Multithreaded Multifrontal Sparse Cholesky Factorization Using Threading Building Blocks

Rostislav Povelikin, Sergey Lebedev, and Iosif Meyerov(✉)

Lobachevsky State University of Nizhni Novgorod, Nizhni Novgorod, Russia
povelikin.rostislav@gmail.com,
sergey.a.lebedev@gmail.com, iosif.meyerov@vmk.unn.ru

Abstract. The multifrontal method is a well-established approach to parallel sparse direct solvers of linear algebraic equations systems with sparse symmetric positive-definite matrices. This paper discusses the approaches and challenges of scalable parallel implementation of the numerical phase of the multifrontal method for shared memory systems based on high-end server CPUs with dozens of cores. The commonly used parallelization schemes are often guided by an elimination tree, containing information about dependencies between logical tasks in a computational loop of the method. We consider a dynamic two-level scheme for the organization of parallel computations. This scheme employs the task-based model with dynamic switching between solving relatively small tasks in parallel and using parallel functions of BLAS for relatively large tasks. There are several problems with the implementation of this scheme, including time-consuming synchronizations and the need for smart memory management. We found a way to improve performance and scaling efficiency using the model of parallelism and memory management tools from the Threading Building Blocks library. Experiments on large symmetric matrices from the SuiteSparse Matrix Collection show that our implementation is competitive with the commercial direct sparse solver Intel MKL PARDISO.

Keywords: Sparse direct methods · Multifrontal method · Parallel computing · High performance computing · Threading building blocks

1 Introduction

Direct methods for solving large sparse systems of linear algebraic equations (SLAEs) with a symmetric positive-definite (SPD) matrix are widely used in numerical simulations in different subject areas. During matrix factorization, the number of nonzero elements in the factor increases by several orders of magnitude compared to the original matrix, which significantly affects the memory requirements and the computation time. In this regard, a special reordering procedure [13] is applied to the original matrix, which rearranges the rows and columns of the matrix in order to reduce the number of nonzero elements in the factor. Next, a symbolic phase of the Cholesky decomposition is performed for the reordered SPD matrix. This numerical procedure analyses the matrix, creates special data structures and allocates necessary memory. The next stage of the solution is a numerical phase of the Cholesky decomposition. At

© Springer Nature Switzerland AG 2019
V. Voevodin and S. Sobolev (Eds.): RuSCDays 2019, CCIS 1129, pp. 75–86, 2019.
https://doi.org/10.1007/978-3-030-36592-9_7

this stage, non-zero elements of the factor are calculated. Next, the solution of two triangular SLAEs is performed and the inverse permutation of the components of the solution is applied [5, 6].

When solving state-of-the-art problems, each stage of the scheme described above is very computationally intensive and requires efficient parallelization for modern supercomputers. Appropriate algorithms have been under development over the past decades. The parallel algorithms for distributed and shared memory systems are implemented in Intel MKL PARDISO [12], MUMPS [2], SuperLU [19], CHOLMOD [4], HSL_MA57 [9], and in other solvers that are widely used in many research projects around the world. However, continuous improvement of multicore architectures motivates further development of high-performance scalable algorithms for such systems [1, 10, 11, 16, 22–25].

In this paper, we focus on achieving the efficiency of parallelization and using the memory subsystem on a high-end multicore computer when performing the numerical phase of the Cholesky decomposition. This phase is often very time-consuming, and its parallelization is a challenging problem. The MUMPS solver was originally developed for distributed memory systems, and then shared memory parallelism support was added to it [15]. For this, a modification of the Geist-Ng algorithm is used, in which prior to the start of the numerical phase the search for the layer of the elimination tree is performed, subtrees with the root belonging to the found layer are processed independently. Another widely used method of parallelization is the use of Direct Acyclic Graph, where the graph describes the dependencies between the nodes of the elimination tree and the sequence of operations inside the node which results in fine-grained parallelism. This approach is used in the HSL_MA57 solver [10]. Earlier, we proposed a dynamic two-level parallelization scheme for the multifrontal method that combines task-based parallelism at the lower levels of the elimination tree and the use of parallel BLAS functions at the upper levels when solving a limited number of large subtasks [18]. In this paper, we address two main problems encountered in the implementation of this scheme: moderate load balancing quality for dozens of computing cores, the need for adaptive selection of the switching point between two ways of parallelization. Further, it will be shown how a suitable usage of the TBB library allows us to overcome these problems, to improve the memory management scheme, and obtain competitive results with Intel MKL PARDISO outperforming it on several matrices.

