Applied Mathematics, 2015, 6, 304-311
Published Online February 2015 in SciRes. http://www.scirp.org/journal/am
http://dx.doi.org/10.4236/am.2015.62028

Scientific Research Publishing

Schur Complement Computations in Intel® Math Kernel Library PARDISO

Alexander Kalinkin, Anton Anders, Roman Anders

Intel Corporation, Software and Services Group (SSG), Novosibirsk, Russia
Email: alexander.a.kalinkin@intel.com, anton.anders@intel.com, roman.anders@intel.com

Received 11 January 2015; accepted 29 January 2015; published 5 February 2015

Abstract

This paper describes a method of calculating the Schur complement of a sparse positive definite matrix *A*. The main idea of this approach is to represent matrix *A* in the form of an elimination tree using a reordering algorithm like METIS and putting columns/rows for which the Schur complement is needed into the top node of the elimination tree. Any problem with a degenerate part of the initial matrix can be resolved with the help of iterative refinement. The proposed approach is close to the "multifrontal" one which was implemented by Ian Duff and others in 1980s. Schur complement computations described in this paper are available in Intel® Math Kernel Library (Intel® MKL). In this paper we present the algorithm for Schur complement computations, experiments that demonstrate a negligible increase in the number of elements in the factored matrix, and comparison with existing alternatives.

Keywords

Multifrontal Method, Direct Method, Sparse Linear System, Schur Complement, HPC, Intel® MKL

1. Introduction

According to F. Zhang [1], the term "Schur complement" was used first by E. Haynsworth [2]. Haynsworth chose this term because of the lemma (Schur determinant lemma) in the paper [3] that was edited by Schur himself. In spite of matrix $A - BD^{-1}C$ being used in this lemma as a secondary term, later this matrix came to play an important role in mathematical algorithms as the Schur complement. It is denoted as $\left(A\middle|D\right) = A_{loc} - BD^{-1}C$. For example, in mathematical statistics, the Schur complement matrix is important in computation of the probability density function of multivariate normal distribution, and in computational mechanics the Schur complement matrix correlates to media stiffness.

Partial solving of systems of linear equations plays an important role in linear algebra for implementation of

How to cite this paper: Kalinkin, A., Anders, A. and Anders, R. (2015) Schur Complement Computations in Intel® Math Kernel Library PARDISO. *Applied Mathematics*, **6**, 304-311. http://dx.doi.org/10.4236/am.2015.62028

efficient preconditioners based on domain decomposition algorithms. Partial solutions usually involve sparse matrices. For this reason Schur complement computations and partial solving have been implemented in Intel® Math Kernel Library (Intel® MKL) [4]. This paper covers the ideas behind the implementation.

There are a number of papers that focused on efficient implementation of the Schur complement. As example, Aleksandrov and Samuel [5] in their paper proposed algorithm to calcluate the Schur complement for Sparse system. Yamazaki and Li published an idea [6] of how to implement Schur complement efficiently on cluster. And we need to mention MUMPS solver [7] that integrated the Schur complement computation a few years ago.

Intel® MKL PARDISO [4] can be considered as one of the multifrontal methods that have been proposed by Duff [8] and further expanded by Liu [9]. This method is divided into three stages. First, the initial matrix undergoes a reordering procedure like the one developed by Karypis [10] [11] in order to represent it in the form of a dependency tree. Then symbolic factorization takes place, where the total number of nonzero elements is computed in LDU decomposition. And finally, factorization of the permuted matrix in the LDU form is performed like the factorization proposed in Amestoy [12]-[16]. In the last stage, both forward and backward substitutions are implemented to compute a solution for the two triangular systems.

