

SparSol - sparse linear systems solver

O. V. DIYANKOV, S. V. KOSHELEV, S. S. KOTEGOV, I. V. KRASNOGOROV, N. N. KUZNETSOVA,
V. Y. PRAVILNIKOV*, and B. L. BECKNER, S. Y. MALIASSOV, I. D. MISHEV,
A. K. USADI†

26th March 2007

Abstract — *SparSol* is a software package intended for the preconditioned iterative solution of large sparse linear systems. It includes sets of iterative methods, preconditioners, scaling and reordering algorithms that allows to choose the optimal combination of algorithms for a particular problem. The paper briefly describes the algorithms implemented and numerically compares the performance of *SparSol* with that of several popular, freely available software packages using test systems arising from hydrocarbon reservoir simulations.

Keywords: Large sparse linear system, Preconditioned iterative methods, Iterative solvers

1. INTRODUCTION

SparSol is a powerful and efficient iterative solver intended for solving linear systems of algebraic equations

$$A \cdot x = b$$

where A is a large sparse non-singular matrix, b is a right-hand side vector and x is a vector of unknowns. The matrix A can be either symmetric or unsymmetric. The system may have a block structure when each matrix block corresponds to a different physical model of a solved problem. Moreover a matrix block may consist of small size blocks corresponding to grid blocks (finite elements) with a few number of unknowns (components). Matrices with such a structure can arise, for example, in hydrocarbon reservoir simulations.

Preconditioned iterative methods with preprocessing are widely used for the solution of such systems. The solution procedure includes the following general steps:

- Transformation of a multi-block structure to a single-block structure (if needed);
- Preprocessing the initial system (applying scaling, reordering algorithms, etc...);

*Neurok TechSoft LLC, Troitsk, Moscow region, Russia

†ExxonMobil Upstream Research Company, Houston, Texas, USA

- Construction of a preconditioner for the preprocessed matrix;
- Solution of the preprocessed system by a preconditioned iterative method;
- Backward transformation of the obtained solution to the original scale and order.

The *SparSol* code has been developed under the contract with the *ExxonMobil Upstream Research Company* and in close cooperation with research teams from the *Institute for Numerical Mathematics of the Russian Academy of Sciences (INM RAS)* and the *Computing Center of the Russian Academy of Sciences (CC RAS)*. The code is implemented in serial, SMP and MPI variants and can be run under MS Windows and many Unix-type operating systems. A set of program interfaces provides an easy-to-use access to the package.

The effectiveness of algorithms is illustrated by numerical results performed for a set of 7 test matrices from the *ExxonMobil* matrix collection available on the site <http://www.aconts.com/XOMMatrices/>. General information on these matrices is presented in the table 1.

Table 1.

General information on matrices from the ExxonMobil matrix collection

Problem	N	Z	Z/N	ZD	ND	POD
CIT-1	17436	344245	19,74	0	4207	98092
CIT-2	249428	5613978	22,51	30	1323	1024619
SBO-1	21700	145122	6,69	1	0	7
SBO-2	111756	888190	7,95	0	0	12
SBO-3	216051	1849317	8,56	0	0	0
SBO-4	93264	667882	7,16	0	0	0
SEO-1	22421	204784	9,13	0	180	849

where N is a number of rows, Z is a number of nonzeros, $S = Z/N$ is an average sparsity, ZD is a number of zero diagonal entries, ND is a number of negative diagonal entries, POD is a number of positive off-diagonal entries.

Numerical experiments, presented in the paper, were performed on an AMD Opteron computer with two 2GHz CPUs and 4GB RAM under Linux SuSe 8.0 operational system. All programs were compiled with GCC 3.4.1.

Due to all test matrices are unsymmetric, we mostly focus on iterative methods with *ILU*-type preconditioning. By default, the *BiCGStab* iterative method [1] with the right *FILU(0.001, 1)* preconditioner (3.2), [2] is used for the solution. Exception is the matrix *CIT-2* for which *Row-Column Scaling* (2.1.3) is applied as the pre-processing. Initial guess is assumed zero, maximal number of iterations is 300 and stopping criterion is

$$|r^v|_2 < 10^{-5} \cdot |b|_2.$$

where r^v means the iterative residual on the v -th iteration and $|\cdot|_2$ means the second vector norm.

