Procedia Computer Science

Volume 80, 2016, Pages 865–874

ICCS 2016. The International Conference on Computational Science

Hybrid direct and iterative solver with library of multi-criteria optimal orderings for *h* adaptive finite element method computations

Hassan AbouEisha[1], Konrad Jopek[3], Bartłomiej Medygrał[3], Mikhail Moshkov[2], Szymon Nosek[3], Anna Paszyńska[4], Maciej Paszyński[3], Keshav Pingali[5*]

[1]Computer Science, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
[2]Applied Mathematics and Computational Science, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
[3]Department of Computer Science, AGH University of Science and Technology, Kraków, Poland
[4]Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, Kraków, Poland
[5]Institute for Computational and Engineering Sciences, The University of Texas at Austin, USA

Abstract

In this paper we present a multi-criteria optimization of element partition trees and resulting orderings for multi-frontal solver algorithms executed for two dimensional h adaptive finite element method. In particular, the problem of optimal ordering of elimination of rows in the sparse matrices resulting from adaptive finite element method computations is reduced to the problem of finding of optimal element partition trees. Given a two dimensional *h* refined mesh, we find all optimal element partition trees by using the dynamic programming approach. An element partition tree defines a prescribed order of elimination of degrees of freedom over the mesh. We utilize three different metrics to estimate the quality of the element partition tree. As the first criterion we consider the number of floating point operations(FLOPs) performed by the multi-frontal solver.

As the second criterion we consider the number of memory transfers (MEMOPS) performed by the multi-frontal solver algorithm. As the third criterion we consider memory usage (NONZEROS) of the multi-frontal direct solver. We show the optimization results for FLOPs vs MEMOPS as well as for the execution time estimated as FLOPs+100*MEMOPS vs NONZEROS. We obtain Pareto fronts with multiple optimal trees, for each mesh, and for each refinement level. We generate a library of optimal elimination trees for small grids with local singularities. We also propose an algorithm that for a given large mesh with identified local sub-grids, each one with local singularity. We compute Schur complements over the sub-grids using the optimal trees from the library, and we submit the sequence of Schur complements into the iterative solver ILUPCG.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.312

Keywords: finite element method; h adaptivity; multi-frontal direct solver; element partition tree; ordering algorithms; ILUPCG

1 Introduction

Multi-frontal solver [1,2] is the state-of-the-art direct solver algorithm for solving sparse systems of linear equations resulting from two dimensional h adaptive finite element method computations. The performance of the multi-frontal solver algorithm depends on the so-called ordering, that can be obtained from element partition tree (see chapter 8 in [3]) prescribing recursive divisions of the computational mesh. The nested-dissections [4] (equally weighted partitions) are optimal only for regular grids. When we switch to adaptive grids, the nested dissections algorithm is no longer optimal. The problem of finding optimal element partition tree defining optimal ordering, is NP-complete, as the problem of finding the minimum fill-in [5]. In this paper we focus on an automatic optimization of the element partition trees using dynamic programming. We augment the results of our previous automatic optimization [6,7,8] by considering three different cost functions.

First, we focus on estimations of the number of floating point operations (FLOPs) performed by the multi-frontal solver algorithm. Second, we focus on estimations of the number of memory transfers (MEMOPS) performed during the factorizations. Third, we focus on estimation on the memory usage (NONZEROS) of the solver. We generate Pareto fronts with optimal element partition trees by using the dynamic programming approach with minimization of FLOPs vs MEMOPS, as well as the execution time of the solver estimated as FLOPs+100*MEMOPS vs the NONZEROS.

Our dynamic programming approach allows us to build a library of optimal element partition trees, for a class of small grids with local singularities. Having a large grid with several local singularities, we run a greedy algorithm that localizes sub grids with local refinements. Finally, for each sub grid we compute Schur complement (we eliminate interior of the sub-grid with respect to its boundary). We do that using the ordering following the element partition tree, as described in chapter 8 of [3], and the GALOIS multi-frontal solver [7], that allows to pass the element partition tree as an input. Having the Schur complement computed, we collect them and call iterative solver ILUPCG from SLATEC library [9]. We show that our approach allows to significantly reduce the number of iterations of the iterative solver, if the local singularities are well separated.