The proposed implementation of the Schur complement continues the work of the authors in the area of multifrontal direct sparse solvers. In Kalinkin [17], the basic algorithm was implemented for symmetric, positive definite matrices. In the presentations [18] and [19], the proposed algorithm was significantly improved by balancing the dependency tree. In [20], the algorithm was expanded to non-positive definite matrices and non-symmetric matrices. In this paper, we propose to move all matrix elements that correlate to Schur complement to the top of the dependency tree in order to improve parallelization of computations.

Let A be a symmetric positive definite sparse matrix (the symmetry and positive definiteness of the matrix is set in order to simplify the algorithm description avoiding the case of degenerate matrix minors):

$$A = \begin{pmatrix} A_{loc} & B^{\mathrm{T}} \\ B & C \end{pmatrix}, \tag{1}$$

where A_{loc} and C are square sparse positive definite matrices, and B is a sparse rectangular matrix. Then we can make the following decomposition, which is similar to a Cholesky decomposition of matrix A:

$$A = \begin{pmatrix} L_{11} & 0 \\ L_{12} & I \end{pmatrix} * \begin{pmatrix} I & 0 \\ 0 & S \end{pmatrix} * \begin{pmatrix} L_{11}^{\mathrm{T}} & L_{12}^{\mathrm{T}} \\ 0 & I \end{pmatrix}, \tag{2}$$

where

$$A_{loc} = L_{11}L_{11}^{\mathrm{T}}; \quad B = L_{12}L_{11}^{\mathrm{T}}; \quad S = C - BA_{loc}^{-1}B^{\mathrm{T}}.$$

The matrix $S = \left(A \middle| A_{loc}\right)$ is the Schur complement. The general approach to computing the Schur complement based on this formula and mathematical kernels can be expressed in the form of pseudocode:

Algorithm 1. Simple Schur complement computational algorithm.

1) Calculate decomposition of $A_{loc} = L_{11}L_{11}^{\mathrm{T}}$ with the factorization step of the direct solver;

2) Calculate $B_{\text{temp}} = A_{loc}^{-1}B^{\mathrm{T}}$ with the solving step applied to multiple right-hand sides;

3) Calculate $C_{\text{temp}} = BB_{\text{temp}}$ as sparse-dense matrix-matrix multiplication;

4) Calculate $S = C - C_{\text{temp}}$ as a difference.

This algorithm has several significant disadvantages that can form barriers for its implementation for large sparse systems. The main disadvantage is in the step 2 of **Algorithm 1** involving the conversion of sparse matrix B^{T} into a dense matrix, which requires allocating a lot of memory for storing temporary data. Also, if we consider B^{T} as a dense matrix a large number of zero elements are processed in multiplication $A_{loc}^{-1}\ B^{\mathrm{T}}$, which would make this step one of the most computational intensive parts of the algorithm and would significantly increase the overall computational time. To prevent this, we propose the following algorithm based on the multifrontal approach which calculates the Schur complement matrix first, and then the factorization of the matrix A without significant memory requirements for the computations to proceed.

2. Schur Complement Computational Algorithm

As in the papers [17]-[20], consider a sparse symmetric matrix A_{loc} as in the left of **Figure 1**, where each shaded block is a sparse sub-matrix and each white block is a zero sub-matrix. Using reordering algorithm procedures [10] [11], this matrix can be rotated to the pattern shown in the right of **Figure 1**. A reordered matrix is more convenient for computations than the initial one since Cholesky decomposition can start simultaneously from several entry points (for the matrix on the right of **Figure 1**, the first, second, fourth, and fifth rows of the matrix L can be calculated independently.

Let us append the original matrix A_{loc} stored in the sparse format with zeroes so that its nonzero pattern matches completely that of the matrix L. The elements of L in row 3 can be computed only after the elements in rows 1 and 2 are computed; similarly, element in row 6 can be computed only after elements in rows 4 and 5 are computed. The elements in the 7th row can be computed last. This allows us to construct the dependency tree [10] [11]: a graph, where each node corresponds to a single row of the matrix and each graph node can be computed only if its children (nodes on which it depends) are computed. A deeper discussion of the algorithm with pseudocode of the distribution of nodes of the tree between processes can be found in [17]. The dependency tree for the matrix is given in **Figure 2** (the number in the center of a node shows the row number).