The performance of any combination of algorithms is compared by its total solution time related to the total solution time of the described above default set of algorithms. Other two important characteristics of applied algorithms are the *preconditioner extension* (i.e. if M is a preconditioner matrix then $ext(M) = nnz(M)/nnz(A)$, where $nnz(\cdot)$ means a number of nonzeros) and the *number of iterations* for convergence. In the tables below we use the following notations: *Prec.ext.* or *Ext* is a preconditioner extension, *Iters* is a number of iterations, *Tot.time* is a relative total solution time. If an iterative method with some combination of algorithms does not achieve the required accuracy for the maximal number of iterations than such variant is considered as diverged and indicated in the tables as '-'.

Note also, that the package implements as well-known algorithms as some new algorithms (or the algorithms whose description the authors did not see in the literature before). The first class of algorithms is referenced to corresponding articles/books while some of new algorithms have no references because have not been published before.

The paper is organized as follows. In sections 2,3,4 we describe the main algorithms implemented. In section 5 we briefly consider some ways how SparSol can be used inside of an external application. In section 6 we compare the performance of our solver to that of several popular, freely available software packages.

2. SYSTEM PREPROCESSING - SCALINGS/REORDERINGS

In many cases preprocessing the linear system can significantly improve the convergence of a preconditioned iterative solver. At this step the initial matrix is transformed to the one whose properties, like matrix profile and scalability, are more conducive to a fast residual reduction. Preprocessing algorithms can be divided into two categories: *scalings* and *reorderings*.

In principal, one is able to apply any combination of preprocessing algorithms, but in practice, one scaling and one or two reorderings is usually sufficient.

Note that preprocessing algorithms affect upon the whole system. Therefore we must transform the vector b before the start of the solution procedure and then transform the obtained approximate solution x to the original scale and ordering.

2.1. Scalings

The main purpose for any scaling procedure is to equalize or normalize, in some way, the magnitudes of matrix entries. Scaling can be particularly useful when applied to matrices arising from problems with multiple unknowns per cell/node and, hence, having rows/columns with significantly different norms. Applying a scaling can be especially important for preconditioners implementing an incomplete factorization based on the strategy of dropping by magnitude of elements (3).

In the general case, a scaling consists of two diagonal matrices D_L and D_R used

to scale the rows and the columns of the initial matrix. Note that for some scalings described below one of these diagonal matrices can be the identity I .

Thus, the initial system $Ax = b$ is transformed to a system

$$D_L \cdot A \cdot D_R \cdot D_R^{-1} \cdot x = D_L \cdot b$$

or

$$\tilde{A} \cdot \tilde{x} = \tilde{b}$$

where $\tilde{A} = D_L \cdot A \cdot D_R$, $\tilde{b} = D_L \cdot b$ and $\tilde{x} = D_R^{-1} \cdot x$.

We implemented the following scalings:

- *Norm Scaling* makes row/column norms of the scaled matrix equal to 1;
- *Diagonal Scaling* makes diagonal entries of the scaled matrix equal to ± 1 (this is very popular technique described in many papers);
- *Row-Column Scaling* approximately equalizes both row and column norms to 1 [3].

2.1.1. Norm Scaling. *Norm Scaling* makes row/column norms ($|A_{i*}|_\alpha / |A_{*j}|_\alpha$) of a scaled matrix equal to 1. Row norm α can be one of the following row norms:

- $|A_{i*}|_{L1} = \frac{\sum_{a_{ij} \neq 0} |a_{ij}|}{nz(A_{i*})}$;
- $|A_{i*}|_{L2} = \sqrt{\frac{\sum_{a_{ij} \neq 0} a_{ij}^2}{nz(A_{i*})}}$;
- $|A_{i*}|_\infty = \max_{a_{ij} \neq 0} |a_{ij}|$;
- $|A_{i*}|_{L1N} = \sum_{a_{ij} \neq 0} |a_{ij}|$;
- $|A_{i*}|_{L2N} = \sqrt{\sum_{a_{ij} \neq 0} a_{ij}^2}$.

The formulas for column norms can be written similarly.