The paper is organized as follows. Section 2 provides a general overview of the multifrontal method. In Sect. 3, the main ideas of two-level task-based parallelization of the multifrontal method for shared memory systems are described. In Sect. 4 we propose the new parallel scheme based on Threading Building Blocks. Section 5 presents numerical results and discussion. Section 6 concludes the paper.

2 Multifrontal Method Overview

The multifrontal method [7, 8] for the numerical phase of the Cholesky decomposition is commonly used in many sparse direct solvers, such as MUMPS, SuiteSparse and others. The advantages of this method include the efficient use of a hierarchical

memory system, as well as simple and local dependencies between iterations, which creates good prospects for parallelization. The main idea of the method is to organize computations using high-performance implementations of operations on dense submatrices of the original matrix. The dependencies between operations are determined by an elimination tree [20] which is constructed during the analysis phase. The number of nodes in the tree corresponds to the dimension of the original matrix N, the nodes are numbered from 1 to N, each node is associated with a column of a factor L. Edges in the tree define the order of calculations of columns of L. The main principle is as follows: before calculating a column associated with some node of the tree we must calculate all columns corresponding to his child nodes. Therefore, the main computational loop of the multifrontal method performs calculations, examining the nodes of the elimination tree in order from leaves to root. In every node a set of operations with dense submatrices is performed (Fig. 1). First, the frontal matrix of the node is calculated using the elements of the column of the original matrix (the `init_frontal_matrix` procedure) and the updating matrices of the child nodes (the `assembly_frontal_matrix` procedure). Then, a partial dense factorization is performed for the frontal matrix (the `factorize` procedure) resulting in the column of the factor and in the update matrix (the `form_update_matrix` procedure), which will be used when building the frontal matrix for the parent node. A more detailed description of the method can be found in [14, 20].

```
1    foreach node i of elimination tree in topological order
2      init_frontal_matrix(F_i)
3        foreach child j of i do
4          U ← U ⊕ U_j
5        end for
6      assembly_frontal_matrix(F,U)
7      factorize(F)
8      form_update_matrix(U_i)
9      L_i ← F_(1,*)
10   end for
```

Fig. 1. High-level overview of the multifrontal method

The multifrontal method can be easily parallelized using parallel implementations of the BLAS library functions to perform operations with frontal matrices. This approach does not scale well due to the lack of resources for parallelization in the lower levels of the elimination tree, where frontal matrices are usually too small. Another approach to parallelization is exploit task-based parallelism (Fig. 2), where the task is to calculate one column of the factor L. The order of calculations and the possibility of parallelization are determined by the elimination tree. This approach can be

implemented using static or dynamic load balancing. In the case of static load balancing, tree nodes are assigned for processing to certain threads. When using dynamic balancing, the nodes of the elimination tree are assigned to be executed by threads during program execution. However, scaling efficiency is limited when processing the nodes of the tree close to the root. This is due to the fact that the number of parallel tasks decreases and some of the threads are not used while the dimension of frontal matrices for the upper part of the tree increases.

```
1   procedure process_node(node of elimination_tree)
2       foreach child of node in elimination_tree do
3           #spawn new task
4               process_node(child)
5       end for
6     #wait for spawned tasks to complete
7       multifrontal_step(node)
8   end procedure

9   procedure multifrontal_step(node of elimination tree)
10    i←number of node in elimination tree
11      init_frontal_matrix(F_i)
12      foreach child j of i do
13          U ← U ⊕ U_j
14      end for
15      assembly_frontal_matrix(F,U)
16      factorize(F)
17      form_update_matrix(U_i)
18      L_i←F(1,*)
19  end procedure
```

