table
integer : father_iel
pointer to father initial mesh element in ELEMS table
(if any)
integer : sbs
0 if the vertex is not located on the interface,
non zero if located on the interface
(if a vertex belongs to more then one sub-domain)
= index of interface node, used by parallel solver

Class:
Node
Attributes:
character(4) : type
type of the node: 'medg' for edge node,
'mfac' for face node,
'mdlb' for middle node of element interior node
integer : order
order of approximation
integer : father
pointer to father node in NODES table
integer : ref_kind
flag coding refinement type of the node
integer, dimension(:) : vertex_sons
dynamically allocated table storing pointers to all
son vertices for broken edge, face or interior nodes
integer, dimension(:) : edge_sons
dynamically allocated table storing pointers to all
edge son nodes for broken edge, face or interior nodes
integer, dimension(:) : face_sons
dynamically allocated table storing pointers to all
face son nodes for broken face or interior nodes
integer, dimension(:) : interior_sons
dynamically allocated table storing pointers to all
interior son nodes for broken interior nodes
double, dimension(:,:) : geom_coord
geometrical degrees of freedom used to express geometry
of curvilinear edges,
expressed as a combination of node shape functions
double, dimension(:,:) : zdofs
degrees of freedom

```
      (coefficients of local shape functions)
      used for local approximation of the solution
integer : sbs
      0 if the vertex is not located on the interface,
      non zero if located on the interface
      (if a vertex belongs to more then one sub-domain)
      = index of interface node, used by parallel solver
integer : kref
      required refinement for the node,
      used during the virtual refinements
```

The classes are stored in the following `ELEMS`, `VERTS` and `NODES` collections

```
Collections of objects:
type(Element), pointer, dimension(:) :: ELEMS
      dynamically allocated table of Element class objects
type(Node), pointer, dimension(:) :: NODES
      dynamically allocated table of Node class objects
type(Vertex), pointer , dimension(:) :: VERTS
      dynamically allocated table of Vertex class objects
```

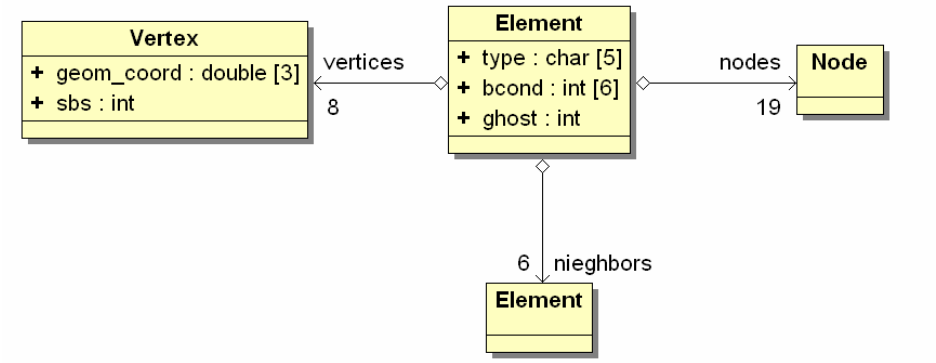


Fig.C.6. Relations between `Element`, `Node` and `Vertex` classes on the level of initial mesh

The relations between classes on the level of initial mesh elements are illustrated in Figure C.6. In three dimensions, each brick element consists of eight vertices, twelve edges, six faces and one interior. The element vertices are represented by `Vertex` class objects, stored on the vertices list aggregated by `Element` class. The element edges are represented by `Node` class objects with `type='mdle'`, element faces by `Node` class objects with `type='mfac'` and element interiors by `Node` class object with `type='mdlb'`. The edges, faces and interior `Node` class objects are stored on the `nodes` list aggregated by `Element` class.

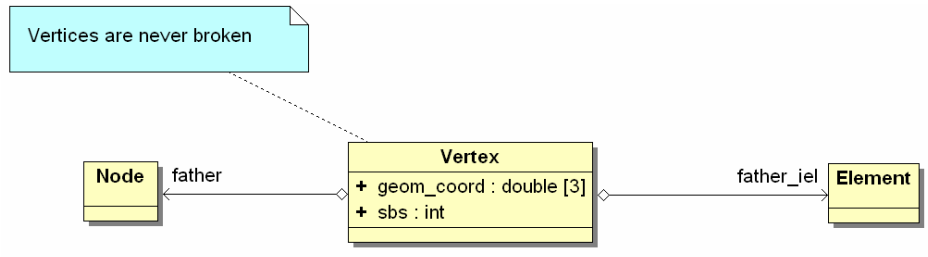


Fig.C.7. Father / son relations of element vertices

As in case of the two-dimensional implementation, the element vertices represented by `Vertex` class object are never broken. However, new `Vertex` class objects may be created as a result of `Node` class objects refinement. Such `Vertex` class objects have `father` pointer set. `Vertex` class objects that belong to initial mesh element has `father_iel` pointer set to Element class object representing initial mesh element. The relations are presented in Figure C.7.

There are many possibilities of breaking a three-dimensional brick element:

a) An element can be broken into two new son elements, with the cutting plane orthogonal to either *x, y* or *z* direction.
b) An element can be broken into four new son elements, with two cutting planes parallel to either *x, y* or *z* direction.
c) An element can be broken into eight new son elements.

Each refinement consists in suitable breaking of selected element edges, faces and interior.

