Computer Assisted Methods in Engineering and Science, **20**: 43–54, 2013.

TWENTY YEARS OF THE CAMES

Architecture of iterative solvers for *hp*-adaptive finite element codes

Przemysław Płaszewski[1], Krzysztof Banaś[1,2]
[1] *Department of Applied Computer Science and Modelling*
AGH University of Science and Technology
Mickiewicza 30, 30-059 Kraków, Poland
[2] *Institute of Computer Science, Cracow University of Technology*
Warszawska 24, 31-155 Kraków, Poland
e-mail: pplaszew@agh.edu.pl, pobanas@cyf-kr.edu.pl

Maciej Paszyński
Department of Computer Science, AGH University of Science and Technology
Mickiewicza 30, 30-059 Kraków, Poland
e-mail: paszynsk@agh.edu.pl

We present a layered architecture for iterative solvers of linear equations, designed to allow for easy integration with existing *hp*-adaptive FEM codes. We discuss interfaces between a solver and an external FEM code and requirements for the FEM code that must be met in order to work with the solver. Our solution is suited to work effectively with stationary as well as time-dependent problems. In this article, we provide an overview of the layered solver's structure and modules of each layer. In subsequent articles, we will present specific implementations of particular layers.

Keywords: solver, FEM, higher-order.

1. Introduction

One of the strengths of higher-order *hp*-adaptive FEM approximations is that for a required level of accuracy much less degrees of freedom (DOFs) are necessary than for linear approximations. Improving solution accuracy of standard *h*-adaptive FEM codes is done through increasing the number of elements of a mesh, while for *hp*-adaptive methods one can work with coarser meshes with fewer elements and locally increase the order of approximation. We assume that, contrary to linear approximation where DOFs are associated with vertices only, in higher order approximation basis functions are associated with other mesh entities, like edges, elements' interiors and faces in 3D.

On the downside, FEM codes that employ such a higher-order approximation require more sophisticated mesh entities management to handle relations between elements and its vertices, edges and faces. In order to assemble the global matrix of a system of linear equations (the global stiffness matrix), the code needs to track for each element which vertices, edges and faces it contains, what are their neighbourhood and the number of associated DOFs. Global matrices of higher-order codes are less sparse than for linear FEM and comprise blocks associated with (shape functions defined for) vertices, edges, faces and elements' interiors. The higher the order of approximation for a particular mesh entity the larger its block, containing entries resulting from integration of all shape functions related to this entity.

Because of the block structure of local and global stiffness matrices, it is advantageous to use special storage schemes and special algorithms for solving systems of linear equations resulting from higher-order approximations. Consequently, the interface between a finite element code and a solver should enable the transfer of information describing block structure of matrices. The problem of creating a flexible interface between FEM codes and linear equations solvers was the subject of investigations described in [1, 2] and recently in [3, 4]. The implementation of direct solvers for *hp*-adaptive codes is presented in [5, 6]

In the present paper, we describe an interface and an architecture for Krylov space iterative solvers equipped with facilities needed to work with higher-order *hp*-adaptive FEM codes. The interface, being a further refinement of the interface described in [7, 8], and an example of implementation that we present take into account the aforementioned mesh entities management and support for block-structured matrices. The design of an interface makes it easy to plug the solvers implementing it into existing FEM codes. As an example of such procedure we couple the developed solver with the general purpose *hp*-FEM code described in and freely distributed with [9].

2. Architecture

Our aim while designing the solver's architecture was to make it possible for the solver to work with a broad range of FEM codes assuming as little as possible about the computations workflow and the way the data is organized inside the FEM code. As there is no single best iterative method, preconditioning mechanism and their implementation that suits all FEM simulations and hardware platforms, we propose modular architecture depicted in Fig. 1.

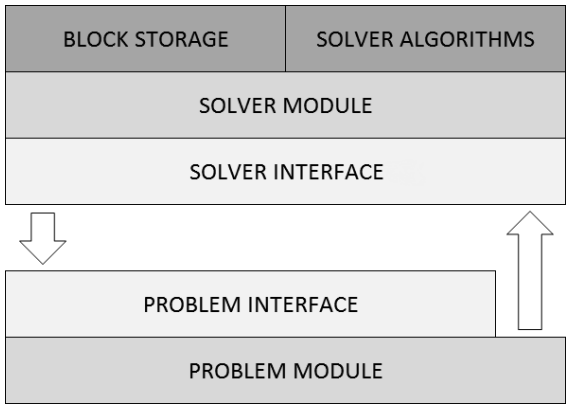


Fig. 1. Solver layers.

2.1. Block Storage and Solver Algorithms

Top layer consists of two modules, Block_Storage and Solver_Algorithms, responsible for data storage and iterative algorithms that can be changed independently of each other and the layers below. Such interchangeability requires well-defined and general enough interfaces between layers and modules in order to accommodate a large variety of possible preconditioners, iterative methods and hardware architectures.

Block_Storage and Solver_Algorithms cooperate closely with each other. Block_Storage implements data structures that store assembled system matrix and preconditioner data. Solver_Algorithms layer is responsible for an iterative method for solving the system of equations (GMRES, CG and their variations). Interface exposed by Block_Storage is composed of functions that perform basic matrix and vector computations required by iterative algorithms such as matrix-vector multiplication, vector norm, and preconditioned residual calculation.