Fig. 2. The task-based parallel multifrontal method

3 Task-Based Two-Level Dynamic Parallel Algorithm

In [17, 18] we analyzed the task-based two-level algorithm. The algorithm employs task-based parallel load balancing, highly effective on lowest levels of the elimination tree, and switches to using parallel BLAS functions for computationally demanding tasks at the upper levels of the tree (Fig. 3). This approach greatly improves performance and scaling efficiency of the implementation.

```
1    procedure process_node(node i of elimination_tree)
2        init_frontal_matrix(F_i)
3        foreach child j of i do
4            U ← U ⊕ U_j
5        end for
6        assembly_frontal_matrix(F,U)
7        factorize(F)
8        form_update_matrix(U_i)
9        L_i←F_(1,*)
10   end procedure
11
12   procedure two-level_parallel_multifrontal
13       set_num_threads(MAX_SYSTEM_THREADS);
14       blas_set_num_threads(1);
15       #parallel section
16       while(there are enough independent tasks)
17           i ← nextTask()
18           process_node(i)
19       end while
20
21       set_num_threads(1);
22       blas_set_num_threads( MAX_SYSTEM_THREADS);
23       while(there is a task)
24           i← nextTask();
25           process_node(i)
26       end while
27   end procedure
```

Fig. 3. The parallel two-level multifrontal algorithm for shared memory systems

4 Exploiting Parallelism Using the Threading Building Blocks Library

Experiments have shown that the approach described above has two key problems. First, the approach assumes an explicit synchronization of the threads during the transition between levels. Such a scheme leads to the useless, from utilization of computing resources point of view, waiting for the completion of processing the nodes of the elimination tree. Taking into account that the second level of the described scheme assumes the absence of parallel processing of independent nodes, this leads to insufficient scalability when using dozens of CPU cores. Secondly, the approach requires the selection of the switching moment between two parallelization schemes, which intricately depends on the various characteristics of the original matrix.

In this paper, we propose a new scheme of parallelization based on the TBB library [27]. TBB was created to develop scalable parallel applications in terms of logical problems, not threads. In addition, Intel MKL implementation of BLAS functions also

```
1   procedure dynamic_parallel_multifrontal
2     set_tbb_shared_thread_pool(MAX_SYSTEM_THREADS);
3
4     parallel foreach leaf node i of elimination_tree do
5       process_node(i)
6       p ← getParentNode(i)
7       while p has no left unprocessed children
8         process_node(p)
9         p ← getParentNode(p)
10      end while
11    end parallel foreach
12  end procedure
```

Fig. 4. The dynamic parallel multifrontal algorithm for shared memory systems

supports the use of the TBB library as a thread manager. Such support and the design of libraries allow the use of nested parallelism mechanisms for organizing the dynamic switching from parallel processing of the elimination tree nodes to parallel BLAS functions (Fig. 4).

The key difference of the proposed approach is the use of the TBB library parallelism model. A model is a dynamic distribution of logical tasks between a shared set of threads. Thus, the logical tasks of parallel processing of the tree nodes and parallel BLAS functions are distributed over the total set of threads. The organization of parallel logical tasks becomes the responsibility of the library's task scheduler. The work of the scheduler is decentralized and distributed among all threads. Each thread has its own local task queue. The thread accesses the queue to get a new task when it finishes executing the current one. If there are no tasks in the local queue, the thread searches for the task in the queue of another thread.

Consider the work of the task scheduler in the case of a new parallel scheme. The main thread puts all the available logical tasks of processing independent nodes in its local queue. Threads created by the TBB library when initializing a shared set of threads find a free logical task in the local queue of the main thread, steal and process it. When the local thread queue is empty, the thread looks for a task in the queues of other threads. Upon completion of the current task, the thread executes a new task for processing the node M, if the processing of the last child node of the node M has been completed. Parallel BLAS functions also generate new logical tasks. Thus, threads process tree nodes and auxiliary tasks of BLAS functions in parallel, dynamically switching between them. Thread finishes its work when it cannot find new tasks neither in its own queue nor in the queues of other threads. The described balancing process eliminates the need for explicit synchronization and the choice of its moment, and also allows us to process independent nodes of the tree in parallel throughout the whole computational loop. Also note that in the implementation of this scheme, we employed the Intel MKL BLAS version, using TBB for multithreading. In general, this scheme allows us to overcome two problems noted at the beginning of this section.