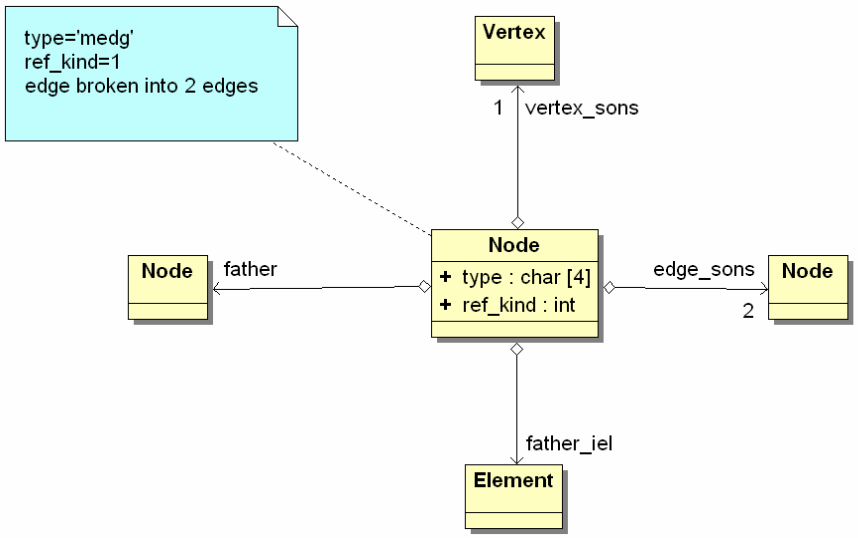


Fig.C.8. Relations for an element edge broken into two new edges

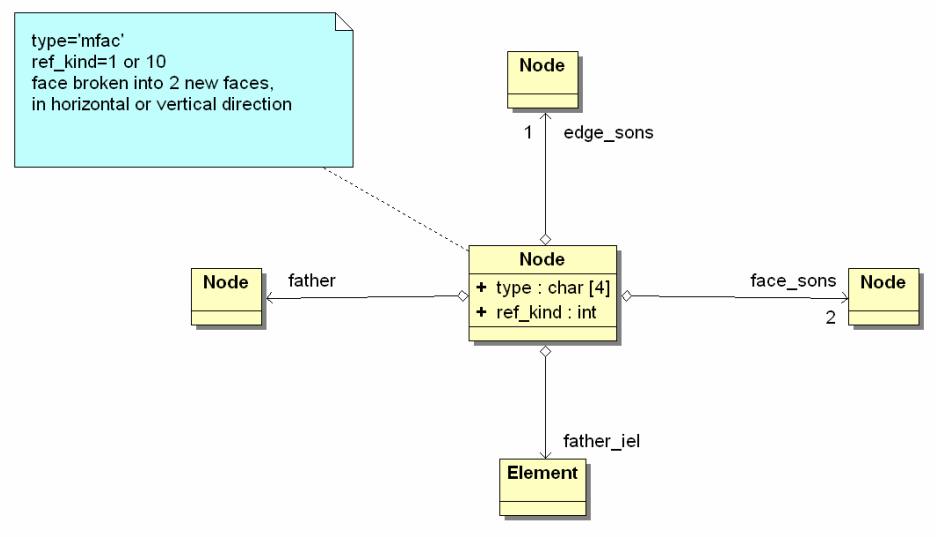


Fig.C.9. Relations for an element face broken into two new faces

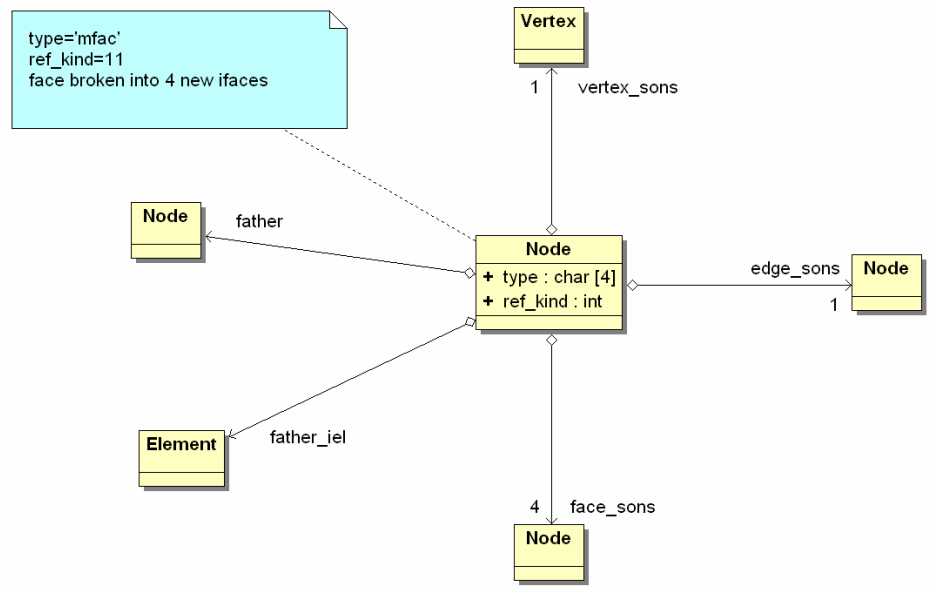


Fig.C.10. Relations for an element face broken into four new faces

When an element edge, represented by `Node` class object with `type='medg'`, is broken, one new element vertex, represented by `Vertex` class object, and two new element edges, represented by `Node` class objects with `type='medg'` are created. The refined `Node` class object stores pointers to newly created objects on `vertex_sons` and `edge_sons` lists. This is illustrated in Figure C.8.