The scaling computes diagonal matrices in the following way

$$D_L^N = \left| \frac{1}{|A_{i*}|_\alpha} \right|, D_R^N = \left| \frac{1}{|A_{*j}|_\alpha} \right|$$

There are four optional variants of the scaling:

1. *Row Norm Scaling* - $D_L = D_L^N$, $D_R = I$ makes all row norms equal to 1;
2. *Column Norm Scaling* - $D_L = I$, $D_R = D_R^N$ makes all column norms equal to 1;
3. *Row then Column Norm Scaling* makes all column norms equal to 1 applying successively *Row* and then *Column Norm Scaling*; this process can be performed iteratively;
4. *Column then Row Norm Scaling* makes all row norms equal to 1 applying successively *Column* and then *Row Norm Scaling*; this process can be performed iteratively.

Note that this scaling does not preserve matrix symmetry.

Results for the solution of test matrices using *Norm scaling* are presented in table 2

Table 2.
Norm Scaling results

Matrix	Row Scaling			Column Scaling			Row+Col Scaling			Col+Row Scaling		
	Prec.ext.	Iters	Tot.time	Prec.ext.	Iters	Tot.time	Prec.ext.	Iters	Tot.time	Prec.ext.	Iters	Tot.time
CIT-1	1,44	25	1,00	1,55	104	2,53	1,52	22	1,06	1,83	25	1,14
CIT-2	1,02	300	-	1,03	300	-	1,02	52	0,68	1,04	207	2,18
SBO-1	1,63	175	1,11	1,64	167	1,10	1,63	194	1,26	1,63	158	1,05
SBO-2	2,23	112	0,86	2,23	106	0,81	2,21	119	0,89	2,22	101	0,79
SBO-3	1,64	154	0,82	1,64	152	0,81	1,65	173	0,91	1,65	157	0,85
SBO-4	1,33	120	1,10	1,33	111	1,03	1,33	117	1,10	1,33	125	1,16
SEO-1	1,25	87	1,07	1,22	69	0,85	1,25	65	0,85	1,22	91	1,13

As seen in the table above, this scaling shows unstable results, i.e. for some matrices (like *SBO-2*, *SBO-3*) it allows to obtain the speed up 20%, but for some matrices it slows down the performance more than 2 times (see for example *CIT-1* and *CIT-2*). Note also that two-side scalings (the last two columns of four) are more robust, but not always more effective.

2.1.2. Diagonal Scaling. *Diagonal Scaling* makes diagonal entries of the scaled matrix equal to ± 1 calculating diagonal matrices by the formulas:

$$D_L = \left| \frac{1}{\sqrt{|a_{ii}|}} \right|, \quad D_R = \left| \frac{\text{sign}(a_{ii})}{\sqrt{|a_{ii}|}} \right|$$

After diagonal scaling an iterative balancing procedure can be applied to considerably reduce the norm of the matrix off-diagonal part in the case of unsymmetric matrices [4].

Note that this scaling preserves matrix symmetry and can be particularly effective for SPD matrices.

2.1.3. Row-Column Scaling. *Row-Column Scaling* approximately equalizes both row and column norms to 1 [3].

This scaling is especially effective for systems arising in discretizations of problems with variable number of unknowns with significantly different scales.

Note that this scaling does not preserve matrix symmetry.

The results on the solution of the test matrices for *Row-Column* and *Diagonal Scalings* are presented in table 2

Table 3.
Row-Column and Diagonal Scalings results

Matrix	Row-Column Scaling			Diagonal Scaling		
	Prec.ext	Iters	Tot.time	Prec.ext	Iters	Tot.time
CIT-1	1,47	21	0,89	1,46	38	1,33
CIT-2	1,02	83	1,00	1,02	154	1,68
SBO-1	1,32	216	1,24	1,34	222	1,27
SBO-2	1,93	135	0,94	1,98	126	0,89
SBO-3	1,46	197	0,98	1,48	147	0,77
SBO-4	1,21	119	1,10	1,22	141	1,25
SEO-1	1,18	99	1,22	1,18	96	1,15

As seen in the table above, these scalings show unstable results, i.e. for some matrices (like *SBO-2*, *SBO-3*) they allow to obtain the acceleration up to 20%, but for some matrices they slow down the performance considerably (see for example *SBO-1*, and *SBO-4*, and *SEO-1*).