Such a representation allows us to modify **Algorithm 1** using the following notation: node Z_j is a child of Z_i if Z_j resides lower than Z_i in the dependency tree (**Figure 2**) and there is a connection from Z_j to Z_i.

Algorithm 2. LL^T decomposition based on the dependency tree.

```
1)  L = A_loc
2)  for i = 1, number_of_tree_nodes do
3)      Z_i = node of tree;
4)      for all Z_j child of Z_i do
5)          Z_{i,j} = Z_j * mask_i Z_j   prepare update of Z_i by j-th child;
6)          Z_i = Z_i − Z_{i,j} ;
7)      end
8)      Calculate LL^T decomposition of Z_i;
9)  end
```

where by $mask_iZ_j$ we denote a submatrix built as intersection of columns corresponding to node Z_i with rows corresponding to node Z_j. In terms of representation in the right of the **Figure 1** that would mean the ij-th square.

To calculate the Schur complement let us add to the representation in the columns and rows of matrices B, B^T, and C to achieve full representation of matrix A as in left part of the **Figure 3**. As one can see, we achieve similar representation to the **Figure 2** with additional rows corresponding to those of matrices B and C in **Figure 3**

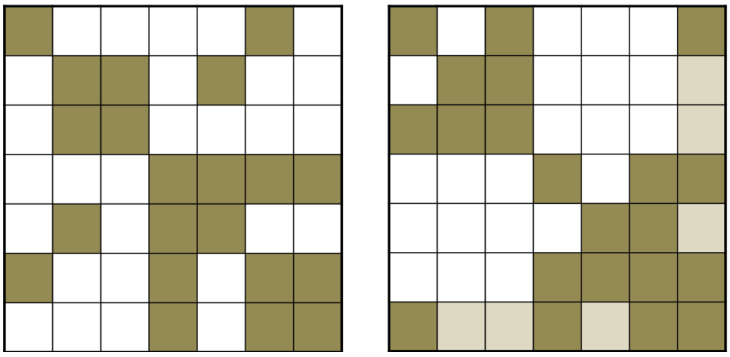

Figure 1. Nonzero pattern of the original matrix (left) and of the same matrix after reordering (right).

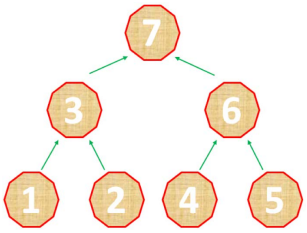

Figure 2. Dependency tree sample.

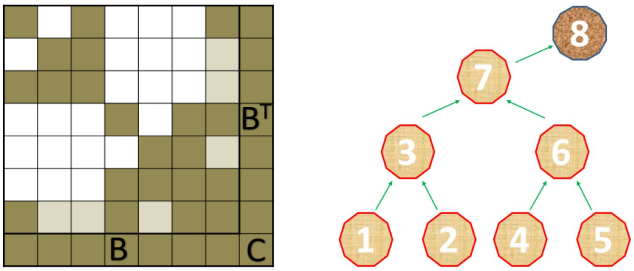

Figure 3. Nonzero pattern of matrix A after reordering of A_{loc} (left) and its tree representation (right).

(right). Note that blocks corresponding to the columns and rows of matrices B^{T}, B, and C are sparse. After factorization of the full matrix A the number of nonzero elements there increases significantly, but our experiments show that the blocks remain sparse and do not become dense.

Let's introduce the following notation: $\tilde{Z}_i$ is Z_i node of the tree expanded by the corresponding rows of the matrix B^{T}, Z_C is a node of the tree corresponding to the matrix C. Then we can modify **Algorithm 2** to take into account the elements of matrices B, B^{T}, and C.