When reimplementing a particular solver (e.g., for porting to some new computer architecture), there are two possible approaches for implementing the two modules of the top layer:

- Re-use existing Block_Storage or Solver_Algorithms module and implement the other one using existing interfaces between the two modules and the layer below. This approach is recommended when there is already Block_Storage optimized for a particular architecture and one needs, for example, to switch from one iterative method to another (e.g., GMRES to CG). And the reverse – for example, when one already has iterative algorithm loop implemented on CPU and want to delegate matrix and vector kernels to GPU.
- Reimplement both modules, when interface between Block_Storage and Solver_Algorithms is limiting – i.e., there is a need for fine-tuned and highly coupled implementation for particular architecture; for example, when off-loading not just the computational kernels to GPU, but whole iterative method logic. In such a case, two top layer modules can even be implemented as one and only the interface with the layer below retained.

2.2. Solver Module

The task of the Solver_Module layer is the organization of solver's work. This layer is independent of the two layers above: Block_Storage and Solver_Algorithms. It orchestrates solver's actions by delegating to the subsequent layers tasks such as assembling of a system matrix, creation of a preconditioner, and solution of a linear equations system. Operations of the Solver_Module are similar to those of the Template Method design pattern – it determines the order and type of tasks while not knowing the implementation's details of each activity.

2.3. Solver Interface

The bottom layer provides an interface exposed to external FEM code. In our architecture, we assume that FEM code:

- Implements the ideas of integration and DOF entities (described in the next section).
- Can return local matrices and vectors that are further assembled into the global system matrix (the stiffness matrix) and the right-hand side vector.

Communication with the FEM code (refered to as Problem_Module – see Fig. 1) is bidirectional. The FEM code must call a function, from Solver_Interface layer, that starts the process of solving the system of linear equations; during the process, solver calls back the FEM code and queries it for information about its mesh entities and their matrices and vectors. Thus the FEM code must be equipped with a thin code layer that implements the Problem_Interface which comprises functions to enumerate entities and return matrices and right-hand side vectors.

Solver_Interface, i.e., functions called by an external FEM code, consists of two sets of functions – basic and advanced interface. Basic interface consists of *solve_lin_sys* function, which initializes the solver, creates a matrix and a preconditioner, then begins the process of solving the system of equations and, at the end, frees resources reserved by the solver. In the case of time-dependent problems, it is possible to use advanced interface (described in Subsec. 2.5), where the FEM code controls the phases of the solver. Therefore, such steps as, for example, allocation of memory by the solver and the creation of data structures can be made only once – before the first time step – or just after each mesh adaptation.

We believe that the proposed architecture minimizes the effort needed to connect an external FEM code to a solver of linear equations, while preserving the flexibility of choosing a right iterative method and a preconditioner needed to obtain optimal performance. In the simplest scenario, one

would only need to implement Problem_Interface layer on top of a FEM code. Since in most FEM codes there already exist functions enumerating elements (and faces or edges) and calculating their local matrices and vectors, functions of Problem_Interfaces will be simple adapters around them. Depending on the problem solved and the hardware architecture employed, one could then use one of the existing Block_Storage and Solver_Algorithms modules or, in most advanced scenario, develop a new one – without the need to change once implemented Problem_Interface.

2.4. Integration and DOF entities

External FEM code, in order to comply with the proposed architecture and solver interface, has to support concepts of integration and DOF entities.

Both concepts are explained in a simple 2D mesh in Fig. 2. This artificial example consists of two elements: a quadrilateral and a triangle. The higher-order FEM degrees of freedom are associated not only with the vertices of elements, but also with the edges and the interiors of elements (for 3D meshes elements' faces play the role of edges). In FEM literature DOFs are often not associated directly with the edges and the interior of an element, but additional nodes on the edges and inside elements are introduced – from the standpoint of our solver architecture both concepts are synonymous. In the interface it is only assumed that with a single mesh entity there may be associated several DOFs (each DOF is associated with a single, scalar or vector basis function).

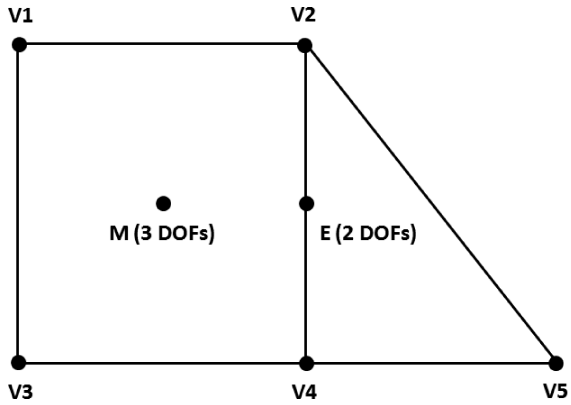


Fig. 2. Integration and DOF entities (description in text).

The quadrilateral in Fig. 2, in addition to the degrees of freedom (shape functions) associated with vertices V1, V2, V3, V4, has also two degrees of freedom associated with edge E and three degrees of freedom (bubble shape functions) associated with the central node (element interior) M. The triangle has three shape functions for vertices V2, V4, V5 and two higher-order shape functions associated with edge E. Note that on the edges between the elements approximations must be compatible[1]. The standard way of achieving this is to ensure that all shape functions, except shape functions associated with vertices V2, V4, and edge E are zero along the edge. Two shape functions, in the quadrilateral and the triangular elements, associated with each DOF of E form a single continuous basis functions, i.e., they are identical along edge E.