Nevertheless, summarizing all results on scalings and comparing them with the results from the table 5 (preconditioned iterative method without preprocessing) one should note that applying one of described scalings produces preconditioner with smaller extension and allows to obtain the solution with at least similar speed in the worst case or significantly faster in the best one. Moreover, for the matrix *CIT-2* only applying of scaling allows to obtain the convergence of the iterative solver. The problem of choosing the optimal scaling is not clear and still open.

2.2. Reorderings

Reorderings influent significantly on the quality of a preconditioner and accelerate the convergence of the iterative method. A reordering can improve both the structure of a matrix minimizing number of fill-ins appeared in incomplete factorizations and the numerical stability of such factorizations.

In general, a reordering permutes the whole system in the following way

$$P_R \cdot A \cdot P_C^{-1} \cdot P_C \cdot x = P_R \cdot b$$

where P_R is a row permutation and P_C is a column permutation. If $P_R = P_C$ than a reordering is symmetric.

The following reorderings are available in *SparSol*

- *RCM* implements the well-known Reverse Cuthill-McKee reordering as a multi-level set traversal algorithm where the next node in the level is chosen by its degree [6].
- *Weighted RCM* implements a Weighted Reverse Cuthill-McKee reordering proposed by I. Kaporin from *CC RAS*; it differs from the standard *RCM* in that the next node in the level is chosen by its magnitude.
- *Multicoloring* implements a Greedy Multicoloring Algorithm [6] which uses a direct traversal of a matrix graph assigning to a node the minimal possible color; a matrix can be preordered by degrees (both forwards and backwards).
- *Diagonal Filtration* reorders a matrix by successively moving diagonally dominant rows to the top of a permuted matrix; diagonal dominance for the i -th row is defined as $|a_{ii}| \geq \text{const} \cdot |A_{i*,i \neq j}|$.
- *Diagonal Dominant Filtration* [7] reorders a matrix by sorting rows in descending order according to their diagonal dominance condition; the diagonal dominance condition for the i -th row is $\frac{\max |a_{ij}|}{|A_{i*,L1}|}$. This reordering generates an unsymmetric permutation.
- *Independent Graphs Reordering* searches for independent sub-graphs in a matrix, joins "small" sub-graphs into bigger ones, and then reorders them in descending order. This is particularly useful when the matrix represents a physical system with sub-regions that are not in communication with each other.
- *Cell Based Reordering* considers each block of a matrix related to a one grid block as one element of a *reduced graph* which can be then reordered by any structural reordering discussed above (e.g. *RCM*). This keeps all elements of a grid block together during permutation of the matrix and can produce much better performance compared to reorderings which destroy this natural clustering.

RCM-like reorderings were observed to be most effective in helping improve convergence performance on the problems tested. Furthermore, reordering is usually more effective in combination with scaling. Numerical results for some of reorderings are presented in the table 4.

As seen in the table, *RCM* or *WRCM* with *Row-Column Scaling* give very good acceleration for all systems, except of *SBO-1*.

Table 4.
RCM numerical results

Matrix	RCM			RCM + Row-Column scaling			WRCM + Row-Column scaling		
	Prec.ext	Iters	Tot.time	Prec.ext	Iters	Tot.time	Prec.ext	Iters	Tot.time
CIT-1	2,5	27	1,33	1,55	13	0,81	1,33	16	0,89
CIT-2	1,12	300	-	1,03	33	0,51	1,02	39	0,73
SBO-1	1,55	211	1,29	1,33	210	1,19	1,31	249	1,47
SBO-2	2,26	102	0,82	1,92	120	0,81	1,94	103	0,74
SBO-3	1,66	139	0,66	1,36	183	0,76	1,28	132	0,60
SBO-4	1,32	100	0,89	1,17	99	0,86	1,11	101	0,92
SEO-1	1,31	62	0,87	1,17	78	0,93	1,11	49	0,72

3. PRECONDITIONERS

Preconditioning is a key procedure for the iterative solution of large sparse linear systems. The convergence rate of the iterative method greatly depends on the quality of the preconditioner. In *SparSol* we implemented a right preconditioning when preconditioner M is use in the form

$$A \cdot M^{-1} \cdot M \cdot x = b$$

The preconditioners available in *SparSol* can be divided into the three classes:

- *Incomplete Cholesky*-type preconditioners construct the matrix M as $A \approx M = U^T \cdot U$ where U is an upper triangular matrix. These preconditioners are especially effective for SPD matrices, but can not be applied to unsymmetric matrices.
- *Incomplete LU*-type preconditioners construct the matrix M as $A \approx M = L \cdot U$ where L is a strictly lower triangular matrix with unit diagonal and U is an upper triangular matrix. These preconditioners can be applied to general sparse matrices.
- *Nested*-type factorizations can be applied to matrices having a special *nested* or *multi-level* structure. Such matrices usually arise in problems which use finite difference discretization on structured grids.

