PMORSy: Parallel Sparse Matrix Ordering Software for Fill-in Minimization[1]

Anna Pirova[2], Iosif Meyerov, Evgeniy Kozinov, Sergey Lebedev

Lobachevsky State University of Nizhni Novgorod, Institute of Information Technology, Mathematics and Mechanics, Nizhni Novgorod, Russia

Abstract.
In this paper we present PMORSy – a new parallel software package for symmetric sparse matrix ordering on shared memory systems. The NP-complete fill-in minimization problem is solved by means of multilevel nested dissection algorithm with modifications for vertex separators. Parallel processing is done in a task-based fashion with the granularity tuning. We employ threading techniques on shared memory using OpenMP 3.0 technology as opposed to the MPI-based approach widely used for parallel sparse matrix ordering. Experimental results on symmetric matrices from the University of Florida Sparse Matrix Collection and matrices from finite-element analysis of three-dimensional strength problems show that our implementation is competitive to the ParMETIS and PT-Scotch libraries both in ordering quality and performance. The PMORSy library is publicly available from the UNN HPC Center web-site.

Keywords. Fill-in minimization, Multilevel nested dissection, Sparse matrix ordering, Cholesky factorization, Parallel computing, Task-based parallel processing

1. Introduction

Sparse systems of linear equations arise from numerical simulations in various scientific and engineering applications. The order of such systems often attains 10^6– 10^8, which makes their solution computationally intensive. Direct methods based on matrix factorization and backward solvers for triangular systems are extensively used due to their accuracy and robustness [25]. Nevertheless, the number of non-zero elements during the factorization step can dramatically increase over the initial matrix. This so-called *fill-in* significantly enlarges memory consumption and highly influences the run time of the most time consuming factorization step. To diminish this effect, symmetric *reordering* of the initial matrix rows and columns is performed.

Unfortunately, the problem of finding the ordering that minimizes the factor fill-in is NP-complete [37]. As for many discrete optimization problems [3, 8, 29], heuristic methods are commonly applied. For the factor fill-in minimization problem two main approaches are based on the minimum degree and nested dissection algorithms. The minimum degree algorithm was proposed by Tinney and Walker in 1969 [36]. It models the Gaussian elimination process and is based on the local factor minimization strategy. Since 1980s a number of modifications of the minimum degree algorithm for improving its run time and

[1] This work was presented at the 5th International Conference on Network Analysis, held in Nizhni Novgorod, Russia, 18–20 May, 2015.
[2] Corresponding author. Email: pirova@itmm.unn.ru

quality have been developed, including Multiple Minimum Degree [27], Approximate Minimum Degree [1], Column Approximate Minimum Degree [7] and others. The nested dissection algorithm for finite element meshes was proposed by George in 1973 [12] and generalized for irregular graphs by Lipton, Rose, Tarjian [26], and George, Liu [11]. It is based on the global factor minimization strategy and divide-and-conquer principle. Since 1993 there have been lots of modifications of the nested dissection algorithm employing the multilevel graph partitioning procedure. This approach was proposed by Bui and Jonse [4] and improved by Hendrickson and Leland [14], Hendrickson and Rothberg [15], Karypis and Kumar [21], and others.

Both minimum degree and nested dissection methods are widely used in practice. The main peculiarity of the minimum degree algorithm is its sequential nature, while divide-and-conquer nested dissection algorithm has a good potential for parallel processing and additionally benefits parallelization in further matrix factorization [10]. The major challenge in the implementation of the nested dissection algorithm is in finding a trade-off between the resulting fill-in and run time on modern computational systems. As for many graph algorithms, there are several factors that significantly affect computational time: list processing, indirect memory accesses, domination of memory operations over arithmetic operations, computational imbalance during divide-and-conquer procedure. These kinds of computations do not naturally fit present-day architectures. Thus, special efforts are required to create an efficient implementation and there is a great interest in new advancements in this area.

There are remarkable achievements in high-performance implementation of the nested dissection algorithm. METIS library employs a sequential version of the multilevel nested dissection scheme [18, 21]. METIS is broadly used in various research and development projects, e.g. as part of Intel MKL PARDISO [16]. Scotch is another software library that offers sequential implementation of sparse matrix ordering algorithms. Scotch utilizes a hybrid approach based on the combination of the incomplete multilevel nested dissection and a modified minimum degree algorithm [30, 31]. Applications usually employ parallel versions of both libraries, ParMETIS [19, 20, 35] and PT-Scotch [5]. These versions are designed for distributed memory systems, take advantage of MPI and demonstrate reasonable speedup.

The advancements of computer architectures over the last decade, especially the development of multicore and manycore architectures, create new opportunities and new challenges in the optimization of ordering algorithms. The simplest way to utilize such architectures in existing MPI-based software is to run an appropriate number of processes on shared memory without any code modification. . A significant drawback of this decision is potential performance loss due to non-use of the most effective parallel technologies for shared memory systems (pthreads, OpenMP, Cilk Plus, and TBB). Another possible solution is to parallelize certain steps of multilevel nested dissection algorithm for shared memory systems. Related work in this field concerns the graph partitioning problem and is implemented in mt-Metis [23]and PT-Scotch [32]. However, even for graphs with 10^7 vertices, the run time of an individual step of the method can be under one second, which remarkably decreases the parallelization potential. A successful OpenMP-based parallel implementation of the nested dissection algorithm is done in commercial MKL PARDISO [17] solver. However, to the authors' knowledge, the algorithm is not published yet and cannot be used aside the solver mentioned.

In this paper we address the problem of parallelization of the nested dissection algorithm for shared memory systems. The key idea is to utilize computational cores via threading techniques employing load balancing in task-based paradigm with the granularity tuning. The proposed algorithm is implemented in software package PMORSy extending the previously developed MORSy library [33], which applies the classical multilevel nested dissection

algorithm with modifications for vertex separators. Parallel version of the library is publicly available from the UNN HPC Center web-site [28].

The rest of the paper is organized as follows: we give a formal problem statement and a brief description of the modified nested dissection algorithm in MORSy, present our parallel algorithm and analyze quality, performance and scalability of the implementation described in comparison to ParMETIS and PT-Scotch on a 16-core CPU. All computational experiments are performed on matrices from the University of Florida Sparse Matrix Collection [6] and matrices generated by LOGOS software [34] during finite-element analysis of three-dimensional strength problems (LOGOS-FEM Collection).

2. Problem Statement

Let $A = (a_{ij})$ be a sparse symmetric n by n matrix. A matrix graph $G = (V, E)$ is defined as follows. Each vertex $v_i \in V$ corresponds to a matrix row i ($i = 1, 2, ..., n$). Each edge corresponds to a non-diagonal non-zero element of the matrix, i.e. $(v_i, v_j) \in E$ if and only if $a_{ij} \neq 0$ ($i, j = 1, 2, ..., n$; $i \neq j$). The set of vertices adjacent to a vertex v is denoted by Adj(v).

When *elimination* of a vertex v from a graph G is performed, the edges between vertices adjacent to the vertex v are added to the graph so that they form a clique, the vertex v is deleted from the set of vertices together with all incident edges:

$$V = V \setminus v;\ E = E \setminus \{(u, v): u \in \mathrm{Adj}(v)\} \cup \{(u_1, u_2): u_1, u_2 \in \mathrm{Adj}(v)\}.$$

The added edges are associated with the elements that become non-zero during the Gaussian elimination of v-th matrix row.