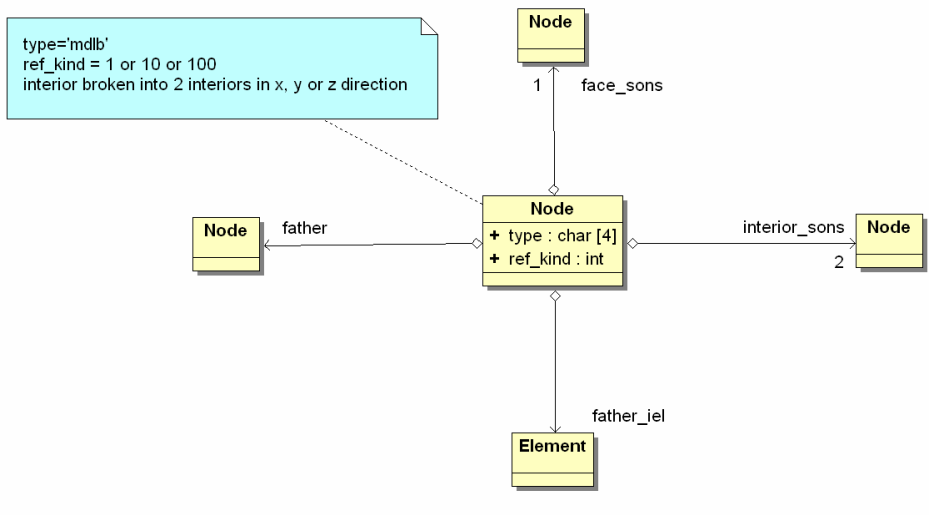


Fig. C.11. Relations for an element interior broken into two new interiors

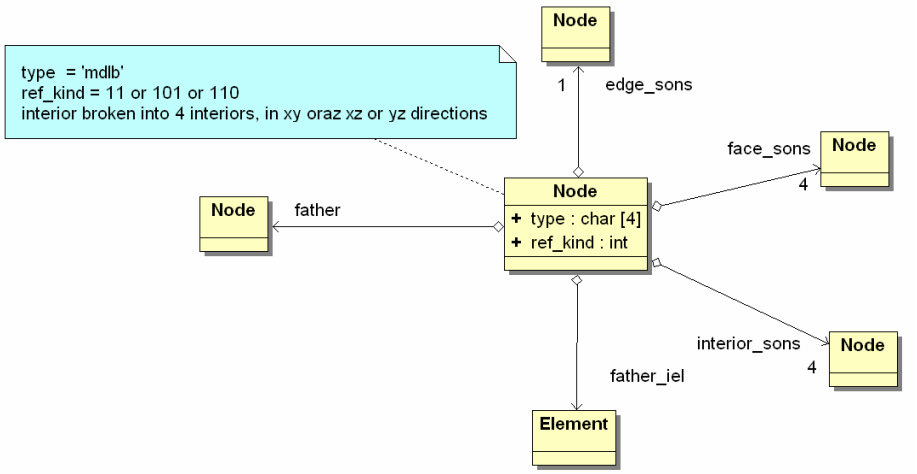


Fig. C.12. Relations for an element interior broken into four new interiors

When an element face, represented by `Node` class object with `type='mfac'`, is broken into two new faces, one new element edge, represented by `Node` class objects with `type='medg'`, and two new element faces, represented by `Node` class objects with `type='mfac'`, are created. The refined `Node` class object stores pointers to newly created objects on `edge_sons` and `face_sons` lists. This is illustrated in Figure C.9.

When an element face, represented by `Node` class object with `type='mfac'`, is broken into four new faces, one new element vertex, represented by `Vertex` class object, four new element edges, represented by `Node` class objects with `type='medg'`, and four new element faces, represented by `Node` class objects with `type='mfac'`, are

created. The refined `Node` class object stores pointers to newly created objects on `vertex_sons`, `edge_sons` and `face_sons` lists. This is illustrated in Figure C.10.

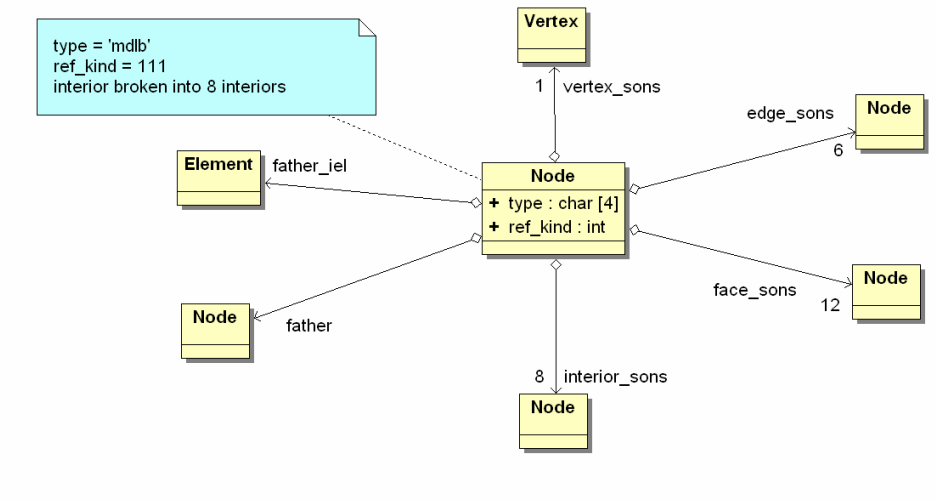


Fig.C.13. Relations for an element interior broken into eight new interiors

When an element interior, represented by `Node` class object with `type='mdlb'`, is broken into two new interiors, one new element face, represented by `Node` class objects with `type='mfac'`, and two new element interiors, represented by `Node` class objects with `type='mdlb'`, are created. The refined `Node` class object stores pointers to newly created objects on `face_sons` and `interior_sons` lists, as presented in Figure C.11.

When an element interior, represented by `Node` class object with `type='mdlb'`, is broken into four new interiors, one new element edge, represented by `Node` class objects with `type='medg'`, four new element faces, represented by `Node` class objects with `type='mfac'`, and four new element interiors, represented by `Node` class objects with `type='mdlb'` ,are created. The refined `Node` class object stores pointers to newly created objects on `edge_sons`, `face_sons` and `interior_sons` lists. This is illustrated in Figure C.12.

When an element interior, represented by `Node` class object with `type='mdlb'`, is broken into eight new interiors, one new element vertex, represented by `Vertex` class object, six new element edge, represented by `Node` class objects with `type='medg'`, twelve new element faces, represented by `Node` class objects with `type='mfac'`, and eight new element interiors, represented by `Node` class objects with `type='mdlb'`, are created. The refined `Node` class object stores pointers to newly created objects on `vertex_sons`, `face edge_sons`, `_sons` and `interior_sons` lists. This is illustrated in Figure C.13.

