

PARALLEL, FULLY AUTOMATIC *hp*-ADAPTIVE 2D FINITE ELEMENT PACKAGE

M. Paszyński^{(1),(2)}, J.Kurtz⁽¹⁾, L.Demkowicz⁽¹⁾

(1) Institute for Computational Engineering and Sciences

The University of Texas at Austin

(2) AGH University of Science and Technology,

Department of Computer Methods in Metallurgy, Cracow, Poland

Abstract

This paper presents a description of par2Dhp - a 2D, parallel fully automatic *hp*-adaptive finite element code. The parallel implementation is an extension of the sequential code 2Dhp90, which generates fully automatic *hp*-approximations for solutions of various boundary value problems. The presented work addresses parallelization of each stage of the automatic *hp*-adaptive algorithm, including decomposition of the computational domain, load balancing and data redistribution, a parallel frontal solver, and algorithms for parallel mesh refinement and mesh reconciliation. The application was written in Fortran 90 and MPI, and the load balancing is done through an interface with the Zoltan library. Numerical results are presented for the model L-shape domain problem, and a highly anisotropic heat conduction (battery) problem from Sandia National Laboratories.

Key words: Automatic *hp*-adaptivity, Finite Element Method, Parallel algorithms, High performance computing

Acknowledgment

The work of the third author has been supported by Air Force under Contract F49620-98-1-0255. The computations reported in this work were done through the National Science Foundation's National Partnership for Advanced Computational Infrastructure. The authors are greatly indebted to Kent Milfeld and Timothy Walsh for numerous discussions on the subject.

1 Introduction

***hp* Finite Elements.** In the most flexible version of the Finite Element Method, elements may have a locally varying element size h , and polynomial order of approximation p . A rationale for using *hp* elements may have different origins. In singularly perturbed problems, e.g. elastic thin-walled structures, elements with $p \geq 5$ allow for avoiding locking [22]; for wave propagation problems, high order elements help to minimize the dispersion (pollution) error [1]. At the same time, small elements are necessary to capture geometrical details present in real world problems. Only the *hp* elements allow for combining small elements of lower order with large elements of high order in the same mesh. The main motivation of this work, however, comes from using *hp*-adaptive discretizations where the distribution of element size h and order p is optimized to deliver the smallest possible problem size (number of *degrees-of-freedom* (d.o.f.)) meeting a prescribed error tolerance criterion. After over a decade of research, a fully automatic, problem independent, *hp*-strategy has been constructed [7, 8, 9] that delivers a sequence of optimally refined *hp* meshes and *exponential convergence* - the error decreases exponentially fast, both in terms of problem size and actual CPU time.

The presented work is motivated with solutions of large Electromagnetic scattering problems involving geometrical singularities (diffraction on edges and points) and scattering from resonating cavities, see e.g. [4]. Resolution of geometric singularities in 2D requires many levels of *h*-refinements (the ratio of the smallest to largest elements may be 10^{-6}). Additionally, resolution of boundary layers (skin effects in EM computations) and edge singularities in 3D calls for anisotropic mesh refinements.

Parallelization of adaptive methods. Parallelization of adaptive codes is difficult. Most implementations for distributed memory parallel computers are based on Domain Decomposition (DD) concepts. The domain of interest is partitioned into sub-domains with each of the sub-domains delegated to a single processor. To maintain scalability, only local information about the mesh may be stored in each processor's memory. The situation is different for Boundary Element (BE) computations where high order discretizations result in a "small" but dense stiffness matrix, and a copy of a mesh supporting data structure can be stored on each processor; the main effort then is directed at parallelizing the integration of element matrices and a linear solver [25].

Among major undertakings to develop a general infrastructure to support DD based parallelization of PDE solvers, one has to list first of all the Sierra Environment [23, 10, 11, 12] developed by Sandia National Labs. Designed to support *h*-adaptivity, the Sierra framework has been used to parallelize several FE codes developed at Sandia [12]. The Sierra environment allows for an arbitrary domain partitioning of a current mesh but it does not support anisotropic mesh refinements.

General environments to support local mesh refinements have been developed in the groups of Joe Saltz, see e.g. [15], Scott Baden, see e.g. [17] and, more recently, Kathy Yelick [27, 20]. None

of them, unfortunately, turned out to be applicable directly to our *hp* codes.

The only parallel *hp* codes that we are aware of, have been developed by Joe Flaherty at RPI [21] in the context of Discontinuous Galerkin (DG) methods, and Abani Patra at UNY at Buffalo [16, 3].

In our first attempt to develop a parallel *hp* code, we managed to develop only a parallel two-grid solver for *hp* meshes and 2D Maxwell equations [2], but we failed miserably with the parallelization of the actual code. This failure motivated us to develop new data structures better suited for parallelization [5, 6].

The presented concept of parallelization is based on the assumption that in the DD based parallel code, each of the processors is executing the sequential code with only minimal upgrades added (to support communication between sub-domains). In order to maintain the load balance during refinements, the mesh has to be frequently repartitioned, and the data structure arrays supporting the new sub-domains, must be generated. In simple terms - elements and nodes are assigned new numbers. This requires reproducing horizontal information, like element-to-nodes connectivities with new numbers, a nightmare for adapted meshes with hanging nodes. To avoid the problem, the new *hp* data structures include the horizontal information *only* for the initial mesh elements and nodes. All other information on elements and nodes resulting from *h* or *p*-refinements is reproduced from *nodal trees* relating parent and children nodes only *vertically*. Contrary to the horizontal information, the trees are much easier to regenerate.

The use of trees has forced us to partition the domain using the initial mesh elements only. In this context, the complexity of the DD step and data regeneration is only slightly higher than for static FE meshes with classical data structures.

Wave propagation problems. The main motivation for developing a parallel version of the *hp* methodology is the solution of large wave propagation problems. The very first issue in addressing this class of problems is the resolution of the wave form of the solution. High order methods turned out to be the most powerful tool in minimizing the *dispersion error*, and other preasymptotic convergence range artifacts classified in the FE community under the common name *pollution error*. As mentioned in the beginning, resolution of complex geometries requires the use of very nonuniform meshes. Employing the same, high order of approximation for both small and large elements is very inefficient and unnecessary. Only the *hp* elements allow for adjusting the element order to its size, and satisfaction of dispersion error criteria in an efficient way [1].

Another issue is rooted in the Galerkin method itself which, for time-harmonic wave propagation problems, becomes stable only asymptotically. The actual *approximation error* becomes equal to the *best approximation error* characterizing approximation properties of a mesh, only for sufficiently fine meshes. Entering the asymptotic region of convergence depends upon wave number *k*, and occurs later with larger *k*, i.e. the “meeting point” of the approximation and best approximation error occurs at a smaller level of the best approximation error with growing wave number *k*. Once

in the asymptotic convergence region, the Galerkin method becomes stable, and further mesh optimization can then be used to capture geometric singularities or near resonance behavior. The *hp* strategy discussed in this paper is based on an interaction between a coarse and fine mesh. The fine mesh is obtained from the coarse mesh through a global *hp*-refinement and, in 3D, results in 30 to 40 times larger number of d.o.f. One of the main points about our strategy is that *only the fine mesh must be in the asymptotic convergence region*; the solution on the coarse mesh may be completely meaningless (hundreds of percent of error) when the mesh optimization process begins, compare [8, 9]. The catch 22 though is that in this situation, the main engine of making the strategy possible - our two-grid solver used to solve the fine grid problem [18], fails to converge. It is only after the coarse grid enters the asymptotic convergence range, that the two-grid iterations become effective. On the practical side, this implies that in the beginning of the mesh optimization process we are forced to use the direct solver for both the coarse and fine mesh solves, and only later we can switch to the orders of magnitude more efficient two-grid iterations. This motivates the development of the parallel version of the algorithm, based on a direct solve for both coarse and fine grids, presented in this contribution.

Challenges. Besides the DD and data structure regeneration issues, the parallelization of our *hp* codes involved two more main challenges - parallelization of the linear solvers, and mesh reconciliation. Even if the decision on optimal *hp*-refinements is made fully in parallel (which may be impossible, as we will discuss later), the execution of optimal *hp*-refinements on the sub-domains results in meshes that are globally illegal. The *hp* meshes supported by our codes satisfy two main regularity assumptions - they must be *1-irregular* (no multiply constrained nodes are allowed) and they must satisfy the *minimum rule* (order for edges is set to the minimum of orders for the neighboring elements). Enforcing both regularity assumptions *a-posteriori*, is commonly called the *mesh reconciliation*, and presents one of major challenges for the parallel implementation of the *hp* methods discussed in this paper.

2 Flowchart for the parallel *hp* algorithm

The input data for the parallel version of the *hp* code includes the following items.

- *Geometry* of the domain is prescribed within the *Geometry Modeling Package* [26]. A 2D domain is described as a union of curvilinear GMP “rectangles” forming a FE-like mesh. Each of the rectangles is identified with the image of reference square $(0, 1)^2$ through an explicit or implicit parametrization supported by GMP. For adjacent rectangles, the corresponding parameterization are compatible with each other, i.e. they yield an identical parametrization for the common edge.
- *Data for (an initial) mesh generation.* Each of the reference squares is covered with a uniform mesh of $m_1 \times m_2$ elements of uniform order $\mathbf{p} = (p_1, p_2)$. The number of horizontal m_1 and

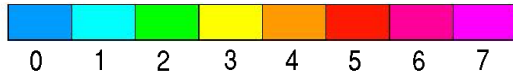


Figure 1: Orders of approximation over elements

vertical m_2 subdivisions must be compatible with the adjacent rectangles, so that the resulting initial mesh is regular. The *minimum rule* is used to determine the order of approximation for edges shared by two rectangles.

- *Material and load data.* It is assumed that the coefficients of the differential equation and boundary operators, as well as source term f and boundary data g , are constant within each GMP rectangle. If the exact (manufactured) solution is known, the load data is determined from the differential equation and boundary conditions.
- *Additional data for post-processing:* definition of elements sets, sections etc., all done within GMP logic.

The hp algorithm produces a sequence of *coarse* and corresponding *fine* meshes that deliver exponential convergence. The initial mesh is the coarse mesh for the first iteration. The algorithm performs the following iterations, see Fig.4. The coarse mesh is globally refined in both element size h (each quad element is divided into four element sons in 2D) and order of approximation $\mathbf{p}=(p_1, p_2)$ (raised uniformly by one), to obtain the fine mesh, see Fig.2. Load balancing using the Zoltan library [28] is performed at the beginning of each iteration step. The problem is solved twice, once on the current coarse mesh, and once on a fine mesh. The FE error on the coarse grid is estimated by simply evaluating (norm of) the difference between the coarse grid and fine grid solutions. If error exceeds a preset tolerance, the coarse grid is hp -refined in an optimal way. The optimal hp -refinements (see Fig.2) are obtained by minimizing the coarse grid *projection based interpolation error* of the fine grid.. Both coarse and fine grid solution are stored in element fashion, using elements local coordinates systems. This localization principle allows us to compute projections of global interpolants locally, over particular elements. The optimal mesh obtained in the current step constitutes the coarse mesh for the next step, see Fig.3. The iterations are stopped when the estimated error is sufficiently small.