Let $\pi = (\pi_1, \pi_2, ..., \pi_n)$ be a permutation of the set of vertices V. *Fill-in* $F(\pi)$ generated by the permutation π is a set of edges added during the consecutive elimination of vertices $\pi_1, \pi_2, ..., \pi_n$. The problem is to find the permutation π^* that minimizes the number of edges in the produced fill-in:

$$\pi^* = \mathrm{argmin}\ \{|F(\pi)|\}.$$

This problem is NP-complete [37] and is usually solved using heuristic approaches.

3. Multilevel Nested Dissection in MORSy

Earlier we presented MORSy – a sequential library for sparse symmetric matrix ordering for fill-in minimization [28, 33] MORSy is written in C and runs under Windows and Linux.

Sparse matrix ordering in MORSy applies the classical multilevel nested dissection method (see the Algorithm 1). The nested dissection algorithm is built on a divide-and-conquer principle. At each step of the method a separator is found, that is, a small number of vertices whose elimination divides the graph into disconnected parts. After that, the algorithm operates recursively on those parts. Finding a graph separator using the *multilevel* method consists of three stages: *coarsening*, *partitioning*, and *uncoarsening*. At the *coarsening* stage, a sequence of graphs $G_1, G_2, \ldots, G_m$ is constructed, where the structure of G_{i+1} coarsens the structure of G_i. At the partitioning stage, the separator and two disconnected parts of the coarsest graph G_m are determined. At the uncoarsening stage, the separator of the coarsest graph is projected back to the original graph G_1 through the sequence of intermediate graphs $G_{m-1}, G_{m-2}, \ldots, G_2$. Obtaining a separator of the graph G_i is performed by projecting the separator of the graph G_{i+1} to G_i and refining it to reduce the separator size and balance the subgraphs. For this purpose, different modifications of iterative methods by Kernighan, Lin [22] and Fiduccia, Mattheyses [9] are applied.

Algorithm 1 Multilevel Nested Dissection in MORSy

Input: Graph $G_A(V_A, E_A)$ constructed from a sparse symmetric matrix A. V_A is a set of graph vertices, E_A is a set of graph edges. Parameter m defines the number of coarsening steps.
Output: *Iperm* – a new numbering of vertices of G_A (rows of A).

1: **function** NDStep(graph $G_0(V_0, E_0)$, parameter m) // the step of the multilevel nested dissection
2: **if** $|V_0|$ is small enough **then**
3: Save the order of separators obtained by automatic nested dissection into S_0.
4: **else**
5: **for** $i = 0$ to m **do**
6: **Coarse** $G_i(V_i, E_i)$ to $G_{i+1}(V_{i+1}, E_{i+1})$
7: **end for**
8: **Initialize Partition** $P_m(S_m, V_{m,1}, V_{m,2})$ of G_m with separator S_m and disconnected parts $V_{m,1}$, $V_{m,2}$
9: **for** $i = m$ **downto** 1 **do**
10: **Project Partition** P_i of G_i to partition $P_{i-1}(S_{i-1}, V_{i-1,1}, V_{i-1,2})$ of G_{i-1}
11: **Refine Partition** P_{i-1}
12: **end for**
13: **Find subgraphs** $G_1^*, G_2^*, ..., G_k^*$ obtained after removing S_0 from G_0
14: **end if**
15: **return** $(S_0, G_1^*, G_2^*, ..., G_k^*)$
16: **end function**

17: **function** MORSyOrdering(graph G_A, parameter m) // main ordering function
18: **Compress** G_A to obtain graph $G(V, E)$
19: Create an empty queue of subgraphs Q, add G to it
20: **while** queue Q is not empty **do**
21: Dequeue $G_0(V_0, E_0)$
22: $(S_0, G_1^*, G_2^*, ..., G_k^*) = \text{NDStep}(G_0(V_0, E_0), m)$
23: Enqueue $G_1^*, G_2^*, ..., G_k^*$
24: Save S_0 into a shared array S
25: **end while**
26: Number vertices from separator array S in reverse order
27: Form *Iperm* by projecting vertices numbers of from G to G_A
28: **return** *Iperm*
29: **end function**

Before the start of the multilevel nested dissection, graph compression is executed by the algorithm described in [15] to reduce the ordering time (line 18 of the Algorithm 1). During the multilevel nested dissection method heavy edge matching or random matching [21] is used at the coarsening stage (lines 5–7 of the Algorithm 1), finding the separator from the rooted level structure is employed at the partitioning stage (line 8 of the Algorithm 1), the modification of the Primitive moves algorithm by Ashcraft and Liu [2] and Hendrickson and Rothberg [15] is used at the refinement stage (lines 9–12 of the Algorithm 1).

The Primitive moves algorithm includes two nested loops. Each iteration of the outer loop over partitions is done as follows. First, we fix the current partition of the graph that defines the separator and two disconnected subgraphs. It is succeeded by moving vertices from the separator to one of the subgraphs according to the rule [2][9] in the inner loop to improve the quality of the partition. If a better partition is found, the outer loop continues with that partition, otherwise it stops.

Algorithm 2 Parallel Multilevel Nested Dissection in PMORSy

Input: Graph $G_A(V_A, E_A)$ constructed from a sparse symmetric matrix A. V_A is a set of graph vertices, E_A is a set of graph edges. Parameter m defines the number of coarsening steps.
Output: *Iperm* – a new numbering of vertices of G_A (rows of A).

```
1: function NDStepParallel(graph G0(V0, E0), parameter m) // the step of the multilevel
                                                          nested dissection
2:   (S0, G1*, G2*, ..., Gk*) = NDStep(G0(V0, E0), m)
3:   for i = 0 to k do
4:   #pragma omp task if (Gi* is big enough)
5:     NDStepParallel(Gi*(Vi, Ei), m)
6:   end for
7: end function
8: function MORSyOrdering(graph GA, parameter m) // main ordering function
9:   Compress GA to graph G(V, E)
10:  #pragma omp parallel
11:    #pragma omp single
12:      NDStepParallel(G(V, E), m)
13:  end omp parallel
14:  Number vertices from separator array S in reverse order
15:  Form Iperm by projecting vertices numbers of from G to GA
16:  return Iperm
17: end function
```

Our modification of the Primitive moves algorithm is the revised condition of performing the vertex move from the separator to one of the disconnected parts. As opposed to the original scheme, we perform moves only to the smaller part of the graph at the beginning of the current iteration of the outer loop.

A more detailed description of the algorithm is presented in [33].

4. Parallel Algorithm for Shared Memory Systems

Parallel ordering in PMORSy is developed in a task-based fashion employing threading techniques and granularity tuning. We parallelize the nested dissection algorithm in MORSy (Algorithm 1) for shared memory systems using task parallelism in OpenMP. For this purpose we modify the reordering algorithm from the iterative form into a recursive one, in which every NDStep() call corresponds to a logical task that can be solved independently by a free thread. Small enough subgraphs are processed as part of the same task (granularity tuning). The automatic OpenMP scheduler provides load balancing and distributes tasks among threads (Algorithm 2).

5. Experimental Results and Discussion

5.1 Test environment

PMORSy was tested using symmetric positive definite matrices from the University of Florida Sparse Matrix Collection [6] (Table 1) and LOGOS-FEM Collection (Table 2). We compared PMORSy with two state-of-the-art libraries: ParMETIS v.4.0.3 and PT-Scotch v.6.0.3. For ParMETIS and PT-Scotch, each matrix was equally distributed between processes. PT-Scotch was run under METIS-comparable interface. ParMETIS and PT-Scotch was run with the default parameters of ordering routines.

