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 $A - BD^\top C$ 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 $(A | D) = A_{\text{loc}} - BD^\top C$. 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
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 calculate 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):

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

where $A_{\text{loc}}$ 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 $A$:

$$A = \begin{pmatrix} L_{11} & 0 \\ L_{12} & I \end{pmatrix} \cdot \begin{pmatrix} I & 0 \\ S & L_{12}^T \end{pmatrix} \cdot \begin{pmatrix} L_{11}^T & L_{12}^T \\ 0 & I \end{pmatrix},$$

where

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

The matrix $S = (A | A_{\text{loc}})$ 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 $A_{\text{loc}} = L_{11} L_{11}^T$ with the factorization step of the direct solver;
2) Calculate $B_{\text{temp}} = A_{\text{loc}}^{-1} B^T$ with the solving step applied to multiple right-hand sides;
3) Calculate $C_{\text{temp}} = B B_{\text{temp}}$ 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 $A_{\text{loc}}^{-1} B^T$, 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 \( Z_i \) is a child of \( Z_j \) 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.

\[
\begin{align*}
&1) \quad L = A_{loc} \\
&2) \quad \text{for } i = 1, \text{number} \_ \text{of} \_ \text{tree} \_ \text{nodes} \text{ do} \\
&3) \quad Z = \text{node of tree}; \\
&4) \quad \text{for all } Z_j \text{ child of } Z_i \text{ do} \\
&5) \quad Z_{ij} = Z_i \* \text{mask}_{Z_j} \quad \text{prepare update of } Z_i \text{ by } j\text{-th child}; \\
&6) \quad Z_i = Z_i - Z_{ij}; \\
&7) \quad \text{end} \\
&8) \quad \text{Calculate } LL^T \text{ decomposition of } Z_i \\
&9) \quad \text{end}
\end{align*}
\]

where by \( \text{mask}_{Z_i} \) 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).](image-url)
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: $\hat{Z}_i$ 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 $i = 1, \text{number of tree nodes}$ do
   3) $\hat{Z}_i = \text{node of tree}$;
   4) for all $\hat{Z}_j$ child of $\hat{Z}_i$ do
      5) $\hat{Z}_{i,j} = \hat{Z}_i \ast \text{mask} \hat{Z}_j$, prepare update of $\hat{Z}_i$ by $j$-th child;
      6) $\hat{Z}_i = \hat{Z}_i - \hat{Z}_{i,j}$;
   end
   7) Calculate $LL^T$ decomposition of $\hat{Z}_i$;
   8) end
9) for $j = 1, \text{number of tree node}$ do
   10) $\hat{Z}_{c,j} = \hat{Z}_j \ast \text{mask} \hat{Z}_c$, prepare update of $\hat{Z}_c$ by $j$-th child;
   11) $Z_{c,j} = Z_{c} - Z_{c,j}$;
   12) end
   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) \( \tilde{Z}_i = \) node of tree;
4) for all \( \tilde{Z}_j \) child of \( \tilde{Z}_i \) do
5) \( \tilde{Z}_{ij} = \tilde{Z}_i \ast \text{mask} \tilde{Z}_j \) prepare update of \( \tilde{Z}_i \) by \( j \)-th child;
6) atomic \( \tilde{Z}_i = \tilde{Z}_i - \tilde{Z}_{ij} \);
7) end
8) Calculate $LL^T$ decomposition of \( \tilde{Z}_i \);
9) end
10) parallel for $j = 1$, number_of_tree_nodes do
11) \( Z_{cj} = \tilde{Z}_j \ast \text{mask} \tilde{Z}_c \) prepare update of \( Z_c \) by \( j \)-th child;
12) atomic \( Z_c = Z_c - Z_{cj} \);
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 matrices from 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


Scientific Research Publishing (SCIRP) is one of the largest Open Access journal publishers. It is currently publishing more than 200 open access, online, peer-reviewed journals covering a wide range of academic disciplines. SCIRP serves the worldwide academic communities and contributes to the progress and application of science with its publication.

Other selected journals from SCIRP are listed as below. Submit your manuscript to us via either submit@scirp.org or Online Submission Portal.