Mesh entities with which degrees of freedom are associated are called, in the interface nomenclature, DOF entities. These comprise the vertices and those edges and interiors that contain degrees of freedom. Thus in our example, square element related DOF entities are the following: four vertices, the edge E and the interior M. For the triangle: three vertices and edge E. The number of degrees of freedom provided by the DOF entities in the example of mesh is as follows: all vertices provide one degree of freedom, the interior of M three degrees, the edge E two[2].

[1]In the case of discontinuous Galerkin codes with which our solver can also be combined [10], it is not required.
[2]In our example we assume that the problem solved is scalar – there is one unknown per grid node.

Due to the nature of finite element approximation, each row of a local and global stiffness matrix is related to one degree of freedom; similarly, each column of the matrices is associated with one DOF. Hence, each entry corresponds to a pair of DOFs. Connectivity between DOFs is understood as the fact that an entry related to the DOFs is non-zero (usually this happens when basis functions associated with both DOFs have overlapping supports, hence local and global stiffness matrices have symmetric structure). Connectivity information, for each non-zero entry in a local or global stiffness matrix, consists of knowledge to which pair of DOFs it corresponds. Two DOFs related to a non-zero stiffness matrix entry are called neighbouring DOFs, and their DOF entities are also called neighbours (in the mesh, such DOF entities are usually also neighbours in the topological sense; however, for irregular meshes with constrained approximation the nature of the neighbourhood of DOF entities is more complex).

The solver interface (and our prototype implementation) support different finite element approximation methods and spaces: standard continuous, H_div, H_curl, and discontinuous Galerkin. Each approximation is related to different choices of shape functions associated with mesh entities. However, the solver is oblivious to these details, in order to work properly it needs only to know contributions to entries in the global stiffness matrix coming from different mesh entities and the connectivities in the global stiffness matrix. Both data have to be provided in the form of local stiffness matrices by the FEM code in the procedure *pdr_comp_stiff_mat* (its interface is discussed in details later).

Local stiffness matrices are provided by the FEM code for another type of entities. It is assumed that local matrices are obtained mainly by numerical integration of FEM weak statements over elements, faces (in 3D) and edges (in 2D). Hence, entities that provide local stiffness matrices are called in the interface – integration entities. All entries to the global stiffness matrix are provided by the procedure *pdr_comp_stiff_mat* called for subsequent mesh entities. In order to incorporate an entry to the global stiffness matrix, the FEM code has to associate it with some integration entity (the interface accepts any mesh entity as integration entity) and provide suitable values and connectivity data.

Each integration entity provides in its stiffness matrix entries associated with its DOF entities. In our example, the quadrilateral element has a total of nine degrees of freedom associated with DOF entities V1, V2, V3, V4, M, E. Its stiffness matrix has dimension nine. Triangle provides the stiffness matrix with dimension five.

The structure of the assembled global matrix for our example is shown in Fig. 3. This symmetric matrix consists of stripes related to the individual DOF entities. In particular ordering presented,

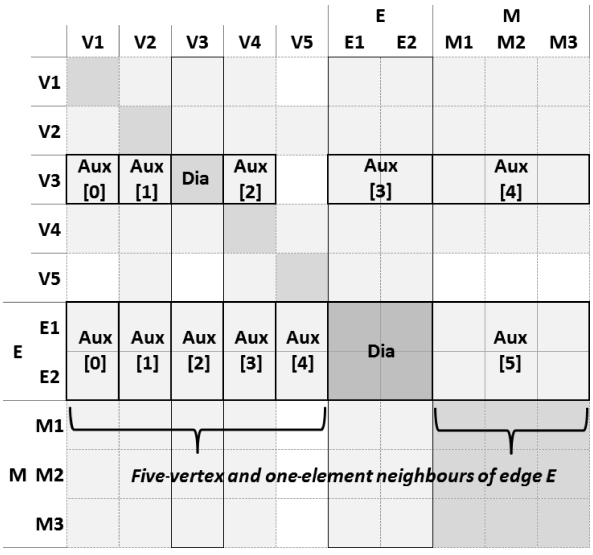


Fig. 3. Structurally symmetric global stiffness matrix structure.

the first five stripes are associated with vertices – each vertex is associated with a single basis function, hence the stripe width is one. Next stripe is associated with the edge E and two higher-order basis functions. The last stripe is associated with the interior M of the rectangular element.

In each of the stripes there is one diagonal matrix block that contains entries related to pairs of DOFs related to a single DOF entity (Dia block) and blocks with entries related to DOFs from a pair of DOF entities. All entries corresponding to pairs of DOFs from a single pair of DOF entities form an off-diagonal block (Aux block).

2.5. Interfaces

The solver interface consists of functions invoked by Problem_Module (i.e., external FEM code). As described previously it is divided into basic interface and advanced interface suited mostly for time-dependent problems. Basic interface consist of a single function:

- *sir_solve_lin_sys*,

which performs the whole solution procedure. Second interface consists of functions:

- *sir_init*, which reads information from solver configuration file and sets up appropriate parameters for solver, preconditioner, etc.;
- *sir_create*, which builds structures needed during solution phase. Inside this function solver communicates with external FEM code in order to get information about integration and DOF entities and their relations;
- *sir_solve*, which assembles and then solves the system of linear equations. During assembling process, local matrices of integration entities are read from external FEM code;
- *sir_free* to free matrix and other structures (preconditioner data, etc.);
- *sir_destroy* to totally erase solver from the data structures (it is assumed that the Solver_Module can hold several instances of solvers in its memory – this feature can be used, e.g., for coupled problems).

