ARI The Bulletin of the Istanbul Technical University VOLUME 54, NUMBER 3
Communicated by Muhittin Gökmen

Incomplete Inverse Preconditioners

Fernando Alvarado
Department of Electrical and Computer Engineering, University of Wisconsin, 53706 Madison, USA
Hasan Dağ
Computational Science and Engineering Program, Informatics Institute, Istanbul Technical University, 34469 Maslak, Istanbul, Turkey
(Received 22 September 2003)

Incomplete *LU* factorization is a valuable preconditioning approach for sparse iterative solvers. An "ideal" but inefficient preconditioner for the iterative solution of $A\,\mathbf{x} = \mathbf{b}$ is A^{-1} itself. This paper describes a preconditioner based on sparse approximations to partitioned representations of A^{-1}, in addition to the results of implementation of the proposed method in a shared memory parallel environment.

The partitioned inverses are normally somewhat sparse. Their sparsity can be enhanced with suitable ordering and partitioning algorithms. Sparse approximations to these partitioned inverse representations can be obtained either by discarding selected nonzero entries of these inverses or by precluding the creation of some inversion fills. Experimental results indicate that the use of these partitioned incomplete inverses as preconditioners results in excellent highly parallel preconditioners.

Keywords: Conjugate gradient, preconditioner, iterative methods, parallel computation, partitioned inverse method.

1. Introduction

Many important practical problems, such as circuit analysis, numerical solution of boundary value problems and partial differential equations, discretization of elliptic boundary value problems by finite difference or finite element methods, give rise to large systems of linear equations. For many large linear systems, direct methods based on variants of Gaussian elimination are not practical, even when sparse matrix methods are used. Iterative methods are used to solve very large linear systems.

For the sake of completeness a brief introduction to iterative methods specifically to the conjugate gradient method [1] is presented next. An iterative method to solve an $n \times n$ linear system starts with an initial approximation $\mathbf{x}^0$ to the solution $\mathbf{x}$ and generates a sequence of vectors $\{\mathbf{x}^i\}_{i=1}^{\infty}$ that converges to the solution $\mathbf{x}$. Consider the linear equations:

$$A\mathbf{x} = \mathbf{b}, \tag{1}$$

where it is assumed for the rest of the paper that $n \times n$ matrix A is symmetric and positive definite. The minimization of the quadratic function

$$Q(\mathbf{x}) = \frac{1}{2}\,\mathbf{x}^T A\,\mathbf{x} - \mathbf{b}^T\mathbf{x} \tag{2}$$

is equivalent to the solution of (1). There are numerous iterative methods to solve (2). Most of these methods are of the form:

$$\mathbf{x}^{i+1} \leftarrow \mathbf{x}^i - \alpha_i\,\mathbf{p}^i \tag{3}$$

where the $\mathbf{p}^i$ vectors represent a sequence of direction vectors and the scalars α_i represent a sequence of scalars which determine the distance to be moved along the $\mathbf{p}^i$ direction at every iteration. For more details about iterative methods refer to [2-6]. Some of the most widely used iterative methods are based on the conjugate gradient method [1]. In theory, a conjugate gradient method converges to an exact solution in at most n iterations if exact arithmetic is used. Thus, a conjugate gradient method is sometimes considered as a direct method. In practice, however, the conjugate gradient method is used as an iterative method.

The actual rate of convergence of a conjugate gradient method depends on the distribution of eigenvalues, which implicitly defines the condition number of A, $\kappa(A)$, defined as $\|A\| \cdot \|A^{-1}\|$. If the two norm is used, $\kappa(A) = \kappa_2(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$. The bigger the condition number, the slower the convergence rate [7-9]. A desire to improve the convergence rate of the conjugate gradient algorithm leads to the preconditioned conjugate gra-

dient algorithm. Preconditioning corresponds to a congruence (linear) transformation:

$$\hat{A} = SAS^T, \tag{4}$$

where S is a nonsingular matrix chosen so that $\kappa(\hat{A}) < \kappa(A)$ [10]. The system to be solved becomes:

$$\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}, \tag{5}$$

where $\hat{\mathbf{x}} = S^{-T}\mathbf{x}$ and $\hat{\mathbf{b}} = S\mathbf{b}$. There are several ways of choosing the matrix S. An ideal preconditioned matrix $\hat{A}$ can be obtained by the choice of $S = A^{-1/2}$ so that $\hat{A} = I$, but this is not a practical choice. The preconditioned conjugate gradient method is rearranged in such a way that it does not make explicit use of matrix S. Instead, it uses matrix A and a new symmetric positive definite matrix M, defined as $(S^TS)^{-1}$. This matrix M is also not computed explicitly. The following algorithm describes the basic preconditioned conjugate gradient method [5].

Initialize:
Select $\mathbf{x}^0$
Let $\mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0$
Solve $M\,\tilde{\mathbf{r}}^0 \leftarrow \mathbf{r}^0$
Let $\mathbf{p}^0 \leftarrow \tilde{\mathbf{r}}^0$

Iterate:
$\alpha_k \leftarrow -(\tilde{\mathbf{r}}^k, \mathbf{r}^k)/(\mathbf{p}^k, A\,\mathbf{p}^k)$
$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k - \alpha_k\mathbf{p}^k$
$\mathbf{r}^{k+1} \leftarrow \mathbf{r}^k + \alpha_k A\,\mathbf{p}^k$
Solve $M\,\tilde{\mathbf{r}}^{k+1} = \mathbf{r}^{k+1}$
$\beta_k \leftarrow (\tilde{\mathbf{r}}^{k+1}, bfr^{k+1})/(\tilde{\mathbf{r}}^k, \mathbf{r}^k)$
$\mathbf{p}^{k+1} \leftarrow \tilde{\mathbf{r}}^{k+1} + \beta_k\mathbf{p}^k$

End:

The direction vectors $\mathbf{p}^i$ are not specified ahead of time, but rather are computed while executing the algorithm. The algorithm is stopped when the norm of the residual vector $||\mathbf{r}^{k+1}||$ becomes sufficiently small. It is desirable to be able to solve the set of equations $M\tilde{\mathbf{r}} = \mathbf{r}$ easily. At the same time, the preconditioner M should make the eigenvalue distribution of the preconditioned system more clustered than that of the original matrix A. A diagonal matrix $M = diag(a_{11}, a_{22}, \cdots, a_{nn})$ makes $M\tilde{\mathbf{r}} = \mathbf{r}$ very easy to solve, but eigenvalue clustering is often not good enough with this choice. A commonly used preconditioner, the ILU preconditioner [4, 5, 6, 11] is introduced in Section 2 for the sake of completeness.

The preconditioned conjugate gradient method (PCG) requires at each iteration two inner products, 3 saxpy operations, one matrix vector product, two scalar comparisons, and a linear solver, i.e., $M\tilde{\mathbf{r}} = \mathbf{r}$. The linear solver can be implemented as a forward and backward substitution process. The dependencies associated with this substitution limit the parallelism of PCG. This paper assumes that the traditional ILU preconditioner is available and concentrate on the solution of the boxed line, that is, $M\tilde{\mathbf{r}} = \mathbf{r}$, in the above algorithm. Specifically, the paper introduces a preconditioner based on the partitioned inverse representation of ILU preconditioner. With this preconditioner the linear-solver part of the above algorithm becomes essentially a matrix-vector product, which can be readily done in parallel.

Partitioned incomplete inverse preconditioners are introduced in Section 3 and experimental results are presented in Section 4.