Note that previous versions of the method used the following memory management scheme for processing the elimination tree: each thread allocated the maximum required amount of memory for any node before processing the tree. The scheme assumed the exclusive use of previously allocated memory. The new task-based model of parallelism cannot exploit this scheme of working with memory producing the data race due to the task-stealing mechanism. The thread resets the memory when it takes the task of processing a new node, waiting for another thread to finish BLAS task of processing the actual node on the same memory, which raises the problem of data races. To solve this problem and also improve memory usage patterns, the following scheme of work with memory is proposed. We need logically bind the memory to the task but not to the thread. Standard memory managers will not cope effectively with such a scheme, as it involves frequent requests for memory allocation and deallocation. Therefore we propose to use the TBB library's scalable memory manager (`tbb::scalable_allocator`). One of the key features of this memory manager is caching freed memory for potential new allocations. This reduces the number of system calls, but increases the level of memory consumption by the application.

5 Numerical Results

5.1 Computational Infrastructure and Test Problems

The computational experiments were performed at a node of a supercomputer with 2x Intel Xeon Gold 6152 (Skylake, 22 cores each), 44 cores overall, 192 GB RAM, Ubuntu 18.04, Intel C++ Compiler, Intel MKL and TBB from the Intel Parallel Studio XE 2019 suite. For benchmarking purposes we used 8 large symmetric positive definite matrices from the SuiteSparse Matrix Collection [26]. All matrices were reordered using permutations computed using the Metis library [13] other libraries can be also used [21, 22]. The PARDISO solver was run with default settings. According to the documentation the `iparm(24)` value was set to 1 when a large number of threads was used.

Table 1. Matrices and their parameters

Matrix	Dimension	Nonzeros	Nonzeros in L
boneS10	914 898	28 191 660	266 173 272
Emilia_923	923 136	20 964 171	1 633 654 176
audikw_1	943 695	39 297 171	1 225 571 121
bone010_M	986 703	12 437 739	363 650 592
bone010	986 703	36 326 514	1 076 191 560
StocF-1465	1 465 137	11 235 263	1 039 392 123
Hook_1498	1 498 023	31 207 734	1 507 528 290
Flan_1565	1 564 794	59 485 419	1 451 334 747

5.2 Results and Discussion

First, we study performance of the two considered parallel schemes in the numerical phase of the Cholesky decomposition. To do this, we run these schemes on 8 large symmetric positive-definite matrices, the parameters of which are given in Table 1. In all experiments we use 44 cores of the Skylake processor (we also make sure that using such a large number of cores does not slow down the computations). We make each run 10 times and take the minimum time. For both schemes, we choose the appropriate strategy of padding the matrix columns with zeros in order to increase the size of the groups of columns with the same sparsity pattern under the upper triangle (the so-called supernodes). For the 'Old Scheme', based on OpenMP, in addition, we empirically choose a relevant moment of switching between parallel tasks solving and using parallel BLAS. Then we select the best results and compare them with the results of MKL PARDISO, also tested 10 times on each matrix.

Our first observation is that the strategy of padding supernodes with zeros on large matrices and huge number of threads affects performance of the considered algorithms. For example, on the matrix Emilia_923 with a largest factor, the computation time of the 'Old Scheme' varies from 33 s to 28 s, and the run time of the 'New Scheme' improves from 19 to 15 s, depending on the padding algorithm (Fig. 5).

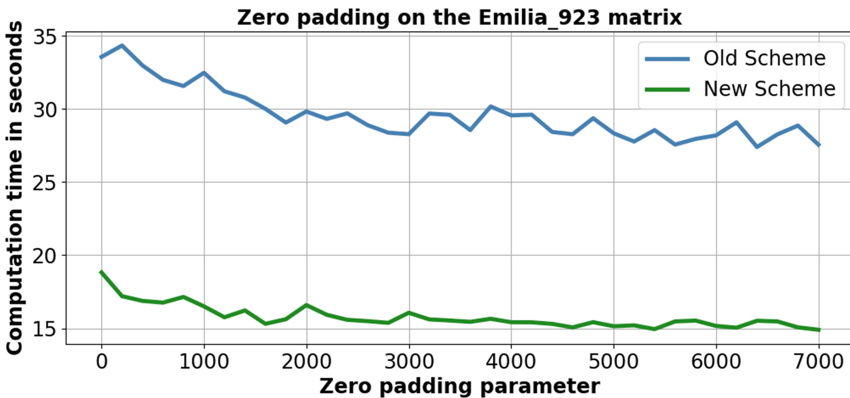

Fig. 5. Zero padding improves performance of the Cholesky decomposition. The parameter value on the x-axis corresponds to a maximum number of zeros added to a supernode.