In time-dependent problems it is more effective to call *sir_create* only at the beginning and after each mesh and/or order adaptation while for the rest of time steps to call only *sir_solve*.

Problem_Module in order to work with the presented solver must be equipped with an interface through which the solver gets required data. Problem_Interface consists of functions:

- *pdr_get_list_ent*, which is used by solver to get list and information about integration and DOF entities;
- *pdr_comp_stiff_mat*, which returns local matrix and right-hand side vector for an integration entity;
- *pdr_read_sol_dofs*, to read current solution from Problem_Module and set it as initial guess (for time dependent problems);
- *pdr_write_sol_dofs*, used by solver to pass the calculated solution back to the FEM code.

The functions described above present basic, minimal interface for single-level sequential algorithms. In case of multigrid or parallel solutions a few more functions exist which will be covered in subsequent articles.

3. Solution process

The detailed description of interfaces is done using a typical sequence of function calls performed during a solution process. We use functions from the described interfaces and a possible implementation of the interface between Solver_Module and both Solver_Algorithms and Block_Storage. The names of functions called by Solver_Module are chosen to clearly define their meaning.

3.1. Setup phase

Solver's setup phase is shown in the Diagram (Fig. 4).

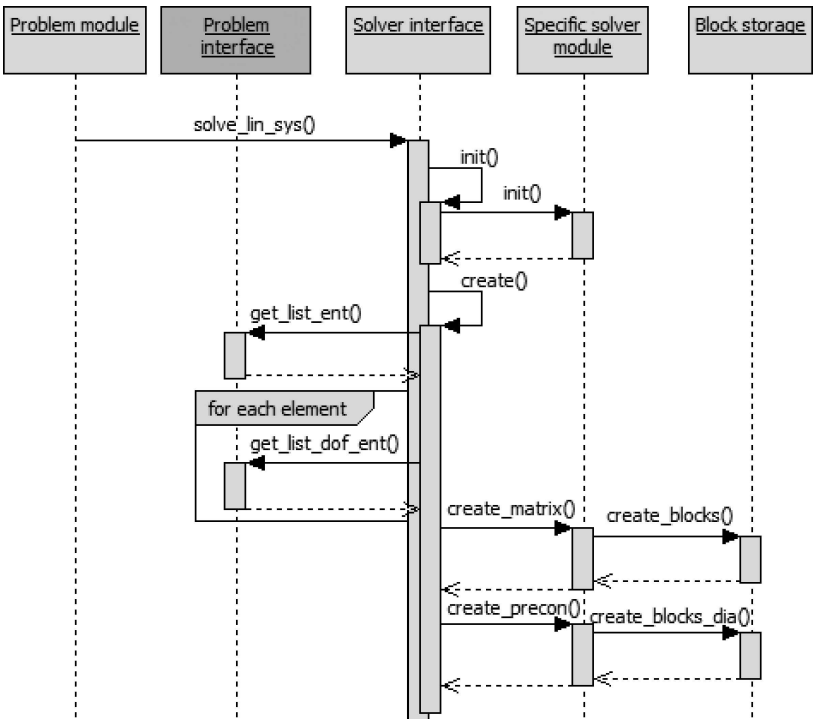


Fig. 4. Setup phase.

During the setup phase, carried out by the functions *sir_init* and *sir_create* of Solver_Interface, memory for matrix blocks is allocated and solver data structures are created – in this phase there are not yet any calculations. Function *sir_init* is used to set the parameters of a specific solver method from Solver_Algorithms layer. These parameters vary depending on the solver and are read from the configuration file. In the case of GMRES iterative solver parameters are, among others, number of restarts and the convergence criteria. Preconditioner parameters and a matrix storage type are also set. In this article, the sparse matrix storage scheme BASIC_BLOCKS is employed. The storage consists in creating a list of DOF entities and storing, for each DOF entity, the corresponding Dia and Aux blocks (shown in Fig. 3).

Function *sir_create* creates data structures that store information about the relationships between elements and related DOF entities, neighbouring DOF entities as well as memory allocations for matrix and preconditioner. The structure of memory allocation for BASIC_BLOCKS is to reserve for each matrix strip (i.e., DOF entity) its Dia and Aux blocks.

As an example of preconditioner block Gauss-Seidel algorithm is used in our description. Block Gauss-Seidel (BGS) preconditioner uses a structure similar to BASIC_BLOCKS storage scheme, with the exception that only Dia blocks are created, that in the case of this preconditioner hold inverses (or factorisations) of Dia blocks from the global stiffness matrix. It is assumed that the

structure is created by function *create_blocks_dia* of Block_Storage layer. In a more sophisticated setting, BGS method can be configured to work with larger Dia blocks, associated with patches of neighboring integration entities and created by assembling the corresponding Dia blocks from BASIC_BLOCKS storage and performing a suitable decomposition. Other types of preconditioners can use different representations, unrelated to actual matrix storage scheme. The important point is that all information about system matrix structure is available when preconditioner setup starts.

