A Multithreaded Recursive and Nonrecursive Parallel Sparse Direct Solver

Ercan Selcuk Bolukbasi and Murat Manguoglu

Abstract Sparse linear system of equations often arises after discretization of the partial differential equations (PDEs) such as computational fluid dynamics, material science, and structural engineering. There are, however, sparse linear systems that are not governed by PDEs, some examples of such applications are circuit simulations, power network analysis, and social network analysis. For solution of sparse linear systems one can choose using either a direct or an iterative method. Direct solvers are based on some factorization of the coefficient matrix such as the LU, QR, or singular value decompositions and are known to be robust. Classical preconditioned iterative solvers, on the other hand, are not as robust as direct solvers and finding an effective preconditioner is often problem dependent. Due to their sequential nature, direct solvers often have limited parallel scalability. In this chapter, we present a new parallel recursive sparse direct solver that is based on the sparse DS factorization. We implement our algorithm using MIT's Cilk programming language which is also a part of the Intel C++ compiler. We show the scalability and robustness of our algorithm and compare it to Pardiso direct solver.

1 Introduction

The chip producers can no longer efficiently increase the clock frequency of a processor. The Moore's law which originally stated that the number of transistors doubled every 2 years [18], is, however, still valid. The Moore's law could be translated as the number of cores double every 2 years today. As a result of this paradigm shift, researchers have been working on parallelizing the existing sequential algorithms in order to effectively use all the cores of the processors. A more innovative approach is to design a completely parallel algorithm with parallelism in mind.

Solution of sparse linear systems is required by many applications in science and engineering. Often, the linear solution step is the main performance bottleneck.

E.S. Bolukbasi • M. Manguoglu (✉)
Department of Computer Engineering, Middle East Technical University, Ankara, Turkey
e-mail: ercan@ceng.metu.edu.tr; manguoglu@ceng.metu.edu.tr

Y. Bazilevs, K. Takizawa (eds.), *Advances in Computational Fluid-Structure Interaction and Flow Simulation*, Modeling and Simulation in Science, Engineering and Technology, DOI 10.1007/978-3-319-40827-9_22

There are two main classes of linear system solvers, namely direct and iterative methods. While iterative solvers are not as robust as direct solvers [19], iterative solvers are considered to scale better on parallel computers. There are many parallel sparse direct solver implementations some of the most well known of these are SuperLU [5, 15], MUMPS [1], PARDISO [23–26], and WSMP [7, 8]. All of these direct solvers, however, are based on some form of the classical LU factorization, performed in parallel. After factorization, the system is solved using the computed factors via forward and backward sweeps. For most sparse systems there are some dependencies between unknowns, which limit the parallelism in factorization phase and, due to fill in, this is even more pronounced during the triangular solution phase.

To alleviate these drawbacks of the existing direct solvers we have developed a new general parallel sparse direct solver [2] based on the sparse DS factorization [3]. The idea of the DS factorization is first introduced in [20–22] for banded linear systems which is called the SPIKE algorithm due the structure of the S matrix. A recursive banded DS factorization is introduced in [21] which applies recursion on the S matrix. Our approach, on the other hand, is to apply the recursion on the smaller reduced system. A generalization of the banded DS factorization to sparse linear systems and its hybrid (direct/iterative) implementation, in which the reduced system is solved iteratively, is given in [16, 17].

Given a banded or sparse linear system of equations and number of blocks,

$$Ax = f \tag{1}$$

The DS decomposition factors A into two matrices D and S such that,

$$A = DS \tag{2}$$

where D is just the block diagonal of A. Hence the splitting $A = D + R$ where R is the remaining block off-diagonal entries of A. There is no computation to obtain D. The S matrix, on the other hand, is obtained by $S = D^{-1}A$ or taking advantage of the fact that the block diagonals of S is identity,