2 Dynamic programming optimization

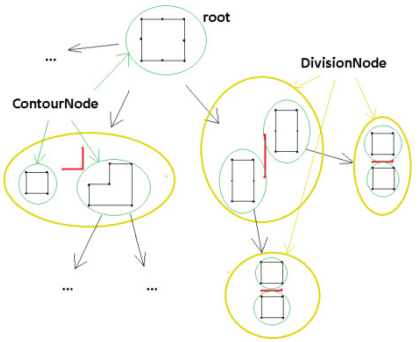


Figure 1. Contour Nodes and Division Nodes of the DAG

Following the ideas already presented in [8], we construct a dynamic programming approach that considers many recursive partitions of the computational mesh. It first constructs a Directed Acyclic

Graph (DAG) (see Figure 1) with the root representing the entire mesh, the son nodes representing possible partitions of the mesh, and leaves representing particular finite elements. Having the DAG we estimate the computational cost of the multi-frontal solver algorithm. The cost at the leaves is equal to the cost of elimination of the interior nodes. The cost at parent nodes is equal to the sum of costs from children nodes plus the cost of elimination of the interface node after merging of the two sub-parts of the mesh. Having the costs computed up to the root, we can go down now and select an optimal element partition tree with minimum cost. An example element partition tree for the mesh with point singularity located in the corner is presented in Figure 2. For more details we refer to [8].

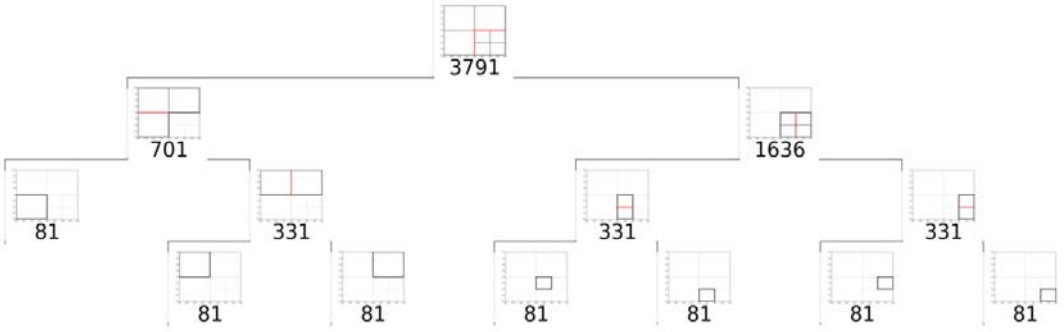


Figure 2. Example of optimal element partition tree for a mesh with point singularity

The same DAG can be used for construction of the set of Pareto optimal points (Pareto front) for bi-criteria optimization of element partition trees with respect to a chosen pair of criteria. In this paper we utilize three different optimization criteria:

- The minimization of the number of floating point operations (FLOPs) performed during the factorization
- The minimization of the number of memory transfers (MEMOPS) performed by our implementation of the GALOIS multi-frontal solver working based on the element partition tree. Notice that the implementation of the mesh-based solver [3] is different from the implementation of the traditional multi-frontal solver like MUMPS solver [11] working based on the sparse matrix, without the knowledge about the structure of the mesh.
- The minimization of the memory usage (NONZEROS) utilized by the mesh-based solver.

We have executed the following numerical experiments. First, we computed the Pareto front for the bi-criteria optimization with respect to FLOPs vs MEMOPS for different meshes and for different refinement levels. The Pareto fronts computed for example grids with point and edge singularities are presented in Figures 3 and 4.

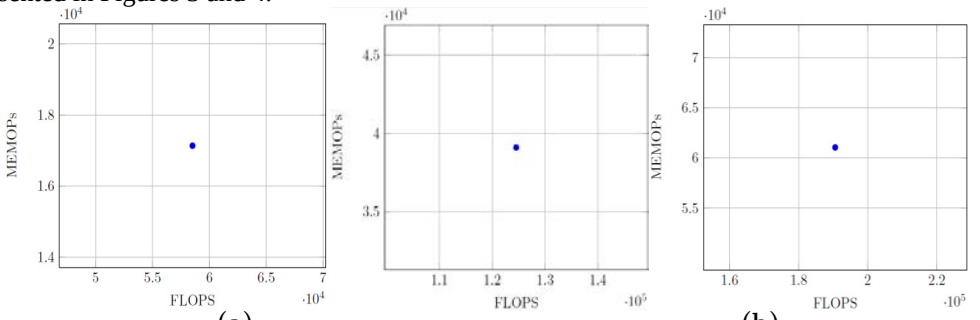


Figure 3. Performance of bi-criteria optimal element partition treees for mesh with central point singularity. Panels (a), (b) and (c) correspond to different refinement levels, from (a)=3, (b)=6, to (c)=9.