Table 1. Description of the test matrices from the University of Florida Sparse Matrix Collection. N is the number of matrix rows, NZ is the number of matrix non-zeros.

Matrix	N	NZ	NZ/N^2
Pwtk	217 918	5 926 171	1.25E-04
Msdoor	415 863	10 328 399	5.97E-05
parabolic_fem	525 825	2 100 225	7.60E-06
tmt_sym	726 713	2 903 835	5.50E-06
boneS10	914 898	28 191 660	3.37E-05
Emilia_923	923 136	20 964 171	2.46E-05
audkiw_1	943 695	39 297 171	4.41E-05
bone010	986 703	36 326 514	3.73E-05
ecology2	999 999	2 997 995	3.00E-06
thermal2	1 228 045	4 904 179	3.25E-06
StocF-1465	1 465 137	11 235 263	5.23E-06
Hook_1498	1 498 023	31 207 734	1.39E-05
Flan_1565	1 564 794	59 485 419	2.43E-05
G3_circuit	1 585 478	4 623 152	1.84E-06

Table 2. Description of the test matrices from LOGOS-FEM Collection. N is the number of matrix rows, NZ is the number of matrix non-zeros.

Matrix	N	NZ	NZ/N^2
cyclik	60 984	1 909 788	5.14E-04
Kamaz_kollekt	135 669	2 168 968	1.18E-04
podves	269 178	2 333 392	3.22E-05
lopatka1	274 104	10 370 664	1.38E-04
Kamaz_gusev	1 429 158	50 191 148	2.46E-05
trubka	2 428 323	70 338 539	1.19E-05
Korpus	2 504 934	93 368 914	8.14E-06
lopatka2	2 545 314	88 273 521	1.36E-05
49_750	2 615 169	97 081 773	1.42E-05
p4_6	4 216 212	144 714 294	8.14E-06

The quality of orderings was evaluated with respect to the number of non-zero elements in the factor of the reordered matrix. The quality of ordering produced by ParMETIS and PT-Scotch depends on the number of processes and usually decreases as this number grows. The quality of PMORSy ordering is the same for serial and parallel implementations. .

PMORSy was tested in two configurations. In the first configuration for each matrix we found the best parameters with respect to factor fill-in. We denote this configuration as 'best quality'. The second configuration allows taking orderings with the quality similar to ParMETIS. We denote this configuration as 'ParMETIS quality'. It shows the time needed by PMORSy for obtaining the ordering with ParMETIS-like quality.

Our experiments were performed on a cluster node with two 8-core Intel Sandy Bridge E5-2660 2.2 GHz, 64GB RAM, running Linux CentOS 6.4. The code was compiled using Intel C++ Compiler from Intel Cluster Studio XE 2013 SP1. Intel MKL library was used for random number generation.

Table 3. The performance of ParMETIS on test matrices using 1, 8 and 16 processes on a single node. NZ is the number of matrix non-zeros.

	Factor NZ			Time, sec.		
# processes / matrix	**1**	**8**	**16**	**1**	**8**	**16**
The University of Florida Sparse Matrix Collection						
pwtk	47 124 530	48 948 344	50 763 733	1.26	0.68	0.76
msdoor	51 483 893	52 234 967	53 160 419	2.13	1.09	1.02
parabolic_fem	25 607 853	25 626 615	25 600 254	4.08	1.09	0.74
tmt_sym	29 507 621	29 283 199	29 872 816	5.91	1.24	0.92
boneS10	268 565 124	268 893 070	275 620 364	10.62	3.13	2.50
Emilia_923	1 636 886 316	1 801 164 151	1 909 074 078	10.39	3.37	2.85
audikw_1	1 216 865 448	1 232 975 819	1 415 589 740	15.89	8.11	5.88
bone010	1 037 288 274	1 101 282 758	1 189 532 331	14.13	4.82	3.95
ecology2	35 641 736	33 907 697	35 058 149	6.72	1.42	0.96
thermal2	50 430 085	50 258 433	50 366 476	10.77	2.07	1.48
StocF-1465	1 035 811 119	1 062 713 367	1 056 205 648	21.79	4.28	3.15
Hook_1498	1 495 017 138	1 553 275 512	1 656 017 944	17.32	5.11	3.80
Flan_1565	1 456 370 148	1 522 924 200	1 582 973 048	22.82	6.09	4.75
G3_circuit	90 625 664	95 168 941	95 711 155	13.40	2.78	2.00
LOGOS-FEM Collection						
cyclik	21 738 321	24 708 907	24 527 841	1.21	0.78	1.36
Kamaz_kollekt	21 201 625	21 762 875	21 635 358	0.92	0.52	0.71
podves	27 510 284	29 252 522	28 549 496	3.06	0.79	0.87
lopatka1	221 594 040	242 367 410	239 491 446	3.54	1.51	1.90
Kamaz_gusev	599 222 021	797 046 640	848 729 839	21.00	7.85	6.32
trubka	1 313 242 750	1 343 085 976	1 379 674 538	49.74	15.01	8.54
Korpus	2 529 773 425	2 655 859 137	2 790 940 899	37.48	10.55	8.13
lopatka2	1 390 295 013	1 523 333 132	1 514 489 295	34.44	9.31	6.86
49_750	2 212 802 334	2 355 219 294	2 406 528 129	37.85	11.46	8.36
p4_6	2 401 305 585	2 438 307 255	2 568 883 906	59.80	15.90	12.78

5.2 ParMETIS and PT-Scotch orderings

Table 3 shows the quality and processing time of ParMETIS on test matrices. For most matrices the factor fill-in increases as the number of processes grows. The average speedup for 16 processes is 4.8 for the University of Florida Sparse Matrix Collection matrices and 3.6 on matrices from LOGOS-FEM Collection.

Table 4 shows the quality and processing time of PT-Scotch on test matrices. The average speedup for 16 processes is 6.9 for matrices from the University of Florida Sparse Matrix Collection and 6.7 on matrices from LOGOS-FEM Collection.

Table 4. The performance of PT-Scotch on test matrices using 1, 8 and 16 processes on a single node. NZ is the number of matrix non-zeros.

	Factor NZ			Time, sec.		
# processes / matrix	1	8	16	1	8	16
The University of Florida Sparse Matrix Collection						
pwtk	49 522 203	50 432 471	52 020 654	3.42	0.69	0.50
msdoor	60 790 300	62 840 085	64 561 972	6.50	1.68	1.30
parabolic_fem	29 078 475	29 944 981	30 622 333	2.58	0.83	0.64
tmt_sym	36 914 916	38 315 065	39 067 222	3.63	0.68	0.46
boneS10	295 951 174	329 567 485	351 489 532	23.80	4.10	2.92
Emilia_923	1 762 468 827	1 799 143 533	1 858 065 713	19.69	3.87	2.97
audikw_1	1 248 785 099	1 253 055 438	1 271 564 572	37.18	9.85	7.02
bone010	1 232 014 110	1 334 369 612	1 254 340 494	28.57	5.26	3.87
ecology2	45 319 057	46 989 800	46 520 507	4.26	0.82	0.60
thermal2	59 264 373	61 301 047	63 325 339	7.16	1.35	0.93
StocF-1465	1 124 071 466	1 166 743 560	1 168 194 673	18.08	3.30	2.32
Hook_1498	1 675 608 963	1 773 634 758	1 834 874 754	31.76	5.78	4.19
Flan_1565	1 548 018 276	1 585 325 535	1 625 188 876	49.19	8.18	5.59
G3_circuit	108 971 392	111 764 770	111 703 925	8.89	2.24	1.46
LOGOS-FEM Collection						
cyclik	24 123 440	27 145 380	26 814 364	1.53	0.41	0.35
Kamaz_kollekt	23 004 028	24 145 762	25 946 752	1.71	0.45	0.37
podves	33 213 664	36 059 421	36 437 044	2.24	0.56	0.47
lopatka1	234 178 088	260 651 063	248 632 544	7.58	1.58	1.27
Kamaz_gusev	631 697 329	663 903 867	704 744 769	42.46	10.86	8.06
trubka	1 540 544 221	1 591 819 760	1 673 724 390	76.81	15.76	10.31
Korpus	3 045 599 880	3 033 801 023	3 229 321 942	82.82	13.76	9.06
lopatka2	1 549 586 946	1 687 010 537	1 737 497 254	75.14	12.23	8.04
49_750	2 476 555 151	2 690 546 946	2 774 807 964	83.98	17.02	10.92
p4_6	3 006 381 215	3 363 028 106	3 266 817 392	126.60	23.61	17.38