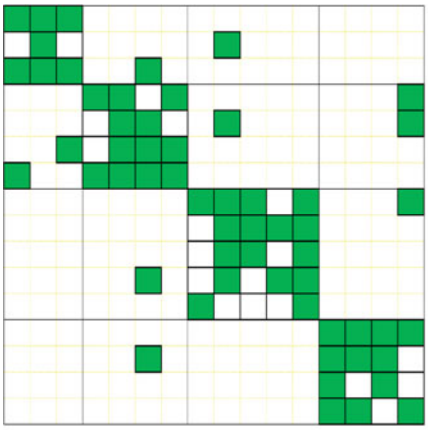
$$S = D^{-1}(D + R). \tag{3}$$

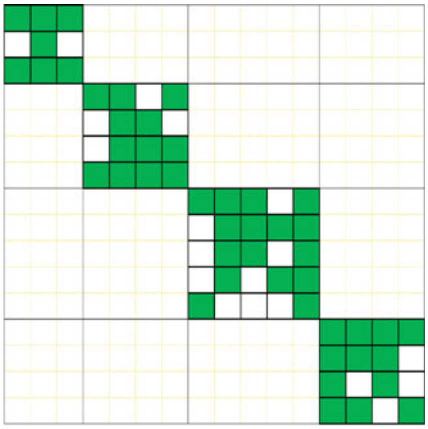
We obtain $S = I + G$ which involves solving independent systems in parallel to obtain $G = D^{-1}R$. After obtaining the DS factorization, if we multiply both sides of the Equation (1) with D^{-1} from left,

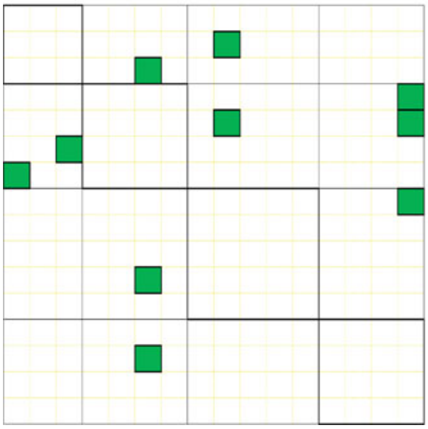
$$D^{-1}Ax = D^{-1}f, \tag{4}$$

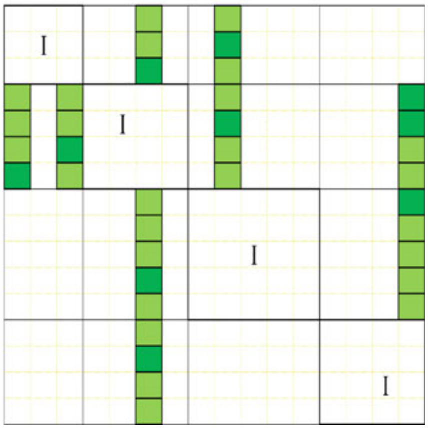
and obtain a new system

$$Sx = g \tag{5}$$


(a) An example sparse coefficient matrix.


(b) D matrix corresponding to the sparse matrix.


(c) R matrix such that $A = D + R$ for the small example.



(d) S matrix for the smaller example, light color indicates the matrix entries can be relatively small if the matrix is diagonally or block diagonally dominant.

Fig. 1 The coefficient matrix and the corresponding D, R, and S matrices using 4 partitions.

where $g = D^{-1}f$. The new system in Equation (5) contains a smaller subsystem which did not exist in the original system of equations. The reduced system is obtained by identifying the indices, c, of the columns which contain nonzeros in R matrix. Then, the reduced system is formed simply by selecting the rows and columns, c, from the S matrix (i.e., $S_{(c,c)}$).

For a small example the sparsity structure of A, D, R, S , and $S_{(c,c)}$ is given in Figures 1(a), 1(b), 1(c), 1(d), 2 respectively.

The reduced system could be formed

$$S_{(c,c)}x_{(c)} = g_{(c)}. \tag{6}$$

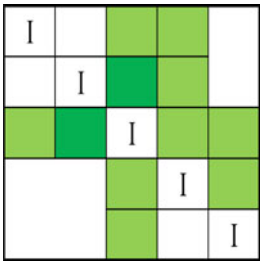


Fig. 2 Extracted reduced system $S_{(c,c)}$.

