Scalability of a parallel Schur complement method given the limitations for memory

N S Nedozhogin, S P Kopysov and A K Novikov
Institute of Mechanics, Udmurt Federal Research Centre Ural Branch of the Russian Academy of Sciences. 34 T. Baramzinoy, Izhevsk, Russia 426067
E-mail: Nedozhogin@inbox.ru

Abstract. In decomposition methods, the memory costs for solving the interface problem are increasing significantly with the number of subdomains increasing. Using of the Schur complement method allows you to reduce the number of iterations when the system is being solved. At the same time, the Schur complement matrix takes up more memory in comparison with global stiffness matrix. This imposes restrictions on the maximum size of the problem for which you can apply this method. Different approaches to reduce the costs and limitations of memory on stage of the construction and solving of the interface system exist. Parallel algorithm of the construction \( S \) with distributed storage of the matrix are considered when implementing using OpenMP and MPI technologies. This approach allows not only to reduce the limits on the maximum size of the solved problem, but also to resolve conflicts of shared memory access by increasing the number of independent parallel tasks.

1. Schur complement method

The Schur complement method is the non-overlapping domain decomposition [1]. However, the Schur complement matrix has more nonzero elements with smaller dimension in comparison with global stiffness matrix [2]. Due to these properties, even the use of sparse matrix storage formats requires considerable memory costs. Block [3] and hierarchical [4, 5] methods with low-rank approximation have received great development due to the current trends in reducing the amount of memory per processor core. In this paper, parallel Schur complement method with distributed storage of the matrix is considered. The distributed construction makes it possible to apply this approach not only in the classical Schur complement method, but also in the hierarchical or block one, or in the case of constructing a preconditioner on the based on the substructures method.

Let the mesh domain \( \Omega \) be split into \( n_\Omega \) non-overlapping subdomains \( \Omega = \bigcup_{i=1}^{s n_\Omega} \Omega_i^{(1)} \), where \( \Omega_i^{(1)} \cap \Omega_j^{(1)} = \emptyset \), when \( i \neq j \). Formally unite the subdomains, preserving the structure of the decomposition (interior nodes remain interior, boundary – boundary). We introduce the decomposition of the second level with number of the subdomains equal number of the computational modules \( n_p \) such that \( \Omega = \bigcup_{k=1}^{k \leq n_p} \Omega_k^{(2)} \), and \( \Omega_k^{(2)} = \bigcup_{i=\frac{k s n_\Omega}{n_p}}^{\frac{(k+1) s n_\Omega}{n_p}} \Omega_i^{(1)} \).

For decomposition the first level, the system of equations is formed in such a way that the unknowns corresponding to the interior and boundary nodes take the form:
In this system, the unknowns stand for the boundary nodes are found from the solution of the system $S^{(1)} u_B = f_B$. Here $S^{(1)}$ is the global boundary stiffness matrix (also known as the Schur complement matrix [2]) and is the sum of the local Schur complement matrices $S^{(1)}_i = A^{(1)}_{BB} - i A^{(1)}_{BI} (A^{(1)}_{II})^{-1} i A^{(1)}_{IB}$ such that $S^{(1)} = \sum_{i=1}^{n_B} S^{(1)}_i$, where index $i$ corresponds to subdomains of the first level.

Schur complement matrix $S^{(2)}_k$ corresponding to these subdomains are supplemented by zeros up to size global matrix $S^{(1)}$ in this $S^{(1)} = \sum_{k=1}^{n_p} S^{(2)}_k$. Note that, Matrices $S^{(2)}_k$ don’t depend on each other during the formation stage. This excludes conflicts when reading elements data and writing results of summing of local matrices $S^{(1)}_i$.