In order to retrieve information about the structure of approximation, solver communicates with the Problem_Module using Problem_Interface functions *pdr_get_list_ent* and *pdr_comp_stiff_mat*. The latter is the same function that returns local stiffness matrices and load vectors, but called with parameters indicating that no computations are performed. The first function returns a list of integration entities and DOF entities for the whole mesh, the second returns DOF entities associated with each integration entity – their types, numbers and the numbers of provided degrees of freedom. Analysis of the relationship between the elements and DOF entities allows for the building of a neighbourhood information structure used later when creating matrix storage structure in Block_Storage layer.

3.2. Solution phase

After preparing data structures in function *sir_create* solver is ready for solution of the system. Solution function operations are depicted in Fig. 5. It is assumed that the solver works in the callback model in which it polls the FEM code element by element for their stiffness matrices. For this purpose, *pdr_comp_stiff_mat* function (from Problem_Interface) needs to be implemented

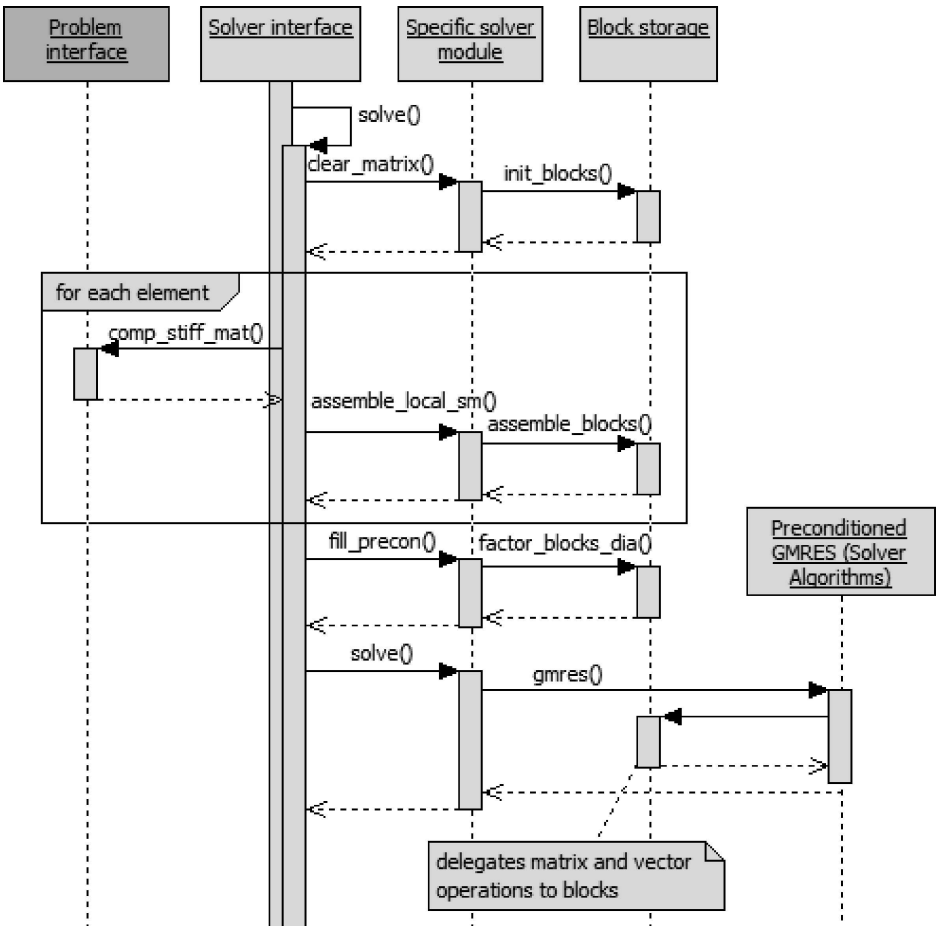


Fig. 5. Solution phase.

in the external FEM code. Then, *assemble_blocks* function from Block_Storage layer assembles the local matrix in the corresponding Aux and Dia blocks in the internal matrix storage structure – information about the relationship between elements and DOF entities gathered in setup phase is used.

When the matrix is assembled, solver is ready to fill preconditioner data structures allocated in setup phase. In the case of simple block GS preconditioner, this step is accomplished by reversing (or factorization) of the assembled preconditioner Dia blocks.

Start of solving the system of equations takes place in the function *sir_solve* of Solver_Algorithms. Basic matrix and vector operations are delegated to the Block_Storage module, as this is the module that knows the specifics of matrix storage structure.

3.3. Post-solve phase

The last stage is to rewrite the vector of solution into structures of external FEM code for later post-processing, etc. For this purpose solver calls *pdr_write_sol_dofs* function of Problem_Interface. In each call to *pdr_write_sol_dofs* solver passes a part of solution vector corresponding to a particular DOF entity, along with entity's number and type.

4. Implementation

As a first test for the proposed architecture, we implemented an example solver that uses GMRES method and basic preconditioners (block Jacobi, block Gauss-Seidel, ILU). The performance of the solver for parallel execution on distributed memory systems is described in [11]. In subsequent articles, we will present different variants of the implementation of both Block_Storage and Solver_Algorithms modules in the context of a layered structure of our solver.