5.3 PMORSy with 'best quality' parameters

This subsection presents a comparison of the time and quality of PMORSy orderings with 'best quality' parameters with ParMETIS and PT-Scotch results (Table 5).

First consider the matrices from the University of Florida Sparse Matrix Collection. Compared to ParMETIS, PMORSy produces orderings of 2% to 16% better quality for 11 matrices out of 14 in case 16 cores are used. As the fill-in advantage usually grows when the number of cores increases, the average gain in factor non-zeros is 8% for 16 cores. For the remaining matrices the size of the factor is 2% to 6% larger compared to ParMETIS. Compared to PT-Scotch using 16 cores, PMORSy gives orderings of better quality for 13 matrices out of 14 (8% to 37%). The factor fill-in disadvantage on the rest matrix is 4%.

For the matrices from LOGOS-FEM Collection, the difference between PMORSy, ParMETIS and PT-Scotch quality is smaller. Compared to ParMETIS, PMORSy produces orderings of 2% to 12% better quality for 9 matrices out of 10 when 16 cores are used.

Table 5. The factor fill-in after PMORSy ordering with 'best quality' settings relative to ParMETIS and PT-Scotch. NZ_PMORSy, NZ_ParMETIS, NZ_PT-Scotch are number of non-zeros after PMORSy, ParMETIS, PT-Scotch ordering respectively. Results less than or equal to 1.0 are better for PMORSy and emphasized in bold italic.

	NZ_PMORSy / NZ_ParMETIS			NZ_PMORSy / NZ_PT-Scotch		
# cores / **matrix**	**1**	**8**	**16**	**1**	**8**	**16**
The University of Florida Sparse Matrix Collection						
Pwtk	***0.98***	***0.95***	***0.91***	***0.94***	***0.92***	***0.89***
Msdoor	1.01	***1.00***	***0.98***	***0.86***	***0.83***	***0.81***
parabolic_fem	***0.97***	***0.97***	***0.97***	***0.86***	***0.83***	***0.81***
tmt_sym	***0.97***	***0.98***	***0.96***	***0.78***	***0.75***	***0.74***
boneS10	1.08	1.08	1.06	***0.98***	***0.88***	***0.83***
Emilia_923	1.01	***0.92***	***0.86***	***0.94***	***0.92***	***0.89***
audikw_1	1.08	1.07	***0.93***	1.06	1.05	1.04
bone010	***0.99***	***0.93***	***0.86***	***0.83***	***0.77***	***0.82***
ecology2	***0.83***	***0.87***	***0.84***	***0.65***	***0.63***	***0.63***
thermal2	1.02	1.02	1.02	***0.87***	***0.84***	***0.81***
StocF-1465	1.04	1.01	1.02	***0.95***	***0.92***	***0.92***
Hook_1498	1.06	1.02	***0.96***	***0.95***	***0.89***	***0.86***
Flan_1565	***0.97***	***0.93***	***0.89***	***0.91***	***0.89***	***0.87***
G3_circuit	***0.99***	***0.95***	***0.94***	***0.83***	***0.80***	***0.81***
LOGOS-FEM Collection						
Cyclik	***1.00***	***0.88***	***0.88***	***0.90***	***0.80***	***0.81***
Kamaz_kollekt	1.04	1.02	1.02	***0.96***	***0.92***	***0.85***
podves	***0.96***	***0.90***	***0.93***	***0.80***	***0.73***	***0.73***
lopatka1	***0.97***	***0.89***	***0.90***	***0.92***	***0.82***	***0.86***
Kamaz_gusev	1.14	***0.86***	***0.81***	1.08	1.03	***0.97***
Trubka	1.03	1.01	***0.98***	***0.88***	***0.85***	***0.81***
Korpus	1.03	***0.98***	***0.93***	***0.85***	***0.86***	***0.80***
lopatka2	1.07	***0.97***	***0.98***	***0.96***	***0.88***	***0.85***
49_750	***0.98***	***0.92***	***0.90***	***0.88***	***0.81***	***0.78***
p4_6	***0.96***	***0.95***	***0.90***	***0.77***	***0.69***	***0.71***

For the remaining matrix the size of the factor is 2% larger. Compared to PT-Scotch, PMORSy works 3% to 29% better on all matrices.

Ordering time of PMORSy compared to ParMETIS and PT-Scotch is shown in Table 6.

Considering the matrices from the University of Florida Sparse Matrix Collection, PMORSy works faster than ParMETIS and PT-Scotch on 8 matrices out of 14. The average advantage is 2.4 times compared to ParMETIS and 3.6 times compared to PT-Scotch. For the rest 6 matrices, PMORSy works slower than ParMETIS and PT-Scotch. The average disadvantage is 1.5 to 2.5 times compared to ParMETIS and 2.2 to 3.8 times compared to PT-Scotch depending on the number of cores.

Table 6. The comparison of PMORSy reordering time with 'best quality' settings compared to ParMETIS and PT-Scotch. T_PMORSy, T_ParMETIS, T_PT-Scotch are reordering time of PMORSy, ParMETIS, PT-Scotch respectively. Results smaller than 1.0 are better for PMORSy and emphasized in bold italic.