For reduce computational costs of RAM, each computational modules forms one row $l$ of the matrix $S^{(1)}_i$ at the same time. The order of the construction of rows is given by subdomains. Parallel algorithm of the construction for $n_p$ computational modules using OpenMP technology on the example of two subdomains would be as follows (note that $n_\Omega \geq n_p$):

```
Algorithm 1. Parallel algorithm of the Schur complement matrix construction $S^{(1)} = \sum_{k=1}^{n_p} S^{(2)}_k$

//Forming of $S^{(2)}_1$
For $i = 1..n_\Omega / n_p$ do
1) Function of the inverse matrix $i (A^{(1)}_{II})^{-1}$;
//Loop of OpenMP
For $l = 1..n_B i$ do
2) $v = i A^{(1)}_{BI} (l,:)$;
3) $v = v \cdot (A^{(1)}_{II})^{-1}$;
4) $S^{(1)}_i (l,:) = i A^{(1)}_{BB} (l,:) - v \cdot A^{(1)}_{IB}$;
5) Record the $S^{(1)}_i (l,:) \text{ elements in the corresponding row of the matrix } S^{(2)}_1$;

//Forming of $S^{(2)}_2$
For $i = n_\Omega / n_p .. n_\Omega$ do
1) Function of the inverse matrix $i (A^{(1)}_{II})^{-1}$;
//Loop of OpenMP
For $l = 1..n_B i$ do
2) $v = i A^{(1)}_{BI} (l,:)$;
3) $v = v \cdot (A^{(1)}_{II})^{-1}$;
4) $S^{(1)}_i (l,:) = i A^{(1)}_{BB} (l,:) - v \cdot A^{(1)}_{IB}$;
5) Record the $S^{(1)}_i (l,:) \text{ elements in the corresponding row of the matrix } S^{(2)}_2$;
```

Based on the computing experiment (See Table 1), run-time costs on each subdomain have the following order (top-down):

1) Inverse of the matrix $i (A^{(1)}_{II})$ (step 1 of Algorithm 1);
2) Record the row $l$ of the local matrix $S^{(1)}_i (l,:) \text{ in the corresponding row of the matrix } S^{(2)}_1$ (step 5 of Algorithm 1);
3) Operations of taking the row from the matrix, the matrix-vectors multiplications and the difference of the vectors (steps 2-4 of Algorithm 1, denote by MatVec).
Taking the row from the matrix, the matrix-vector multiplications and the difference of the vectors (see steps 2-4 of Algorithm 1) are parallelized using OpenMP technology, that for these operations gave an acceleration close to linear. Also, in the future, it is planned to parallelize these operations for use on graphics accelerators using CUDA technology.

 Recording of the row \( l \) of the local matrix \( S_i \) \((1)\) to the corresponding row of the matrix \( S_k \) \((2)\) largely depends on the matrix storage formats used. Matrices \( S_k \) \((2)\) is formed in a compressed CSR format, because of its large size. Using arrays of vectors (C++ container std::vector) allows to reduce memory costs to a minimum, but the complexity of adding new elements to such storage formats is \( O(n_B) \), where \( n_B \) is the number of equations corresponding to the boundary nodes. To reduce the costs of adding new elements of the row, we use sorted associative containers (C++ container std::map) and not sorted (C++ container std::unordered_map). In the sorted container, the complexity of finding and adding a new element is \( O(\ln n_B) \) (due to its red-black tree structure), but it significantly increase memory costs to store additional two pointers. The non-sorted container, for these operations has complexity form \( O(1) \) to \( O(n_B) \) (depending on the hash function) and request smaller memory sizes, compared to the container std::map, which is optimal for use in the constructing of matrices \( S_k \) \((2)\).

### Table 1. Characteristic formation times in one subdomain

| \( i(A_{H})^{-1} \) | MatVec \( N_B \) | Assembly \( S^{(1)} \) \( N_{i_{\text{min}}} \) |
|------------------|----------------|-----------------------------|
| LU | SM | \( n_\Omega = 512 \) | |
| 95.4972 | 41.69505 | 0.340893 | 1.53655 |
| 4.07287 | 1.947 | 0.130277 | 0.6955 |
| \( n_\Omega = 1024 \) | | |

### 2. Matrix inversion

To inverse the matrix at Step 1 of Algorithm 1, LU-factorisation and Sherman-Morrison methods [6] were considered.