Figure 6 presents a comparison of the best computation time of the considered algorithms and MKL PARDISO. Experiments have shown that the 'Old Scheme' loses PARDISO, whereas the 'New Scheme' is ahead on matrices with a large factor size, showing comparable or slightly worse results on the other matrices.

Further, we investigated how the considered algorithms are scaled with an increase in the number of threads involved in the computations. Figure 7 shows the performance results for the matrix Emilia_923. The results show that with the use of a small number of cores, the 'Old Scheme' works better due to the low overhead of the work of the scheduler. However, with an increase in the number of cores, the 'New Scheme' demonstrates its advantage and continues to scale, while the computation time of the 'Old Scheme' ceases to decrease. Note that all the algorithms do not show good scaling efficiency when using 44 cores, even on the large matrices from the SuiteSparse Collection.

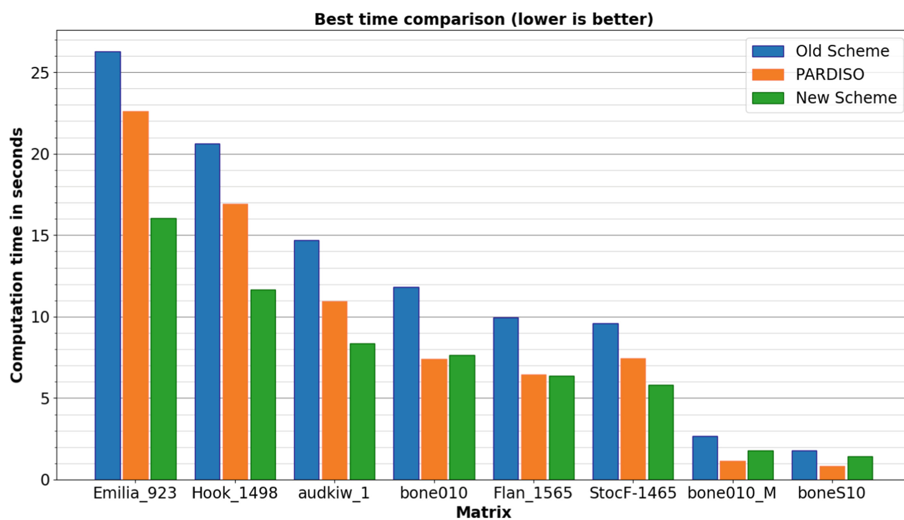

Fig. 6. Computation times of the numerical phase of the Cholesky decomposition. Three implementations are compared: Intel MKL PARDISO, the scheme based on OpenMP ('Old Scheme'), and the proposed scheme based on TBB ('New Scheme'). Time is given in seconds.

Let us make sure that the 'New Scheme' is significantly ahead of the 'Old Scheme' due to the decrease in the spin-time when using a large number of cores. To do this, we use the Intel Amplifier profiler. The results of the profiling of both implementations are shown in Figs. 8 and 9, respectively. The profiles contain basic hotspots and allow us to understand what the bottleneck of each algorithm is. Thus, in the 'Old Scheme', more than half of the time is spent waiting for the barrier to reach when synchronizing OpenMP threads. On the contrary, in the 'New Scheme', the vast majority of time is spent on calculations in several functions of BLAS, which indicates a more efficient implementation in terms of performance. It should be noted that the difference in the names of the BLAS functions used in the considered algorithms is likely caused by differences in the implementation of the parallel BLAS by means of OpenMP and TBB inside the MKL library.