3 Parallel data structures and load balancing

Since the parallel code described in this paper was developed as an extension of the serial code 2Dhp90, we will begin with a short description of the data structures in the serial code, and discuss how they have been modified in the parallel code. The data structures are organized into three layers with geometry representation at the bottom, initial mesh elements in the middle, and refinement trees on top, see Fig.5. In general, geometry and initial mesh elements are static, while

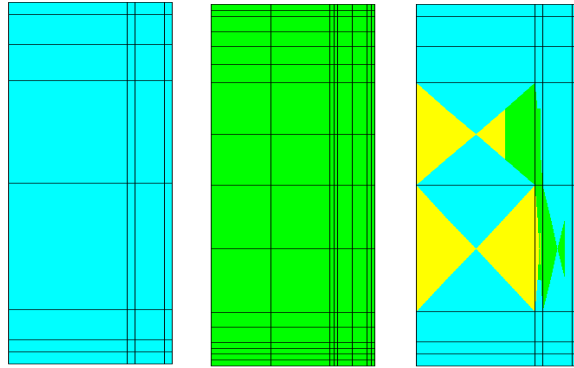


Figure 2: Left picture: Initial mesh = coarse mesh. Middle picture: Fine mesh in the first iteration. Right picture: Optimal mesh after the first iteration

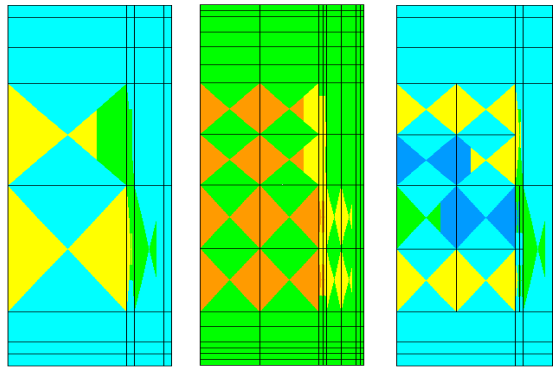


Figure 3: Left picture: Optimal mesh after the first iteration = coarse mesh for the second iteration. Middle picture: Fine mesh in the second iteration. Right picture: Optimal mesh after the second iteration

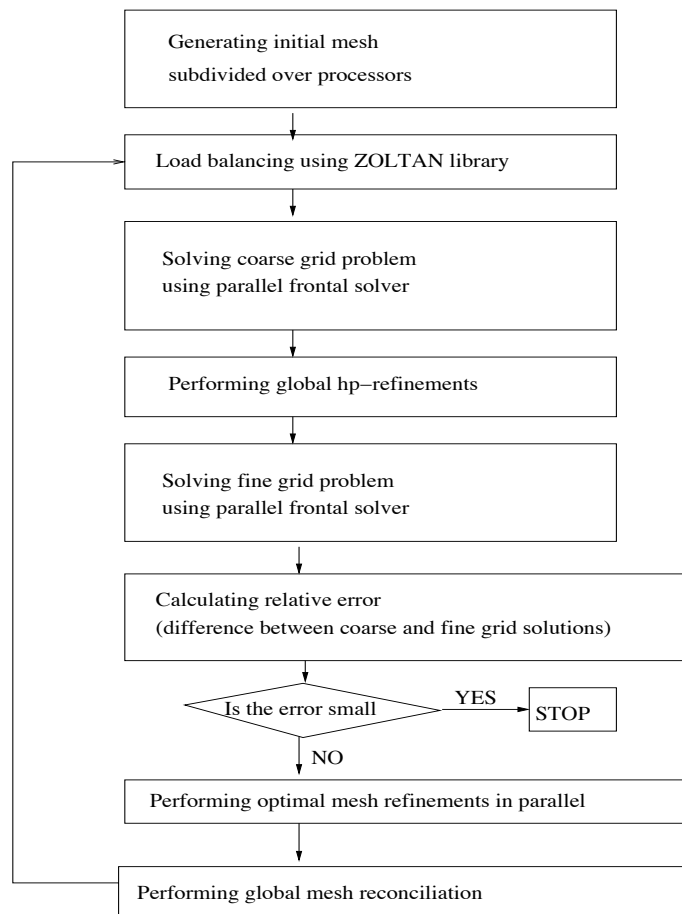


Figure 4: Flow chart for parallel fully automatic hp -adaptive finite element algorithm

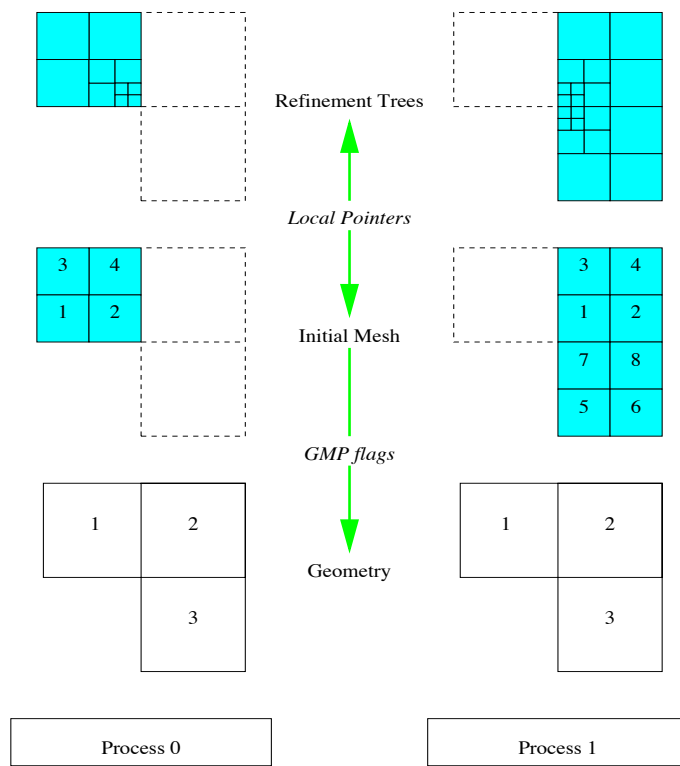


Figure 5: Global geometry, divided initial mesh, and refinements

refinement trees are grown dynamically. In the parallel version, the geometry is global, while initial mesh elements and refinement trees are local. This will be clarified in the following discussion.

The lowest level, geometry representation, is provided by the Geometrical Modeling Package (GMP, [26]). The geometry is represented as a collection of points, curves, and rectangular figures. In the parallel version, geometry is global, i.e. each process constructs an identical copy of the geometry for the entire computational domain.

At the next initial mesh level, a truly parallel execution begins. Each process takes responsibility for a part of the global geometry, and generates initial mesh elements only for that part. For each initial mesh element we store the following data,

nodes - local pointers to four vertex nodes, four edge nodes, and a single middle node (to be described below),

orient - orientations for the four edge nodes,

bcond - boundary condition flags for the four edges,

neig - local pointers to the four neighboring initial mesh elements, or zero if a particular neighbor resides on another process or doesn't exist because the element is adjacent to the boundary,

geom_interf - an interface to GMP that identifies the geometric figure in which the element resides and its local element number within that figure.

Because the global geometry is known to each process, the **geom_interf** flag is a unique global identifier for each element, and provides the basis for communication between processes. For example, when we need to find all the neighbors for an element, we can first look at the local pointers **neig** to identify all neighbors that reside on the same process. If one of these is zero we can jump down one layer into the geometry data and either discover that the element is adjacent to the boundary, or that the neighbor simply resides on another process, in which case we can identify it by using its **geom_interf** flag.

With initial mesh elements thus constructed, each process initializes the final layer, refinement trees, by adding vertex, mid-edge, and mid-quad nodes to the mesh. For vertices we store the following information,

bcond - a boundary condition flag

father - for initial mesh vertices, this is a pointer to one of the initial mesh elements containing the vertex; for vertices resulting (later) from the refinement of mid-edge or mid-quad nodes, this is a pointer to the parent node that was refined,

coord - geometric coordinates,

zdofs - solution degrees of freedom.

For mid-edge and mid-quad nodes alike, we store,

type - identifies the node as mid-edge or mid-quad,

order - polynomial order of approximation (possibly anisotropic for mid-quad nodes),

bcond - a boundary condition flag,

ref_kind - h -refinement flag,

father - for initial mesh nodes, this is a pointer to one of the initial mesh elements containing the node; for nodes resulting (later) from the refinement of mid-edge or mid-quad nodes, this is a pointer to the parent node that was refined,

sons - for nodes that have been h -refined, we store pointers to the nodes generated by that refinement; a mid-edge node may have zero or three sons (two mid-edge nodes and a vertex), and a mid-quad node may have zero, three (two mid-quads and a mid-edge), or nine (four mid-quads, four mid-edges, and a vertex) sons,

coord - geometry degrees of freedom,

zdofs - solution degrees of freedom.

Notice that nodes do not store horizontal connectivities, i.e. a mid-quad node does not store pointers to the adjacent vertices or mid-edge nodes. Only father-son relationships are stored, and efficient algorithms are used to dynamically reconstruct horizontal connectivities in the refined mesh.

While the initial mesh is distributed evenly across all processes, adaptive refinements lead to load imbalance and we must obtain a new domain decomposition to rebalance the load. The most general approach would allow for any partitioning of the current active mesh. However, this would be an enormous departure from the serial code. Instead, we only allow a partitioning along lines in the initial mesh, i.e. each initial mesh element, along with its refinement trees, is treated as a unit.

Our data decompositions are obtained through a simple interface with the Zoltan library [28]. For each initial mesh element, we provide Zoltan with the corresponding global identifier (the `geom_intf` flag), spatial coordinates of the element centroid, and a weight representing the computational cost for that element and its refinement tree (currently we use a weight that scales with the order of approximation p like the cost for element stiffness matrix integration, i.e. $O(p^4)$ in 2D).

Having gathered this information from each process, Zoltan employs one of its load balancing algorithms providing a new decomposition of the domain with the weights equitably distributed between sub-domains. Currently, we are using a Hilbert space filling curve-based algorithm, based

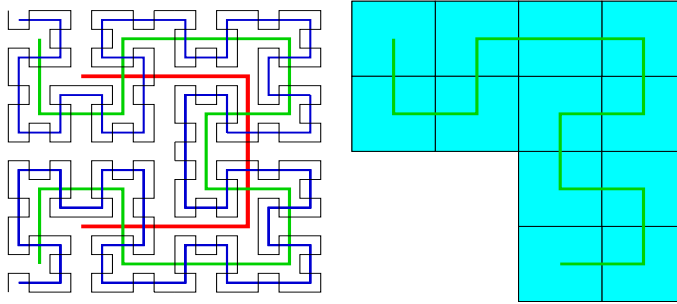


Figure 6: Refinement levels for a Hilbert space filling curve, and the one that passes through initial mesh elements

on a non-intersecting curve passing through the element centroids, see Fig. 6. This curve is cut into segments (one for each processor) in such a way that the computational weights are equitably distributed between segments. Because the initial mesh is static, this curve never changes; only the weights and therefore the cuts change as the mesh is refined. The new data decomposition is returned to each process in the form of a list of elements to be exported (and to which process) and a list of elements to be imported (and from which process). We perform the necessary data migration using the Message Passing Interface (MPI).

After data migration each process has to reassemble those elements it did not export to other processes, and those elements it imported from other processes, into a single data structure. The elements are sorted based on their `geom_intf` flags, first by geometric figure numbers, and then by local element numbers within a figure. The new data structure is then assembled, element by element. This process is actually quite simple since the refinement trees do not store horizontal connectivities. We only need to reconstruct the horizontal connectivities (i.e. pointers to neighbors) on the initial mesh level and the refinement trees are assimilated automatically!

4 Parallel frontal solver

Both the coarse mesh and fine mesh problems are solved using a parallel frontal solver (in the future a parallel version of a two grid solver will be developed for solving the fine mesh problem [18]). The frontal solver 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 [13].

The solver performs forward elimination first, and then backward substitution (just like the standard Gaussian elimination scheme). It visits each 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).