4.1. Storage

Our solver provides a general method for storing the matrix in blocks. It can be adapted and optimized for the architecture and problem solved by the FEM code.

Dia and Aux blocks for approximations of higher-orders and/or vector equations can reach large sizes – BASIC_BLOCKS storage is suited for such cases. Block_Storage type BASIC_BLOCKS (Fig. 3) is a simple and often inefficient way to store the matrix. Especially for scalar problems and linear approximations where all blocks would have dimension one. For such problems, classical CRS scheme applied for the whole global matrix would perform better. Critical for optimal performance of the solver is to use storage schemes and matrix kernels (such as matrix-vector multiplication) appropriate for a given architecture and problem solved. Efficient storage types must be developed together with numerical kernels that operate on them and attention must be paid to particular hardware specifications – like optimal memory access patterns.

5. Integration with *hp*-FEM code

In *hp*-FEM code [9] we implemented the Problem_Interface layer, whose task is to realize the interface required by the solver.

In addition to adding a new layer of communication, no further changes were required in an existing FEM code, although *hp*-FEM was originally designed to work with frontal solver.

This layer receives the incoming requests from the solver, utilizes the appropriate existing features of the *hp*-FEM code, such as functions that return the stiffness matrix, gather relevant information about the relationship between elements and DOF entities, and then returns the data

to the solver. Before calling Solver_Interface function *sir_solve_lin_sys* it is necessary to perform any required setup specific to the external FEM code.

5.1. Results

In Figs. 6 and 7 we present some results for *hp*-FEM 2D code with our solver configured to use BASIC_BLOCKS and simple BGS preconditioner. As solution algorithm, we used GMRES with 20, 50, 100 and 200 Krylov vectors. All code is sequential. Tests were performed for a simple test problem of adaptivity: Laplace equation on L-shaped domain. Increase of number of degrees of freedom was obtained with automatic *hp*-adaptation available in the *hp*-FEM code [9], with maximal degree of approximation p equal to nine. Subsequent steps of the algorithm lead to very small elements (with a linear size approx. million times smaller than in the initial mesh) near the corner singularity and large elements with high p far from the singularity. The final mesh is depicted, using subsequent magnifications, in Fig. 8.

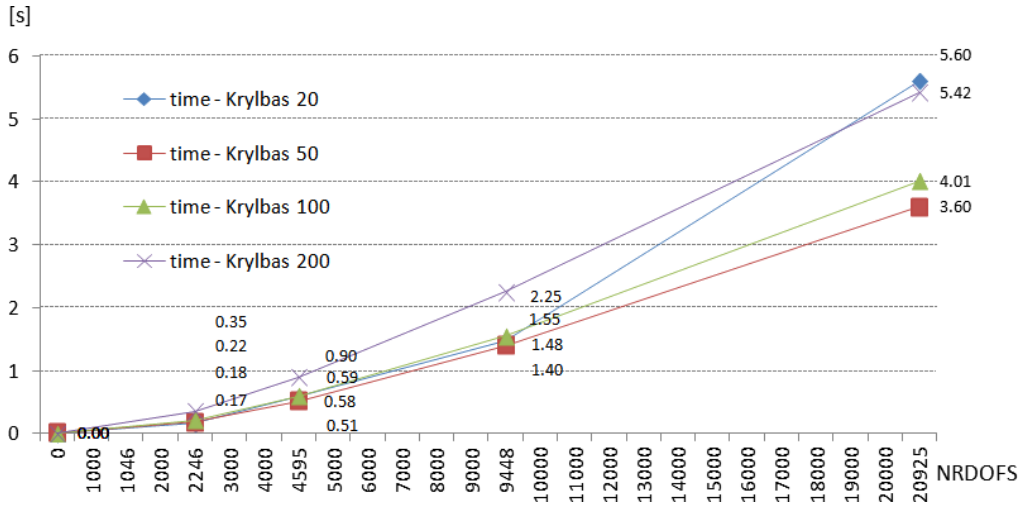


Fig. 6. Solution time for different numbers of iterations in a single restart (the numbers of basis vectors in the Krylov space) and different numbers of degrees of freedom in subsequent steps of adaptive solution of the L-shape domain 2D problem.