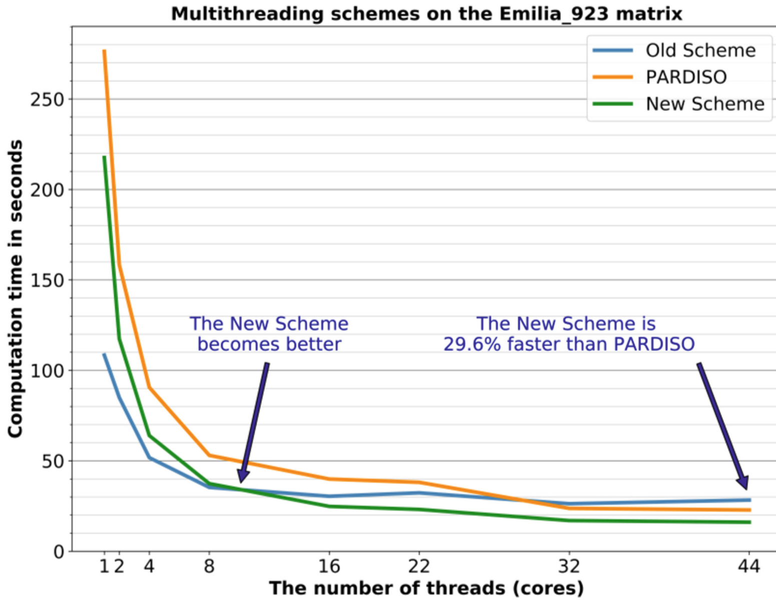

Fig. 7. Computation times of the scheme based on OpenMP ('Old Scheme'), MKL PARDISO, and the proposed scheme based on TBB ('New Scheme') when factorizing the Emilia_923 matrix. The number of threads (cores) varies from 1 to 44. Time is given in seconds.

Function	Module	CPU Time
__kmp_fork_barrier	libiomp5.so	658.379s
[MKL BLAS]@dsyrk	libmkl_intel_thread.so	204.680s
__intel_avx_rep_memset	omp_solver	58.498s
[MKL LAPACK]@dpotrf	libmkl_intel_thread.so	54.043s
omp_driver_recursive	libmkl_intel_thread.so	47.210s
[Others]		142.479s

Fig. 8. Hotspots of the 'Old Scheme' collected by Intel Amplifier

Function	Module	CPU Time
[MKL BLAS]@avx512_dgemm_kernel_nocopy_NT_b1	libmkl_avx512.so	203.477s
[MKL BLAS]@avx512_dgemm_kernel_0	libmkl_avx512.so	147.496s
[vmlinux]	vmlinux	31.096s
cblas_daxpyi	libmkl_intel_lp64.so	24.258s
__intel_avx_rep_memcpy	tbb_solver	22.252s
[Others]		100.119s

Fig. 9. Hotspots of the proposed 'New Scheme' collected by Intel Amplifier

6 Conclusion

In this paper, we presented a new scheme for the organization of parallelism when performing the numerical phase of the Cholesky decomposition for symmetric positive-definite sparse matrices. This scheme is based on the transparent creation of logical tasks that can encapsulate both the processing of the next node of the elimination tree, and individual BLAS operations. It results in a flexible load balancing scheme that allows us to dynamically assign tasks to threads, utilizing the available computational resources. The scheme is implemented using the TBB library and uses the library's scalable allocators for smart memory management. The results of experiments on the two 22-core Intel Skylake CPUs show that the performance and strong scaling efficiency of the described implementation is competitive to Intel MKL PARDISO.