The smaller reduced subsystem can be solved independently and the complete solution can be retrieved also in parallel using one of the following two methods:

$$x = g - G_{(:,c)}x_{(c)} \tag{7}$$

or equivalently

$$x = g - D^{-1}(R_{(:,c)}x_{(c)}). \tag{8}$$

The difference between these approaches is that the first one requires G matrix to be formed completely, while in the second approach we only need to compute a partial G just enough to form $S_{(c,c)}$. However, it involves additional triangular solves which are independent and completely parallel.

Note that the size of the reduced system is highly dependent on the initial structure of the matrix. In fact sparse matrix partitioning tools that are designed to minimize the communication volume in parallel sparse matrix vector multiplication and load balance can be used in sparse DS factorization. The objective of the partitioning in DS factorization is to decrease the size of the reduced system and, hence, to improve the parallel scalability. Furthermore, for the factorization to exist, D must be nonsingular. To achieve this, one can apply a nonsymmetric permutation to strengthen the diagonal entries.

The rest of the chapter is organized as follows. In Section 2, we introduce the new recursive sparse solver and its variations. Programming and computing environments are described and numerical results on sparse linear systems obtained from the University of Florida Sparse Matrix Collection are given in Section 3. Finally, we conclude and summarize the results in Section 4.

2 The Recursive Sparse DS Factorization

Before we apply the recursive algorithm on the linear system, we apply symmetric and nonsymmetric permutations as mentioned before. In the following pseudocode the linear systems $Ax = f$ are assumed to be the permuted system.

Algorithm 1 The Algorithm

```
1: procedure RDS($A, x, f, t$)        ▷ to solve $Ax = f$ with $t$ number of threads
2:     $D + R \leftarrow A$
3:     Identify nonzero columns of $R$ and store their indices in $c$
4:     (a) $[G, g] \leftarrow D^{-1}[R, f]$
5:     (b) $g \leftarrow D^{-1}f$
6:     $S \leftarrow I + G$
7:     if $t \geq 4$ then
8:         RDS($S_{(c,c)}, x_{(c)}, g_{(c)}, t/2$)
9:     else
10:        $x_{(c)} \leftarrow S_{(c,c)}^{-1} g_{(c)}$
11:    end if
12:    (a) $x \leftarrow g - G_{(:,c)} x_{(c)}$
13:    (b) $x \leftarrow g - D^{-1}(R_{(:,c)} x_{(c)})$
14: end procedure
```

The pseudocode of the recursive DS factorization is given as follows:

Two options are indicated with (a) and (b). They are mathematically equivalent but computationally not. If we choose option (a), the G matrix needs to be formed explicitly which is expensive since linear systems with multiple right-hand sides need to be solved in parallel where D is the coefficient matrix (Line 4). Obtaining the final solution in option (a) is just a matrix vector multiplication (Line 12). Option b, on the other hand, requires a matrix vector multiplication followed by a parallel triangular solve with a single right-hand side (Line 13) and G is no longer need to be computed (Line 5). In our implementation, in all variations, we are (sequentially) using Pardiso direct solver for each of the diagonal blocks in D. Even if we choose option (b), we still need to solve the reduced system. Note that the reduced system is formed using only certain entries from G. The system we form to solve for G has R matrix as the right-hand side. This allows us to use the feature that is provided in Pardiso to allow one to compute only certain entries of the solution vector if the right-hand side is sparse. Therefore, in order to keep the computational costs lower, if we choose option (b) we use the sparse right-hand side feature of Pardiso and compute just some entries of G that is required to form the reduced system. Before factorization, we reorder and partition the initial matrix once, and since the smaller reduced system maintains a similar sparsity structure as the reordered original system we do not need to repartition at every recursive call.