For the central point singularity there is only one Pareto optimal point and only one type of bi-optimal element partition trees. For the edge singularity we have the entire Pareto front with many points and many types of optimal element partition trees. Next, we have combined the FLOPs+100*MEMOPS to reflect execution time following the observation that memory transfer in average takes 100 more time than single FLOP.

We then performed the triple-optimization of element partition trees with execution time (FLOPs+100*MEMOPS) vs NONZEROS. For all the grids we obtained Pareto fronts with several optimal trees, presented in Figures 5 and 6. Finally, in Figures 7 and 8 we present exemplary triple criteria optimal element partition trees, that provide minimum execution time. We have generated the tri-criteria optimal element partition trees for different number of refinement levels for different grids with local singularities, we choose the minimum execution time trees and stored them in the element partition trees library.

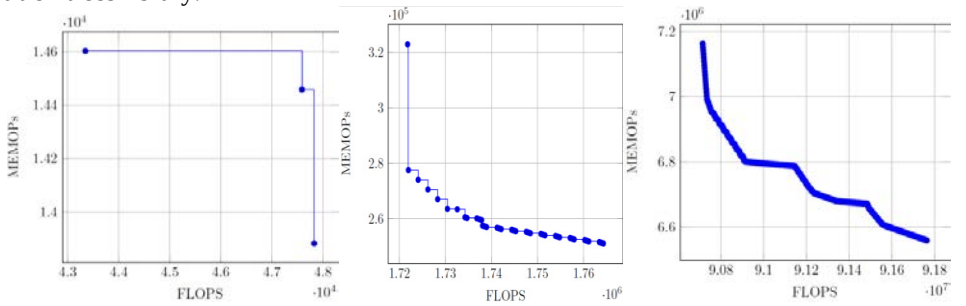


Figure 4. Performance of pareto fronts with optimal grids for mesh with edge singularity. Panels (a), (b) and (c) correspond to different refinement levels, from (a)=3, (b)=6 to (c)=9.

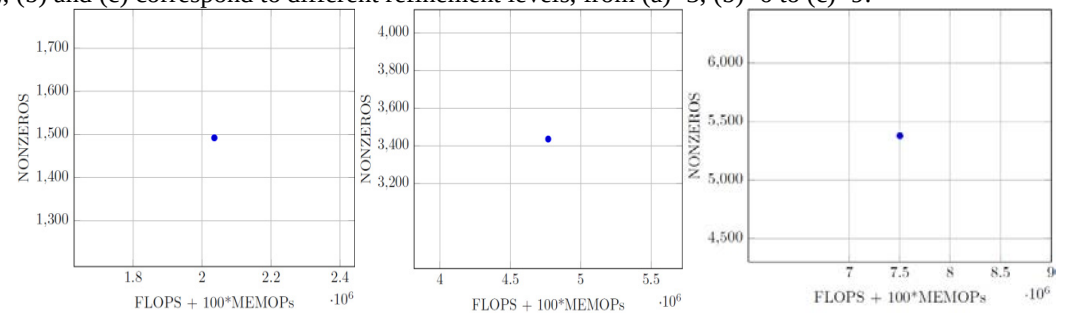


Figure 5. Performance of tri-criteria optimal element partition trees for mesh with central point singularity. Panels (a), (b), and (c) correspond to different refinement levels, from (a)=3, (b)=6 to (c)=9.