References

1. Agullo, E., Buttari, A., Guermouche, A., Lopez, F.: Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. ACM Trans. Math. Softw. (TOMS) **43**(2), 13 (2016). https://doi.org/10.1145/2898348
2. Amestoy, P.R., Duff, I.S., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. **23**(1), 15–41 (2001). https://doi.org/10.1137/s0895479899358194
3. Amestoy, P.R., Duff, I.S., L'excellent, J.Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods Appl. Mech. Eng. **184**(2–4), 501–520 (2000). https://doi.org/10.1016/S0045-7825(99)00242-X
4. Chen, Y., Davis, T.A., Hager, W.W., Rajamanickam, S.: Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. ACM Trans. Math. Softw. (TOMS) **35**(3), 22 (2008). https://doi.org/10.1145/1391989.1391995
5. Davis, T.A.: Direct Methods for Sparse Linear Systems, vol. 2. Siam, Philadelphia (2006)
6. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Oxford (2017)
7. Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear. ACM Trans. Math. Softw. (TOMS) **9**(3), 302–325 (1983). https://doi.org/10.1145/356044.356047
8. Duff, I.S., Reid, J.K.: The multifrontal solution of unsymmetric sets of linear equations. SIAM J. Sci. Stat. Comput. **5**(3), 633–641 (1984). https://doi.org/10.1137/0905045
9. Duff, I., Hogg, J., Lopez, F.: A new sparse symmetric indefinite solver using A Posteriori Threshold Pivoting (2018)
10. Duff, I., Lopez, F.: Experiments with sparse Cholesky using a parametrized task graph implementation. In: Wyrzykowski, R., Dongarra, J., Deelman, E., Karczewski, K. (eds.) PPAM 2017. LNCS, vol. 10777, pp. 197–206. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78024-5_18
11. Hogg, J.D., Reid, J.K., Scott, J.A.: Design of a multicore sparse Cholesky factorization using DAGs. SIAM J. Sci. Comput. **32**(6), 3627–3649 (2010). https://doi.org/10.1137/090757216
12. Kalinkin, A., Anders, A., Anders, R.: Intel® math kernel library parallel direct sparse solver for clusters. In: EAGE Workshop on High Performance Computing for Upstream (2014). https://doi.org/10.3997/2214-4609.20141926

13. Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. J. Parallel Distrib. Comput. **48**(1), 71–95 (1998). https://doi.org/10.1006/jpdc.1997.1403
14. L'Excellent, J.Y.: Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects. Ph.D. thesis, Ecole normale superieure de lyon-ENS LYON (2012)
15. L'Excellent, J.Y., Sid-Lakhdar, W.M.: A study of shared-memory parallelism in a multifrontal solver. Parallel Comput. **40**(3–4), 34–46 (2014). https://doi.org/10.1016/j.parco.2014.02.003
16. LaSalle, D., Karypis, G.: Efficient nested dissection for multicore architectures. In: Träff, J. L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 467–478. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_36
17. Lebedev, S., Akhmedzhanov, D., Kozinov, E., Meyerov, I., Pirova, A., Sysoyev, A.: Dynamic parallelization strategies for multifrontal sparse Cholesky factorization. In: Malyshkin, V. (ed.) International Conference on Parallel Computing Technologies, pp. 68–79. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21909-7_7
18. Lebedev, S., Meyerov, I., Kozinov, E., Akhmedzhanov, D., Pirova, A., Sysoyev, A.: Two-level parallel strategy for multifrontal sparse Cholesky factorization. Vestnik UGATU **19**(3 (69)), 178–189 (2015)
19. Li, X.S., Demmel, J.W.: SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans. Math. Softw. (TOMS) **29**(2), 110–140 (2003). https://doi.org/10.1145/779359.779361
20. Liu, J.W.: The multifrontal method for sparse matrix solution: theory and practice. SIAM Rev. **34**(1), 82–109 (1992). https://doi.org/10.1137/1034004
21. Pellegrini, F.: Scotch and libScotch 6.0 User's Guide. Technical report, LaBRI (2012)
22. Pirova, A., Meyerov, I., Kozinov, E., Lebedev, S.: PMORSy: parallel sparse matrix ordering software for fill-in minimization. Optim. Methods Softw. **32**(2), 274–289 (2017). https://doi.org/10.1080/10556788.2016.1193177
23. Schreiber, R.: A new implementation of sparse Gaussian elimination. ACM Trans. Math. Softw. (TOMS) **8**(3), 256–276 (1982). https://doi.org/10.1145/356004.356006
24. Sid-Lakhdar, W.M.: Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures. PhD Thesis, prepared at ENS Lyon (2014)
25. Tang, M., Gadou, M., Rennich, S., Davis, T.A., Ranka, S.: Optimized sparse Cholesky factorization on hybrid multicore architectures. J. Comput. Sci. **26**, 246–253 (2018). https://doi.org/10.1016/j.jocs.2018.04.008
26. The SuiteSparse matrix collection. https://sparse.tamu.edu
27. The Threading Building Blocks library. https://www.threadingbuildingblocks.org