3 Numerical Results

We implement our algorithms using the Intel Cilk Plus which is an extension of C and C++ languages [11]. In our work it is used to ensure efficient multithreading with recursion. Intel MKL is a library of mathematical subroutines such as BLAS, LAPACK, ScaLAPACK, and sparse linear system solvers [12]. In our implementations, all BLAS and Sparse BLAS operations are called from the available routines in Intel MKL version 10.3 [10]. As the direct solver we use Pardiso version 4.1.2.

For symmetric and nonsymmetric permutations we use METIS and HSL MC64, respectively. METIS is a collection of algorithms for graph partitioning, finite element mesh partitioning, and fill-reducing ordering for sparse matrices [13, 14]. It is used in our work to gather the nonzero values in the diagonal blocks as much as possible by setting it to minimize the communication volume. METIS finds a symmetric permutation of matrices and works on undirected graphs. In order to obtain a partitioning for nonsymmetric matrices, we use a symmetric matrix that matches to the structure of our nonsymmetric matrix (i.e., using $(|A|^T + |A|)$). Version 5.1.0 of METIS is used in our work. Some of the matrices that are used in our experiments have zero values on their main diagonals. Since having even one zero value in the main diagonal means that our matrix is indefinite and the diagonal blocks could be singular, we apply a nonsymmetric permutation. HSL MC64 is a collection of Fortran codes to find a column permutation vector to ensure that the matrix will have only nonzero entries on its main diagonal [9]. The permutation vector created by HSL MC64 is used if the matrix is indefinite.

In addition to two recursive variations of the DS factorization based sparse solver, we have also implemented two nonrecursive variations where the reduced system is directly solved via Pardiso. For comparison there are many available parallel sparse direct solvers, in this chapter we compared our results with multithreaded Pardiso direct solver. For many problem types, Pardiso has been shown to be one of most efficient direct solvers available today [6]. Furthermore, Pardiso is provided as a part of Intel MKL.

In summary, we implemented 4 variations of the DS factorization based sparse solver:

- Nonrecursive DS factorization using the sparse right-hand side feature of PARDISO in its computations (DS-NR-SP)
- Nonrecursive DS factorization without using the sparse right-hand side feature of PARDISO (DS-NR-NS)
- Recursive DS factorization using the sparse right-hand side feature of PARDISO (DS-RE-SP)
- Recursive DS factorization without using the sparse right-hand side feature of PARDISO (DS-RE-NS)

In our naming convention RE, NR, SP, and NS stand for recursive algorithm, nonrecursive algorithm, using the sparse right-hand side feature (i.e., not computing the G matrix explicitly) and not using the sparse right-hand side feature (i.e., computing the G matrix), respectively.

For all runs using the sparse DS factorization based solver, we set the number of partitions to be equal to the number of threads. The matrices used for testing are retrieved from University of Florida Sparse Matrix collection [4]. The properties of matrices are given in Table 1. We use a right-hand side vector that consists of all ones.

Table 1 Properties of test matrices, n, nnz, and dd stand for the number of rows and columns and the number of nonzeros, respectively.

#	Matrix Name	n	nnz	Problem domain
1	**ASIC_320k**	321,821	1,931,828	Sandia, circuit simulation
2	**ASIC_680ks**	682,712	1,693,767	Circuit simulation
3	**crashbasis**	160,000	1,750,416	Optimization
4	**ecology2**	999,999	4,995,991	2D/3D
5	**Freescale1**	3,428,755	17,052,626	Circuit simulation
6	**hvdc2**	189,860	1,339,638	Power network
7	**Kaufhold**	160,000	1,750,416	Counter-example
8	**Lin**	256,000	1,766,400	Structural
9	**majorbasis**	160,000	1,750,416	Optimization
10	**Raj1**	263,743	1,300,261	Circuit simulation
11	**rajat21**	411,676	1,876,011	Circuit simulation
12	**scircuit**	170,998	958,936	Circuit simulation
13	**stomach**	213,360	3,021,648	2D/3D
14	**torso3**	259,156	4,429,042	2D/3D
15	**transient**	178,866	961,368	Circuit simulation
16	**xenon2**	157,464	3,866,688	Materials