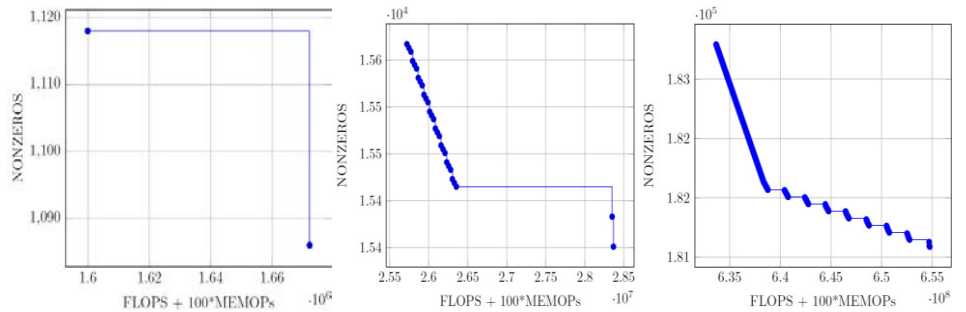


Figure 6. Performance of tri-criteria optimal element partition trees for mesh with edge singularity. Panels (a), (b), and (c) correspond to different refinement levels, from (a)=3, (b)=6 to (c)=9.

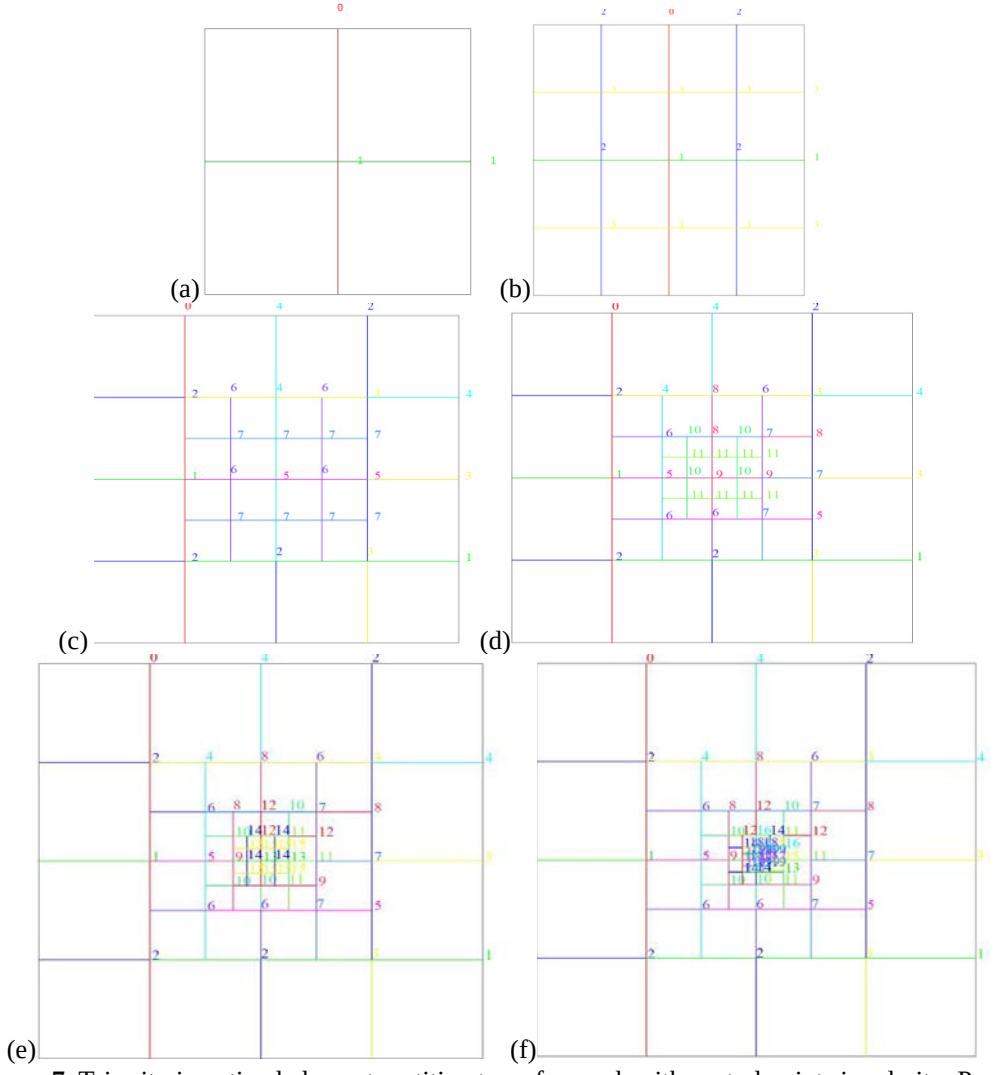
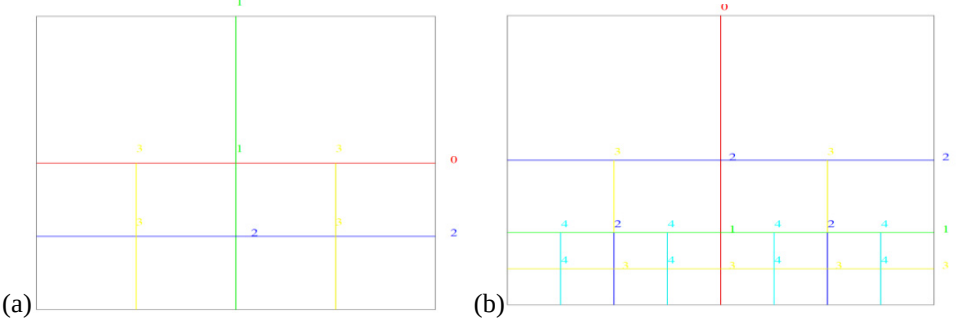


