Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1584 – 1593

Procedia Computer Science

2013 International Conference on Computational Science

Performance analysis of iterative solvers of linear equations for hp-adaptive finite element method

Przemysław Płaszewski[a,*], Krzysztof Banaś[a,b]

[a]AGH University of Science and Technology, Al. A. Mickiewicza 30, 30-059 Kraków, Poland [b]Cracow University of Technology, Warszawska 24, 31-155 Kraków, Poland

Abstract

The paper presents performance considerations for Krylov space iterative solvers used in hp-adaptive finite element codes. The design of optimal data structures and algorithms in terms of memory requirements and computing efficiency is shown. Preliminary computational results prove the correctness of basic assumptions and indicate the direction of further research.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science

Keywords: hp-adaptive; finite element method; iterative solvers

1. Motivation - characteristics of hp-adaptation and their impact on linear solvers

Adaptivity in finite element codes lead to large-scale problems. The systems of linear equations that have to be solved efficiently for these problems possess some specific features that influence the form of matrices of linear equations (finite element stiffness matrices) and the solution procedures. These features depend on particular forms of adaptivity adopted.

Increasing the order of approximation results in new basis functions assigned to mesh entities like edges (for 2D and 3D), faces (for 3D only) and elements' interiors (so called bubble functions). With p-adaptation, order p is not uniform for the whole mesh and varies between mesh entities (see Fig. 1a for a simple example of such a mesh). Some mesh regions may remain only linearly approximated with basis functions present just for vertices, one for each vertex, and not the edges or faces. Other regions may contain entities with as many as few tenths of polynomial basis functions of different order.

* Corresponding author.
E-mail address: pplaszew@agh.edu.pl

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.326

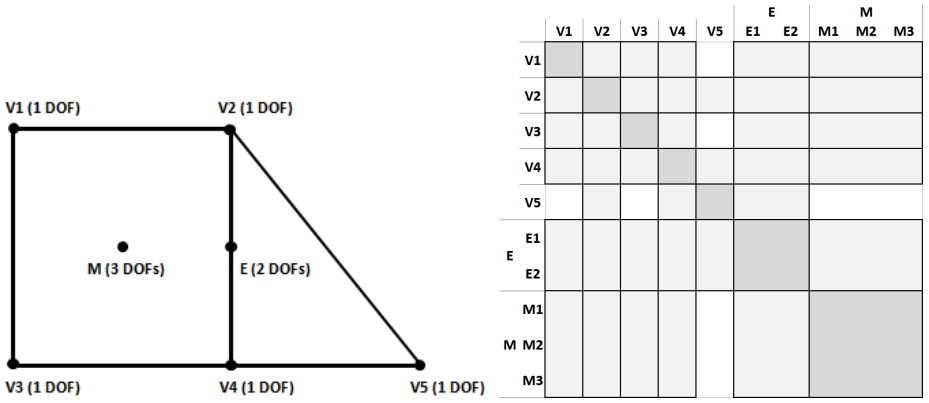

Fig. 1. (a) Finite element mesh with higher order entities E and M; (b) Block structure of a matrix corresponding with mesh (a)

It is natural to group basis functions related to a particular mesh entity (we will call mesh entities that have basis functions, i.e. provide degrees of freedom, DOF entities). For stiffness matrices such groupings result in a, typical for higher order problems, block structure shown in Fig. 1b. Blocks are formed by entries related to basis functions corresponding to a pair of mesh entities. Such dense blocks for high orders p (especially for vector problems with more than one approximated function) may have hundreds of entries.

hp-adaptation adds to non-uniform order distribution present in p-adaptation, local variations of mesh density. It leads to high irregularities in the number of neighbours of mesh entities (as neighbouring entities we understand those with overlapping basis function's support) and their blocks dimensions. In non-refined or only h-refined parts where vertices are the only DOF entities, blocks are small and numbers of neighbours for a single entity remain small, as faces/edges and elements do not have basis functions assigned.

All these lead to a special structure of matrices of linear equations. The purpose of the present paper is to analyse this structure and to specify the set of requirements for iterative solvers specifically designed for this kind of systems. The further parts of the paper describe the structure and the performance of a solver that tries to fulfil these requirements.