	T_PMORSy / T_ParMETIS			T_PMORSy / T_PT-Scotch		
# cores / **matrix**	**1**	**8**	**16**	**1**	**8**	**16**
The University of Florida Sparse Matrix Collection						
Pwtk	***0.29***	***0.22***	***0.20***	***0.11***	***0.22***	***0.30***
Msdoor	***0.37***	***0.26***	***0.26***	***0.12***	***0.17***	***0.20***
parabolic_fem	1.27	1.35	1.71	2.02	1.77	1.97
tmt_sym	1.65	2.22	2.58	2.68	4.06	5.20
boneS10	***0.44***	***0.47***	***0.52***	***0.20***	***0.36***	***0.45***
Emilia_923	***0.45***	***0.51***	***0.56***	***0.24***	***0.44***	***0.53***
audikw_1	***0.51***	***0.34***	***0.41***	***0.22***	***0.28***	***0.35***
bone010	***0.54***	***0.54***	***0.65***	***0.27***	***0.50***	***0.67***
ecology2	1.44	2.11	2.98	2.28	3.64	4.75
thermal2	1.63	2.31	2.75	2.45	3.57	4.40
StocF-1465	1.78	2.54	2.99	2.15	3.30	4.07
Hook_1498	***0.56***	***0.65***	***0.86***	***0.31***	***0.57***	***0.78***
Flan_1565	***0.55***	***0.63***	***0.71***	***0.26***	***0.47***	***0.61***
G3_circuit	1.20	1.56	1.78	1.80	1.94	2.45
LOGOS-FEM Collection						
cyclik	***0.49***	***0.35***	***0.18***	***0.39***	***0.66***	***0.72***
Kamaz_kollekt	***0.53***	***0.40***	***0.31***	***0.29***	***0.46***	***0.60***
podves	1.03	1.35	1.32	1.41	1.92	2.40
lopatka1	***0.33***	***0.30***	***0.23***	***0.16***	***0.29***	***0.34***
Kamaz_gusev	***0.40***	***0.35***	***0.40***	***0.20***	***0.26***	***0.31***
trubka	***0.95***	***0.87***	1.28	***0.62***	***0.83***	1.06
Korpus	***0.39***	***0.46***	***0.53***	***0.18***	***0.35***	***0.47***
lopatka2	***0.66***	***0.73***	***0.93***	***0.30***	***0.56***	***0.79***
49_750	***0.46***	***0.46***	***0.54***	***0.21***	***0.31***	***0.41***
P4_6	***0.59***	***0.64***	***0.68***	***0.28***	***0.43***	***0.50***

Finally, consider the matrices from LOGOS-FEM Collection. PMORSy works faster than ParMETIS and PT-Scotch on all but one matrix for 1 to 8 cores and on 8 of 10 matrices for 16 cores. The disadvantage on the rest matrices is up to 1.3 times compared to ParMETIS and is 1.4 to 2.4 times compared to PT-Scotch.

As a result, PMORSy works better than ParMETIS in the matter of both quality and time on 5 matrices out of 14 from the University of Florida Sparse Matrix Collection and on 4 matrices out of 10 from LOGOS-FEM Collection. Compared to PT-Scotch, it works better on 5 matrices out of 14 from the University of Florida Sparse Matrix Collection and on 7 matrices out of 10 from LOGOS-FEM Collection. Only on one matrix from two collections PMORSy is behind ParMETIS in both speed and quality.

Table 7. The comparison of PMORSy reordering time with 'ParMETIS quality' settings relative to ParMETIS. T_PMORSy, T_ParMETIS are reordering time of PMORSy and ParMETIS respectively. Results smaller than 1.0 are better for PMORSy and emphasized in bold italic.

	T_PMORSy / T_ParMETIS		
# cores / **matrix**	**1**	**8**	**16**
The University of Florida Sparse Matrix Collection			
Pwtk	***0.29***	***0.20***	***0.17***
Msdoor	***0.29***	***0.23***	***0.21***
parabolic_fem	***0.85***	***0.94***	1.19
tmt_sym	***0.97***	1.28	1.43
boneS10	***0.44***	***0.48***	***0.52***
Emilia_923	***0.47***	***0.40***	***0.32***
audikw_1	***0.51***	***0.34***	***0.25***
bone010	***0.48***	***0.54***	***0.36***
ecology2	***0.87***	1.15	1.48
thermal2	1.63	2.32	2.71
StocF-1465	1.31	1.88	2.22
Hook_1498	***0.56***	***0.64***	***0.54***
Flan_1565	***0.35***	***0.40***	***0.44***
G3_circuit	***0.90***	***0.99***	1.16
LOGOS-FEM Collection			
cyclik	***0.49***	***0.26***	***0.14***
Kamaz_kollekt	***0.53***	***0.38***	***0.29***
podves	***0.96***	***0.96***	1.02
lopatka1	***0.30***	***0.29***	***0.22***
Kamaz_gusev	***0.40***	***0.34***	***0.36***
trubka	***0.77***	***0.71***	1.05
Korpus	***0.39***	***0.37***	***0.54***
lopatka2	***0.66***	***0.54***	***0.49***
49_750	***0.46***	***0.42***	***0.51***
p4_6	***0.34***	***0.42***	***0.42***

5.4 PMORSy with 'ParMETIS quality' parameters

We compare PMORSy and ParMETIS with close orderings in terms of factor fill-in (Table 7). In this case, for the majority of matrices processing time is reducing, because it is allowed to obtain more non-zero elements in the factor, than with better quality adjustments. Exception to this tendency applies when PMORSy with maximal quality adjustments is behind ParMETIS.

Our experiments show that for the University of Florida Sparse Matrix Collection PMORSy works faster than ParMETIS on 12, 10 and 8 matrices out of 14 when ParMETIS uses 1, 8 and 16 cores, respectively. For LOGOS-FEM Collection, PMORSy works faster than ParMETIS on all matrices when 1 to 8 cores are used and on all but two matrices on 16 cores. On the remaining matrices PMORSy works 2% to 5% slower.

6. Results and Discussion

6.1 Performance and scalability

First we investigate behaviour of the parallel algorithm, particularly in terms of task mapping to threads. For that purpose we run multithreaded PMORSy on the *pwtk* matrix (8 threads) and on the *G3_circuit* matrix (16 threads). Then we build a task tree: nodes of the tree correspond to logical tasks (each task executed in a NDStepParallel() function call). Each node gets a value that is the number of vertices in the subgraph considered. For each generation of a subtask there is an edge of the tree. The task tree is essentially a recursion tree. Subtasks executed in the same thread are marked with the same color. The root vertex is placed in the center of the figure. Radial tree levels correspond to the depth of recursion. Tasks with the number of vertices under the granularity limit do not generate new tasks and thus are leaves of the tree. They are solved completely in the same thread. GraphViz library [13] was utilized to build the trees presented in this subsection.

The tree shown on Figure 1 is built for a relatively small matrix *pwtk*, so the described task tree building concepts can be observed. For a good understanding of the whole picture we consider a similar run for 16 threads and the matrix *G3_circuit* (figure 2). In distinction with the graph with a small number of vertices, black circles correspond to edges of the graph and visually separate levels from each other. Appearance of colored sectors implies subtrees are usually processed by the same thread as their root vertex. This results in a favorable memory access pattern as threads can reuse data in cache. The figure also shows that the algorithm creates a balanced task tree, in which the number of levels for different subtrees varies insignificantly.

For a more detailed analysis we continue by running the profiler Intel Amplifier XE to find the hotspots. We found that partition refinement is the most time consuming step of the algorithm taking 35% to 55% of the total time depending on the test matrix. Running time of that step is proportional to the number of the partition improvement attempts significantly affecting the fill-in. We found that introducing additional bounds on iteration number allowed twofold speedup of this step without significant quality degradation for most matrices. The second most time consuming stage of the algorithm is graph coarsening, which takes 20% to 30% of reordering time. Its execution time depends on the quantity of random numbers used for building a matching. Further analysis shows that, as quite usual for graph algorithms, poor branch prediction and irregular memory access pattern resulting in significant number of the last level cache and translation look aside buffer misses are the main reasons influencing the execution time. Thus, despite quite good task mapping, generally the problem of working with memory in graph-based algorithms is not resolved entirely in PMORSy and is one of the most promising fields to research.

Bringing scaling issue into analysis, mediocre speedup should be noted on 16 cores against 1 core. The reason behind it is that few largest tasks group near the root of the tree causing undersaturation of 16 threads for a significant portion of the computational time. This problem can be addressed by parallelization in the intra-task level. Two-level parallel scheme, combining data and task parallelism for solving large problems is planned to be developed to eliminate the effect described.