Below we consider these preconditioner classes one at a time.

3.1. IC-type preconditioners

Preconditioners of this type construct the matrix M as

$$A \approx M = U^T \cdot U$$

where U is an upper triangular matrix.

We implemented the following *IC*-type preconditioners:

- *IC(0)* is the simplest variant of Incomplete Cholesky factorization without extension of the original pattern [18].
- *RIC2S* developed by I.Kaporin from *CC RAS* implements high quality preconditioning of a general symmetric positive definite matrix based on its $U^T U + U^T R + R^T U$ -decomposition [14].
- *FIC* implements *IC* factorization with extension of the original pattern basing on preconditioner techniques similar to that of the *FILU* preconditioner [2].

Preconditioners of this type can be applied to SPD systems only, that is why the discussion of their features is out of the scope of this paper.

3.2. ILU-type preconditioners

Preconditioners of this type construct the matrix M as

$$A \approx M = L \cdot U$$

where L is a strictly lower triangular matrix with unit diagonal and U is an upper triangular matrix with the main diagonal.

It is well-known fact that *complete LU* factorization can be very expensive in the sense of required computational and memory resources. That is why one has to introduce some rules to drop "non-essential" entries during factorization. Unlike the strategy based on "levels of fill-in" (*ILU(k)*-type preconditioners [21]), we implemented the preconditioning strategy base on dropping by magnitudes. We discard elements those magnitudes are less than a given constant named *dropping tolerance* multiplied to the norm of the corresponding row

$$|m_{ij}| < \varepsilon \cdot |A_{i*}|_{\alpha}$$

We implemented the following *ILU*-type preconditioners:

- *ILU(0)* is the simplest variant of Incomplete LU factorization without extension of the original pattern when all elements outside of the pattern are discarded [6].
- *RILU* is the relaxed variant of Incomplete LU factorization without extension of the original pattern where discarded entries multiplied to the some *relaxation coefficient* ω are subtracted from the corresponding diagonal entry [22].
- *FILU* is a powerful and flexible preconditioner combining several preconditioning techniques and several dropping strategies. The basic *FILU* algorithm extends the original pattern of a matrix using two drop tolerances - one for

adding new entries into preconditioning stencil and another for dropping sufficiently small new entries from the preconditioned row. We also developed special *split LU* row norm which uses separate norms for L and U parts in dropping rules [2].

- *ILU2* developed by I.Kaporin from *CC RAS* implements high quality preconditioning of a general sparse matrix based on its $LU + LR_1 + R_2^T U$ -decomposition [17].

Preconditioners of this type can be applied to a general sparse linear system.

The numerical results for basic *FILU* algorithm are presented in the table 5. The results from this table are used for comparison in all other tables (for *CIT-2* for comparison we use default *FILU* preprocessed by *Row-Column Scaling* (2.1.3)).

Table 5.
Plain *FILU*(0.001,1) results

Matrix name	Prec.ext	Iters	Tot.time
CIT-1	2.03	13	1.0
CIT-2	-	-	-
SBO-1	1.68	144	1.0
SBO-2	2.39	114	1.0
SBO-3	1.68	177	1.0
SBO-4	1.40	91	1.0
SEO-1	1.26	74	1.0

3.3. NF-type preconditioners

Nested Factorization is a powerful and flexible method intended for the preconditioned iterative solution of sparse linear systems arising in finite difference discretizations on 2D/3D structured grids [5]. It can be extended to unstructured grids by generalizing its underlying concept. *NF* does not form L and U factors explicitly. Instead, *NF* exploits the nested structure of the original matrix by defining the preconditioner matrix in the sequence of level matrices with the diagonal matrix on the lowest level. Thus we need to store only the diagonal matrix while all others matrices are recalculated in the construction and solution procedures.