2. Iterative solvers for hp-adapted problems

Direct solvers based on Gaussian elimination variants, although currently very efficient, due to the large memory requirements are poorly suited to solve very large problems. They are also very sensitive to the matrix bandwidth, in particular for 3D problems where the use of proper reordering is necessary [2] – much less important for iterative solvers.

The main advantage of direct solvers, which is a guarantee to obtain an accurate solution for each nonsingular system of equations, is overshadowed by the biggest drawback, which is the significant investment of memory and time required for problems with a very large number of degrees of freedom.

In the present paper we put our focus on iterative solvers based on Krylov methods such as CG or GMRES. In the case of sufficiently good convergence, they offer significantly lower computational complexity and memory consumption.

Solver design appropriate for hp-adapted problems requires consideration of the block structure of these problems. Matrix storage method has a significant impact on the efficiency of matrix-vector operations used in iterative algorithms. CRS and derivatives do not take into account the problem structure and are characterized

by a large number of indirect memory references. Standard compressed block storages are not effective for large irregularities in the size and distribution of blocks.

In addition to the effective implementation of the basic matrix-vector operations, a fundamental issue for the performance of an iterative solver is to provide good convergence. This is done by selecting an appropriate strong preconditioner.

The goal, which should be taken into account while implementing an iterative solver, is to obtain linear scalability in terms of the number of degrees of freedom. Scalability is achieved by providing:

- constant number of iterations (this is the role of strong preconditioner)
- scalable construction of the preconditioner

The better the preconditioner is, the more its matrix (which usually is never constructed) resembles the matrix of the system, **A**. It should always be borne in mind that the better the preconditioner, the more time is needed for its construction, which is not always amortized by fewer iterations of the main loop of an iterative algorithm.

Two types of preconditioners can be identified:

- single-level
- multi-level (including popular two-level v-cycle)

Block-Jacobi or block-Gauss-Seidel preconditioners (and their generalizations in the form of additive and multiplicative Schwarz preconditioners) form a popular group of preconditioning methods [6]. In both approaches, suitable blocks on the main diagonal are inverted and used to approximate the inverse of the original matrix **A**.

An obvious choice for preconditioner blocks are blocks associated with degrees of freedom related to a single mesh entity. Inverting such a block lead to the solution of a small problem for the subdomain consisting of supports of basis functions for DOFs within the block.

Local, single DOF entity, solutions, for obvious reasons, do not include long-range interactions. To remedy this situation and improve the preconditioner performance local subproblems can be extended to local patches of elements, the larger the patch, the better the approximation of the inverse of matrix **A**. Expanding the patches significantly increases computational effort required to factorize matrix fragment. In the extreme case, the patch would cover the entire mesh - of course factorizing the whole matrix **A** (if possible at all) would much exceed the time required for the iterative solution with a much weaker preconditioner. Appropriate balance must be found.

Multilevel methods provide an alternative way to take into account long-range interactions, without a significant increase in computational effort [8]. This is due to use of coarser meshes hierarchy and the approximate solution of residual form of system of equations on each mesh.

For the systems with good initial conditioning, even a single level block preconditioners allow one to solve large problems in a reasonable time, even though they did not achieve linear scalability, as it would be the case when using the MG.

3. Solver architecture

In the current section we describe a Krylov space solver designed for systems obtained with hp-adaptive FEM. Currently the code have single level, problem structure aware, preconditioning based on multiplicative Schwarz method, in which the approximation of the matrix $\mathbf{A}^{-1}$ is obtained through local solutions for individual main diagonal blocks or larger blocks related to patches composed of neighbouring DOF entities. The extension of the solver for multigrid preconditioners, which significantly improves the conditioning of the system by eliminating the low-frequency error components, is the goal of the next phase of its development.

A detailed description of the solver architecture can be found in [8]. We briefly present main ideas of its structure below.