Figure 7. Tri-criteria optimal element partition trees for mesh with central point singularity. Panels (a)-(f) correspond to different refinement levels, from (a)=1 to (f)=6.



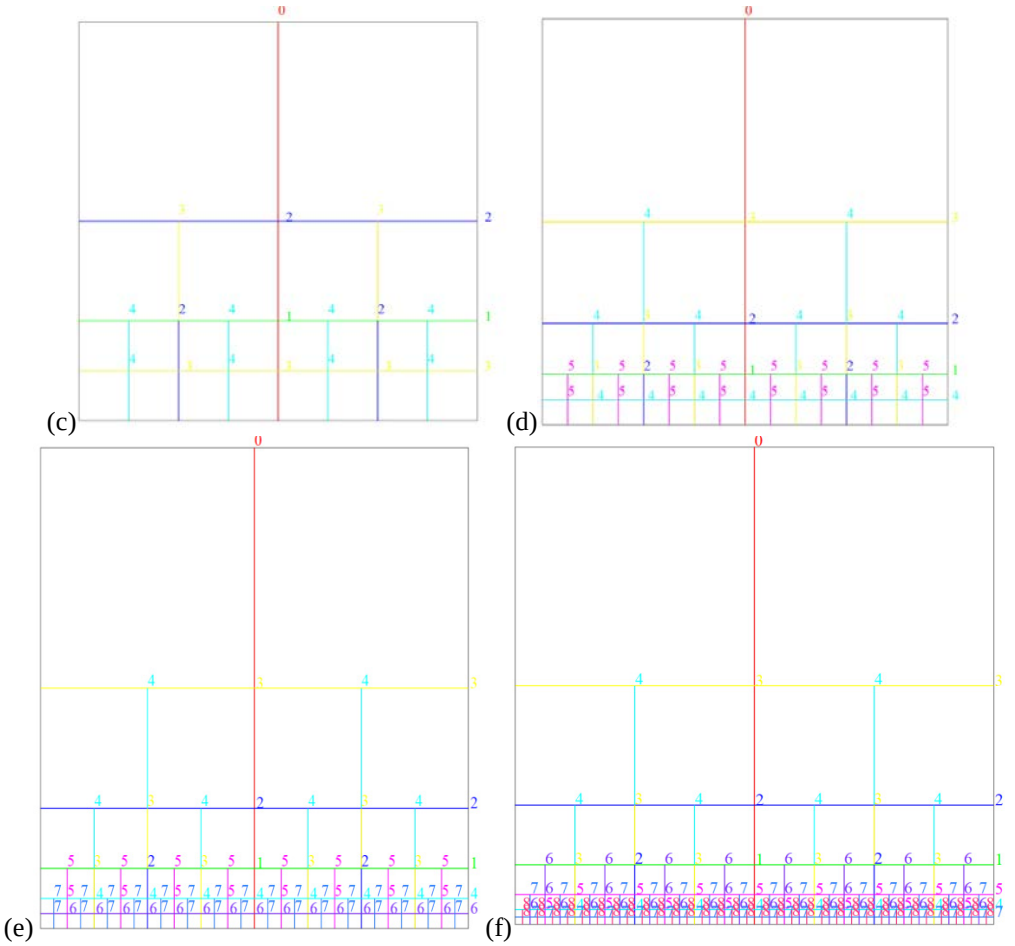


Figure 8. Tri-criteria optimal element partition trees for mesh with edge singularity. Panels (a)-(f) correspond to different refinement levels, from (a)=1 to (f)=6.