We implemented the following *NF*-type preconditioners [23]:

- *Nested Factorization (NF)* for the preconditioned iterative solution of sparse linear systems arising in discretizations on structured grids
- *Generalized NF (GNF)* for the preconditioned iterative solution of sparse linear systems arising in discretizations of problems with several unknowns per finite elements on structured grids.

- *Unstructured NF (UNF)* for the preconditioned iterative solution of sparse linear systems arising in discretizations on unstructured grids. *UNF* uses the special *Chain Reordering* to recover the nested structure for a general sparse matrix.

The discussions on the preconditioners of this type is out of the scope of this paper.

4. ITERATIVE METHODS

The following iterative methods are implemented in *SparSol*:

- *CG*: Conjugate Gradient, usually used for SPD matrices [11].
- *CGS*: Conjugate Gradient Squared [20];
- *BiCGStab*: BiConjugate Gradient Stabilized [1];
- *BiCGStab2*: BiConjugate Gradient 2 Stabilized [10];
- *BiCGStabL*: BiConjugate Gradient Stabilized (L) [12];
- *GMRes*: Generalized Minimal Residual [9];
- *DQGMRes*: Direct Quasi Generalized Minimal Residual [6];
- *TFQMR*: Transpose-Free Quasi-Minimal Residual [6].

The numerical comparison of implemented iterative methods is presented in the tables 6 and 7.

Table 6.

Numerical results for *CGS*, *BiCGStab*, *BiCGStab2* and *GMRes* iterative methods. *Row-Column Scaling* and *FILU(0.001,1)* were used.

Matrix	CGS			BiCGStab			BiCGStab2			GMRes(32)		
	Ext	Iters	Tot.time	Ext	Iters	Tot.time	Ext	Iters	Tot.time	Ext	Iters	Tot.time
CIT-1	1,45	24	0,92	1,45	21	1,00	1,45	22	0,94	1,45	13	0,78
CIT-2	1,02	300	-	1,02	83	1,00	1,02	49	0,77	1,02	26	0,59
SBO-1	1,32	286	1,60	1,32	216	1,00	1,32	205	1,21	1,32	245	2,61
SBO-2	1,93	230	1,52	1,93	135	1,00	1,93	132	0,93	1,93	102	1,41
SBO-3	1,46	268	1,31	1,46	197	1,00	1,46	176	0,91	1,46	176	1,72
SBO-4	1,21	124	1,14	1,21	119	1,00	1,21	112	1,03	1,21	109	2,12
SEO-1	1,18	96	1,15	1,18	99	1,00	1,18	76	0,98	1,18	59	1,20

Table 7.

Numerical results for *BiCGStabL*, *DQGMres* and *TFQMR* iterative methods. Row-Column Scaling and *FILU(0.001,1)* were used.

Matrix	BiCGStabL(16)			DQGMres(32,8)			TFQMR		
	Ext	Iters	Tot.time	Ext	Iters	Tot.time	Ext	Iters	Tot.time
CIT-1	1,45	16	1,25	1,45	21	0,94	1,45	22	0,97
CIT-2	1,02	80	2,17	1,02	58	1,02	1,02	72	0,94
SBO-1	1,32	144	2,56	1,32	300	-	1,32	269	1,63
SBO-2	1,93	80	1,54	1,93	151	1,79	1,93	203	1,44
SBO-3	1,46	144	2,26	1,46	187	1,61	1,46	231	1,22
SBO-4	1,21	96	2,54	1,21	186	1,21	1,21	186	1,79
SEO-1	1,18	64	2,09	1,18	91	1,67	1,18	94	1,24

As seen in tables, *BiCGStab* exhibits fast and stable convergence for most of systems tested and is therefore *SparSols* default iterative solver. *BiCGStab2* also shows good performance and can be chosen as an alternative of *BiCGStab*.

But as in cases of all previous algorithms considered, we can note that there is no unique algorithm which provides the best results for all problems.