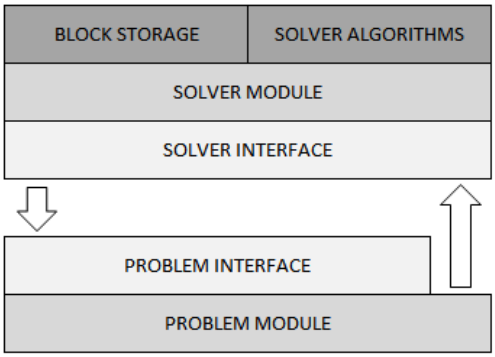

Fig. 2. Layered solver architecture

Fig. 2 presents the layered architecture of the code. Our aim was to decouple solver as much as possible from external FEM code model, so it could cooperate with broad spectrum of FEM codes.

Block Storage and Solver Algorithms layers are responsible for data storage and iterative algorithms that can be changed independently of each other and the layers below.

Particular way of storing the system matrix and preconditioner data is encapsulated in Block Storage. The main iterative algorithms (GMRES, CG and their variants) reside in Solver Algorithms layer. Solver Algorithms delegates storage-dependent operations to Block Storage. Thanks to such design matrix storage manner, type of preconditioner and iterative algorithm can be implemented independently, even without a need to recompile the whole solver [1, 4].

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, preconditioned residual calculation. It should be noted that it is possible thereby to adapt to different hardware architectures. For example, main loop of an iterative algorithm can be performed on the CPU, and the individual kernels, like for example calculation of preconditioned residuum, can be implemented in Block Storage and executed on the GPU.

Solver Module layer organizes the solver work by delegating to the subsequent layers tasks such as assembling of a system matrix, creation of a preconditioner, solution of a linear equations system.

Solver Interface provides an interface exposed to external FEM code. In our architecture, we assume that FEM code is able to enumerate its DOF entities in block-aware manner and return local matrices and vectors for assembling. Our goal was to provide an architecture where developing new storage schemes and preconditioners would not require changing anything in external FEM code and/or solver interface.

3.1. Block storage

For higher degrees of approximation (when a single mesh object, DOF entity in our nomenclature, can be linked to a number of basis functions) or approximation of vector problems basis functions in a natural way are joined into groups. Such a clustering of the basis functions induces division of a global vector of unknowns (the vector of coefficients of the linear combination of basis functions) into smaller vectors. The set of degrees of freedom corresponding to the functions associated with a single DOF entity is called elementary vector of unknowns. Elementary vectors are disjoint and sum up to the whole vector of unknowns. Right-hand-side vector **b** and the matrix **A** are divided in similar way.

A single elementary block of matrix **A** corresponds to a pair of DOF entities (later in the paper such blocks are called Aux - auxiliary blocks). The exceptions are the main diagonal blocks (called Dia blocks) that correspond to a single DOF entity. The block is non-zero if the corresponding grid objects are close to each other - they are neighbours.

For each DOF entity, we allocate a structure that holds this entity's matrix stripe, i.e. one Dia block, off diagonal Aux blocks (one for each neighbour) and the corresponding right-hand-side vector part, see Fig. 3.

Dia blocks are always square, Aux blocks may be rectangular. For the purpose of preconditioning, elementary blocks may by further grouped into larger blocks associated with groups of mesh objects. It is easy to imagine the blocks associated with all objects forming a single finite element - again one square diagonal block and several off-diagonal blocks associated with element neighbours would be formed. Going forward, one can create blocks associated with small groups of adjacent elements (patches) and even further blocks representing large sub-areas of a mesh. An important feature of the blocks is that elementary blocks contain only nonzero entries. For larger blocks, this is not true. For example, Aux blocks associated with a single element contain the zero entries of the matrix **A**. When creating even larger patches, zeros appear in diagonal blocks. For blocks associated with large sub-areas, their structure is similar to the structure of the matrix **A**.