We have implemented a parallel version of the solver, using the domain decomposition approach [24]. The computational domain is subdivided into d sub-domains and d.o.f. are enumerated in the

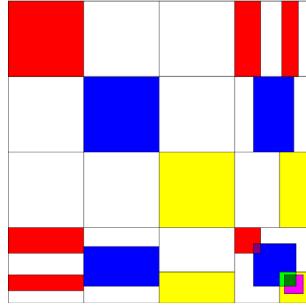


Figure 7: Global matrix after the domain decomposition

following way,

- All internal d.o.f. from 1-st sub-domain,
- All internal d.o.f. from 2-nd sub-domain,
- ...
- All internal d.o.f. from d -th sub-domain,
- All d.o.f. from the global interface.

The structure of the global stiffness matrix is (see also Fig.7),

$$\begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \dots & & \dots \\ & & & A_d & B_d \\ C_1 & C_2 & \dots & C_d & A_s \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \\ x_s \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_d \\ b_s \end{bmatrix} \quad (4.1)$$

or equivalently,

$$\begin{bmatrix} A_I & B^T \\ C & A_s \end{bmatrix} \begin{bmatrix} x_I \\ x_s \end{bmatrix} = \begin{bmatrix} b_I \\ b_s \end{bmatrix} \quad (4.2)$$

The Schur complement of system (4.2) is

$$\hat{A}x_s = \hat{b} \quad (4.3)$$

$$\hat{A} = A_s - CA_I^{-1}B^T \quad (4.4)$$

$$\hat{b} = b_s - CA_I^{-1}b_I \quad (4.5)$$

$$A_Ix_I + B^Tx_s = b_I \quad (4.6)$$

The resulting global matrix has sparse form, except for the part connected with d.o.f. from the global interface. Upon the static condensation of the sub-domains' interior d.o.f. (computation of

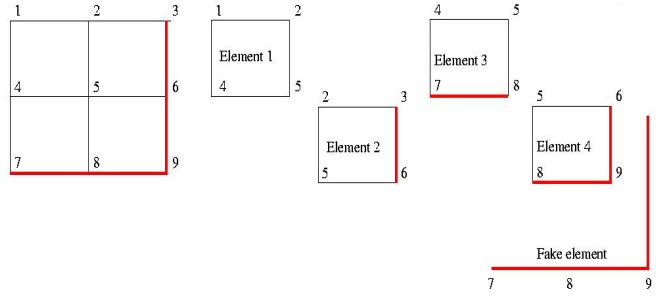


Figure 8: Finite elements together with the fake element over a sub-domain, processed by the parallel frontal solver

the Schur complements), we formulate and solve the global wire-frame problem. Finally, we solve a local interior problem on each sub-domain in parallel, using the wire-frame problem solution.

A practical implementation is achieved by adding a so-called “fake” element [24] at the end of the list of elements browsed by the frontal solver on each sub-domain. The “fake” element contains all d.o.f. from that sub-domain that contribute to the global interface problem (see Fig.8), and tricks the frontal solver into leaving the associated d.o.f. in the front. We perform the following operations,

- Run the forward elimination stage of the frontal solver in parallel with fake elements on each sub-domain. The final front matrices obtained on each sub-domain are precisely the sub-domain contributions to the global Schur complement.
- Accumulate obtained matrices and solve the wire-frame problem (sequential).
- Run the backward substitution stage over each sub-domain in parallel.

Going into details, let n = total number of d.o.f.,

$n_{\text{internal}}^{(i)}$ = number of interior d.o.f. of sub-domain i .

$n_{\text{interface}}^{(i)}$ = number of interface d.o.f. of sub-domain i .

$n_{\text{interface}}$ = global number of interface d.o.f..

We define the following inclusions:

$$P_i : M \left(n_{\text{internal}}^{(i)} + n_{\text{interface}}^{(i)} \times n_{\text{internal}}^{(i)} + n_{\text{interface}}^{(i)} \right) \rightarrow M (n \times n) \quad (4.7)$$

$$P_{bi} : M \left(n_{\text{interface}}^{(i)} \times n_{\text{internal}}^{(i)} \right) \rightarrow \left(n_{\text{interface}}^{(i)} + n_{\text{interface}}^{(i)} \right) \quad (4.8)$$

$$P_{ib} : M \left(n_{\text{internal}}^{(i)} \times n_{\text{interface}}^{(i)} \right) \rightarrow \left(n_{\text{interface}}^{(i)} + n_{\text{interface}}^{(i)} \right) \quad (4.9)$$

The global matrix can then be represented by the following decomposition

$$A = \sum_{i=1}^p P_i A^{(i)} P_i^T \quad (4.10)$$

where

$$P_i A^{(i)} P_i^T \begin{bmatrix} 0 & & & & \\ & \ddots & & & \\ & & A_i & & P_{ib} B_i P_{ib}^T \\ & & & \ddots & \\ & & P_{bi} C_i P_{bi}^T & & P_{bb} A_s^i P_{bb}^T \end{bmatrix} \quad (4.11)$$

The global matrix can be stored in a distributed manner, over d processors (see Fig.9).

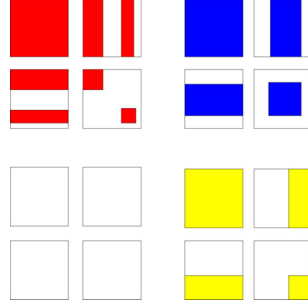


Figure 9: Distribution of the global matrix into processors

The forward elimination stage of the frontal solver can be run over each sub-domain to assemble the system,

$$\begin{bmatrix} A_i & B_i \\ C_i & A_s^i \end{bmatrix} \begin{bmatrix} x_i \\ x_s^i \end{bmatrix} = \begin{bmatrix} b_i \\ b_s^i \end{bmatrix} \quad (4.12)$$

where A_i , B_i , C_i and b_i are fully assembled, and A_s^i is partially assembled.

The frontal solver eliminates all internal d.o.f., but keeps all interface d.o.f. After processing all elements except the fake element, the front is A_s^{i*} , see Fig.10.

The global wire-frame interface matrices are then computed, comp. (4.13) and (4.14), see Fig.10.

$$\hat{A} = \sum_{i=1}^p P A_s^{(i)*} P^T \quad (4.13)$$

$$\hat{b} = \sum_{i=1}^p P b_s^{(i)*} P^T \quad (4.14)$$

The wire-frame problem is currently formulated and solved on a separate processor sequentially (in the future, the problem will be solved using the PLAPACK [19] parallel solver). Finally, the backward substitution stage runs on each sub-domain separately, see Fig.11.

5 The sequential hp -algorithm

In this section we discuss the 2D version of our hp mesh optimization algorithm. The logic of the algorithm is the same for both elliptic and Maxwell problems, only the projections used are

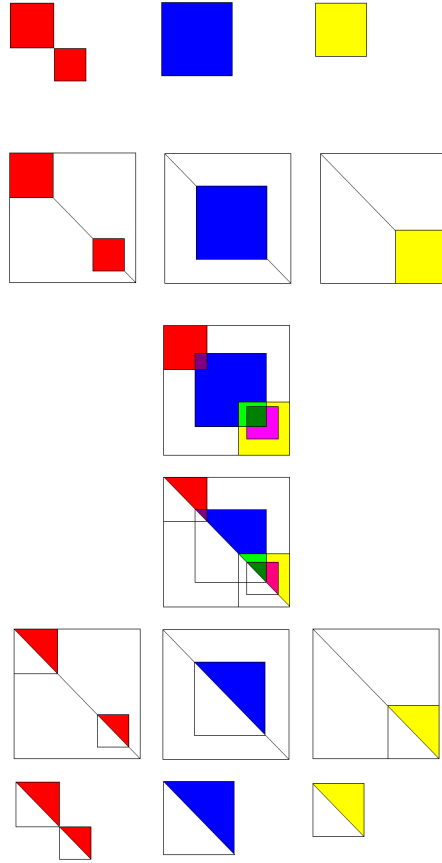


Figure 10: First row: Front matrices A_s^{i*} in contributing slave processes. Second row: $P_{bb}A_s^{i*}P_{bb}^T$ distribution of the interface problem part of the global matrix into processors. Third row: The interface problem $\hat{A} = \sum_{i=1}^p PA_s^{(i)*}P^T$ formulated on a master process. Fourth row: Global interface problem after forward elimination stage. Fifth row: Distribution of the upper triangular form of the interface problem into processors. Sixth row: Upper triangular forms of front matrices on slave processes

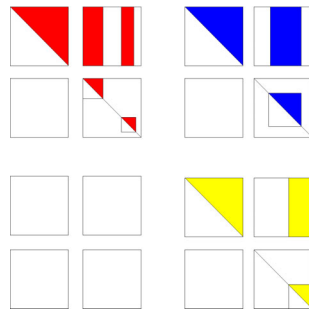


Figure 11: Distributed global matrix before the parallel backward substitution

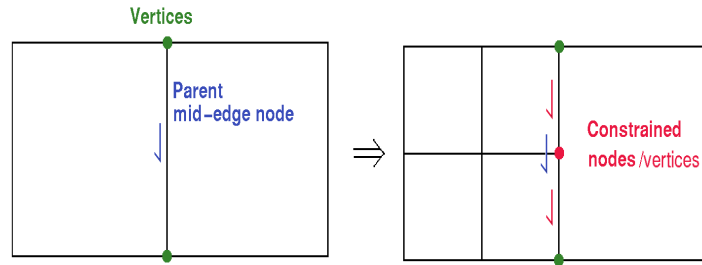


Figure 12: Isotropic refinement of a finite element. mid-edge nodes and one common vertex of two newly created elements are called *constrained nodes*

different.

Mesh regularity

- A mesh, consisting of quads, is called *regular*, if the intersection of any two elements in the mesh is either empty, reduces to a single vertex, or consists of a *whole* common edge shared by the elements.
- An *isotropic h-refinement* occurs (in 2D) when an existing element is broken into four elements sons.
- An *anisotropic h-refinement* occurs when an existing element is broken into two elements sons, in either the horizontal or vertical direction.
- An edge consists of two vertices and one mid-edge node. When an element is refined, in isotropic or anisotropic manner, some of its edges are broken.
- During the isotropic *h-refinement*, an element is broken into four smaller sons, as presented in Fig.12. As a result of the refinement, a *big* element, shown in Fig.12 on the right, has two smaller *neighbors*. In such a situation, two mid-edge nodes of smaller elements and one common vertex of both smaller elements remain constrained with no new nodes generated in global data structure arrays *NODES* and *NVERS*. The d.o.f. of constrained nodes are *forced* to match appropriate linear combinations of *parent nodes* d.o.f in order to enforce the continuity of the approximation (hence the name) but they do not exist in the data structure arrays.
- A mesh with constrained nodes is called an *irregular mesh*. It may happen that smaller elements need to be broken once again, as it is presented in Fig.13. In such a case newly created constrained nodes are said to be *multiply constrained*.

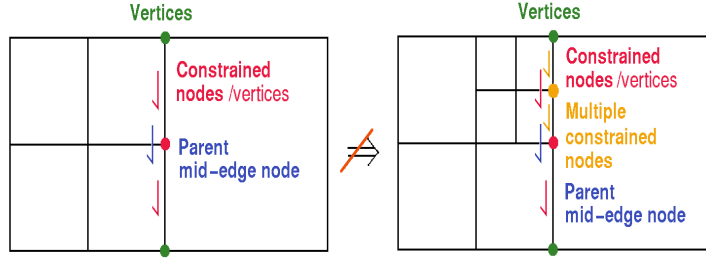


Figure 13: Next refinement leads to the *multiply constrained nodes*.