For a given non-singular matrix \( A \) there exists a decomposition on upper triangular matrix \( U \) and a lower triangular matrix \( L \). If matrices \( A, U, L \) are invertible, then \( A^{-1} = U^{-1}L^{-1} \). In the construction \( L \) and \( U \), products with nonzero elements of the parent matrix \( [7] \) were summed, since the factorization was carried out for matrices stored in the CSR format. Only sequential implementations of the LU-factorization on CPU was considered. This approach showed the best results for small matrices. Features of this algorithm don’t allow to effective use all the capabilities of the massively parallel GPU architecture.

A parallel version of the Sherman-Morrison method [8] implemented using OpenMP technology was considered.

#### Algorithm 2. Parallel algorithm of Sherman-Morrison method

1) \( x, y \) - vectors of dimension \( n \);
   For \( k = 1..n \) do
   //OpenMP
   2) \( x^T = A_{H}(k,:) - e_k \);
   //OpenMP
   3) \( y = x^TA_{\text{inv}} \);
   //Loop of OpenMP on \( i \)
   For \( i = 1..n \) do
      For \( j = 1..n \) do
         4) \( A_{\text{inv}}(j,i) = A_{\text{inv}}(j,i) - \frac{y_i A_{\text{inv}}(j,k)}{y_{k+1}} \);
LU-factorization is faster than the Sherman-Morrison method in sequential implementations. In split of it, LU-factorization doesn’t parallelize well. Using parallel version of the Sherman-Morrison method provided a parallel acceleration about two times. To reduce the number of operations and the computational load, matrix $\left(A^{(3)}_H\right)^{-1}$ is stored in its entirety before the completion of the constructing of $S^{(1)}$ in the current subdomains, since it is not known how many non-zero elements contain an inverse matrix. To solve the system of equations on the inner subdomains nodes $A^{(1)}_I u_i = f_i - A^{(1)}_B \bar{u}_b$, Krylov subspace methods [9] are used.

3. Solving the interface system

Implementation of conjugate gradients method for solving the system $S^{(1)} u_b = f_b$ on one of $n_p$ computing node is shown in Algorithm 3. Here the system matrix is represented in the form $S^{(1)} = \sum_{k=1}^{n_p} S^{(2)}_k$. Index $k$ corresponding to the number of computational module is omitted.

**Algorithm 3.** Algorithm of the preconditioned conjugate gradient method for distributed matrix

1) $r, z, p, u \in \mathbb{R}^{n_B}$  
2) $r_0 \leftarrow f_b$;  
3) $u_0 \leftarrow 0$;  
4) $z_0 = M r_0$; // $M \sim S^{-1}$ – preconditioner  
5) $p_0 \leftarrow z_0$;  
6) $\rho_0 = (r_0, z_0)$; // point of synchronize  
While $\frac{\|r_i\|}{\|f_b\|} > \varepsilon$ do  
7) Assembly $p_i$; // Using MPI::Allreduce  
8) $q_i = S^{(2)} p_i$; // $k$ independent operations on each MPI process  
9) $a_i = (r_i, z_i)/(q_i, p_i)$; // point of synchronize with help MPI::Allreduce  
10) $u_{i+1} = u_i + a_i p_i$;  
11) $r_{i+1} = r_i - a_i q_i$;  
12) $z_{i+1} = M r_{i+1}$; // $M \sim S^{-1}$ – preconditioner  
13) $\rho_{i+1} = (r_{i+1}, z_{i+1})$; // point of synchronize with help MPI::Allreduce  
14) $\beta_{i+1} = \rho_{i+1}/\rho_i$;  
15) $p_{i+1} = r_{i+1} + \beta_{i+1} p_i$;  
16) $i = i + 1$;

In a matrix-vector product (See steps 7-8 of Algorithm 3) operation $q_i = S^{(2)} p_i$ is performed in two steps. First, the assembly of the vector $p_i$ from all MPI processes, then $n_p$ independent operations $(q_k)_i = S^{(2)}_k p_i$. The result of products is the set of vectors $q_k$ such that $q_k \in \mathbb{R}^{n_B}$ and $q = \sum_{k=1}^{n_p} q_k$. Summed of vector $q$ are not required. Vector is stored by parts on various compute nodes.