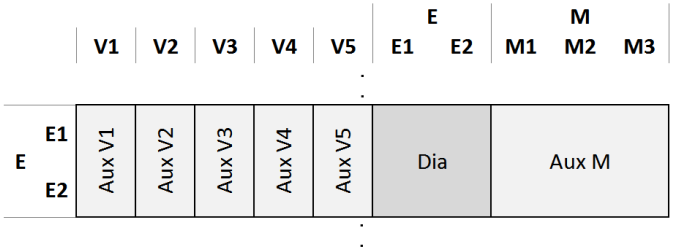

Fig. 3. Matrix stripe of DOF entity E with Dia and Aux blocks

A sparse matrix structure is used by all solvers used in FEM programs. Simple reasoning for the structural cubic mesh and continuous scalar linear approximation shows that the number of nonzero terms in a single row **A** is fixed at 27. In our test of 2D hp-adapted problems (see section 6) maximum number of elementary blocks in a matrix stripe associated with single DOF entity was 37 and the average was around 13 (dimensions of blocks depended on the order p of particular DOF entity and its neighbours). In large-scale computations, where the number of degrees of freedom exceeds several tens of millions, sparsity of a matrix exceeds 99,99%, even in the case of high degrees of approximation and unstructured grids.

4. Solver operations

The main sequence of solver computations is as follows:

Setup phase:

1. Query the problem module for integration and DOF entities.
2. For each DOF entity allocate a structure that holds matrix stripe corresponding with it. Stripe is composed of a diagonal (Dia) block and off-diagonal (Aux) blocks. Diagonal blocks are of dimensions *nxn* where *n* is the number of DOFs associated with the entity, equal to the number of shape functions defined for it. Off-diagonal blocks for every neighbour are of dimensions *nxm*, where *m* depends on the order of approximation of neighbouring entity. The part of right hand side vector associated with DOF entity is also allocated.

3. Next, the preconditioner blocks are allocated. They will be later factorized. In the simplest case, diagonal blocks are allocated for every DOF entity. In more advanced case, blocks are bigger and contain entries from patches of neighbouring DOF entities.

Solve phase:

4. Blocks are zeroed.
5. For each element and each boundary face, problem module is queried for its stiffness matrix and right hand side vector. Entries are then distributed over previously allocated blocks.
6. Diagonal blocks in preconditioner structure are filled and then factorized.
7. Restarted GMRES loop is started.
    a) In every iteration residual of the left preconditioned system of equations $\mathbf{r_i = M^{-1}(b-Ax_{i-1})}$ is calculated in Block Storage function, called *compreres*, using the data in matrix blocks and preconditioner blocks structures.
8. After GMRES converges, solution is written back into problem module structures.

The key to the success of an iterative method is effective implementation of the fourth step in solution phase, where the residual of the preconditioned system, with improved condition number, is calculated. In simple Schwarz preconditioners, matrix $\mathbf{M^{-1}}$ is not formed explicitly, instead LU-decomposed in setup phase blocks of **A** are used. In a loop (or possibly several loops) over all preconditioner blocks, suitable small systems are solved.

Similarly in the case of multi-level preconditioning we would have each level matrix factorized and then sequence of smoothings of the error for every level in (7a) (using restriction and prolongation operators to transfer information between levels).

As mentioned in section 3, preconditioner blocks can be single Dia blocks associated with a single DOF entity (Block type 1 in our code) or larger blocks covering also neighbours for stronger preconditioning (Block type 2). Note that increasing the strength with bigger blocks extends factorization time needed to process the matrix and the calculation of residual in *compreres*. The ability to configure the size of patches knowing the neighbourhood structure of the problem is something that distinguishes our storage scheme from standard compressed storages.

Computations of the left preconditioned system of equations take place in *compreres* function consisting of calculating the residuum $\mathbf{r_{i-1} = b-Ax_{i-1}}$ and solving the system with preconditioner matrix $\mathbf{r_i = M^{-1}r_{i-1}}$ are performed in a loop over matrix stripes (DOF entities). For each stripe all its elementary blocks are multiplied by elementary vectors of global solution vector with *Nb* calls to *dgemv*, where *Nb* is the number of blocks in stripe. Then one small problem is solved with one call to *dgetrs* LAPACK routine (blocks processed earlier, in setup phase, with *dgetrf* LAPACK routine for LU-decomposition, are used).

5. Computational complexity

We focus on the complexity of compreres function, as it is the main component of iterative loop. Remaining operations outside of compreres like *daxpy* are much less complex and can be ignored. Gram–Schmidt orthonormalization complexity depends on the count of Krylov vectors and can also be ignored, as with restarted GMRES this count is low.