For more technical details on the three-dimensional implementation *parhp3D90*, see Paszyński, Demkowicz 2006.

Appendix D – Direct solver algorithms

In this section we present the algorithms for three parallel solvers. The first two parallel solvers have been implemented without the graph grammar model presented in this work. The third parallel solver algorithm has been created on the basis of the presented graph grammar model. The parallel algorithms have been interfaced with the developed self-adaptive *hp*-FEM in order to test the scalability of the parallel algorithms.

D.1 Multiple front algorithm

The multiple front solver has been created and implemented by Walsh, Demkowicz 1999. The multiple front solver algorithm is an extension of the Gaussian elimination algorithm, where assembly and elimination are performed together on the so-called frontal sub-matrix of the global matrix (Geng, Oden 2006). The algorithm first performs forward elimination, and then backward substitution (just like the standard Gaussian elimination scheme). It visits each active element, assembles the element contribution into the frontal matrix, and performs elimination of those degrees of freedom that won't be visited in the future (whose final contribution comes from the current element).

The parallel version of the multiple front solver algorithm implemented by Walsh, Demkowicz 1999 is based on the domain decomposition paradigm. The computational domain is subdivided into p sub-domains, as illustrated in Figure D.1. Note that each sub-domain corresponds to a graph of of a single super-task, introduced in Section 2.4. The multiple front solver algorithm starts with the aggregation of element local matrices into a single sub-domain matrix. It is done by browsing elements one by one, and aggregating element local matrices into one sub-domain matrix. The resulting sub-domain local matrices

$$\begin{bmatrix} \mathbf{A}_i & \mathbf{B}_i \\ \mathbf{C}_i & \mathbf{A}_s{}^i \end{bmatrix} \begin{Bmatrix} \mathbf{x}_i \\ \mathbf{x}_s{}^i \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_i \\ \mathbf{b}_s{}^i \end{Bmatrix} \tag{D.1}$$

are generated on each sub-domain, as it is presented on panel (a) in Figure D.1.

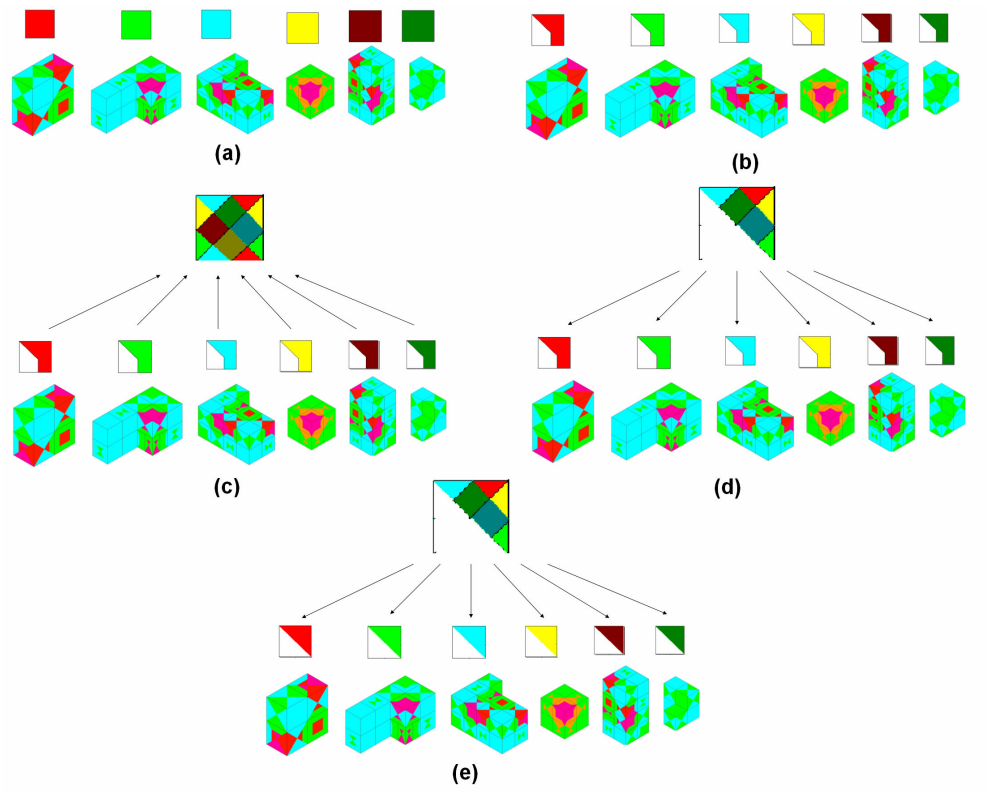


Fig.D.1. Execution of the multiple front solver algorithm on the Fichera problem

The fully assembled degrees of freedom, related to interiors and external boundaries of sub-domains, are eliminated by executing partial forward elimination,

$$\begin{bmatrix} \mathbf{A}_i & \mathbf{B}_i \\ \mathbf{C}_i & \mathbf{A}_s{}^i \end{bmatrix} \begin{Bmatrix} \mathbf{x}_i \\ \mathbf{x}_s{}^i \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_i \\ \mathbf{b}_s{}^i \end{Bmatrix} \rightarrow \begin{bmatrix} \mathbf{U}_i & \mathbf{B}_i^* \\ \mathbf{0} & \mathbf{A}_s{}^{i*} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_i \\ \mathbf{x}_s{}^i \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_i^* \\ \mathbf{b}_s{}^{i*} \end{Bmatrix} \tag{D.2}$$

which is also illustrated on panel (b) in Figure D.1. Next, the Schur complement contributions $\mathbf{A}_s{}^{i*}$ are sent into a separate master processors and collected into a global interface problem

$$\hat{\mathbf{A}}\,\mathbf{x}_s = \hat{\mathbf{b}} \tag{D.3}$$

$$\hat{\mathbf{A}} = \sum_{i=1}^{p} \mathbf{P}_i\, \mathbf{A}_s{}^{i*}\, \mathbf{P}_i{}^T \tag{D.4}$$

$$\hat{\mathbf{b}} = \sum_{i=1}^{p} \mathbf{P}_i\, \mathbf{b}_s{}^{i*}\, \mathbf{P}_i{}^T \tag{D.5}$$

where $\mathbf{P}_i$ stands for the permutation matrices, transforming sub-domain's local ordering of degrees of freedom located on the interface into a global ordering on the interface

$$\mathbf{P}_i : \mathbf{M}\left(n^i_{\text{interface}} \times n^i_{\text{interface}}\right) \rightarrow \mathbf{M}\left(n^i_{\text{interface}} \times n^i_{\text{interface}}\right) \tag{D.6}$$