3 Identification of candidates for local singularities

In this section we propose an algorithm for the localization of candidates for local singularities. The algorithm obtains a mesh that has been already *h* refined a few times by a classical mesh adaptation algorithm. The algorithm works under the following assumptions:

- The initial computational mesh is two dimensional, and consists of rectangular finite elements.
- The initial finite elements are numbered topologically, from left to right, from top to bottom.
- The refined mesh is obtained by performing a sequence of isotropic refinements from the initial mesh.
- When constructing the *h* refined mesh, only isotropic *h* refinement is allowed. Each element of the initial mesh has assigned refinement level equal to 1.
- When the adaptive algorithm breaks an element into four children, the refinement level of the children
  is equal to the refinement level of the parent element plus one.

- The mesh fulfils the one-irregularity rule, which specifies that an element edge can be broken only once without breaking adjacent large element.

```
1 Create list of n initial mesh element trees sorted
   according to highest level of edges on the elements boundary
2  Store neighbors of each element
3  repeat
4     Select a sub-list with highest level of edges on the patch
boundaries
5    loop through adjacent pairs v and w from the sub-list
6    Merge v and w into a new patch r
7      Remove v and w from the list
8    Merge neighbors of v and w to new list for r
9    Add r to the forest in such a way that
       the list is sorted according to highest level of boundary
edges
10   end loop
11 until all the boundary edges are at first or second refinement
level
```

The execution of the algorithm on the mesh with two point singularities over an initial mesh elements is presented in Figure 9, and summarized below:

1. The refined initial mesh elements are sorted according to the maximum refinement level of the edges on their boundary, from the largest to the smallest, see step (a) in Figure 9, where some elements have edges from the fifth refinement level, and the other have edges from the second level.
2. We select the sub-list with highest level refinement for edges located on the boundary, in this case this is the fifth level, compare step (a) in Figure 9.
3. We join adjacent elements with edges from the fifth level into pairs, see step (b) in Figure 9.
4. We sort the list again, compare step (b) in Figure 9, where some elements have edges from the fifth refinement level, and the others have edges from the second level.
5. We select again the sub-list with highest refinement level, see step (b) in Figure 9.
6. We join adjacent elements with edges from the fifth level into pairs, see step (c) in Figure 9
7. We sort the list again, see step (c) in Figure 9.
8. This time all the edges from the boundary of patches of elements have second refinement level.

The algorithm has identified two patches with point singularities, and the other initial mesh elements remained untouched. After having all the candidates for point singularities selected, we execute a hybrid algorithm that can be summarized in the following way:

- identify the local singularity and find its optimal element partition tree in the library generated by the dynamic programming approach,
- perform static condensation over local singularity with the element partition tree from the library,
- perform static condensation for other initial mesh elements,
- collect resulting Schur complements and call ILUPCG solver.