For each iteration, we have one call to compreres which for single-level block Gauss-Seidel preconditioner can be described by the following simplified pseudo-code with annotations concerning computational complexity:

```
for each DOF entity
{
    for each neighbour (Aux block)
```

```
        ddia*daux*{2 flops}        //dgemv
    +
    ddia*{2 flops}                 //daxpy
    +
    ddia*(ddia-1)*{2 flops}        //dgetrs
}
```

Number of floating-point operations is equal to:

$$2 \cdot \left(\sum_{i=1}^{ndofent} \left(ddia(i) \cdot \sum_{j=1}^{nngb(i)} daux(j) \right) + \sum_{i=1}^{ndofent} ddia(i) + \sum_{i=1}^{ndofent} ddia(i) \cdot (ddia(i) - 1) \right)$$

Where:

ddia - number of entity's degrees of freedom (dimension of Dia block),
daux - number of neighbour's degrees of freedom (columns of Aux block),
ndofent - number of DOF entities,
nngb - number of entity's neighbours.

Simplifying:

$$2 \cdot ndofent \cdot (avgnngb \cdot avgsaux + avgsdia)$$

Where:

avgnngb – average number of neighbours of DOF entity,
avgsaux – average size of Aux block,
avgsdia – average size of Dia block.

Average number of DOF entity's neighbours saturates for p-adapted problems where most of mesh entities have basis functions assigned and can be treated as a constant dependent on geometry (i.e. types of finite elements used) and dimension of the mesh. H-adaptation increases *ndofent* since new vertices are added and, when is applied mainly for elements with low order *p*, decreases *avgsaux* and *avgsdia* (new blocks are small). P-adaptation does not affect *ndofent* (at least after certain point in p-adaptation process) and increases *avgsaux* and *avgsdia* since blocks that belong to faces/edges and elements grow as new basis functions are added.

Cumulative effect of hp-adaptation on number of operations strongly depends on the kind of problem solved and the fact whether p adaptations or h adaptations dominate.

Iterative methods for sparse systems consume much less memory than direct methods. Required size, equal to the complexity of single iteration of the method, is limited to a small multiple of the size occupied by the non-zero terms of the matrix, the size of which is of the same order of magnitude as the number of unknowns (as the number of blocks in a single matrix stripe is limited by the geometry of the mesh). As a result, iterative methods have the linear memory complexity allowing, in most cases, the implementation of algorithms "in-core" - as opposed to direct methods [3].

Block storage is optimal as only non-zero elements of the matrix are stored. It also improves memory management during matrix operations (less indirect addressing), and contributes to a better use of the cache. Due to contiguous block storage, we achieve register blocking for lower *p* and cache blocking for higher *p*.

6. Experimental hp problems analysis and results

In Table 1, we present some data for our test problem: Laplace equation on 2D L-shaped domain (in table, middle nodes is a shorthand notation for DOF entities being interiors of elements). As a FEM code we used hp-FEM 2D [5] with built-in automatic hp- and h-adaptation procedures. An initial mesh was first hp-refined to 34193 degrees of freedom and then to 590507 DOFs. For comparison, we also performed h-adaptation, of the same initial mesh, to 29219 DOFs with mostly linear approximation and some edges and elements fixed at $p=2$.

Solver was configured to work sequentially, with block Gauss-Seidel (multiplicative Schwarz) preconditioner. Two types of preconditioner blocks were used: Block type 1 (single DOF entity Dia blocks) and Block type 2 (Dia blocks of an entity and its neighbours). Performance achieved is summarized in Table 2.

As expected increasing the total number of degrees of freedom with hp-adaptation brought increases in the maximum and average sizes of Dia and Aux blocks as DOF entities (edges and elements) received additional basis functions with increasing average order p. Increase in Dia blocks number (i.e. number of DOF entities) was due to h part of hp-adaptation creating new elements and also p part assigning basis functions to those edges and elements that previously did not provide degrees of freedom. For h-adapted problem the total number of DOFs was equal to number of matrix stripes, as all DOF entities had only one associated basis function (thus the dimension of all Dia and Aux blocks was 1).