This is also illustrated on panel (c) in Figure D.1. Note that the system of equations (2.20) corresponds to the so-called Schur complement

$$\hat{\mathbf{A}} = \mathbf{A}_s - \mathbf{C}_I{}^T \mathbf{A}_I{}^{-1} \mathbf{B}_I \tag{D.7}$$

$$\hat{\mathbf{b}} = \mathbf{b}_s - \mathbf{C}_I{}^T \mathbf{A}_I{}^{-1} \mathbf{b}_I \tag{D.8}$$

of the global system

$$\begin{bmatrix} \mathbf{A}_1 & & & \mathbf{B}_1 \\ & \cdots & & \cdots \\ & & \mathbf{A}_p & \mathbf{B}_p \\ \mathbf{C}_1 & \cdots & \mathbf{C}_p & \mathbf{A}_s \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \cdots \\ \mathbf{x}_p \\ \mathbf{x}_s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_1 \\ \cdots \\ \mathbf{b}_p \\ \mathbf{b}_s \end{Bmatrix} \tag{D.9}$$

which can be expressed in a concise form

$$\begin{bmatrix} \mathbf{A}_I & \mathbf{B}_I \\ \mathbf{C}_I{}^T & \mathbf{A}_s \end{bmatrix} \begin{Bmatrix} \mathbf{x}_I \\ \mathbf{x}_s \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_I \\ \mathbf{b}_s \end{Bmatrix} \tag{D.10}$$

In other words, the Schur complement matrix (D.7) and the corresponding right-hand-side vector (D.8) can be easily obtained by executing partial forward eliminations on sub-domains and summing up renumbered sub-matrices $\mathbf{A}_s{}^{i^*}$ and sub-vectors $\mathbf{b}_s{}^{i^*}$, which is expressed in (D.4) and (D.5).

The next step is to solve the global interface problem (D.3) (see panel (d) in Figure D.1). The interface problem solution is broadcasted into sub-domains, and substituted to the right-hand side of the equation, with the Schur complement sub-matrix replaced by the identity matrix

$$\begin{bmatrix} \mathbf{U}_i & \mathbf{B}_i{}^* \\ \mathbf{0} & \mathbf{A}_s{}^{i^*} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_i \\ \mathbf{x}_s{}^i \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_i{}^* \\ \mathbf{b}_s{}^{i^*} \end{Bmatrix} \rightarrow \begin{bmatrix} \mathbf{U}_i & \mathbf{B}_i{}^* \\ \mathbf{0} & 1 \end{bmatrix} \begin{Bmatrix} \mathbf{x}_i \\ \mathbf{x}_s{}^i \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_i{}^* \\ \mathbf{P}_i{}^{-T} \mathbf{x}_s \mathbf{P}_i{}^{-1} \end{Bmatrix} \tag{D.11}$$

The system (D.11) is solved by executing backward substitution on each sub-domain, which is illustrated on panel (e) in Figure D.1.

The multiple front solver has the following disadvantages. To solve the problem distributed into sub-domains, first we need to formulate and solve the interface problem. For multiple sub-domains the interface problem is large, its matrix is dense, and the interface problem solution time may be over 90% of the total solver execution time. Besides, the formulation of the global interface problem is very expensive, since each sub-domain must send its Schur complement to a single master processor. This is a bottleneck for the parallel solver algorithm.

D.2 Multi-level parallel direct solver algorithm

This section describes the multi-level parallel direct solver algorithm designed in order to overcome some disadvantages of the multiple front solver algorithm. The solver works with several sub-domains (corresponding to super-tasks defined in Section 2.4). It is assumed that each sub-domain is assigned to a single processor. The solver follows the following algorithmic steps (see Figures D.2 and D.3):