For all numerical experiments, we use a single node of the Nar cluster. Nar is the High Performance Computing Facility of Middle East Technical University Department of Computer Engineering. A single node of Nar contains 2 x Intel Xeon E5430 Quad-Core CPU (2.66 GHz, 12 MP L2 Cache, 1333 MHz FSB) and 16 GB Memory. Nar uses an open source Linux distribution developed by Fermi National Accelerator Laboratory (Fermilab) and European Organization for Nuclear Research (CERN), Scientific Linux v5.2 64bit, as its operating system. Since each node has 8 cores, we run the algorithms using up to 8 threads.

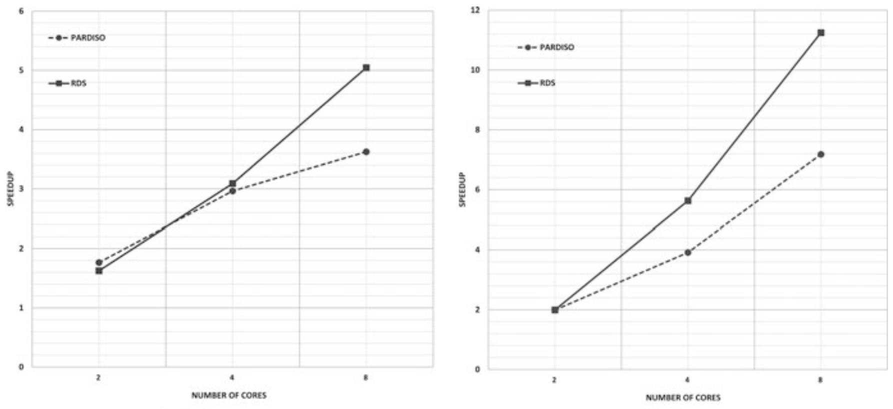
In Table 2, the speed improvement of the recursive DS factorization algorithm (RDS) and multithreaded Pardiso is compared to single threaded Pardiso runs. Timings include the total time to reorder, factorize, and solve the given linear system. We ran all the variations of the RDS algorithm. In the table, due to limited space, all RDS runs presented are using DS-RE-SP variation of the algorithm except for three cases. For matrix #6 we are using DS-RE-NS, for matrices #10 and #15 we are using DS-NR-SP variations of the algorithm since DS-RE-SP does not give a comparable relative residual to multithreaded Pardiso. Also note that the recursive versions of the solver is defined when the number of partitions is equal to or greater than 4. Hence, we are using the nonrecursive DS-NR-SP for all cases if the number of threads is 2. In Table 3, 2-norm of the final relative residuals are given. Based on the results of the runs, the proposed algorithm is faster than Pardiso using 8 threads for 10 cases out of 16 obtaining comparable relative residuals. We note that for the cases where RDS performs worse than multithreaded Pardiso, sequential solution time is very short and hence we could not expect much improvement to begin with.

Table 2 Speedup of multithreaded Pardiso and RDS algorithms compared to the Sequential Pardiso time for all test matrices using 2,4, and 8 threads.

#	Sequential	PARDISO			RDS		
	Time (s)	**2**	**4**	**8**	**2**	**4**	**8**
1	13,88	1,92	2,74	3,39	1,93	2,71	3,52
2	104,30	1,99	3,91	7,18	1,99	3,84	7,04
3	3,74	1,89	3,34	3,60	1,96	3,90	7,33
4	7,49	1,89	3,39	4,57	1,92	3,82	7,57
5	7,64	1,84	3,03	3,37	1,91	3,78	6,42
6	0,45	1,29	1,55	1,96	-	5,63	11,25
7	0,04	1,33	2,00	2,00	0,31	0,40	0,27
8	81,50	1,90	3,37	3,35	0,93	2,59	6,17
9	14,50	1,90	3,42	4,41	1,99	3,61	5,27
10	0,84	1,83	2,80	4,42	1,65	2,47	4,00
11	1,05	1,88	3,00	1,67	1,78	2,69	3,75
12	1,02	1,06	1,52	1,28	0,68	0,92	0,97
13	10,73	1,95	3,55	5,09	1,96	3,78	5,80
14	59,32	1,89	3,54	3,16	1,96	3,66	3,11
15	0,43	1,79	2,69	3,91	1,48	1,95	2,39
16	15,95	1,91	3,56	4,68	1,95	3,68	5,86

Table 3 Relative residual norms for RDS and Pardiso using 2, 4, and 8 threads.