Maximum, minimum and average number of neighbours was similar across all three versions of the problem, as it depends on dimension of a problem and types of elements used and not on the increasing order of approximation (although it would be smaller for uniform linear approximation where edges and elements' interiors do not provide basis functions and are not counted as neighbours).

Distribution of number of degrees of freedom per DOF entity (i.e. dimension of Dia block and stripe height) is presented in Fig. 4 for both hp-adapted variants. One can observe as peaks move to the right with further hp-adaptation. Nevertheless, the number of small blocks associated with vertices is still large compared with number of bigger blocks associated with higher order entities. Maximum number of dofs of single DOF entity (element's interior) is 36 and 64 for 34193 dofs and 590507 dofs problems respectively.

Performance of single GMRES iteration (*compreres* function) increased with increase in average order p as expected. This is because bigger blocks (stored contiguously in memory) provide more work for *dgemv* and *dgetrs* functions per single transfer from memory to cache (see "Time per single floating-point op." in Table 2). On the contrary, with small blocks, cache misses between short calls to *dgemv* and *dgetrs* dominate which is manifested in order-of-magnitude longer time per single floating-point operation in low order h-adapted problem. For example, when storing matrix in CRS format one could expect better performance with blocks associated with vertices, as those would be stored contiguously. However, for bigger blocks, performance would be worse as many (at least the number of height of a block) memory-cache transfers would be necessary

Compreres execution (and thus GMRES iteration) time scales almost linearly with increasing number of degrees of freedom, provided that ratio between small and large blocks is kept relatively constant, as it happens in our hp-adaptation. If one switched from hp to h adaptation while increasing the number of DOFs, *compreres* times would divert from linear increase due to the negative effect of smaller blocks on performance.

Unfortunately, single-level BGS preconditioner cannot keep the same number of GMRES iterations for the bigger problem. It is obvious that better preconditioning is needed in order to sustain linear scalability achieved by *compreres*. For Block type 1 there is 4 times increase in number of iterations. Situation is slightly better for Block type 2 (2.8 times increase), although this gain is diminished by increased execution times of *compreres*.

Table 1. hp- and h-adapted problems details

Degrees of freedom	**34193 (hp)**	**590507 (hp)**	**29219 (h)**
Middle nodes providing dofs:	1457	15536	1645
Dofs from middle nodes:	21755	412224	1645
Edges providing dofs:	2202	24276	20326
Dofs from edge nodes:	11692	169542	20326
Vertex count:	746	8741	7248
Stripe max summed blocks width:	323	523	33
Stripe min summed blocks width:	10	21	6
Stripe average summed blocks width:	75,82	111,2	12,58
Max stripe height (dofs/DOF ent.; Dia dim,):	36	64	1
Average stripe height (related to average *p*):	7,76	12,16	1
Max non-zeroes in stripe:	2304	6400	33
Min non-zeroes in stripe:	10	51	6
Average non-zeroes in stripe:	434,62	943,11	12,58
Max blocks count in stripe (= neighbours + 1):	37	37	33
Min blocks count in stripe (= neighbours + 1):	7	7	6
Average blocks count in stripe:	13,38	12,99	12,58
Average neighbours count in stripe:	**12,38**	**11,99**	**11,58**
Non-zeroes total:	1914517	45790643	367575
Max entries in Dia block:	1296	4096	1
Average entries in Dia block:	**112,33**	**316,31**	**1**
Max entries in Aux block:	216	512	1
Average entries in Aux block:	**26,04**	**52,28**	**1**
Dia blocks count:	4405	48553	29219
Aux blocks count:	54524	582056	338356
Total entries in Dia blocks:	494837	15357743	29219
Total entries in Aux blocks:	1419680	30432900	338356