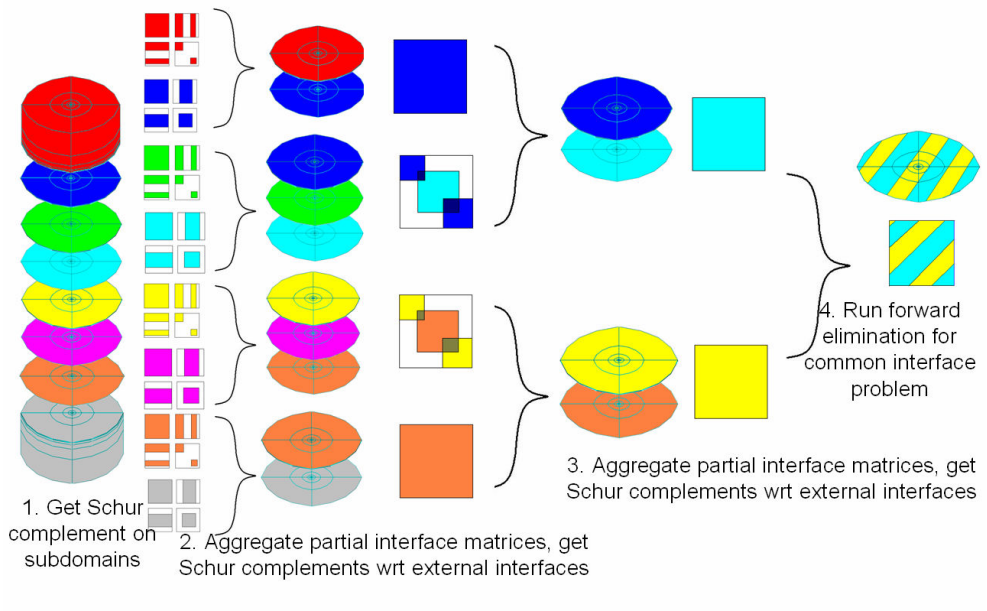


Fig.D.2. Schematic description of multiple partial forward eliminations

a) first, the degrees of freedom from the interior of each sub-domain are eliminated with respect to the degrees of freedom on the interface. This is done using a instance of sequential MUMPS solver on each sub-domain. The MUMPS performs partial forward elimination on each sub-domain to calculate a local contribution to the global interface problem,
b) the sub-domains are joined into pairs, and local contributions to the global interface problem are assembled within these pairs. Note that this procedure requires a new numbering of the interface degrees of freedom within each pair of processors. This is the main technical difficulty in implementing the multi-level scheme. Some interface degrees of freedom are fully assembled and can be eliminated now. Some other interface degrees of freedom are not fully assembled yet and cannot be eliminated,
c) the processors are joined into sets of 4 processors (two pairs of the previous step in each set). All local contributions to the global interface problem are assembled. The degrees of freedom that belong to both pairs are eliminated. Note that the degrees of freedom which belong to other processors, not included in this set, are not eliminated yet,
d) the procedure is repeated recursively until there is only one common interface problem matrix, with all remaining interface degrees of freedom aggregated,
e) this common interface problem matrix is much smaller than the global interface problem matrix, since it corresponds to a common part of the interface shared between two groups of processors. In general, this part of interface is assigned to a cross-section of the domain.

f) the solution of the common interface is broadcast back into two groups of processors. The local solutions are extracted using built maps and are substituted to Schur complement matrices from the corresponding step. The backward substitution is executed to obtain further contributions to the global interface problem solution,
g) the procedure is repeated until we complete the solution of the global interface problem,
h) the last step is to execute the backward substitution on sub-domains, and in this way the global problem is finally solved.

This algorithm works on the sub-domains with an arbitrary shape. First, the solver eliminates the nodes common to both sub-domains in a pair, leaving other interface nodes untouched. After joining two pairs into a new set of four sub-domains, the nodes that belong to both pairs are eliminated, while the nodes that belong to some other sub-domains remain uneliminated. The process is repeated until we reach the root of the elimination tree.

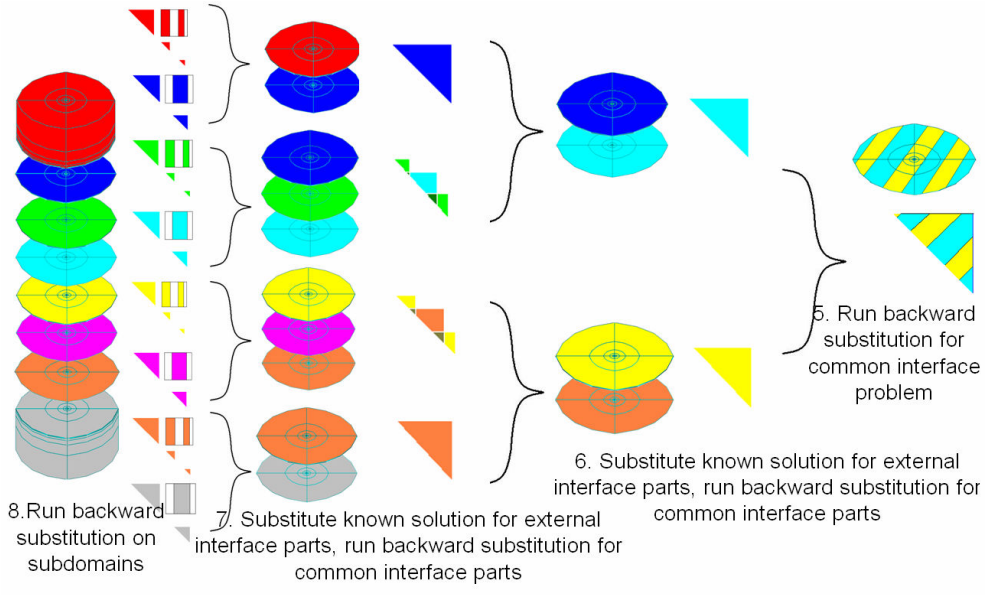


Fig.D.3. Schematic description of multiple partial backward substitutions

The main improvement of the multi-level parallel direct solver algorithm, with respect to the multiple front solver algorithm, is the partition of the interface problem solution into multiple levels, where partial forward eliminations are recursively computed.

D.3 Multi-level multi-frontal direct sub-structuring parallel solver algorithm

The multi-level multi-frontal direct substructuring parallel solver algorithm will be called shortly *the recursive solver*. The recursive solver algorithm follows the graph grammar model of the parallel direct solver introduced in Section 2.1.4.

The parallel recursive solver algorithm has been implemented on the distributed memory architecture. The computational tasks have been defined on the level of initial mesh elements. The groups of initial mesh elements have been agglomerated together to create a single super-task. Each super-task has been assigned (mapped) into a single processor. Thus, the parallel recursive solver algorithm works on three-level elimination trees: the refinement trees that grow from initial mesh elements every time an element is *h* refined; the tree of connectivities of the initial mesh elements; and the tree of connectivities of super-tasks, resulting from the distribution of the computational mesh into multiple sub-domains. An example of the three-level elimination tree is presented in Figure D.4.