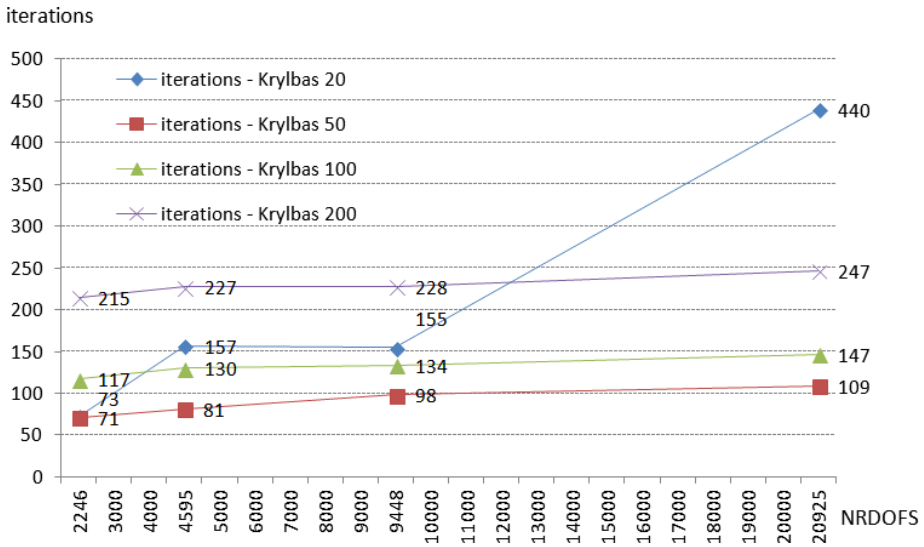
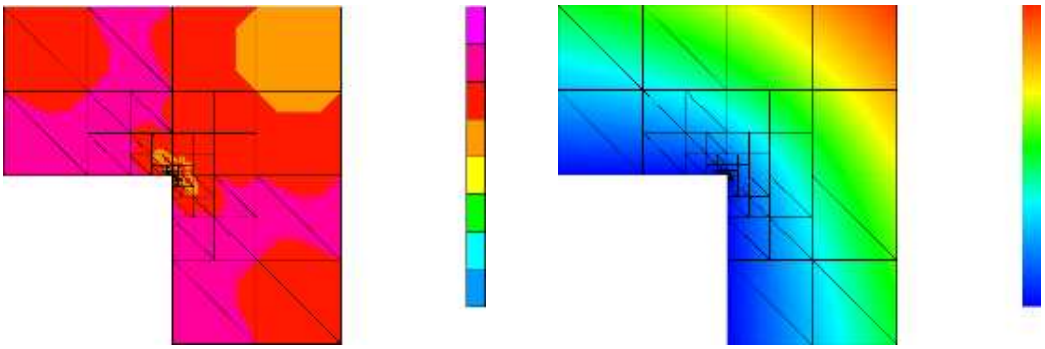
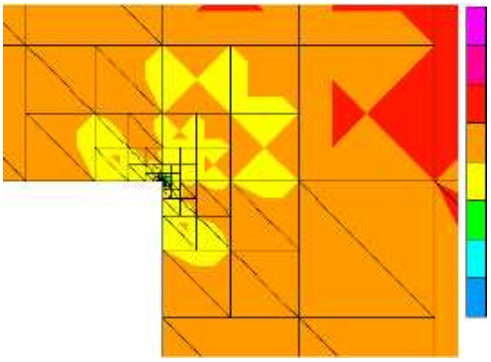


Fig. 7. Number of GMRES iterations for different numbers of iterations in a single restart and for different numbers of degrees of freedom in subsequent steps of adaptive solution of the L-shape domain 2D problem.

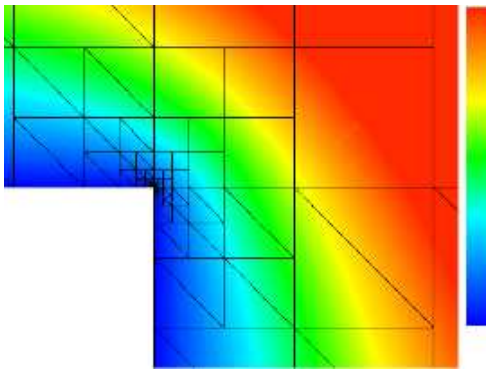


L-shape domain – mesh – original size

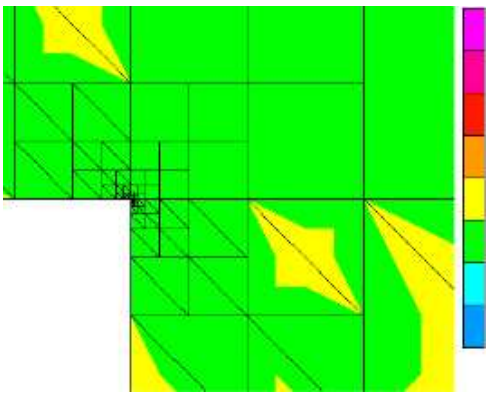
mesh and solution – original size



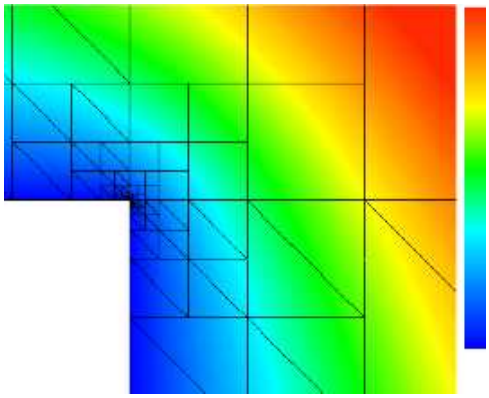
L-shape domain – mesh – magnification ×100