Table 2. Solver performance

Degrees of freedom	**34193 (hp)**	**590507 (hp)**	**29219 (h)**
BLOCK TYPE 1			
GMRES iterations:	47	189	117
Total time spent in compreres [s]:	0,44841	33,55433	0,92562
Time per single compreres call [s]:	0,00954	0,177	0,00791
Floating-point operations per compreres call:	3828890	91585252	735150
Time per single floating-point op.: [s]	2,492e-9	1,939e-9	1,076e-8
Flops (of compreres):	**402 MFLOPS**	**516 MFLOPS**	**93 MFLOPS**
BLOCK TYPE 2			
Num iterations:	17	48	56
Total time spent in compreres [s]:	2,54866	126,45591	5,15513

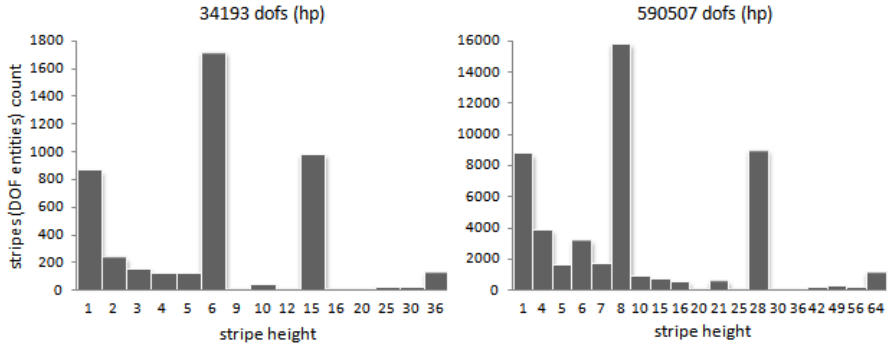

Fig. 4. Matrix stripes height (dofs/DOF entity) distribution

7. Conclusions

In this article, we presented the considerations to be taken into account when creating a solver for hp-adaptation problems. To ensure linear scalability of a solver it is necessary to use a preconditioner that fixes number of iterations with increasing number of degrees of freedom and scalable implementation of a single iteration of GMRES. Critical to ensure optimum performance is using a block matrix storage scheme.

We plan to implement multigrid type of preconditioner. We believe that it will help to reduce the number of iterations, thus providing nearly linear scalability. In order to provide better performance for a single iteration it seems significant to accelerate calculations for small blocks - for example, by grouping them into contiguous memory regions, which should result in cache blocking.

Acknowledgements

The work has been supported by Polish National Science Center grants NN 519 447739 and DEC-2011/01/B/ST6/00674.

References

[1] A. Dedner, R. Klofkorn, M. Nolte, M. Ohlberger, A Generic Interface for Parallel and Adaptive Scientifc Computing: Abstraction Principles and the DUNE-FEM Module, Computing 90 (3-4) (2010) 165-196.
[2] M. Paszynski, D. Pardo, C. Torres-Verdin, L. F. Demkowicz, V. M. Calo, A parallel direct solver for the self-adaptive hp finite element method, J. Parallel Distrib. Comput. 70 (3) (2010) 270-281.
[3] M. Paszynski, D. Pardo, A. Paszynska, L. F. Demkowicz, Out-of-core multi-frontal solver for multi-physics hp adaptive problems, Procedia CS 4 (2011) 1788-1797.
[4] K. Banas, On a modular architecture for finite element systems. I. Sequential codes, Computing and Visualization in Science 8 (2005).
[5] L. Demkowicz, Computing with hp-Adaptive Finite Elements, Vol. 1: One and Two Dimensional Elliptic and Maxwell Problems, Chapman and Hall/CRC, 2006.
[6] K. Banas, M.F.Wheeler, Preconditioning GMRES for discontinuous Galerkin approximations, Computer Assisted Mechanics and Engineering Sciences 11 (2004) 47-62.
[7] K. Banas, Scalability analysis for a multigrid linear equations solver, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics, Proceedings of VIIth International Conference, PPAM 2007, Gdansk, Poland, 2007, Vol. 4967 of Lecture Notes in Computer Science, Springer, 2008, pp. 1265-1274.
[8] P. Plaszewski, M. Paszynski, K. Banas, Architecture of iterative solvers for hp-adaptive finite element codes, Submitted to CAMES: Computer Assisted Methods in Engineering and Science.