Algorithm 3. LL^{T} decomposition based on the dependency tree.

1) $L = A$;
2) parallel **for** $i = 1$, *number_of_tree_nodes* **do**
3) $\quad \tilde{Z}_i$ = node of tree;
4) $\quad$ **for** *all* $\tilde{Z}_j$ *child of* $\tilde{Z}_i$ **do**
5) $\quad\quad \tilde{Z}_{i,j} = \tilde{Z}_j * mask_i \tilde{Z}_j$ prepare update of $\tilde{Z}_i$ by j-th child;
6) $\quad\quad \tilde{Z}_i = \tilde{Z}_i - \tilde{Z}_{i,j}$;
7) $\quad$ **end**
8) $\quad$ Calculate LL^{T} decomposition of $\tilde{Z}_i$;
9) **end**
10) **for** $j = 1$, *number_of_tree_node* **do**
11) $\quad Z_{C,j} = \tilde{Z}_j * mask_C \tilde{Z}_j$ prepare update of Z_C by j-th child;
12) $\quad Z_C = Z_C - Z_{C,j}$;
13) **end**

This algorithm produces $Z_C = S$. In fact, the **Algorithm 3** fully corresponds to the simple **Algorithm 2** without calculations of the LL^{T} decomposition of the last submatrix.

The approach proposed can be implemented on a parallel computers with a small modification of **Algorithm 3**.

Algorithm 4. Parallel implementation of LL^T decomposition based on the dependency tree.

1) $L = A$;
2) **for** $i = 1$, *number_of_tree_nodes* **do**
3) $\quad \tilde{Z}_i$ = node of tree;
4) $\quad$ **for** *all* $\tilde{Z}_j$ *child of* $\tilde{Z}_i$ **do**
5) $\quad\quad \tilde{Z}_{i,j} = \tilde{Z}_j * mask_i \tilde{Z}_j$ prepare update of $\tilde{Z}_i$ by j-th child;
6) $\quad\quad$ atomic $\tilde{Z}_i = \tilde{Z}_i - \tilde{Z}_{i,j}$;
7) $\quad$ **end**
8) $\quad$ Calculate LL^T decomposition of $\tilde{Z}_i$;
9) **end**
10) parallel **for** $j = 1$, *number_of_tree_node* **do**
11) $\quad Z_{C,j} = \tilde{Z}_j * mask_C \tilde{Z}_j$ prepare update of Z_C by j-th child;
12) $\quad$ atomic $Z_C = Z_C - Z_{C,j}$;
13) **end**

Approach presented in **Algorithm 4** allows us to implement the Schur complement of sparse matrix in Intel® Math Kernel Library.

3. Experiments

For all experiments we used a compute node with two Intel® Xeon® processors E5-2697 v3 (35MB cache, 2.60 GHz) with 64GB RAM, MUMPS version 4.10.0 [7], Intel MKL 11.2 Update 1 [4].

Figure 4 shows a cubic domain in which we apply seven-point approximation for a Laplace operator with mesh size $nx = ny = nz = 70$ to generate matrix A, and its cut-off through one of the axes as a domain for which we want to calculate the Schur complement (**Figure 4** (left)).

Figure 4 shows the portrait of matrix A before factorization (center) and the portrait of matrix L after factorization (right). One can see that the sparsity of L in the Schur complement columns decreased versus the sparsity of the part of L that corresponds to matrix A_{loc}, though it stays sparse and overall the number of nonzero elements increases slightly. For this test, we see that the number of nonzero elements is only five percent higher in the case when we calculate the Schur complement (**Algorithm 3**) compared to the case without Schur complement calculations (straight factorization).

In **Figure 5** and **Figure 6** we compare the performance of the implemented functionality with the similar functionality provided by the MUMPS package [7]. We compare the time needed to compute Schur complement matrix and return it in the dense format. The last 5000 rows and columns of the matrices presented are chosen for Schur complement computations.