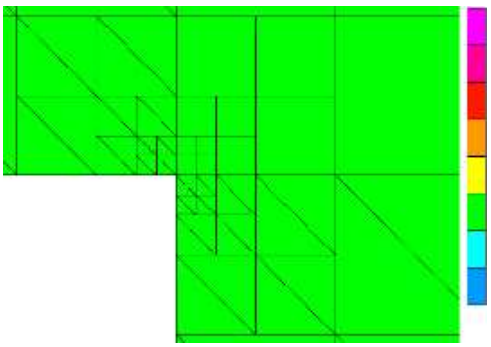
mesh and solution – magnification ×100



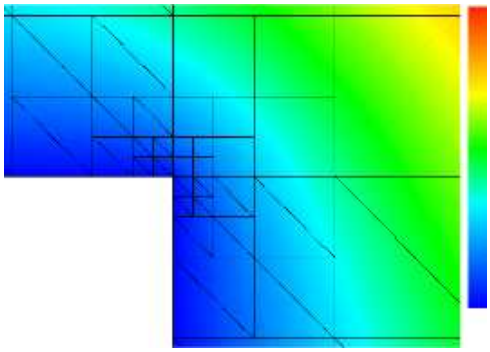
L-shape domain – mesh – magnification ×10000



mesh and solution – magnification ×10000



L-shape domain – mesh – magnification ×1000000



mesh and solution – magnification ×1000000

Fig. 8. Subsequent magnifications of the mesh obtained by automatic *hp*-adaptations for the L-shaped domain problem, leading to the solution with error in energy norm less than 0.001%. Mesh colours denote degree of approximation (from 1 to 8), solution colours – values of solution (color range changes for each image).

The solver works well with a highly unstructured mesh and non-uniform approximation. Despite very simple preconditioning, it converges to the solution. It can be seen that for the small number of Krylov vectors in a single restart the convergence may be difficult to obtain. However, also for too big numbers of base vectors in the Krylov space the convergence deteriorates, the fact that can be attributed to round-off errors that spoil the orthogonality property required for bases of linear spaces.

6. Conclusions

In this article, we presented a layered solver architecture adapted to cooperate with external FEM codes, including those applying higher-order approximation and *hp*-adaptivity.

With clearly defined interfaces in C language and concepts of integration and DOF entities "gluing" our solver with external FEM codes, such as *hp*-FEM code written in Fortran, turned out to be relatively easy. Layered architecture allows the interchangeability of algorithms for solving systems of linear equations, preconditioning methods and storage schemes without the need for changes in both solver and problem interfaces.

Such an architecture should also allow for easy adaptation of solvers to changing computer and processor architectures and programming environments. In subsequent articles, we will present such adaptations, e.g., for GPU architecture and OpenCL environments as well as advanced storage schemes and solver algorithms, such as multigrid preconditioning.

Acknowledgement

The work has been supported by Polish National Science Center grant no. NN 519 447739.

References

[1] R. Clay, K. Mish, I. Otero, L. Taylor, A. Williams. *An Annotated Reference Guide to the Finite Element Interface (FEI) Specification*. Sandia Report SAND99-8229, Sandia National Laboratories (1999).

[2] D.E. Bernholdt, B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T.L. Dahlgren, K. Damevski, W.R. Elwasif, T.G.W. Epperly, M. Govindaraju, D.S. Katz, J.A. Kohl, M. Krishnan, G. Kumfert, J.W. Larson, S. Lefantzi, M.J. Lewis, A.D. Malony, L.C. Mclnnes, J. Nieplocha, B. Norris, S.G. Parker, J. Ray, S. Shende, T.L. Windus, S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, **20**: 163–202, 2006.

[3] M. Blatt, P. Bastian. On the generic parallelisation of iterative solvers for the finite element method. *Int. J. Comput. Sci. Engrg.*, **4**(1): 56–69, 2008.

[4] A. Dedner, R. Klöfkorn, M. Nolte, M. Ohlberger. A Generic Interface for Parallel and Adaptive Scientific Computing: Abstraction Principles and the DUNE-FEM Module. *Computing*, **90**(3–4): 165–196, 2010.

[5] M. Paszynski, D. Pardo, C. Torres-Verdín, L.F. Demkowicz, V.M. Calo. A parallel direct solver for the self-adaptive *hp* finite element method. *J. Parallel Distrib. Comput.*, **70**(3): 270–281 (2010).

[6] M. Paszynski, D. Pardo, A. Paszynska, L.F. Demkowicz. Out-of-core multi-frontal solver for multi-physics *hp* adaptive problems. *Procedia CS*, **4**: 1788–1797, 2011.

[7] K. Banaś. On a modular architecture for finite element systems. I. Sequential codes. *Computing and Visualization in Science*, **8**: 35–47, 2005.

[8] K. Banaś. A modular design for parallel adaptive finite element computational kernels, in: M. Bubak, G. van Albada, P. Sloot, J. Dongarra [Eds.], *Computational Science – ICCS 2004, 4th International Conference, Kraków, Poland, June 2004, Proceedings, Part II, Vol. 3037 of Lecture Notes in Computer Science*, 155–162 Springer, 2004.

[9] L. Demkowicz. *Computing with hp-Adaptive Finite Elements, Vol. 1: One and Two Dimensional Elliptic and Maxwell Problems*. Chapman and Hall/CRC, 2006.

[10] K. Banaś, M.F. Wheeler. Preconditioning GMRES for discontinuous Galerkin approximations. *Computer Assisted Mechanics and Engineering Sciences*, **11**: 47–62, 2004.

[11] K. Banaś. Scalability analysis for a multigrid linear equations solver, [in:] R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski [Eds.], *Parallel Processing and Applied Mathematics, Proceedings of VIIth International Conference, PPAM 2007, Gdansk, Poland, 2007, Vol. 4967 of Lecture Notes in Computer Science*, 1265–1274 Springer, 2008.