The leaves of refinement trees act as the active finite elements. A root of each refinement tree (that is, an initial mesh element) is also a leaf of the connectivity tree for initial mesh elements. The root of this tree (that is an entire super-task) is also a leaf of the connectivity tree for super-tasks. The parallel recursive solver algorithm browses the elimination trees, starting from the level of active elements, through the level of refinement, then through the level of initial mesh elements, and finally through the level of super-tasks. For each tree node, the partial contributions from children nodes are aggregated, the fully assembled degrees of freedom are localized, and the Schur complement of the fully assembled degrees of freedom is computed, with respect to the unassembled (or only partially assembled) degrees of freedom. The procedure is recursively repeated until we reach the root of the connectivity tree for the super-tasks. Finally, the backward substitutions are recursively executed from the root node down to the refinement tree leaves.

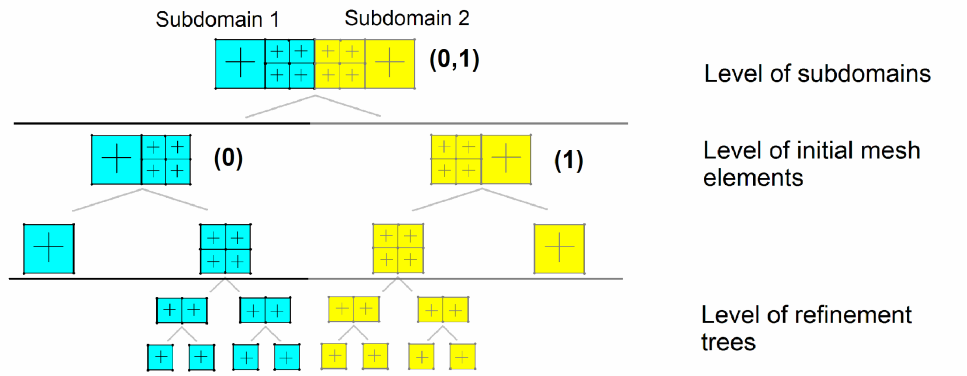


Fig.D.4. Example of three-level elimination tree

We employ the following version of the parallel recursive solver algorithm, with emphasis on the three-level elimination tree.

```
matrix function recursive_solver(tree_node,level)
if level = super-tasks tree and
    only one processor is assigned to tree_node then
  get root_node of initial mesh elements tree assigned
    to current processor
  new_level = initial mesh elements tree
  return recursive_solver(root_node,new_level)
else if level = initial mesh elements tree and
    only one initial mesh element is assigned to tree_node
then
  get root_node of refinement tree assigned
    to initial mesh element
  new_level = refinement tree
  return recursive_solver(root_node,new_level)
else if level = refinement tree and
    this is an active element, a leaf of refinement tree then
  new_matrix = the active element local stiffness matrix
else
  new_matrix = 0
  do for each son_node of tree_node
    if current processor is assigned to son_node then
      matrix_contribution = recursive_solver(son_node,level)
    if current processor is ⌊n/2⌋+1 on the list of n processors
        assigned to tree_node then
      send matrix_contribution to 1st processor
        from the list of processors assigned to tree_node
    endif
    if current processor is 1st processor
          assigned to tree_node then
       merge matrix_contribution into new_matrix
       if there are more than one processor
            assigned to tree_node then
          receive matrix_contribution from ⌊n/2⌋+1 processor
            from the list of n processors
               assigned to tree_node
          merge matrix_contribution into new_matrix
      endif
    endif
  enddo
  decide which degrees of freedom can be eliminated
    from new_matrix
  compute Schur complement
  return Schur complement sub-matrix
end
```

Each node from the elimination tree of super-tasks belongs to several processors. The list of these processors is stored at the node. The leaves of elimination tree of super-tasks are assigned to a single processor. The processor assigned to a leaf of super-tasks tree is also assigned to a branch of the tree growing from the super-task node. This branch includes an initial mesh element tree as well as the refinement trees related to a super-task.

Each tree node has to sone nodes. The algorithm recursively browses the tree nodes assigned to the current processor. If the current processor is not included in the list of processors assigned to a son node, the entire branch is omitted on that processor. When more than one processor (for instance, n processors) are assigned to a tree node, the processors from the first one up to the processor $\left\lfloor \frac{n}{2} \right\rfloor + 1$ from that list process the first branch, and the first processor keeps the matrix contribution resulting from the first branch.

The processors from $\left\lfloor \frac{n}{2} \right\rfloor + 1$ up to the last processor from that list of n processors, process the second branch, and processor $\left\lfloor \frac{n}{2} \right\rfloor + 1$ from that list keeps the matrix contribution resulting from the second branch. In such case, the processor $\left\lfloor \frac{n}{2} \right\rfloor + 1$ from that list sends its matrix contribution to the first processor from the list, and the first processor merges its own matrix and the received matrix.

When the algorithm reaches a tree node representing a single super-task, a tree of initial mesh elements related to the super-task is recovered, and the sequential algorithm is executed on the processor assigned to this super-task. Besides, when the algorithm reaches a single initial mesh element, the refinement tree assigned to the initial mesh element is recovered, and the algorithm follows the refinement tree.

The multi-level multi-frontal direct sub-structuring parallel solver requires BLAS3 routines providing partial forward elimination and backward substitution on dense matrices. The BLAS3 routines need to be executed at each node of the elimination tree. The sequential MUMPS solver is used to compute Schur complements at tree nodes.

The MUMPS solver returns to the user the Schur complement sub-matrix, but it does not return the right-hand-side vector. Thus, in the current version of the parallel recursive solver, the right-hand side is stored as an additional column of the matrix, to enforce the inclusion of the right-hand side into the Schur complement procedure, so we assume that the matrix is non-symmetric. The Schur complement procedure can be understood as partial forward elimination that is stopped before processing unassembled degrees of freedom.