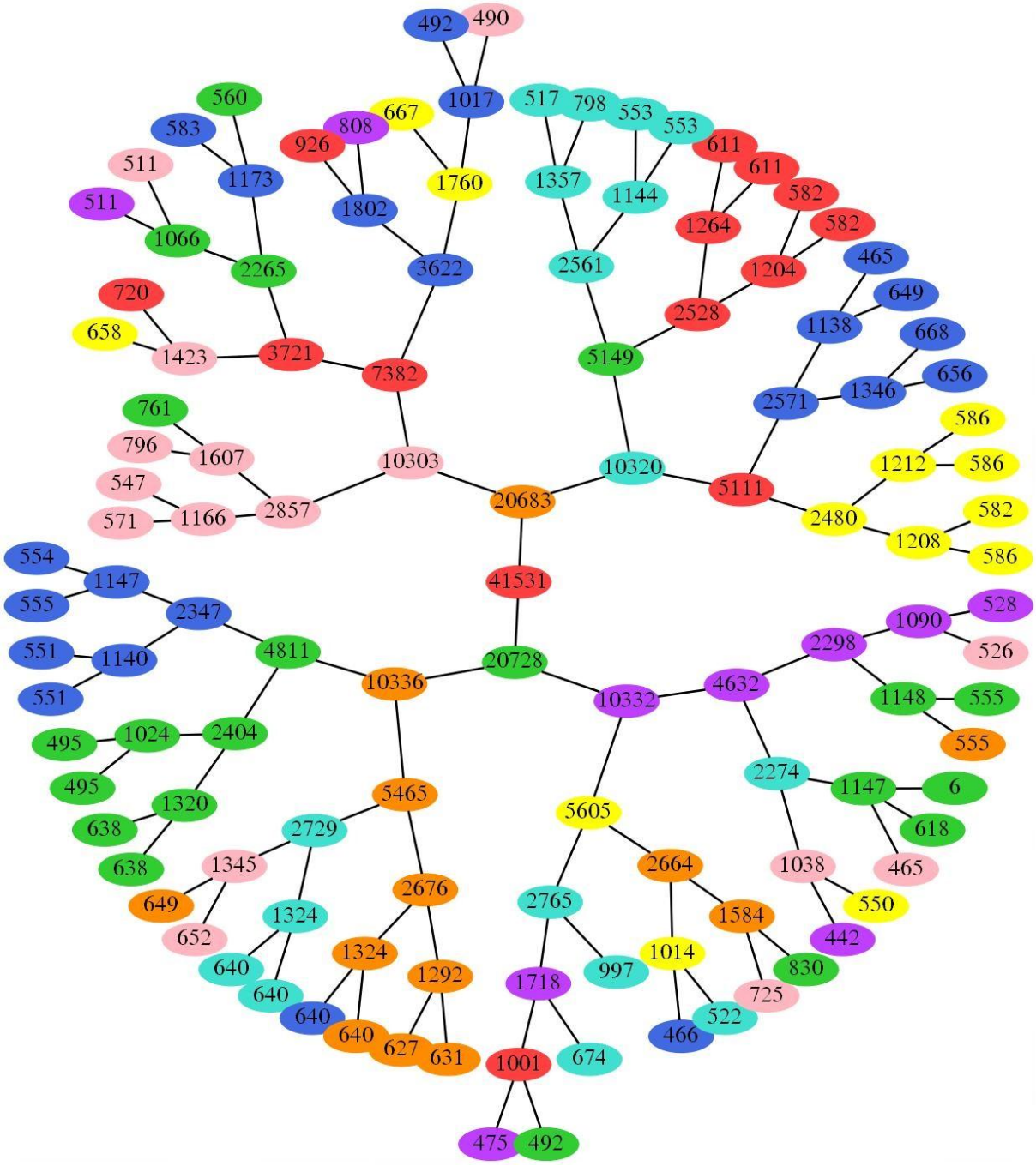

Fig. 1. Task mapping for *pwtk* matrix (Florida Matrix Collection) on 8 threads. Logical tasks are nodes of the graph, dependencies between them are edges. Descendant nodes correspond to the tasks generated after parent node calculating. Same colored nodes are processed by the same thread. Values of the nodes correspond to the number of graph nodes for which the separator search took place. Tasks for graph with less than 1000 nodes are considered as non-generating.