5. USING SPARSOL

An extended, flexible set of program interfaces provides an access to *SparSol* algorithms and allows its easy use in any application written in C++, C and Fortran. The interfaces provide access to both the complete solve procedure as well as to its separate algorithms to allow its use within client solution algorithms. The internal matrix representation data storage format is *CRS* (Compressed Row Storage). However, a matrix can be loaded into the library in several popular formats. *SparSol* interfaces allow I/O of data in many widely-used file formats. Moreover, the special XML ASCII or binary formats allow storage of large systems on disk in a flexible and compact form (see for more details [24]).

The code is supplied with detailed HTML documentation.

6. COMPARISON WITH OTHER PACKAGES

The performance of the *SparSol* package has been compared to that of several popular iterative solvers *SparseKit2* [8], *PETSc* [13] and with the direct solver *PARDISO* [19] for seven test problems from the ExxonMobil public matrix collection *(see 1).

The numerical results for *SparSol* are presented in the table 8. Emphasize that the table contains results for *default* variant of the solver which can be significantly (up to several times) improved by combining various algorithms and turning their

parameters.

Table 8.

SparSol results obtained using *BiCGStab* with *FILU(0.001,1)*. For *CIT-2* results were obtained using *Row-Column Scaling* and *RCM* as preprocessing.

Problem	Prec.ext	Prec.time [%]	IM time [%]	Iters	Tot.time [%]
CIT-1	2,03	0,61	0,36	13	1,00
CIT-2	1,03	0,14	0,60	32	1,00
SBO-1	1,68	0,10	0,89	144	1,00
SBO-2	2,38	0,18	0,81	114	1,00
SBO-3	1,68	0,11	0,88	177	1,00
SBO-4	1,4	0,17	0,82	91	1,00
SEO-1	1,26	0,15	0,83	74	1,00

Note that the sum of 3rd and 4th columns of the above table is less than 1.0 due to the solver spends some time on an additional work like the system check, results storing and so on. For *CIT-2* a part of the total solution time is spent for the scaling and reordering computation and applying.

Table 9 contains results for the *ILU(k)* preconditioner from *PETSc*, table 10 - for *ILUT($\epsilon, lfill$)* from *SparseKit2* and table 11 - for the direct solver *PARDISO* with default parameters.

Table 9.

PETSc ILU(k) results for the best *k*

Problem	k	Iterations number	T_{ILUK}/T_{FILU}
CIT-1	1	26	2.61
CIT-2	-	-	-
SBO-1	1	74	1.38
SBO-2	3	163	7.10
SBO-3	3	530	19.53
SBO-4	3	606	36.57
SEO-1	4	315	49.00

These results indicate that *SparSol* is faster and more robust than the packages compared for the set of matrices tested. For these matrices *FILU* preconditioner generates a high-quality LU factorization allowing standard iterative methods to converge faster and usually take fewer iterations.

7. CONCLUSION

An effective, robust iterative solver has been developed. Results presented in this paper show that *SparSol* solver is faster than several popular, freely available packages for the set of matrices tested. The numerical results presented in the paper show that there is no the sole set of algorithms which is optimal for all problems. That is why

Table 10.
SPARSKIT2 $ILUT(10^{-7}, 15)$ results

Problem	Prec. extension	Iterations number	T_{ILUT}/T_{FILU}
CIT-1	1.11	300	-
CIT-2	0.85	300	-
SBO-1	3.58	46	1.30
SBO-2	3.06	53	1.52
SBO-3	2.75	75	1.22
SBO-4	2.84	35	1.42
SEO-1	2.20	63	1.96

Table 11.
Results of *PARDISO* with default parameters

Problem	LU extension	$T_{PARDISO}/T_{FILU}$
CIT-1	19.35	7.94
CIT-2	-	-
SBO-1	14.86	1.41
SBO-2	29.22	2.35
SBO-3	37.64	3.11
SBO-4	35.46	4.90
SEO-1	29.19	4.81

we implement in the *SparSol* package a wide set of various algorithms that allows us to choose optimal algorithms combination for systems arising in discretizations of PDEs of different types. But the turning of the optimal solver parameters is very non-trivial problem.

Potential future development includes:

- Development of effective parallel partitioning
- Development of an effective, robust parallel preconditioner
- Development of an Unstructured Nested Factorization for matrices produced on 3D on unstructured grids
- Development of parallel multilevel preconditioners
- Optimization and profiling of implemented algorithms

8. ACKNOWLEDGMENTS

The development of *SparSol* was funded by *ExxonMobil Upstream Research Company (XOM URC)*. We are grateful to the management of *XOM URC* for the permission to publish this paper. We are also very grateful to anonymous referee whose critical remarks and comments allow us to improve a text significantly.