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{b}_2 \\ \mathbf{0} & \mathbf{0} & 1 \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} & \hat{\mathbf{b}}_1 \\ \mathbf{0} & \hat{\mathbf{A}}_{22} & \hat{\mathbf{b}}_2 \\ \mathbf{0} & \mathbf{0} & 1 \end{bmatrix} \tag{D.12}$$

The MUMPS solver returns to the user the $\hat{\mathbf{A}}_{22}$ and $\hat{\mathbf{b}}_{22}$ parts of (D.12), together with the last row of the matrix. The obtained Schur complement, $\hat{\mathbf{A}}_{22}$ and $\hat{\mathbf{b}}_{22}$, is employed to build the system of equations for the father element node. Every time the Schur complement at any tree node is obtained, the current sequential instance of the solver

MUMPS is turned off to reduce the memory usage. The forward elimination is followed by the recursive backward substitution which follows the reversed pattern - it starts from the root of the elimination tree, and goes down to the leaves of the tree. In order to perform partial backward substitution, the following system of equations must be reconstructed:

$$\begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{Bmatrix} = \begin{Bmatrix} \hat{\mathbf{b}}_1 \\ \hat{\mathbf{x}}_2 \end{Bmatrix} \tag{D.13}$$

Since the MUMPS does not return $\mathbf{U}_{11}$, $\mathbf{U}_{12}$ and $\hat{\mathbf{b}}_1$, they are recomputed during backward substitution, by performing full forward elimination on the tree node sub-matrix. It is done by executing the full forward elimination on the original matrix, with $\mathbf{A}_{22}$ sub-matrix replaced by the identity matrix, and the bottom part of the right-hand-side $\mathbf{b}_2$ replaced by the solution $\hat{\mathbf{x}}_2$ obtained from the father node:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{Bmatrix} = \begin{Bmatrix} \mathbf{b}_1 \\ \hat{\mathbf{x}}_2 \end{Bmatrix} \rightarrow \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{Bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{Bmatrix} = \begin{Bmatrix} \hat{\mathbf{b}}_1 \\ \hat{\mathbf{x}}_2 \end{Bmatrix} \tag{D.14}$$

This corresponds to the definition of the Dirichlet boundary conditions for the local interface. Note that this full forward elimination is much faster than obtaining Schur complement, since the new system is sparser. Finally, the backward substitution is executed on (D.14), and a new contribution to the solution $\mathbf{x}_1$ is obtained. This trick allows us to minimize the memory usage, since the partial Schur complements are not stored in the memory, but recomputed.

D.4 Multi-level multi-frontal direct sub-structuring parallel solver algorithm with reutilization of partial LU factorizations

The LU factorizations computed by the parallel recursive solver algorithm at each node of the elimination tree can be stored at tree nodes for further reutilization. Every time the mesh is refined, the LU factorizations from the unrefined parts of the mesh can be reutilized. There is a need to recompute LU factorization for the refined elements, as well as for the whole path from any refined leaf up to the root of the elimination tree. An example of the reutilization of partial LU factorizations after performing two local refinements is presented in Figure D.5.

The extension of the parallel recursive solver algorithm using the reutilization of partial LU factorization can be expressed by the following recursive procedure. The modification of the original procedure is denoted by grey colour.

```
matrix function recursive_solver(tree_node,level)
if tree_node has been refined then
  if tree_node has no son nodes then
    eliminate leaf element stiffness matrix internal nodes
    return Schur complement sub-matrix
  else if tree_node has son nodes then
```

```
    new_matrix = zero
    do for each son_node of tree_node
      if current processor is assigned to son_node then
        matrix_contribution=recursive_solver(son_node,level)
        if current processor is ⌊n/2⌋+1 on the list of
            n processors assigned to tree_node then
          send matrix_contribution to 1st processor
            from the list of processors assigned to tree_node
        else if current processor is 1st processor
            assigned to tree_node then
          merge matrix_contribution into new_matrix
          if there are more than one processor
              assigned to tree_node then
            receive matrix_contribution from ⌊n/2⌋+1 processor
              from the list of n processors
              assigned to tree_node
            merge matrix_contribution into new_matrix
          endif
        else
          return zero
      endif
    enddo
    decide which degrees of freedom can be eliminated from
new_matrix
    compute Schur complement
    store Schur complement sub-matrix at tree_node
    return Schur complement sub-matrix
  endif
else
  get the Schur complement sub-matrix from tree_node
endif
```

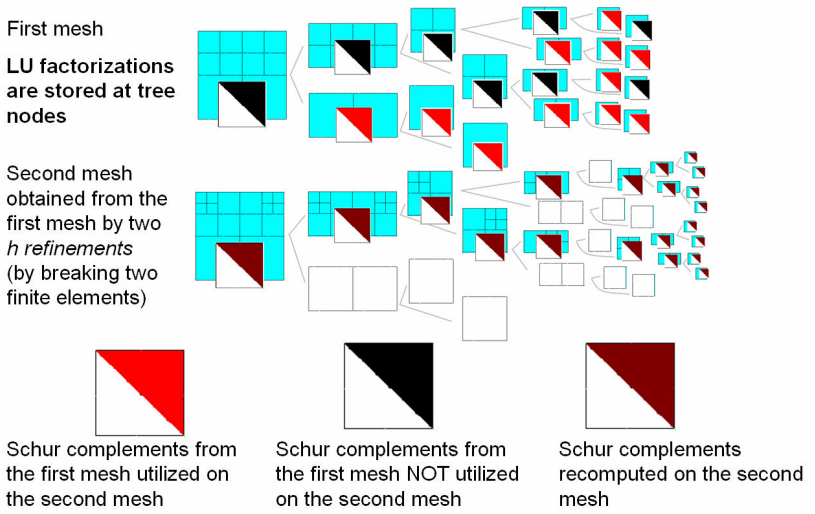


Fig.D.5. The problem is solved for the first mesh. All LU factorizations (black and red) are computed. Then, the mesh is refined, and the problem is solved again. Red LU factorizations are reutilized from the previous mesh, but all brown LU factorizations must be recomputed. Black LU factorizations from the previous mesh are deleted