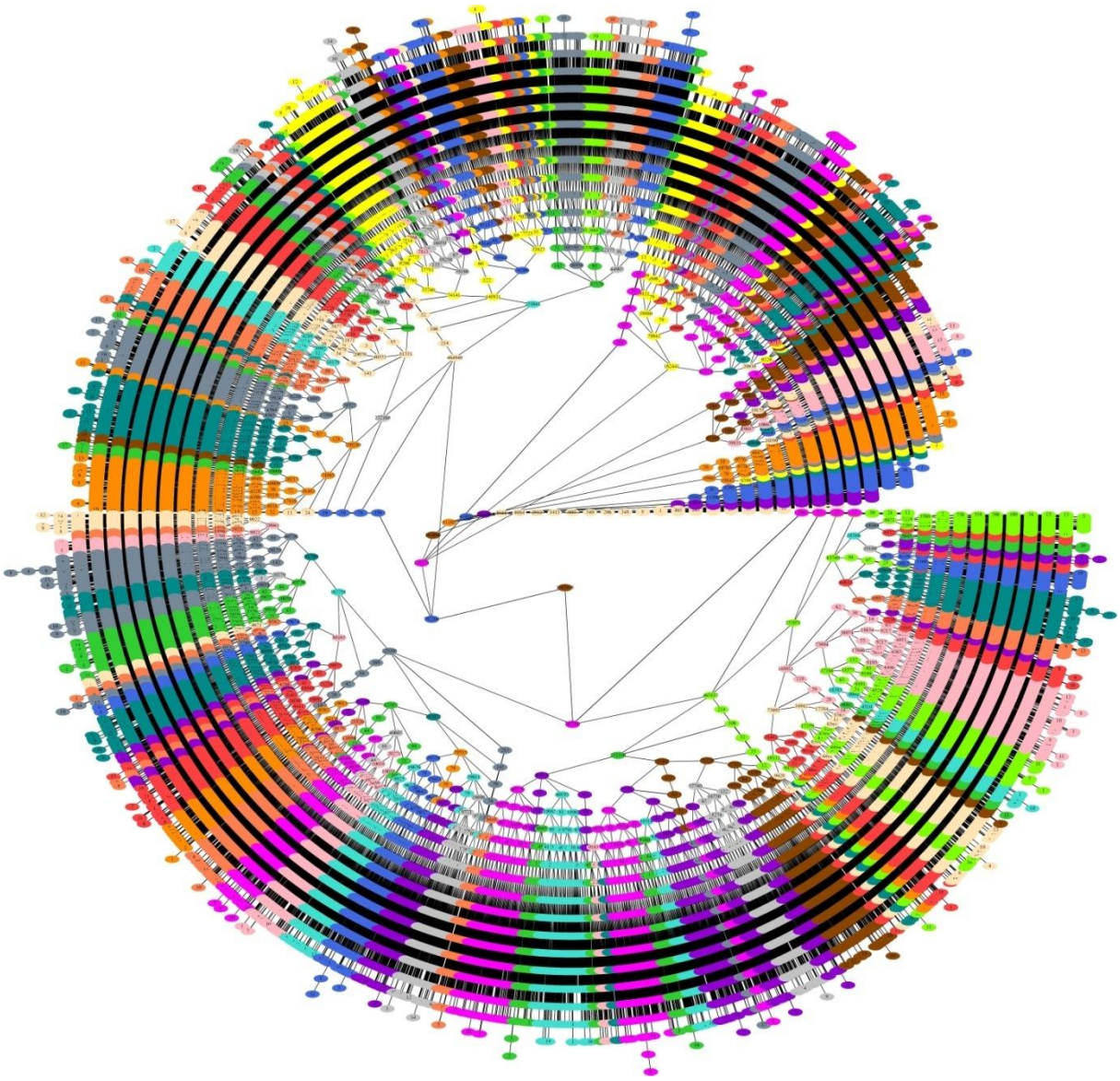

Fig. 2. Task mapping for *G3_circuit* matrix (Florida Matrix Collection) on 16 threads. Logical tasks are nodes of the graph, dependencies between them are edges. Descendant nodes correspond to the tasks generated after parent node calculating. Same colored nodes are processed by the same thread. Digits in the nodes coincide with the number of graph nodes for which the separator search took place. Tasks for graph with less than 3000 nodes are considered as non-generating.

6.2 Comparison to the state-of-the-art libraries

In this subsection we summarize the numerical results of PMORSy in two configurations, ParMETIS, and PT-Scotch. All the runs used 16 cores of the cluster node. The comparison of performance is shown on the left-hand side of Figure 3. Each bar of the graph corresponds to run time given in seconds. On average, PMORSy in the 'ParMETIS quality' configuration performs 40% faster than in the 'best quality' configuration. PMORSy in both configurations outperforms ParMETIS and PT-Scotch on 16 matrices out of 24 with the maximum advantage about 7.0 times on the *cyclik* matrix. Nonetheless, on the remaining 8 matrices ParMETIS and PT-Scotch work faster than PMORSy, up to 5.2 times on the *tmt_sym* matrix. The comparison of orderings quality is presented on the right-hand side of Figure 3. Each bar of the graph corresponds to the number on non-zero elements after factorization given in millions.

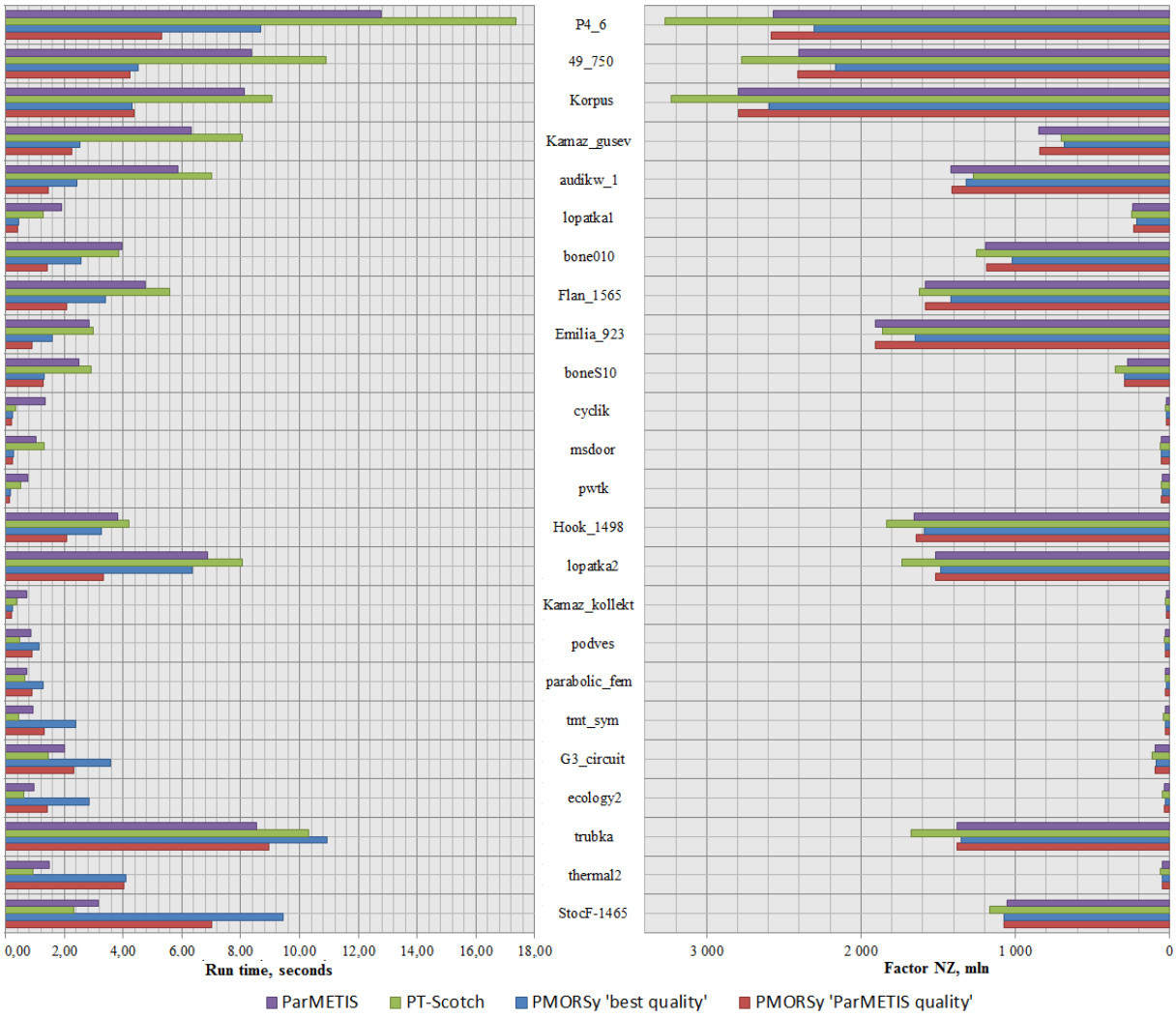

Fig. 3. The comparison of PMORSy, ParMETIS, and PT-Scotch with respect to run time (on the left) and orderings quality (on the right). The results are given on 24 test matrices from the University of Florida and LOGOS-FEM sparse matrix collections. 16 cores were used in all runs. Run time is given in seconds. Orderings quality is shown in the number (in million) of non-zero elements after factorization. Test matrices are ranked as follows: matrices, better for PMORSy in terms of run time compared to ParMETIS and PT-Scotch, are shown on the top.

PMORSy in the best configuration of the parameters produces up to 20% better or competitive orderings compared to ParMETIS on 20 matrices out of 24 and compared to PT-Scotch on 23 matrices out of 24. The disadvantage on the other matrices is at most 6%.

Compared to ParMETIS and PT-Scotch our implementation is better on the matrices that allow structure compression and have quite a large average number of vertices after compression (from 10 to 26 in our experiments). On the contrary, for the matrices with a large number of vertices and a small number of edges after the compression, ParMETIS outperforms our implementation. A possible reason is that the data structures used in our implementation are not well-suited for matrices with a small average vertex degree after structure compression due to relatively low cache efficiency and poor branch prediction. We plan to try other data structures and, probably, implement some heuristic rules to choose a data structure depending on graph properties.

7. Summary and Future Work

We have presented a new parallel nested dissection algorithm for shared memory systems. Parallel processing is performed using task-based technique. The algorithm is implemented using OpenMP technology as an extension to MORSy library [33].