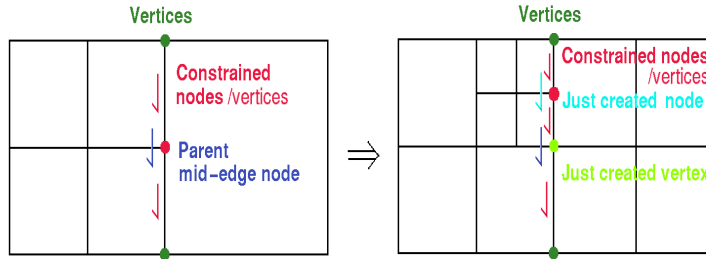


Figure 14: Enforcing breaking neighboring elements, in order to enforce *1-irregularity rule*.

- It is desirable for many reasons to avoid multiply constrained nodes, and limit ourselves to *1-irregular* meshes only. We enforce this *1-irregularity rule* by breaking only those elements, which do not contain constrained nodes. In the case of quads, it is necessary to check if any adjacent mid-edge nodes are constrained. If a constrained node is found on some element edge, we must first break the neighboring element, as shown in Fig.14.
- The fine grid solution is stored in an element fashion - for each of the four sons of a coarse grid element. Translating global d.o.f. into an element local d.o.f. involves using element to nodes connectivities, taking into account global orientations of mid-edge nodes, and the whole machinery of constrained approximation. Calculation of the element local d.o.f. is one of the essential components of the *hp* codes.
- The determination of optimal *hp*-refinements based on *projection-based interpolation* takes place *locally* over each element of the coarse mesh, without any communication with neighboring elements or any other data structures.

The *hp* algorithm

We continue now with a short discussion of operational aspects of the algorithm focusing on those aspects that will be modified in the parallel version.

The main idea of the algorithm is to use the fine grid solution to stage a competition between various possible refinements, by comparing the corresponding decrease rates of the projection-based interpolation error,

$$\text{rate} = \frac{\|u_{h/2,p+1} - w_{hp}\| - \|u_{h/2,p+1} - w_{hp}^{new}\|}{\text{ndof}_{added}}. \quad (5.15)$$

Here $u_{h/2,p+1}$ denotes the fine grid solution, w_{hp} is the current coarse grid interpolant, w_{hp}^{new} is the interpolant corresponding to a (new) grid being tested, and ndof_{added} denotes the *number of local d.o.f. added*, and represents the cost of investment. The mesh optimization is done hierarchically, first optimal edge refinements are determined, and then optimal refinements for (interiors of) elements are found. In the current version of the algorithm, the topology of the new mesh (h -refinements) is determined entirely by optimal edge refinements, and so called *isotropy flags* for coarse grid elements. The isotropy flags reflect the isotropic/anisotropic character of the error function, and are calculated during the evaluation of the global error [7, 8]. The mesh optimization over elements (refined or not in h) consists only of determining the optimal assignment of element orders of approximation.

The algorithm consists of the following main steps.

1. For each element in the coarse grid, the difference between coarse and fine grid solutions is computed, and the corresponding isotropy flags indicating the need for isotropic or anisotropic refinements are determined.
2. For each edge in the coarse mesh, the locally optimal refinement is determined (the edge may be p -refined, or h -refined with optimal orders for the new mid-edge nodes selected), and the corresponding error decrease rate is computed. The computation of the rate requires accessing information about elements neighboring the edge (element size h_1, h_2).
3. Looping over all edges in the mesh, the maximum edge error decrease rate is computed. This requires some global communication. Once the optimal rate is determined, all edges are revisited, and we invest into refining only those edges that deliver error decrease rates within a prescribed percentage of the maximum rate.
4. Based on optimal edge refinements and element isotropy flags, a decision is made about how (and if) to h -refine each element. The optimal h -refinements are made, enforcing the 1-irregularity rule.
5. As a result of enforcing the 1-irregularity rule, some coarse grid edges are *involuntary* h -refined. Those edges are visited again, and the optimal distribution of orders is determined for the new edges, based on comparing the resulting error decrease rates with the maximum rate. The approximation on all coarse grid edges, refined or not, must be fully known before the element stage of the algorithm.

6. Optimal element error decrease rate for each coarse grid element (h -refined or not) is determined. Looping over all elements in the mesh, we compute the maximum element rate. We revisit all elements again and use the maximum element rate to select optimal distribution of orders p for all refined elements.

6 Parallel mesh refinements and mesh reconciliation

We turn now our attention to the parallelization of the hp mesh optimization algorithm.

The parallel version of the algorithm requires a global mesh reconciliation step after refinements have been performed over each sub-domain in parallel. The mesh must be globally consistent. Maintaining the 1-irregularity of the mesh (to avoid the logical nightmare of multiply constrained nodes) requires that an element edge can be broken only once without also breaking the neighboring element. The ultimate goal is to enforce the same rules for element edges situated on the global interface as for those within a sub-domain interior. A final step enforces the minimum rule to ensure that the order of approximation for each edge is set to the minimum of the orders of approximation in the neighboring element interiors.

Our mesh reconciliation algorithm requires an exchange between processors of the following items: refinement trees for edges located on the interface, locations of newly created constrained nodes situated on the interface edges, and the interior orders of approximation for elements adjacent to the interface.

We can summarize the main logic of the algorithm in the following stages,

1. determine the optimal refinements (almost entirely in parallel) and perform them in parallel,
2. exchange information about edge refinement trees, constrained nodes and orders of approximation along the interface,
3. mesh reconciliation.

A repetition of stages (2) and (3) may be required if some of the interface edges were modified during the last iteration.

Example of the parallel mesh refinements. Let us consider the following example. We focus on three initial mesh elements, each one located in a separate sub-domain. The element from sub-domain 1, is adjacent to the element from sub-domain 2, along its right edge, and to the element from sub-domain 3, along its bottom edge, as shown in Fig.15. Only the element from sub-domain 3 is broken. The edges located on the global interface are treated in the same way as internal nodes, so the constrained node is created on the upper edge of the element.

The next hp -refinement iteration is started, and the element from sub-domain 3 is broken once again, see Fig.16. The upper edge of the element is located on the global interface, so the edge

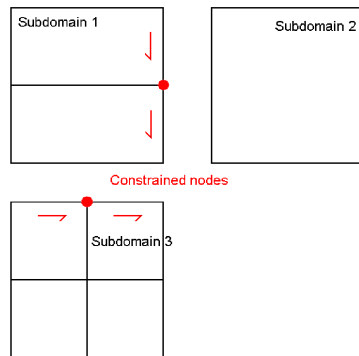


Figure 15: Example of mesh reconciliation process: starting mesh

needs to be treated in the same way as an internal edge. The interface edge must be broken, since the mesh must remain 1-irregular, and multiply constrained nodes are not allowed. Next, a new constrained node is created on the interface edge, see again Fig.16.

The mesh must be globally consistent, so the element neighboring the edge in sub-domain 1 must be broken, see Fig.17. This is done in a *mesh reconciliation* routine, which is called after parallel mesh refinements. But the mesh is still not globally consistent. The mesh reconciliation algorithm must be called once again, in order to break the element located on the second sub-domain, see Fig.18.

This example illustrates one important problem =, which arose during implementation of the parallel mesh refinements algorithm. The problem is related to the need for exchanging refinement tree data over the interface edges. In other words, it is necessary to make the mesh globally consistent, and enforce the breaking of appropriate elements adjacent to the global interface.

Other problems, follow from the need to break interface edges with constrained nodes, as illustrated in Fig.16. These are related to the need for establishing orders of interface edges with constrained nodes, correction factors, and the problem of constrained nodes located at the same place on both sub-domains. All of those problems are explained in detail in the following subsections.

Coding the refinement trees. The refinement trees are exchanged between neighboring sub-domains in a compressed binary format. For example, let us consider the binary description of two refinement trees located on the interface edge, presented in Fig.19. Code 1 denotes a broken edge, while code 0 denotes an edge that is not broken. There is an isomorphism between refinement trees and their binary descriptions, so it is enough to exchange binary descriptions between neighboring sub-domains, compare a description of a refinement tree from current sub-domain with a received description of a refinement tree from neighboring sub-domain, and break some edges in the current refinement tree, if they are different from received ones. The above example shows that the refine-

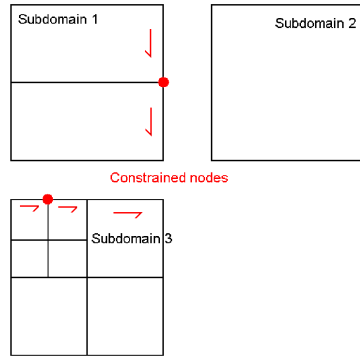


Figure 16: Example of mesh reconciliation process: mesh after refinements process

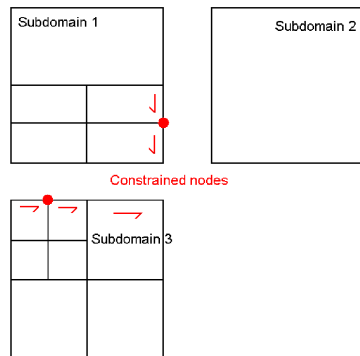


Figure 17: Example of mesh reconciliation process: mesh after the first run of reconciliation routine

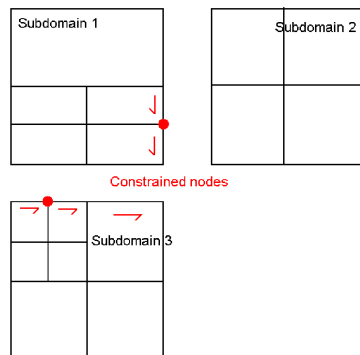


Figure 18: Example of mesh reconciliation process: mesh after the second run of reconciliation routine

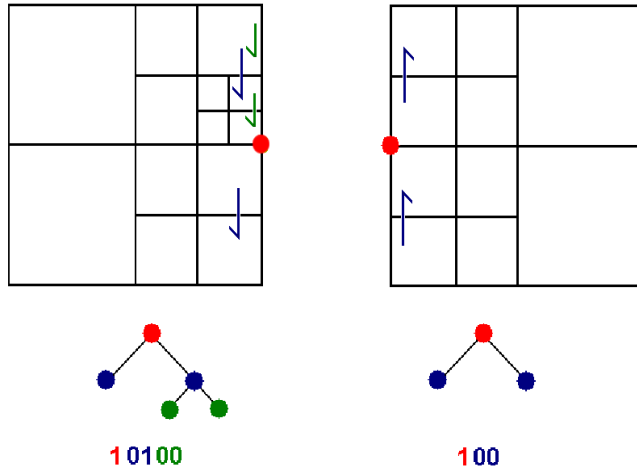


Figure 19: Refinement trees and their representation (before mesh reconciliation)

ment trees need to be exchanged as long as there were any changes made to interface edges during the current step of the mesh reconciliation process.

Orders of approximation for interface edges with constrained nodes. Let us consider a situation, when one edge of an element neighboring interface edge with constrained node was set to be broken in one sub-domain, but in the other sub-domain the interface edge is not to be refined, as in Fig.20 on the left. The element is broken, and the interface edge with constrained node is also broken, as in Fig.20 on the right. To establish orders of approximation for the broken mid-edge node sons, it is necessary to interpolate the fine grid solution over the broken edge. But on the current sub-domain there is no active finite element having entire edge that has just been broken. It is necessary to ask neighboring sub-domain for orders of approximation of just broken interface edge. This example illustrates a need for:

- sending / receiving data for interface edges constrained nodes to neighboring sub-domains,
- calculating the propositions of orders of approximation that can be use on neighboring sub-domain, in a case when the interface edge with constrained node will be broken,
- sending these propositions to neighboring sub-domains.