For **Figure 5** we chose 2 matrices from Florida Matrix collection [21]: Fault_639 with about 600 K rows and columns and 27 M nonzero elements, and Serena with 1.3 M rows and columns and 64 M nonzero elements. On the x axis we plotted the number of threads on the compute node used for computation of the Schur complement. One can see that the time for computing Schur complement is almost the same for a small number of threads, but the time needed for Intel MKL PARDISO solver decreases when the number of threads increases.

For **Figure 6** we chose 2 matricesfrom Florida Matrix collection [21]: Geo_1438 with about 1.4 M rows and columns and 602 M nonzero elements, and Flan_1565 with 1.5 M rows and columns and 114 M nonzero elements. As before, on the x axis we plotted the number of threads on the compute node used for computation of the Schur complement. Notice that overall picture does not change significantly. The main difference between this set of matrices and the previous one is in sparsity—average number of nonzero elements per row. In the first set of experiments (Fault_639 and Serena) we used sparse matrices with fewer than 50 nonzero elements per

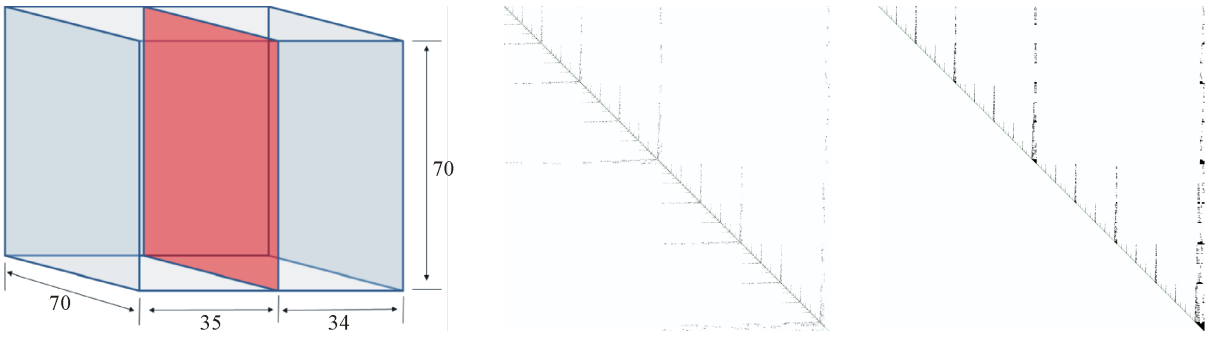

Figure 4. A domain with a dividing plane corresponding to Schur submatrix (left), portraits of the matrix before (center) and after factorization (right).

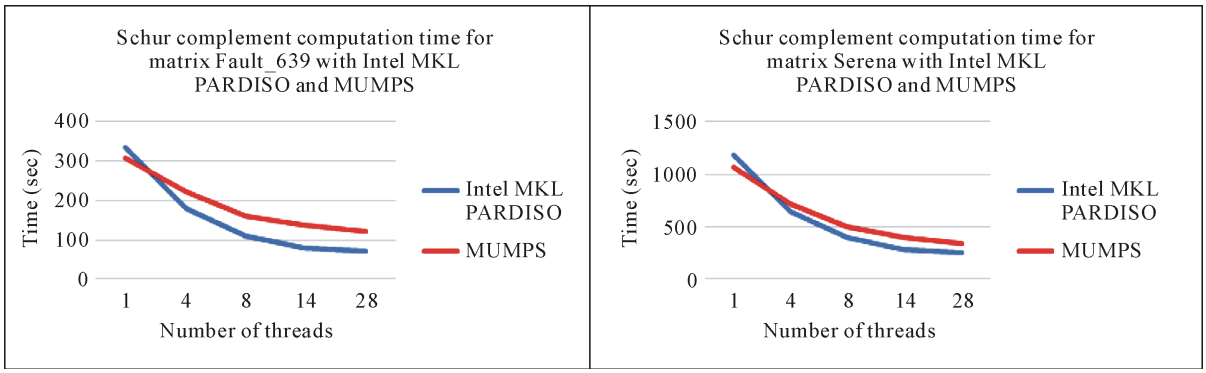

Figure 5. Schur complement computational time for matrices Fault 639 and Serena with Intel MKL PARDISO and MUMPS.