REFERENCES

1. H. A. van der Vorst, Bi-CGSTAB: A fast and smooth converging variant of Bi-CG for the solution of non-symmetric linear systems. In: *SIAM Journal on Scientific and Statistical Computing*, 12, pp 631-644 (1992).
2. O. V. Diyankov et al., FILU - an efficient incomplete LU preconditioner. In: *Russian Journal of Numerical Analysis and Mathematical Modeling*, Vol.22, 4, (2007).
3. V.F. de Almeida, A.M. Chapman, and J.J. Derby, On Equilibration and Sparse Factorization of Matrices Arising in Finite Element Solutions of Partial Differential Equations. In: *Numer. Methods Partial Different. Equ.* 16, 11–29 (2000).
4. I. E. Kaporin, Explicitly preconditioned conjugate gradient method for the solution of nonsymmetric linear systems. In: *Int.J.Computer Math.*, 50, pp 169-187 (1992).
5. J. R. Appleyard and I. M. Cheshire, Nested Factorization. In: *The 7th SPE Symposium on Reservoir Simulation* (1983)
6. Y. Saad, Iterative Methods for Sparse Linear Systems. *PWS Publishing, New York.* (1996).
7. Y. Saad, and B. Suchoemel ARMS: An algebraic recursive multilevel solver for general sparse linear systems. In: *Numerical Linear Algebra with Applications*, 9, pp 359-378 (2002). (1996).
8. Y. Saad, SPARSEKIT: A basic tool for sparse matrix computations. *Tech. Rep. 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA* (1990).
9. Y. Saad, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. In: *SIAM J. Sci. Statist. Comput.*, 7, pp. 856–869. (1986)
10. H. A. Van der Vorst, T. F. Chan I, Linear system solvers: sparse iterative methods.
11. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, Ch. Romine, H. van der Vorst, Templates for the solution of linear systems: building blocks for iterative methods, electronic version.
12. G. L. G. Sleijpen, D. R. Fokkema, BiCGStab(L) for linear equations involving unsymmetric matrices with complex spectrum, electronic version.
13. PETSc Home Portable, Extensible Toolkit for Scientific Computation. <http://www-unix.mcs.anl.gov/petsc/>
14. I. E. Kaporin, High quality preconditioning of a general symmetric positive definite matrix based on its $U^T U + U^T R + R^T U$ -decomposition. In: *Numer. Linear Algebra Appls*, 5, pp 483-509 (1998).
15. I. E. Kaporin, Superliner convergence in minimum residual iterations. In: *Numer. Linear Algebra Appls*, 12, pp 453-470 (2005)
16. I. E. Kaporin, I. Konshin, Optimization of Matrix Scaling in CGSOL code. *NeurOK Techsoft Report, June* (2003)
17. I. E. Kaporin, Second Order ILU Preconditioning for Nonsymmetric Matrices *International Conference on Matrix Methods and Operator Equations, June 20 - 25, Moscow, Russia* (2005)
18. D. Kershaw, The incomplete Choleskyconjugate gradient method for the iterative solution of systems of linear equations. In: *J. Comput. Phys.*, 26, pp. 43–65 (1978)
19. O. Sheck and K. Gärtner, PARDISO: a high performance serial and parallel sparse liner aolver in semiconductor device simulations *Future Generation Computer Systems*, 789(1): 1-9 (2001).
20. P. Sonneveld, CGS, a fast Lanczos type solver for nonsymmetric linear systems. In: *SIAM J. Sci. Statist. Comput.*, 10, pp. 36–52 (1989)

21. J. W. Watts, III, A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. In: *Soc. Petrol. Eng.*, 21, 345 (1981)
22. T. F. Chan and H. Van Der Vorst, Approximate and Incomplete Factorizations. In: <http://igitur-archive.library.uu.nl/math/2001-0621-115821/proc.pdf>
23. O. V. Diyankov et al., The family of Nested Factorizations. In: *Russian Journal of Numerical Analysis and Mathematical Modeling*, Vol.22, 4, (2007).
24. <http://www.aconts.com/XOMMatrices>