Experimental results on matrices from the University of Florida Sparse Matrix Collection and LOGOS-FEM Collection show the competitiveness of our implementation with the widely used ParMETIS and PT-Scotch libraries. The quality of orderings produced by PMORSy is independent on the number of threads. For the most of test matrices, the number of non-zeros in the resulting factor obtained by PMORSy is better than that of ParMETIS and PT-Scotch. In terms of performance parallel PMORSy works faster on a half of test matrices. When PMORSy and ParMETIS give orderings with close number of non-zeros, PMORSy works faster on most test matrices.

While the paper was under review, a new version of mt-Metis library was introduced at the Euro-Par conference [24]. The authors of the library presented a two-level parallelization scheme of the multilevel nested dissection method. The scheme combines a custom task scheduling and parallelism of computationally intensive steps of the multilevel method. Our first experiments show that mt-Metis outperforms ParMETIS 1.5 times on average which is in good agreement with [24]. The trend of performance of PMORSy compared to mt-Metis is generally the same as for comparison to ParMETIS, with PMORSy being advantageous on matrices with greater average vertex degree after structure compression. However, the advantage on these matrices is smaller and the disadvantage on other matrices is larger.

The fact that the new version of mt-Metis, as well as PMORSy, outperforms ParMETIS on shared memory shows that hybrid (e.g. MPI + OpenMP) parallel algorithms are likely best suited for modern cluster systems with multicore nodes. Developing such an algorithm and its efficient implementation is the main direction of our future work.

Acknowledgments

The authors would like to thank Dmitry Akhmedzhanov, Sergey Bastrakov, Alexey Liniov, Alexander Sysoyev, and Nikolai Zolotykh for useful comments and discussions.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

The study was partially supported by the RFBR, research project No. 14-01-3145514 and by the grant 02.B.49.21.0003 of The Ministry of education and science of the Russian Federation.

References

[1] P.R. Amestoy, T. Davis, and I. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl. 17(4) (1996), pp. 886–905.

[2] C. Aschcraft and J.W.H. Liu, *A partition improvement algorithm for generalized nested dissection*, Tech. Rep. BCSTECH-94-020, Boeing Computer Services, Seattle, WA, 1994.

[3] Ch.-Ed. Bichot, P. Siarry (eds.), *Graph Partitioning*, John Wiley and Sons, New York, 2013.

[4] T.N. Bui and C. Jones, *A heuristic for reducing fill-in in sparse matrix factorization*, No. DOE/ER/25151--1-Vol. 1; CONF-930331-Vol. 1. SIAM, Philadelphia, PA, 1993.
[5] C. Chevalier and F. Pellegrini, *PT-Scotch: A tool for efficient parallel graph ordering*, Parallel Comput. 34(6) (2008), pp. 318–331.
[6] T.A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software 38(1) (2011), pp. 1.
[7] T.A. Davis, J.R. Gilbert, S. I. Larimore, and E. G. Ng, *A column approximate minimum degree ordering algorithm*, ACM Trans. Math. Software 30(3) (2004), pp. 353–376.
[8] A. Ferreira and P.M. Pardalos, *Solving Combinatorial Optimization Problems in Parallel-Methods and Techniques*. Lecture Notes in Comput. Sci., Springer Verlag, Berlin, Vol. 1054, 1996.
[9] C.M. Fiduccia and R.M. Mattheyses, *A linear time heuristic for improving network partitions*, 19th IEEE Conference on Design Automation (1982), pp. 175–181.
[10] A. George, M.T. Heath, J.W.H. Liu, and E. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor*, SIAM J. on Scientific and Stat. Comp. 9(2) (1988), pp. 327-340.
[11] A. George and J.W.H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal. 15(5) (1978), pp. 1053–1069.
[12] A. George, *Nested dissection of a regular finite element mesh*. SIAM J. Numer. Anal. 10(2) (1973), pp. 345–363.
[13] GraphViz, Graph Visualization Software; software available at http://www.graphviz.org/.
[14] B. Hendrickson and R. Leland, *A multi-level algorithm for partitioning graphs*. Tech. Rep. SAND93-1301, Sandia National Laboratories, 1993.
[15] B. Hendrickson and E. Rothberg, *Improving the run time and quality of nested dissection ordering*, SIAM J. Sci. Comput. 20 (1999), pp. 468–489.
[16] Intel Math Kernel Library Reference Manual. Available at http://software.intel.com/sites/default/files/managed/9d/c8/mklman.pdf
[17] Intel MKL PARDISO – Parallel Direct Sparse Solver Interface. Available at https://software.intel.com/en-us/node/521677.
[18] G. Karipis, *METIS. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices . V. 5.0*. Tech. Rep., University Of Minnesota, Department Of Computer Science And Engineering, 2011.
[19] G. Karypis and V. Kumar, *ParMetis: Parallel graph partitioning and sparse matrix ordering library*. Tech. Rep. TR 97-060, Department Of Computer Science, University Of Minnesota, 1997.
[20] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel Distrib. Comput. 48 (1998), pp. 71–85.
[21] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput. 20(1) (1998), pp. 359–392.
[22] B.W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical J., 29 (1970), pp. 291–307.
[23] D. Lasalle and G. Karypis, *Multi-threaded graph partitioning*. Parallel & Distributed Processing (IPDPS), 2013, pp. 225-236.
[24] D. LaSalle and G. Karypis, *Efficient Nested Dissection for Multicore Architectures*. Euro-Par 2015: Parallel Processing, 2015, Springer Berlin Heidelberg, pp. 467–478.
[25] J.-Y. L'Excellent, *Multifrontal methods: parallelism, memory usage and numerical aspects*, Doctoral diss., École normale supérieure de Lyon, 2012.

[26] R.J. Lipton, D.J. Rose, and R.E. Tarjan, *Generalized nested dissection*, SIAM J. Numer. Anal. 16(2) (1979), pp. 346–358.
[27] J.W.H. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Trans. Math. Software 11(2) (1985), pp. 141–153.
[28] MORSy software; software available at http://hpc-education.unn.ru/research/overview/sparse-algebra/morsy.
[29] P.M. Pardalos (Ed.), *Parallel processing of discrete problems*. The IMA Volumes in Mathematics and its Applications, Springer-Verlag, Vol. 106, 1999.
[30] F. Pellegrini, J. Roman, and P. Amestoy, *Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering*, Concurrency: Practice And Experience 12 (2000), pp. 68–84.
[31] F. Pellegrini, *Scotch and Libscotch 6.0 User's guide*, Tech. Rep., LABRI, 2012.
[32] F. Pellegrini, *Shared memory parallel algorithms in Scotch 6*. Available at http://graal.ens-lyon.fr/mumps/doc/ud_2013/pellegrini.pdf.
[33] A. Pirova and I. Meyerov, *MORSy – a new tool for sparse matrix reordering*, An Int. Conf. On Engineering And Applied Sciences Optimization, 2014, pp. 1952-1964.
[34] V.N. Rechkin, et al. *Software package LOGOS. Module for solving the quasi-static strength problems and modal analysis*, Proc. of XIII International seminar 'Supercomputations and mathematical modeling', Sarov, 2011 (in Russian).
[35] K. Schloegel, G. Karypis, and V. Kumar. *Parallel multilevel algorithms for multi-constraint graph partitioning*, Euro-Par 2000 Parallel Processing, 2000, pp. 296–310.
[36] W. Tinney and J. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proc. of the IEEE 55(11), 1967, pp. 1801–1809.
[37] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Algebraic and Discrete Methods 2(1) (1981), pp. 77–79.