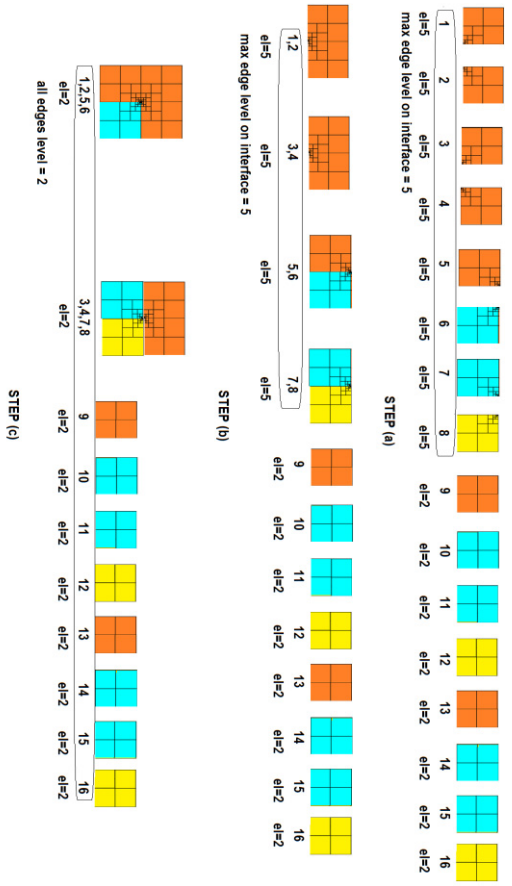


Figure 9. Execution of the algorithm for localizing of submeshes with point singularities on a 2D mesh that initially contained 16 elements (4x4 configuration) with two point singularities

4 Numerical Results

We have executed the algorithm for identification of local singularities on two grids presented in Figure 10. Later, we take the triple-criteria element partition trees from the library of trees and compute the Schur complements with respect to the boundary. Next, we run the ILUPCG iterative solver from the SLATEC library [20] on the collected Schur complements.

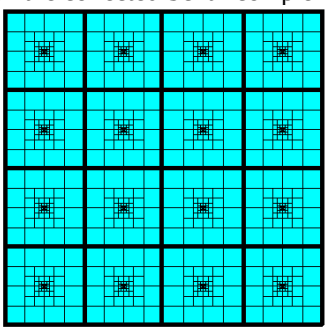


Figure 10. Identification of local singularities over the 4x4 point singularities.

We compare the resulting number of iterations of the ILUPCG algorithm executed over the collection of the Schur complements, with the number of iterations of the ILUPCG algorithm executed over the entire mesh, with element static condensation only. The computations of the local Schur complements are performed by GALOIS direct solver executed for local singularities with element partition trees taken from the library.

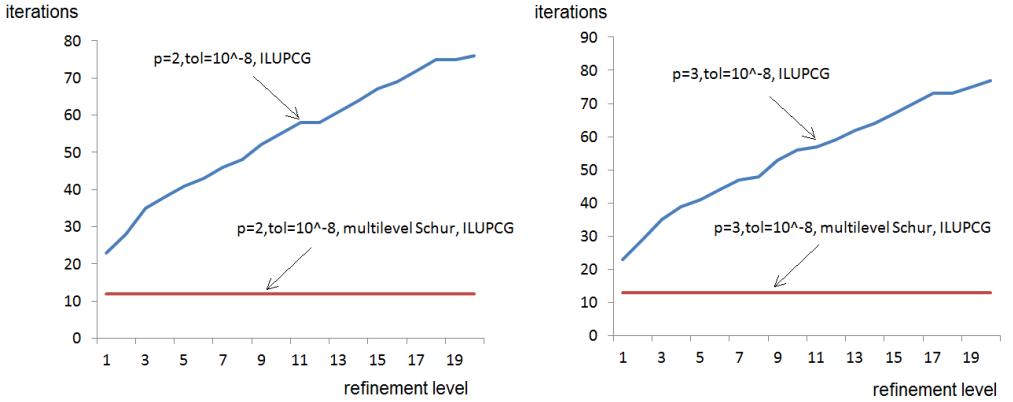


Figure 11. Left panel: Comparison on the number of iterations of ILUPCG solver for two dimensional mesh with quadratic polynomials and 4x4 singularities, for hybrid and standard algorithm. **Right panel:** Comparison on the number of iterations of ILUPCG solver for two dimensional mesh with cubic polynomials and 4x4 singularities, for hybrid and standard algorithms.

Table 1. Number of iterations of ILUPCG for the mesh from panel (a) in Figure 12

# refinements	1	2	3	4	5	6
#iterations	43	59	61	72	81	124

5 Conclusions

The methodology presented in this paper allows to speedup iterative solvers by computing Schur complements over local singularities and solving the top problem with iterative solver ILUPCG or similar one, with constant number of iterations. We have a library of optimal element partition trees for typical configurations of singularities that speed up the Schur complement process. However, for the grids partitoned like the one presented in Figure 12, we have a large edge singularity and cutting it into pieces like on panel (a) does not reduce the number of iterations of ILUPCG. Actually the number of iterations is reverse proportional to the smallest element diameter, and since the edge singularity has small elements on the interface, we observe growth in the number of iterations, presented in Table 1. To solve this problem and have constant number of iterations we need to defragment the mesh into big sub-mesh with the entire edge singularity, like the one on panel (b). So we need our greedy algorithm for the separation. However, for the large sub-grid we do not have optimal element partition tree in our library, and we need to run one of the heuristic algorithms [12,13,14,15,4,6] and compute Schur complement with quasi-optimal ordering.