Correction factors. The third problem, that is hidden inside the sequential code, is connected with the need of calculating some correction factors over elements neighboring an interface edge. The decision of the refinement kind over an edge situated on the global interface is made using an interpolation of the fine grid solution projected onto the edge, with some *correction factor*. The factor is defined as a function of the area of elements neighboring the edge. Thus, a communication

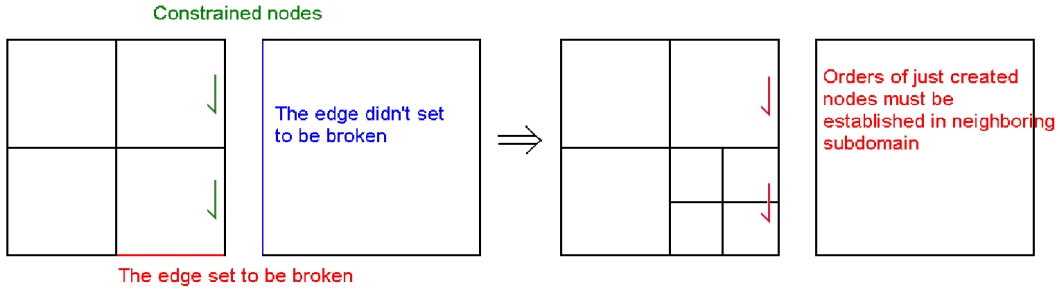


Figure 20: Breaking an element having constrained node on the interface (before mesh reconciliation)

is required in order to obtain the correction factors from elements neighboring the interface edge, located on some other sub-domain.

Constrained nodes located at the same place on both neighboring sub-domains. The last problem arose from the methodology of treating interface edges. When an interface edge is broken, then a constrained node is created, since interface edge is treated at the same way as an internal edge. If the same interface edge is broken on both adjacent elements located on two neighboring sub-domains, then constrained node will be created at the same place on the edge, in both sub-domains. The refinement trees are identical for this edge, however those constrained nodes must be removed. Thus, a communication is required in order to locate and remove those constrained nodes, which are located at the same place on both neighboring sub-domains.

7 Results and discussion

L-shape domain problem

We begin with numerical results for the standard L-shape domain [6] problem. The par2Dhp code was run for the initial mesh presented in Fig.21, over 4 processors, with 3 sub-domains and 1 process responsible for the wire-frame problem solution. After 8 iterations of automatic parallel *hp*-adaptive strategy the optimal mesh was reached, presented in Fig. 22. The optimal mesh delivers a solution shown in Fig. 23 with a 1 % relative error (in energy norm).

The Sandia battery problem

The second problem presented here is a battery problem from Sandia National Laboratories. The problem consists in solving orthotropic heat equation,

$$\nabla (K \nabla u) = f \tag{7.16}$$

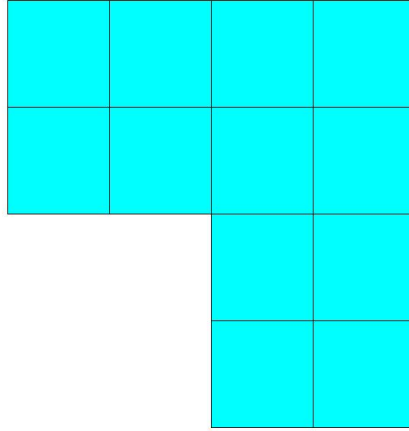


Figure 21: Initial mesh for the Laplace equation over the L-shape domain problem

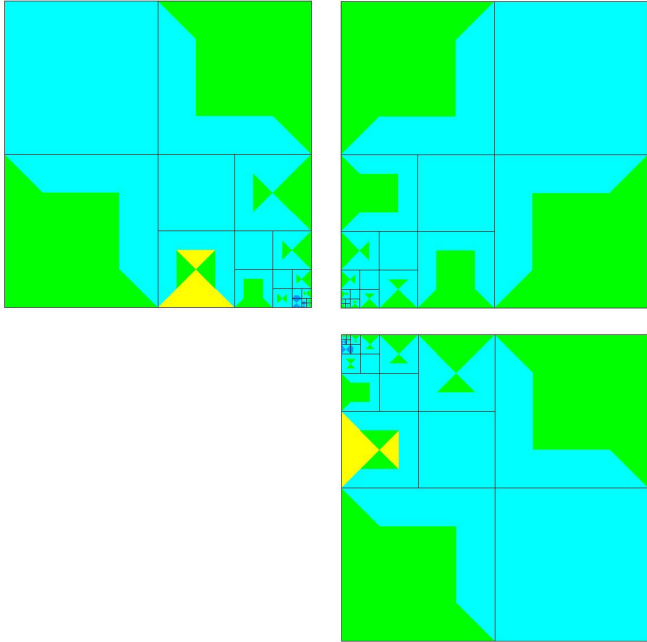


Figure 22: Optimal hp mesh delivering accuracy of 1 % error

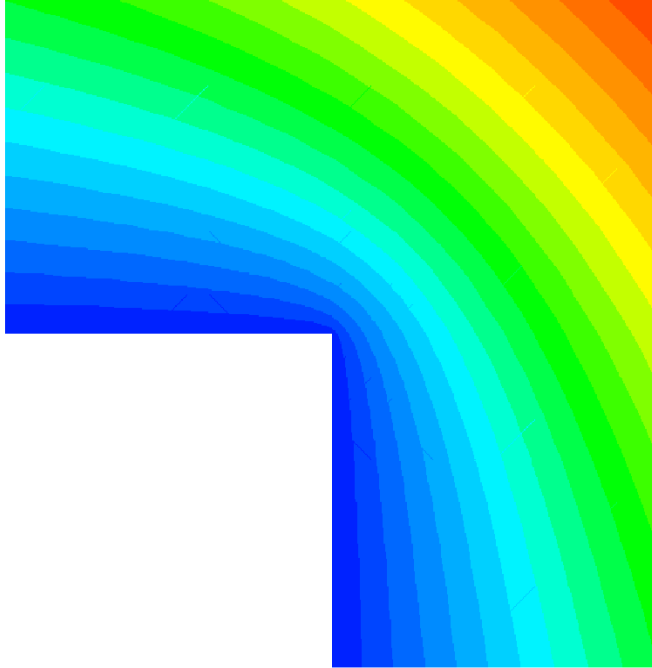


Figure 23: Solution of the Laplace equation over the L-shape domain with 1 % relative error. Exponential convergence is obtained to the accuracy of 1 % relative error

$$K = \begin{bmatrix} K_x & 0 \\ 0 & K_y \end{bmatrix} \quad (7.17)$$

over the domain with 5 different materials, some orthotropic, some not, see Fig. 24. There is a large number of jumps in the material data, which generate singularities. Determining an optimal mesh requires highly anisotropic refinements.

The par2Dhp code was run for the initial mesh presented in Fig. 25, over 16 and 32 processors, with 15 or 31 sub-domains respectively, and process 1 responsible for the wire-frame problem solution. The optimal hp mesh was reached after 45 iterations, together with the corresponding mesh partition shown in Figures 26 and 27. The solution on optimal mesh, see Fig. 28, has a 0.1 % relative error. The sequence of refined meshes delivers exponential convergence rates, illustrated in Fig. 29.

Discussion on the efficiency of the parallel algorithm

In this subsection we would like to discuss efficiency of our parallel algorithm, on the basis of some comparison of numerical calculations for the Sandia's battery problem, performed by using 1, 16 and 32 processors.

First of all, we measured the computation time for each part of the sequential code, during all performed iterations, as presented in Fig. 30. The total computation time varied from 23 seconds in

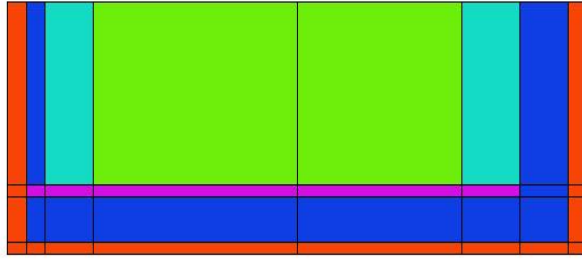


Figure 24: Different materials on the battery problem domain

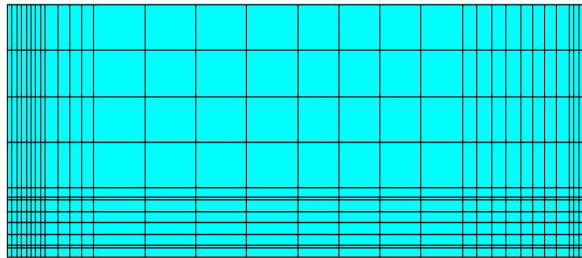


Figure 25: Initial mesh for the battery problem

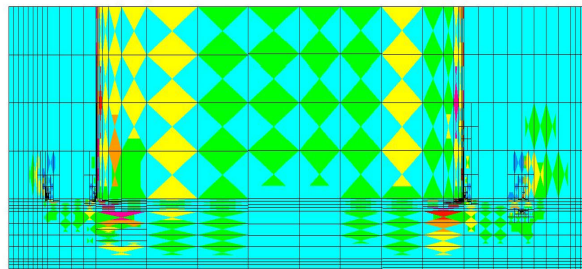


Figure 26: The optimal mesh giving 0.1% relative error of the solution for the battery problem

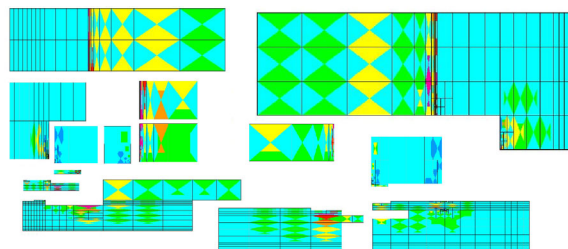


Figure 27: Subdivision of the optimal mesh into processors

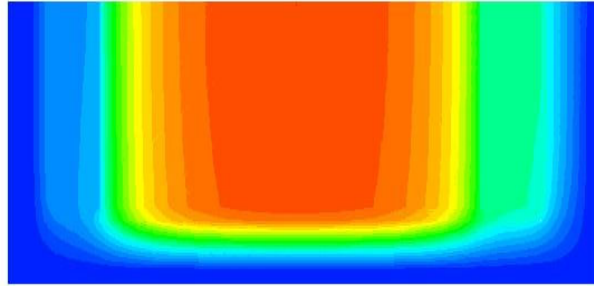


Figure 28: Solution of the battery problem with 0.1 % relative error

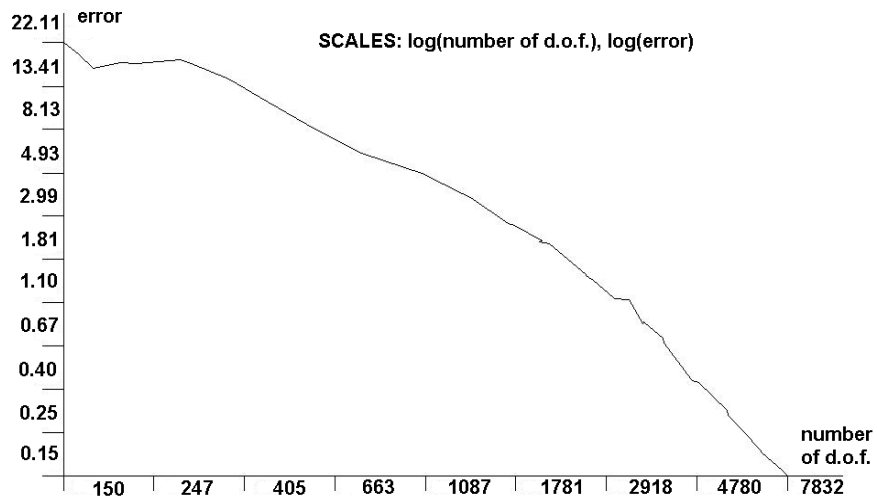


Figure 29: Exponential convergence rate for the parallel calculations of the anisotropic battery problem from Sandia National Laboratories