Parallelization of the operations with vectors is realized as follows: operands are parts of vectors consisting of components corresponding to mesh nodes of the subdomain $\Omega^{(2)}_i$ for each parallel MPI process. The result vector assembly are not required, each MPI process jobs with its local vectors, and synchronization occurs when the scalar products $(r, z), (q, p), (r, r)$ are computed. Scalar products are performed in two steps. First, local scalar products for the part vectors in parallel MPI processes, then summation of the local sums (on this step, implicit synchronization occurs). Inside the MPI process, vectors operations are parallelized using OpenMP technology with number of the thread equal number of the CPU cores on computational nodes.
4. Results
Numerical experiments were carried out on the test problem of the theory of elasticity. The computational domain had the form of a parallelepiped. Mesh consisted of 539000 hexagonal cells. 8 threads of the OpenMP are running inside each MPI process. In Table 2, the result of experiment are presented for dimension on 512 and 1024 subdomains with using 2, 4, 6 MPI processes. The presented results are obtained on 6 computational nodes, each of which contains two processors Intel Xeon E5-2609 and 64 GB RAM. For \( n_\Omega = 512 \), this problem took approximately 100GB of RAM, for \( n_\Omega = 1024 \), problem took approximately 80GB of RAM.

| \( n_p \) | \( \sum_i S^{(1)}_i \) | \( S^{(1)} u_B = f_B \) | \( iA^{(1)}_{ii} iu_i = \Lambda f_i - iA^{(1)}_{ii} u_B \) |
|---|---|---|---|
| | LU | SM | # | Time | # | Time |
| \( n_\Omega = 512 \) | | | | | | |
| 2 | 39954.6 | 14936.8 | 2324 | 1640.37 | 43 | 4.72359 |
| 4 | 18086.6 | 7615.15 | 2325 | 1391.23 | 43 | 2.80635 |
| 6 | 11713.4 | 5014.66 | 2325 | 1498.37 | 43 | 1.9185 |
| \( n_\Omega = 1024 \) | | | | | | |
| 2 | 5128.47 | 2818.16 | 2600 | 1464.09 | 35 | 3.15815 |
| 4 | 2622.52 | 1411.97 | 2600 | 1397.77 | 35 | 2.10205 |
| 6 | 1802.32 | 965.803 | 2600 | 1469.18 | 35 | 1.74885 |

The considered algorithm allows to significantly reduce the memory limits for the Schur complement method. The partitioning of the matrix \( S^{(1)} \) and its distributed storage makes it possible to realize coarse-grained parallelism at the formation stage of Schur complement matrix, to reduce exchanges and to exclude synchronous access to one memory cell during the local matrices addition. At the same time, an increase in the number of involved computational modules permits achieving an acceleration close to linear at the stage of matrix construction. At the solver stage, the distributed storage of the matrix \( S^{(1)} \) allows solving systems with a dimension proportional to the memory size of the involved computational modules. Scalability is limited to assembling the results of scalar products of vectors and depends only on the restrictions on the speed to data exchange over the network.

Acknowledgments
The reported study was partially funded by RFBR according to the research projects № 17-01-00402-a.

References
[1] Toselli A and Widlund O B 2005 Springer Series in Computational Mathematics 34
[2] Kopysov S P, Kuzmin I M, Nedozhogin N S, Novikov A K and Sagdeeva Y A 2014 Journal of Supercomputing 69 81-88
[3] Amestoy P R, Buttari A, L’Excelent J-Y and Mary T 2017 SIAM J. Of Scienti fic Computing 39 A1710-A1740
[4] Chaves G, Turkiyyah G and Keyes D E 2017 Domain Decomposition Methods in Science and Engineering XXIII 135-143
[5] Chen C, Pouransari H, Rajamanickam S, Boman E G and Darve E 2018 Parallel Computing 74 46-64
[6] Sherman J and Morrison W J Ann. Math. Statistics 21 124-127
[7] Kopysov S P, Kuzmin I M, Nedozhogin N S and Novikov A K 2012 Uchenye zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki 154 202-215
[8] Nedozhogin N S, Sarmakeeva A S and Kopysov S P 2014 Bulletin of the South Ural State University. Series “Computational Mathematics and Software Engineering” 3 101-108
[9] Saad Y Iterative Methods for Sparse Linear System (Boston: PWS Publiching company) 447