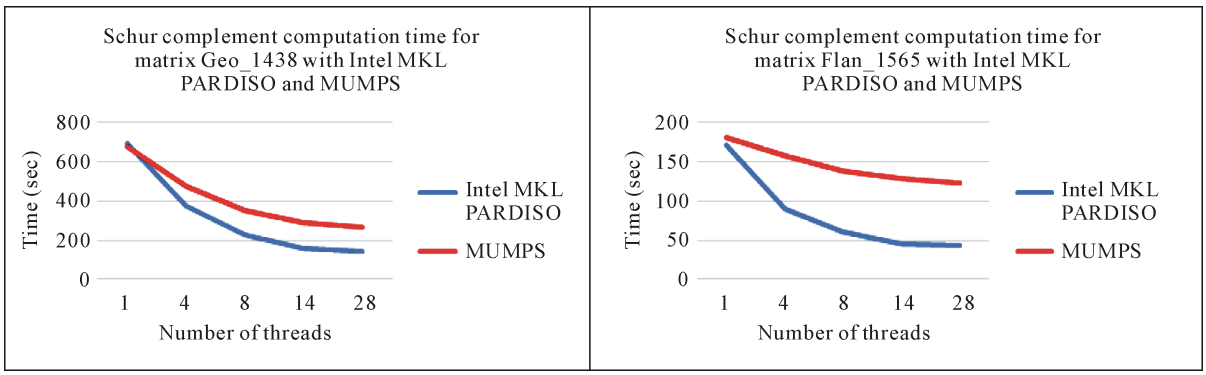

Figure 6. Schur complement computational time for matrices Geo 1438 and Flan 1565 with Intel® MKL PARDISO and MUMPS.

row on average, while the sparsity of Flan_1565 is about 70 nonzero elements per row and the sparsity of Geo_1438 is more than 400 nonzero elements per row. In both cases the time for Schur complement computations is almost the same when the number of threads is small for the Intel MKL and MUMPS, but the time needed for Intel MKL PARDISO solver significantly decreases when the number of threads increases. Moreover, comparison of **Figure 5** and **Figure 6** indicates that the performance of Intel MKL PARDISO becomes better if sparsity increases.

4. Conclusion

We demonstrated an approach that calculates the Schur complement for a sparse matrix implemented in Intel Math Kernel Library using the Intel MKL PARDISO interface. This implementation allows one to use a Schur complement for sparse matrices appearing in various mathematical applications, from statistical analysis to algebraic solvers. The proposed approach shows good scalability in terms of computational time and better performance than similar approaches proposed elsewhere.

References

[1] Zhang, F. (2005) The Schur Complement and Its Applications, Series: Numerical Methods and Algorithms, Vol. 4, Springer, USA.

[2] Haynsworth, E.V. (1968) On the Schur Complement. Basel Mathematical Notes, BMN 20, 17 p.

[3] Schur, I. (1986) On Power Series Which Are Bounded in the Interior of the Unit Circle, Series: Operator Theory: Advances and Applications, Birkhauser, Basel, Vol. 18, 31-59.

[4] Intel Math Kernel Library http://software.intel.com/en-us/intel-mkl

[5] Aleksandrov, V. and Samuel, H. (2010) The Schur Complement Method and Solution of Large-Scale Geophysical Problems. Bayerisches Geoinstitut (BGI). http://karel.troja.mff.cuni.cz/documents/2010-ML-Aleksandrov.pdf

[6] Yamazaki, I. and Li, S.X. (2010) On Techniques to Improve Robustness and Scalability of the Schur Complement Method. 9*th International Conference on High Performance Computing for Computational Science*, Berkeley, 22-25 June 2010, 14 p.