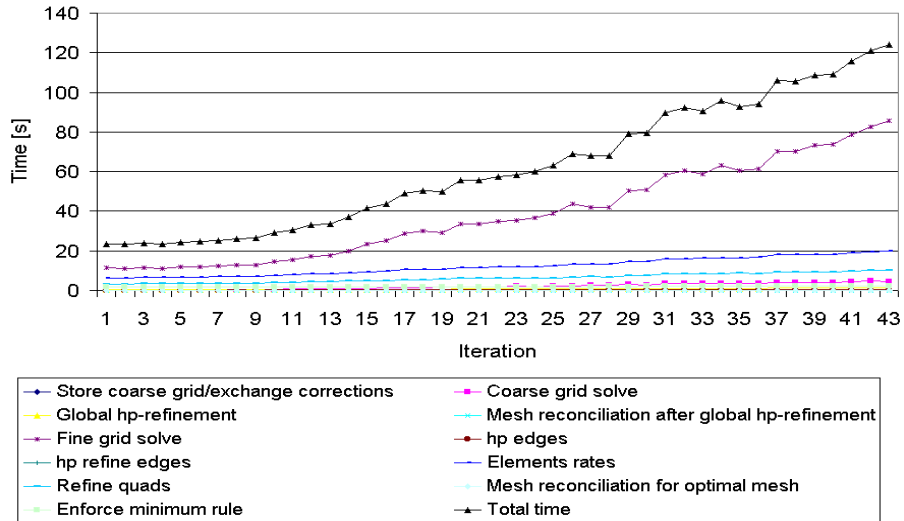


Figure 30: Computation time for each part of the sequential code, measured during each iteration

the first iteration, up to 124 seconds in the last one. The highest influence in the total computation time had the “Fine grid solve” part as expected. That time was uniformly raising from 11s in the first iteration, when the fine grid problem size was 20.000 d.o.f. up to 85s, when the find grid problem size was about 80.000 d.o.f. The next most expensive part was the “Elements rate” part, where optimal refinements for each finite element are determined, by comparing error decrease rates for all possible hp refinements. The algorithm requires solution of some local systems of equations over each particular finite element, and its cost is raising with the number of degrees of freedom. The next most expensive componenets were “Refine quads” routine, estimating optimal orders of approximation for each finite element, and the “Coarse grid solve”.

We compare now the graph with a similar result for the parallel execution using 16 processors, as presented in Fig. 31. We observe that the number of iterations required in order to obtain the same 0.1% accuracy as in the serial code, was lower, equal to 37. This can be explained by the better accuracy of the parallel frontal solver, in comparison with the sequential one. The total computation time for the parallel execution varied from about 4 seconds in the first iteration up to 18 seconds in the last one. Again, the most expensive part was the “Fine grid solve” part, for which execution time varied from about 2 seconds up to about 9 seconds. Next most expensive components were “ hp refine edges” and “Enforce minimum rule” routines, where inter-processors communication is required. The “Refine quads” part was still essential, and the reason for that will be explained. Next, we plot a similar representation for the parallel execution using 32 processors, shown in Fig. 32. Surprisingly, the overall shape of the 32 processors case is similar to that drawn for 16 processors.

We have also measured times for particular stages of the dominating fine grid problem solve

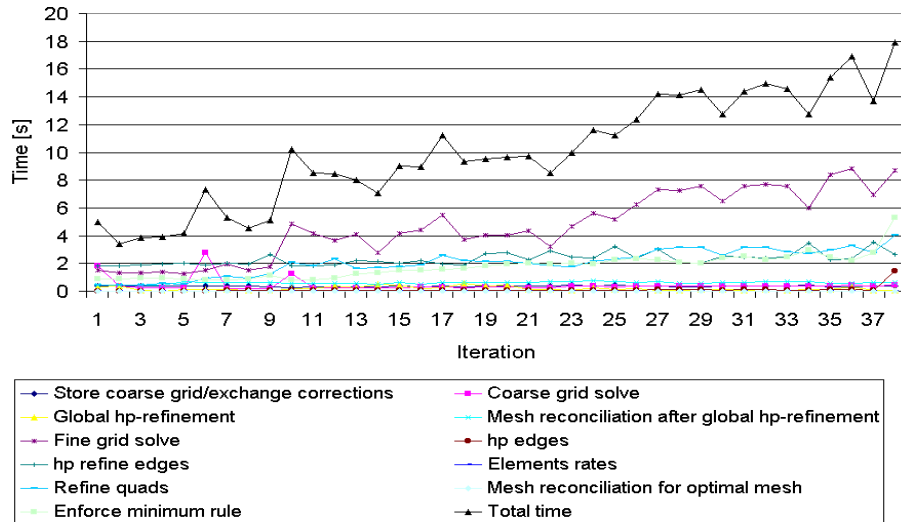


Figure 31: Computation time for each part of the parallel code executed over 16 processors, measured during each iteration

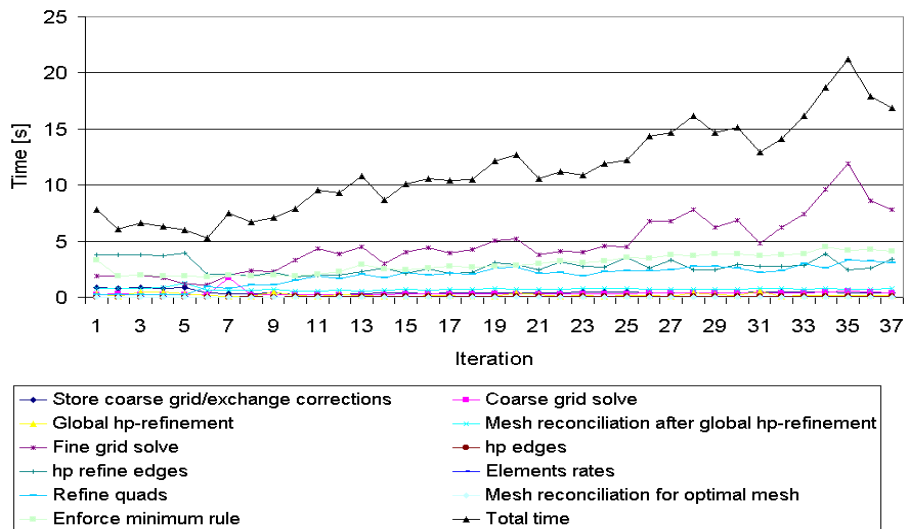


Figure 32: Computation time for each part of the parallel code executed over 32 processors, measured during each iteration

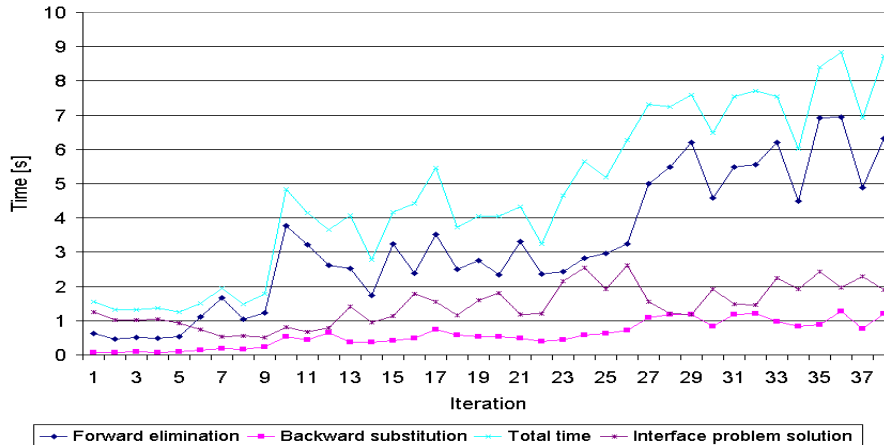


Figure 33: Computation time for each part of the fine grid problem solver, executed on 16 processors, measured during each iteration

part. These are: forward elimination and backward substitution for all slave processors, and the interface problem solution for the master process (including communication during assembling of the interface problem matrix). We present computation times for both 16 and 32 processors, shown in Figures 33 and 34. We can conclude from these Figures, that the interface problem solution time dominates for 32 processors run, and this motivates our future work plans on using the PLAPACK parallel solver.

In order to investigate the problem with a lack of significant difference between computation times for 16 and 32 processors, we performed some measurements of loads over all processors, during particular iteration. Fig. 35 presents measured distribution of computational load (as input into ZOLTAN load balancing library), for each of 32 processors, in particular iterations. The great surprise was that after 6 iterations, when load was almost uniformly distributed over each processor, the load began to rise dramatically on about 14 processors, whilst other processors had zero load and empty sub-domains. The reason for such behaviour can be explained by the following example. Let us consider a case, when there are 10 initial mesh elements, and the load on the first element is equal to 10, whilst the load on each other element is equal to 1. The optimal load balancing for the case of 4 processors, as presented in Fig. 36 is that processor one has the first element with the highest load, and all other nine elements are assigned to processor 2. In other words, processors 3 and 4 have empty sub-domains!

In our *hp*-adaptive code the load balancing is done on the level of initial mesh elements only. In the Sandia battery problem there are about four areas in our computational domain where singularities are present, and most of the *hp*-refinements was required in the neighborhood of these areas. There are about twelve initial mesh elements which completely cover these areas with singularities. After some initial steps, where global *hp*-refinements take place, the algorithm starts

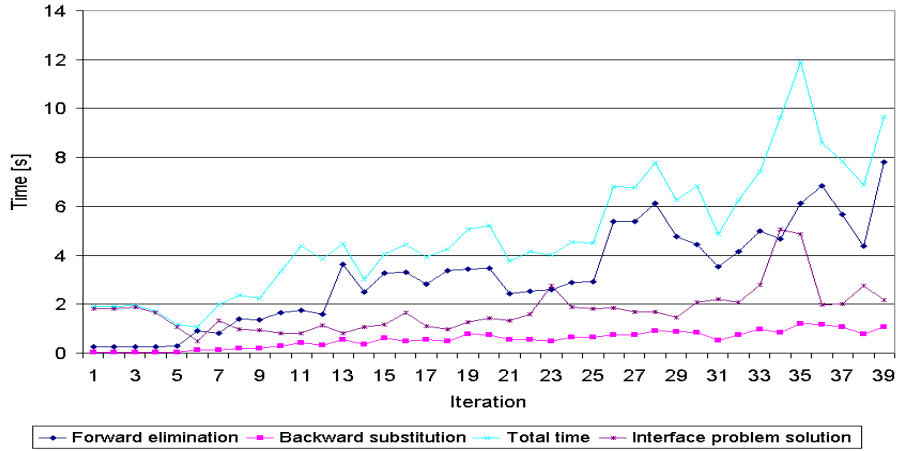


Figure 34: Computation time for each part of the fine grid problem solver, executed on 32 processors, measured during each iteration