Acknowledgement

The work presented in this paper has been supported by National Science Centre, Poland grant no. DEC-2015/17/B/ST6/01867 and by King Abdullah University of Science and Technology (KAUST).

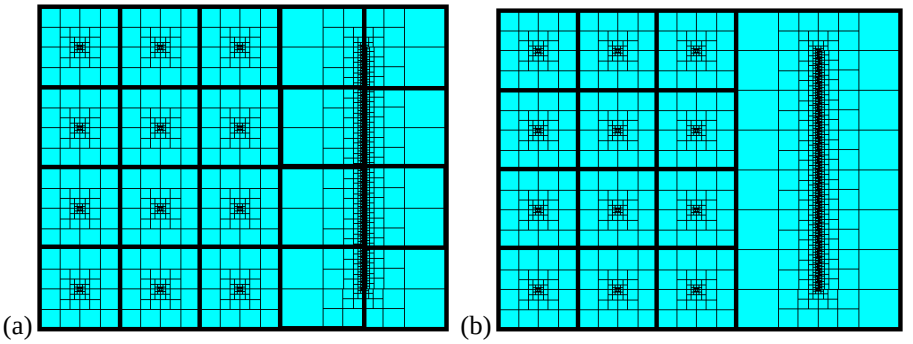


Figure 12. Identification of local singularities over the mesh 3x4 point singularities and one edge singularity

References

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

[2] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent and S. Pralet, Hybrid scheduling for the parallel solution of linear systems. Parallel Computing 32(2) 136-156 (2006)

[3] M. Paszyński, Fast Solvers for Mesh Based Computations, Taylor and Francis, CRC Press (2016)

[4] A. George, J.W.-H. Liu, An automatic nested dissection algorithm for irregular finite element problems. SIAM Journal of Numerical Analysis 15, 1053-1069 (1978)

[5] M. Yannakakis, Computing the minimum fill-in is NP-complete, SIAM Journal on Algebraic Discrete Methods 2, 77-79 (1981)

[6] Paszyńska A., Paszyński M., Jopek K., Woźniak M., Goik D., Gurgul P., AbouEisha H., Moshkov M., Calo V. M., Lenharth V. M., Nguyen D., Pingali K., Quasi-Optimal Elimination Trees for 2D Grids with Singularities, Scientific Programming, Volume 2015 Article ID 303024:1-18 (2015)

[7] H. AbouEisha, V. M. Calo, K. Jopek, M. Moshkov, A. Paszynska, M. Paszynski, M. Skotniczny, Optimization of element partition trees for two dimensional h refined meshes, submitted to Computers and Mathematics with Applications

[8] H. AbouEisha, M. Moshkov, V. Calo, M. Paszynski, D. Goik, K. Jopek, Dynamic Programming Algorithm for Generation of Optimal Elimination Trees for Multi-frontal Direct Solver Over h-refined Grids, Procedia Computer Science, 29, 947-959 (2014)

[9] SLATEC Common Mathematical Library http://www.netlib.org/slatec/

[10] Intel® 64 and IA-32 Architectures Optimization Reference Manual

[11] MUlti-frontal Massively Parallel Sparse direct solver MUMPS, http://mumps.enseeiht.fr/

[12] J. Schulze, Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods, BIT, 41, 4 (2001)

[13] P. Heggernes, S.C. Eisenstat, G. Kumfert, A. Pothen, The Computational Complexity of the Minimum Degree Algorithm, ICASE Report No. 2001-42, (2001).

[14] P. R. Amestoy, T. A. Davis, I. S. Du, An Approximate Minimum Degree Ordering Algorithm, SIAM Journal of Matrix Analysis & Application, 17, 4 (1996) 886-905.

[15] G.W. Flake, R.E. Tarjan, K. Tsioutsiouliklis, Graph clustering and minimum cut trees, Internet Mathematics 1 (2003), 385-408.