#	PARDISO			RDS		
	2	**4**	**8**	**2**	**4**	**8**
1	1,41E-10	9,03E-11	5,31E-11	1,12E-15	1,47E-15	1,16E-15
2	9,45E-08	9,45E-08	9,45E-08	6,19E-10	8,40E-10	8,13E-10
3	2,20E-15	2,20E-15	2,19E-15	2,20E-15	2,01E-14	2,44E-14
4	1,31E-16	1,31E-16	1,29E-16	1,32E-16	2,04E-16	1,71E-16
5	1,79E-10	1,79E-10	1,85E-10	4,47E-15	7,11E-15	1,03E-15
6	2,80E-09	2,80E-09	2,85E-09	-	9,51E-11	1,24E-10
7	9,72E-15	9,72E-15	9,72E-15	1,26E-16	3,14E-04	6,51E-16
8	8,81E-16	8,07E-16	7,16E-16	5,63E-16	3,70E-13	6,13E-13
9	1,71E-15	1,71E-15	1,70E-15	1,09E-16	9,61E-05	2,91E-11
10	6,16E-10	4,50E-10	6,47E-10	4,67E-07	1,21E-08	1,04E-08
11	7,97E-06	8,38E-06	1,16E-05	4,93E-07	3,18E-04	9,96E-05
12	2,03E-09	1,60E-09	1,59E-09	4,63E-15	1,73E-15	1,44E-15
13	7,71E-16	7,64E-16	7,52E-16	7,65E-16	3,20E-15	3,27E-15
14	1,02E-15	1,01E-15	9,93E-16	4,80E-15	2,11E-15	8,59E-16
15	1,42E-06	1,56E-06	1,68E-06	1,96E-10	1,80E-10	1,47E-10
16	4,39E-12	4,37E-12	4,35E-12	9,41E-12	2,21E-11	5,23E-11



(a) Average (b) Best

Fig. 3 The average and the best speed improvements obtained using RDS and Pardiso compared to the sequential time with Pardiso.

In Figure 3(a), the average speed improvement of RDS is compared against the average speed improvement for Pardiso for all problems. The improvement of RDS more pronounced as the number of threads (i.e., cores) is increased from 4 to 8.

In Figure 3(b), we plot the best speed improvement for both RDS and Pardiso. Again, the improvement is more pronounced as the number of threads increase.

4 Conclusions

We present a recursive sparse DS factorization based direct solver. The results show that on a multicore environment, the scalability of the proposed algorithm is better than the classical LU factorization based solvers in most examples. The improvement is more pronounced if the number of cores is large.

Acknowledgements The HPC resources were provided by the Department of Computer Engineering, Middle East Technical University. This work has been supported by TUBITAK Career Award EEAG111E238, METU BAP-08-11-2011-128, and Turkish Academy of Sciences Distinguished Young Scientist Award TUBA-GEBIP/2012-19.