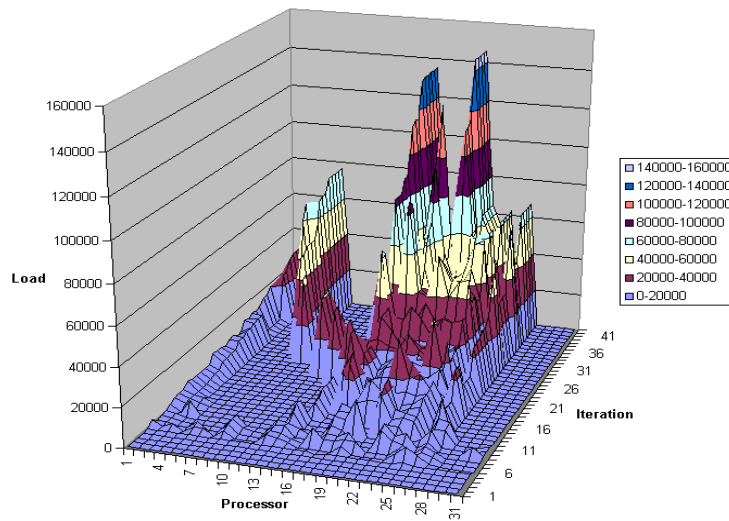


Figure 35: Load distribution over 32 processors during particular iterations

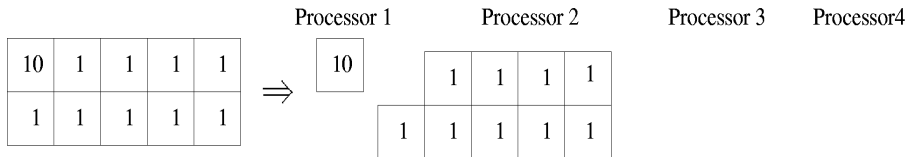


Figure 36: Optimal load balance for mesh containing one finite element with load higher then total load of all other finite elements

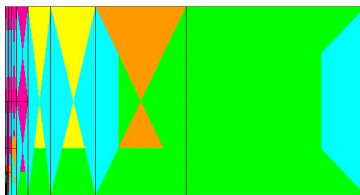


Figure 37: Highly hp -refined initial mesh element in the neighborhood of a strong singularity

strong hp -refinements around the detected singularities, as presented in Fig. 37. These refinements are performed exclusively within these twelve initial mesh elements. The load for some of these elements is one order of magnitude higher than overall load for all other elements. The optimal load balancing proposed by ZOLTAN resulted in using only fourteen sub-domains: some with only one initial mesh element with high load, and others with remaining initial mesh elements, whose total load was smaller than the load of those highly hp -refined initial mesh elements. In conclusion, the use of large number of processors is therefore motivated for problems with large number of singularities.

8 Conclusions and Future Work

In the paper, a parallel version of the fully automatic hp -adaptive FE method, has been presented. The three main technical challenges in parallelizing the sequential version of the code included:

- mesh repartitioning accompanied by regeneration of hp data structure arrays after each mesh refinement, necessary to maintain load balancing,
- parallelization of a (sequential) frontal solver,
- parallel execution of optimal hp mesh refinements and following mesh reconciliation to enforce mesh regularity rules.

The par2Dhp package can be run on any parallel machine, equipped with Fortran 90 compiler, MPI communication platform, and a C++ compiler, required for rebuilding of the ZOLTAN library.

It is possible to select required load balancing algorithm by simply changing value of “LB.METHOD” parameter in ZOLTAN library interface. Thus, load balancing algorithms can be fit to a specific problem requirements.

Mesh is refined fully in parallel, and the algorithm enforces the rule that nodes located on the global interface are treated at the same manner as internal nodes. The 1-irregularity rule of the refined mesh is enforced in the parallel version of the code. This implies that the parallel mesh refinements algorithm delivers a similar sequence of refined meshes as the serial one.

We discuss now shortly our current and future work.

Parallelization of the 3D code. First and foremost, the presented work has been a 'warm up project' before starting the actual parallelization of the 3D *hp* code [18]. Both 2D and 3D codes share an identical data structure which gives us a hope to overcome the complexity of the 3D *hp* coding. The Zoltan/MPI framework has passed the 2D exam and seems to be an optimal environment for doing the 3D work as well.

Solvers. The linear solver presented in this paper does not scale with the number of sub-domains. This is due to the solution of the wire-frame interface problem on a single processor. We are in process of replacing this step with the parallel dense solver from *PLAPACK* [19]. An ideal choice to solve the wire-frame problem - a parallel iterative solver, in context of non-definite wave-propagation problems, is not available. We are also in process of parallelizing our 2grid solver. The distributed memory implementation calls for a different logic of implementation, compared with the workstation version. The main bottle neck in terms of resources is the memory available on a single processor.

Selecting the number of processors with respect to the number of singularities. From the presented discussion it follows that the use of large number of processors is motivated for problems with large number of singularities. We therefore conclude that some initial analysis of the problem should be performed in order to estimate reasonable number of processors, with respect to the number of singularities detected in the computational domain. It can be done automatically, by returning some processors back to system, when the load balancing algorithm starts turning off some sub-domains because of large load associated with other sub-domains where strong singularities are detected.

New approach to *hp*-Adaptivity. Finally, we mention our continuing work on determining optimal *hp* refinements. The determination of optimal mesh refinements is done in stages: for coarse mesh edges, then faces, and finally element interiors. In our current implementation, each stage is followed with the execution of the optimal refinements, e.g. the optimal refinements for faces are determined *after* performing optimal refinements for edges. Enforcement of 1-irregular meshes produces *unwanted refinements* changing the scenarios and complicating tremendously the logic of the algorithm, especially for a parallel implementation. In order to resolve the problem, we are working on a new version of the algorithm in which none of the refinements takes place until all optimal edge, face and element interiors refinements are determined. The determination of optimal refinements will take place in a stand alone code which should facilitate the parallel implementation as well.

References

- [1] M. Ainsworth, "Discrete Dispersion Relation for hp -Version Finite Element Approximation at High Wave Number", *SIAM Journal on Numerical Analysis*, to appear.
- [2] A. Bajer, W. Rachowicz, T. Walsh, and L. Demkowicz, "A Two-Grid Parallel Solver for Time Harmonic Maxwell's Equations and hp Meshes", *Proceedings of Second European Conference on Computational Mechanics*, Cracow, Jun.25 - Jun.29, (2001).
- [3] A. C. Bauer, A. K. Patra, "Robust and Efficient Domain Decomposition Preconditioners for Adaptive hp Finite Element Approximations of Linear Elasticity with and without Discontinuous Coefficients", *International Journal for Numerical Methods in Engineering* **59**(3), (2004) p.337-364
- [4] J. D'Angelo, I. Mayergoyz, "Large-Scale Finite Element Scattering Analysis on Massively Parallel Computers", *Finite Element Software for Microwave Engineering*, Eds. T. Itoh, G. Pelosi and P.P. Silvester, Wiley & Sons, (1996)
- [5] L. Demkowicz, "2D hp -Adaptive Finite Element Package (2Dhp90) Version 2.0", *TICAM Report 02-06* (2002)
- [6] L. Demkowicz, D. Pardo, W. Rachowicz, "3D hp -Adaptive Finite Element Package (3Dhp90) Version 2.0, The Ultimate (?) Data Structure for Three-Dimensional, Anisotropic hp Refinements", *TICAM Report 02-24* (2002)
- [7] L. Demkowicz, W. Rachowicz, and Ph. Devloo, "A Fully Automatic hp -Adaptivity", *Journal of Scientific Computing*; **17**(1-3), p.127-155.
- [8] L. Demkowicz, " hp -Adaptive Finite Elements for Time-Harmonic Maxwell Equations", *Topics in Computational Wave Propagation*, Eds. M. Ainsworth, P. Davies, D. Duncan, P. Martin, B. Rynne, *Lecture Notes in Computational Science and Engineering*, Springer Verlag, Berlin (2003)
- [9] L. Demkowicz, "Fully Automatic hp -Adaptivity for Maxwell's Equations", *TICAM Report 03-45*, (2003).
- [10] H. C. Edwards, "SIERRA Framework Version 3: Core Services Theory and Design." *SAND2002-3616* Albuquerque, NM: Sandia National Laboratories, (2002)
- [11] H. C. Edwards, J. R. Stewart, J. D. Zepper, "Mathematical Abstractions of the SIERRA Computational Mechanics Framework." *Proceedings of the 5th World Congress Comp. Mech.*, Vienna Austria, (2002)

- [12] H. C. Edwards, J. R. Stewart, "SIERRA, A Software Environment for Developing Complex Multiphysics Applications." *Computational Fluid and Solid Mechanics Proc. First MIT Conf.*, Cambridge MA, (2001)
- [13] P.Geng, T.J.Oden,R.A.van de Geijn, "A Parallel Multifrontal Algorithm and Its Implementation", *Comput. Methods Appl. Mech. Engrg* 149 (1997) p.289-301
- [14] I.Foster, *Designing and Building Parallel Programs* <http://www-unix.mcs.anl.gov/dbpp/>
- [15] Y-S. Hwang, B. Moon, S.D. Sharma, R. Ponnusamy, R. Das, and J.H. Saltz, "Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines", *Software - Practice and Experience*, **25**(6), (1995) p.597-621
- [16] A. Laszloffy, J. Long, A. K. Patra, "Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive *hp* finite element simulations" *Parallel Computing* **26**, (2000) p.1765-1788.
- [17] J. Merlin, S. Baden, S. Fink, and B. Chapman, "Multiple Data Parallelism with HPF and KeLP", *Elsevier Science*, preprint (1998)
- [18] D.Pardo, L. Demkowicz, "Integration of *hp*-Adaptivity and Multigrid. I. A Two Grid Solver for *hp* Finite Elements", *TICAM Report 02-33* (2002)
- [19] "PLAPACK: Parallel Linear Algebra Package", <http://www.cs.utexas.edu/users/plapack/>
- [20] G. Pike, L. Semenzato, P. Colella, P. N. Hilfinger, "Parallel 3D Adaptive Mesh Refinement in Titanium" *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March (1999) <http://www.cs.berkeley.edu/projects/titanium>
- [21] J. F. Remacle, Xiangrong Li, M. S. Shephard, J. E. Flaherty "Anisotropic Adaptive Simulations of Transient Flows using Discontinuous Galerkin Methods" *Int. J. Numer. Meth. Engng.* **00**(1-6), (2000)
- [22] Ch. Schwab, *p and hp-Finite Element Methods*, Clarendon Press, Oxford, (1998)
- [23] J. R. Stewart, H. C. Edwards, " SIERRA Framework Version 3: h-Adaptivity Design and Use." *SAND2002-4016* Albuquerque, NM: Sandia National Laboratories, (2002)
- [24] T.Walsh, L. Demkowicz, "A Parallel Multifrontal Solver for *hp*-Adaptive Finite Elements", *TICAM Report 99-01* (1999)
- [25] T. Walsh, and L. Demkowicz, "*hp* Boundary Element Modeling of the External Human Auditory System - Goal Oriented Adaptivity with Multiple Load Vectors", *Computer Methods in Applied Mechanics and Engineering*, **192**, (2003) p.125-146

- [26] Dong Xue, L. Demkowicz, “Geometrical Modeling Package. Version 2.0”, *TICAM Report 02-30* (2002)
- [27] “Titanium: A High-Performance Java Dialect” K. Yelick et al., *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, February (1998) <http://www.cs.berkeley.edu/projects/titanium>
- [28] “Zoltan: Data-Management Services for Parallel Applications”, <http://www.cs.sandia.gov/Zoltan/>

9 Appendix: Algorithms and data structures for parallel mesh refinement and mesh reconciliation

We follow now with a detailed description of our parallel mesh refinement and mesh reconciliation algorithms. The parallel mesh refinement algorithm was created by adding some new stages to the original sequential one, following from the considerations presented in previous chapter.