[7] MUMPS http://mumps.enseeiht.fr/

[8] Duff, I.S. and Reid, J.K. (1983) The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Transactions on Mathematical Software*, **9**, 302-325. http://dx.doi.org/10.1145/356044.356047

[9] Liu, J.W.H. (1992) The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Review*, **34**, 82-109. http://dx.doi.org/10.1137/1034004

[10] Karypis, G. and Kumar, V. (1996) Parallel Multilevel Graph Partitioning. *Proceedings of the* 10*th International Parallel Processing Symposium*, Honolulu, 15-19 April 1996, 314-319.

[11] Karypis, G. and Kumar, V. (1998) A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, **48**, 71-85 http://dx.doi.org/10.1006/jpdc.1997.1403

[12] Amestoy, P.R., Duff, I.S., Pralet, S. and Voemel, C. (2003) Adapting a Parallel Sparse Direct Solver to Architectures with Clusters of SMPs. *Parallel Computing*, **29**, 1645-1668. http://dx.doi.org/10.1016/j.parco.2003.05.010

[13] Amestoy, P.R., Duff, I.S. and Vomel, C. (2005) Task Scheduling in an Asynchronous Distributed Memory Multifrontal Solver. *SIAM Journal on Matrix Analysis and Applications*, **26**, 544-565. http://dx.doi.org/10.1137/S0895479802419877

[14] Amestoy, P.R., Guermouche, A., L'Excellent, J.-Y. and Pralet, S. (2006) Hybrid Scheduling for the Parallel Solution of Linear Systems. *Parallel Computing*, **32**, 136-156. http://dx.doi.org/10.1016/j.parco.2005.07.004

[15] Amestoy, P.R. and Duff, I.S. (1993) Memory Management Issues in Sparse Multifrontal Methods on Multiprocessors. *International Journal of High Performance Computing Applications*, **7**, 64-82. http://dx.doi.org/10.1177/109434209300700105

[16] Amestoy, P.R., Duff, I.S., L'Excellent, J.-Y. and Koster, J. (2001) A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, **23**, 15-41. http://dx.doi.org/10.1137/S0895479899358194

[17] Kalinkin, A. (2013) Intel Direct Sparse Solver for Clusters, a Research Project for Solving Large Sparse Systems of Linear Algebraic Equations on Clusters. *Sparse Days Meeting* 2013 *at CERFACS*, Toulouse, 17-18 June 2013. http://www.cerfacs.fr/6-27085-Sparse-Days-2013.php

[18] Kalinkin, A. (2013) Sparse Linear Algebra Support in Intel Math Kernel Library. *Sparse Linear Algebra Solvers for High Performance Computing Workshop*, Scarman House, University of Warwick, 8-9 July 2013. http://www2.warwick.ac.uk/fac/sci/dcs/research/pcav/linear solvers/programme/

[19] Kalinkin, A. and Arturov, K. (2013) Asynchronous Approach to Memory Management in Sparse Multifrontal Methods on Multiprocessors. *Applied Mathematics*, **4**, 33-39. http://dx.doi.org/10.4236/am.2013.412A004

[20] Kalinkin, A., Anders, A. and Anders, R. (2014) Intel® Math Kernel Library Parallel Direct Sparse Solver for Clusters. *EAGE Workshop on High Performance Computing for Upstream*, Chania, Crete, 7-10 September 2014. http://www.eage.org/events/index.php?evp=12682&ActiveMenu=2&Opendivs=s3 http://dx.doi.org/10.3997/2214-4609.20141926

[21] Davis, T.A. and Hu, Y. (2011) The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, **38**, 1:1-1:25. http://www.cise.ufl.edu/research/sparse/matrices