References

1. Amestoy, P., Duff, I., L'Excellent, J.Y., Koster, J.: Mumps: a general purpose distributed memory sparse solver. In: Sørevik, T., Manne, F., Gebremedhin, A., Moe, R. (eds.) Applied Parallel Computing. New Paradigms for HPC in Industry and Academia. Lecture Notes in Computer Science, vol. 1947, pp. 121–130. Springer, Berlin, Heidelberg (2001)
2. Bolukbasi, E.: A new multi-threaded and recursive direct algorithm for parallel solution of sparse linear systems. Master's thesis, Middle East Technical University, Ankara (2013)

3. Bolukbasi, E.S., Manguoglu, M.: A new multi-threaded and recursive direct algorithm for parallel solution of sparse linear systems. 5th International Conference of the ERCIM (European Research Consortium for Informatics and Mathematics) Working Group on Computing & Statistics (2012)
4. Davis, T.A., Hu, Y.: The university of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011). doi:10.1145/2049662.2049663. http://doi.acm.org/10.1145/2049662.2049663
5. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. SIAM J. Matrix Anal. Appl. **20**(3), 720–755 (1999)
6. Gould, N.I.M., Scott, J.A., Hu, Y.: A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. ACM Trans. Math. Softw. **33**(2) (2007). doi:10.1145/1236463.1236465. http://doi.acm.org/10.1145/1236463.1236465
7. Gupta, A.: WSMP: Watson sparse matrix package (part-i: direct solution of symmetric sparse systems). IBM TJ Watson Research Center, Yorktown Heights, NY. Technical Report, RC, vol. 21886 (2000)
8. Gupta, A.: Wsmp: Watson sparse matrix package (part-ii: direct solution of general sparse systems). Technical Report, Technical Report RC 21888 (98472). IBM TJ Watson Research Center, Yorktown Heights, NY (2000). http://www.cs.umn.edu/~agupta/doc/wssmp-paper.ps
9. HSL: A collection of Fortran codes for large scale scientific computation (2011). http://www.hsl.rl.ac.uk
10. Intel. Intel MKL 10.3 release notes (2012). http://software.intel.com/en-us/articles/intel-mkl-103-release-notes
11. Intel: Intel Cilk plus (2013). http://software.intel.com/en-us/intel-cilk-plus
12. Intel: Intel math kernel library 11.0 (2013). http://software.intel.com/en-us/intel-mkl
13. Karypis, G.: Metis - serial graph partitioning and fill-reducing matrix ordering (2013). http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
14. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998). doi:10.1137/S1064827595287997. http://epubs.siam.org/doi/abs/10.1137/S1064827595287997
15. Li, X.S.: An overview of SuperLU: Algorithms, implementation, and user interface. ACM Trans. Math. Softw. **31**(3), 302–325 (2005)
16. Manguoglu, M.: A domain-decomposing parallel sparse linear system solver. J. Comput. Appl. Math. **236**(3), 319–325 (2011)
17. Manguoglu, M.: Parallel solution of sparse linear systems. In: High-Performance Scientific Computing, pp. 171–184. Springer, London (2012)
18. Moore, G.E.: Cramming more components onto integrated circuits. IEEE Solid State Circuits Soc. Newsl. **11**(5), 33–35 (2006) [Reprinted from Electronics **38**(8), 114 ff. (1965)]. doi: 10.1109/N-SSC.2006.4785860
19. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003)
20. Sameh, A.H., Kuck, D.J.: On stable parallel linear system solvers. J. ACM **25**(1), 81–91 (1978)
21. Sameh, A.H., Polizzi, E.: A parallel hybrid banded system solver: the spike algorithm. Parallel Comput. **32**(2), 177–194 (2006)
22. Sameh, A.H., Polizzi, E.: Spike: a parallel environment for solving banded linear systems. Comput. Fluids **36**(1), 113–120 (2007)
23. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with pardiso. Futur. Gener. Comput. Syst. **20**(3), 475–487 (2004)
24. Schenk, O., Gärtner, K.: On fast factorization pivoting methods for sparse symmetric indefinite systems. Electron. Trans. Numer. Anal. **23**, 158–179 (2006)
25. Schenk, O., Wächter, A., Hagemann, M.: Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. Comput. Optim. Appl. **36**(2–3), 321–341 (2007)
26. Schenk, O., Bollhöfer, M., Römer, R.A.: On large-scale diagonalization techniques for the Anderson model of localization. SIAM Rev. **50**(1), 91–112 (2008)