Global variables

- MAXTREENODES - maximum number of refinement tree nodes, used during packing current sub-domain refinement trees
- MAXTREES - maximum number of refinement trees, used during packing current sub-domain refinement trees
- MAXCONSTRAINS - maximum number of interface edges constrained nodes, used during packing current sub-domain interface edges
- NUMB_MODIFIED_EDGES - number of edges of initial mesh elements that have been modified during mesh refinement process on current sub-domain
- NUMB_CONSTRAINED_NODES - number of interface nodes that have been made constrained in this loop of the refinement algorithm on current sub-domain
- NUMB_RECV_MODIFIED_EDGES - number of initial mesh element edges that have been modified during mesh refinement process on neighboring sub-domains
- NUMB_RECV_CONSTRAINED_NODES - number of interface nodes that have been made constrained on neighboring sub-domains
- NR_INTEDGE - number of interface edges with constrained nodes
- NDIMEN - number of spatial dimensions.

Objects

Object: Refinement tree over the initial mesh element edge.

Name: refined_edge

Description:

Type:

'*coord1(NDIMEN)*', '*coord2(NDIMEN)*' - geometrical coordinates of initial mesh element vertices.

'*ref_kind(MAXTREENODES)*' - table of refinement types for the mid-edge nodes from the refinement tree (1 the mid-edge node has been refined, 0 the mid-edge node has not been refined).

'*iorder(MAXTREENODES)*' - table with orders of approximation for mid-edge node sons in the tree.

'*medg*' - The corresponding mid-edge node index of initial mesh element edge from current sub-domain.

Object: Interface edge with constrained node

Name: constrained_edge

Description:

Type:

'*medg*' - index of the mid-edge node of broken edge located on the global interface.

'*mdle*' - index of the middle node of finite element neighboring the edge on current sub-domain.

'*order*' - order of approximation of the mid-edge node (having the index '*medg*')

'*orders(2)*' - orders of approximation of sons of refined mid-edge node

'*edge_orient*' - orientation flag of the edge

For a detailed description of *elem*, *node* and *vertex* objects, we refer to the description of sequential 2Dhp90 package [5]. We report here only additional properties of those objects, added in parallel version of the code.

Object: Finite element node

Name: node

Description:

Type:

'*numb_neig*' - for interface nodes, the number of neighboring elements (1 or 2) in other sub-domain, 0 for other nodes

'*nord_neig(2)*' - for interface nodes, the order of the neighboring elements middle nodes (1 or 2) from other sub-domain, 0 for other nodes

'*nord_prop(2)*' - for interface nodes, the proposition of orders for sons of broken constrained node situated on the interface, 0 for other nodes (use by *hp_refine_edges* routine)

'*correction*' - correction for the element edge located on the interface, calculated over the element neighboring the edge on other sub-domain (use by *exchange_correction* routine), 0 for internal nodes

'*estimated*' - flag indicating if the orders for the interface node sons were 0 - taken from its parent

(in *break_imedg* routine) or 1 - estimated and enforced in *hp_refine_edges*).

Data structure arrays:

- RECV_MODIFIED_EDGES(MAXTREES) - table of refinement trees over initial mesh element edges received from other sub-domains.
- RECV_CONSTRAINED_NODES(MAXCONSTRAINS,NDIMEN) - table of geometrical coordinates of interface edges constrained nodes received from other sub-domains.
- INTEDGES(:) - dynamic table of interface edges with constrained nodes from current sub-domain.

Parallel mesh refinements algorithm

Store coarse grid. A temporary copy of the current coarse grid data structure is created.

Store coarse grid interface. The routine fills INTEDGES table with data for interface edges constrained nodes. Those data will be sent to neighboring sub-domains, for the reasons explained in paragraph **Orders of approximation for interface edges with constrained nodes**.

Exchange corrections. The routine exchanges information about the correction factors for global interface edges, and stores those factors in NODES(interface mid-edges)%*correction* property. The need for the correction factors is explained in paragraph **Correction factors**.

Solve coarse grid problem. The routine solves the problem over the coarse mesh

Global *hp*-refinements The routine breaks each element into four sons, and increases the order of approximation p by one over each son-element. In the parallel version of the code, the routine requires that interface edges are not broken and the constrained nodes are also created on the global interface.

Mesh reconciliation. The routine "repairs" the mesh that was refined in parallel, since the mesh must be globally consistent. The details of the routine are discussed in the paragraph **Parallel mesh reconciliation algorithm**.

Solve fine grid problem. The routine solves the problem over the fine mesh.

Compute sub-domain element edges error. The routine calculates the error over each finite element edge, by comparing coarse grid and fine grid solutions.

Calculate global element edges error. The routine calculates the maximum element edges error over entire mesh. It is done by calling

```
mpi_allreduce(local_max_edge_error,global_max_edge_error,
```

```
MPI_MAX,MPI_COMM_WORLD)
```

The *hp* edges routine. The routine determines refinement flags for coarse mesh element edges. A detailed algorithm is as follows:

```
Loop through coarse mesh elements
  Loop through element edges
    If edge has not been broken:
      Calculate optimal refinement kind for the edge
      (h-refinement or p-refinement)
    End of loop through element edges
  End of loop through coarse mesh elements
```

The *hp* refine edges routine (*h*-refinements): Based on optimal refinement flags for edges (estimated by *hp* edges routine), routine refines the mesh accordingly, modifies the edge refinement flags for involuntary *h*-refined edges (due to the mesh regularity rules), and determines the final approximation order for edges. In the following, we present the detailed description of the routine. Those parts of the routine that were added in the parallel version of the code, are denoted with **bold characters**. The routine needs to be modified, in order to solve the interface edges with constrained nodes problem, explained the paragraph **Orders of approximation for interface edges with constrained nodes**.

```
Loop through coarse mesh elements
  Ask neighboring sub-domain for proposition of approximation
  orders for interface edges constrained nodes,
  stored in INTEDGES table
  Store received orders propositions in NODES(:)%nord_prop
End of loop through coarse mesh elements
Loop through coarse mesh elements
  Loop through element edges
    If an element edges were set to be broken (in hp_edges):
      Set an element to be vertically or horizontally broken
    End of loop through element edges
  End of loop through elements
Loop through elements which were set to be broken
  Break an element (together with its neighbors,
```

in order to enforce 1-irregularity rule,
 see [5] for more details)
 End of loop through elements set to be broken
Loop through elements which were set to be broken
If an element has constrained node on the interface:
Break interface edge
End of loop through elements set to be broken

Mesh reconciliation. In the middle of the *hp* refine edges routine, where all *h*-refinements were performed, it is necessary to call so-called mesh reconciliation routine. The routine "repairs" the mesh that was refined in parallel, since the mesh must be globally consistent. Details of the routine will be discussed in the paragraph **Parallel mesh reconciliation algorithm**. The routine must be called after creating all new elements, before starting estimation of optimal orders for all finite element edges.

The *hp* refine edges routine (*p*-refinements): The remaining part of the *hp* refine edges routine is estimating optimal orders of approximation for all finite element edges (also for those that were involuntary refined during *h*-refinement and mesh reconciliation processes).

Loop through coarse mesh edges
 If an edge was not set to be broken,
 but it was broken:
 Set an edge for estimation of new nodes orders
 If an edge was set to be broken,
 and it was broken:
 Set an edge for estimation of new nodes orders
 End of loop through coarse mesh edges
Loop through coarse mesh interface edges
with constrained nodes
(stored in INTEDGES table)
If an edge was broken:
Set an edge for estimation of new nodes orders
End of loop through coarse mesh interface edges
 Loop through coarse mesh edges
 If an edge was set for estimation of new nodes orders:
 Estimate new nodes orders
 Enforce orders for new nodes
 End of loop through coarse mesh edges

Loop through coarse mesh interface edges
with constrained nodes
(stored in INTEDGES table)

If an edge was set for estimation of new nodes orders:

Read new orders from NODES(:)%*nord_prop*

Enforce orders for new nodes

End of loop through coarse mesh interface edges

Compute sub-domain elements error rate. The routine calculates the error over each finite element, by comparing coarse grid and fine grid solutions.

Compute global elements error rate. The routine calculates the maximum elements error rate over entire mesh. It is done by calling

```
mpi_allreduce(local_max_element_error,global_max_element_error,  
MPI_MAX,MPI_COMM_WORLD)
```

Refine elements. The routine set optimal orders of approximation for middle nodes of those elements, which error rate is less than 0.33 percent of the maximum error rate.

Enforce minimum rule. The routine enforces the minimum rule over the refined mesh. Here is presented the detailed description of the routine. Those parts of the routine that were added in the parallel version of the code, are denoted by **bold characters**.

```
Loop through active mesh elements  
Set element middle node horizontal approximation order  
as a maximum of its horizontal mid-edge nodes orders  
Set element middle node vertical approximation order  
as a maximum of its vertical mid-edge nodes orders  
End of loop through active mesh elements  
Ask neighboring sub-domains for orders of approximation  
of element middle nodes neighboring the global interface  
Store number of received neighboring elements in  
NODES(:)%numb_neig for interface edges  
Store received neighboring elements middle node orders in  
NODES(:)%nord_neig for interface nodes  
Loop through active mesh elements
```

```

Loop through element edges
  Set order of approximation of the mid-edge node
  as a minimum of neighboring elements middle node orders
  (using NODES(:)%numb_neig
  and NODES(:)%nord_neig for interface edges)
End of loop through element edges
End of loop through active mesh element

```

Details of the parallel mesh reconciliation algorithm

Pack refinement trees of sub-domain interface edges. The routine browses all interface edges in current sub-domain and pack theirs refinement trees.

It is done by calling the following routine for the mid-edge node of initial mesh element:

```

Pack refinement kind of the node
(0 for unrefined edge, 1 for refined edge)
If the edge is refined:
  Call the routine recursively for the first son node
  Call the routine recursively for the second son node

```

The routine creates a binary description of the refinement trees, as it is shown in Fig.19.

Exchange refinement trees. The routine sends to and received from the neighboring sub-domains the binary description of refinement trees over interface edges.

Compare refinement trees. The routine compares received and current refinement trees, enforces refinement when the received tree has additional branches, and sets orders of approximation for new created nodes (as equal to orders of approximation of received refinement tree node sons). If the refinement of the current tree is required, then the comparison algorithm is restarted (from the initial mesh element mid-edge node). Here is presented the detailed description of the routine. Since entire routine is presented only in parallel version of the code, we do not emphasis any parts of the routine.

The following routine is called for the initial mesh element mid-edge node and the roots of received refinement kinds tree:

```

If refinement kind of current node is equal to
refinement kind of received node:

```

```

If both refinement kinds are equal to 0:
    return
Else if both refinement kinds are equal to 1:
    Call the routine for the first son of current tree node
    and the first son of received tree node
    Call the routine for the second son of current tree node
    and the second son of received tree node
Else if received refinement kind is equal to 1
and current refinement kind is equal to 0
    If current node is constrained:
        Break interface edge
        Set orders of approximation for new nodes
        as equal to orders of approximation of
        corresponding nodes in received refinement tree
    Else if current node is unconstrained:
        Break the element neighboring the edge
        Break interface edge
    End if
    Restart the comparisons
Else if received refinement kind is equal to 0
and current refinement kind is equal to 1
    Exit comparison
    (another exchange of refinement trees is required)
End if

```

Find constrained nodes. The routine browses through all interface edges in current sub-domain and packs their constrained nodes.

Exchange refinement trees. The routine sends to and received from the neighboring sub-domains the packed constrained nodes of interface edges.

Compare constrained nodes. The routine compares received and current constrained nodes, and breaks those nodes, that are constrained in both current and neighboring sub-domain. The motivation for the routine is explained in the paragraph **Constrained nodes located at the same place on both neighboring sub-domains.**

The mesh reconciliation algorithm is performed as long, as there won't be any difference between current and received refinement trees over the interface edges.