2. Incomplete LU (ILU) Preconditioners

Given a symmetric positive definite matrix A, the Cholesky factorization of A is

$$A = L\,D\,L^T. \tag{6}$$

For large systems A is usually sparse. Due to fill-in, L can be considerably less sparse than A. Thus instead of using L, an approximation to L denoted by $\widetilde{L}$ can be used:

$$A = \widetilde{L}\,\widetilde{D}\,\widetilde{L}^T + B \tag{7}$$

where $B \neq 0$ is an error matrix, and $\widetilde{L}$ is a unit lower triangular matrix, which is more sparse than L. This matrix implicitly defines $M = \widetilde{L}\widetilde{L}^T$ (M is never computed explicitly) and it is called an *incomplete factorization* of A [5] for general matrix A and is called *incomplete Cholesky* for symmetric matrix A. The term incomplete LU will be used in this paper even though matrix A is assumed to be symmetric.

There are several possible ways of constructing and defining $\widetilde{L}$. One possible way is to construct L, and then discard those entries within L that correspond to zero positions in A. This approach is inefficient in that it requires the computation of the entire L matrix, which is often

a costly step. However, in a limited number of experiments it has led to superior numerical performance for least squares problems [12].

A more efficient way to obtain ILU factors is to perform an ordinary factorization of a matrix, but preclude the creation of any new nonzeros. That is, all computation involving fills is suppressed during the factorization process. This simple departure from ordinary LU factorization is the "level 0" ILU algorithm [11].

The numerical performance of an ILU algorithm can be improved if some fill-ins are permitted to occur. The simplest possibility is to permit the occurrence of fills that involve original matrix entries, but preclude the creation of fill entries that depend on prior fills. This is the "level 1" ILU algorithm. Further levels of fills based on prior fills may be permitted, defining higher level incomplete factorization algorithms. The more levels that are included, the closer $\widetilde{L}$ can be expected to be to L. However, more accuracy also implies greater density. It has been observed that the number of iterations of the conjugate gradient method does not depend heavily on the number of nonzeros (fills) precluded, rather it depends on the norm of the error matrix B [13]. This paper assumes that the (incomplete) factors of A have already been computed.

3. Partitioned Incomplete Inverse Preconditioners

This section provides a brief review of the partitioned inverse method (known also as W-matrix method) for solving sparse linear equations [14, 15]. A sparse set of linear equations (1) can be solved in three steps: forward substitution, diagonal scaling and back substitution. The three steps is preceded by an ordering to reduce factorization fills and factorization of A into $L\,D\,L^T$.

$$L\mathbf{y} = \mathbf{b}, \quad D\mathbf{z} = \mathbf{y}, \quad L^T\mathbf{x} = \mathbf{z}. \tag{8}$$

Define:

$$W = L^{-1}. \tag{9}$$

The forward and back substitution steps (8) are replaced with the matrix-vector products:

$$\mathbf{y} = W\,\mathbf{b}, \quad \mathbf{z} = D^{-1}\,\mathbf{y}, \quad \mathbf{x} = W^T\,\mathbf{z}. \tag{10}$$

These products are quite amenable to parallel processing. The matrix L can also be expressed as the product of elementary matrices:

$$L = L_1 L_2 \cdots L_n, \tag{11}$$

where the elementary matrix L_i is an identity matrix except for its i^{th} column, which contains column i of L. The inverse of L can be written as:

$$W = L^{-1} = W_n W_{n-1} \cdots W_1. \tag{12}$$

The matrix W is a unit lower triangular matrix that is usually considerably denser than L. Its graph is the transitive closure of the graph associated with L [14].

An alternate representation for W is based on grouping the elementary inverse factors W_i into m groups of adjacent factor, with $m < n$:

$$W = \widehat{W}_m \widehat{W}_{m-1} \cdots \widehat{W}_1 \tag{13}$$

where each $\widehat{W}_k$ is the aggregate of several elementary inverse factor W_j. By aggregating the product in (12) into m factors (partitions) rather than just one factor (partition), the combined sparsity structure of these m factors of W can be the same as the structure of L itself. With suitable ordering and partitioning algorithms, it is usually possible to have $m \ll n$ [14-17]. With the partitioning of W the solution to linear system (1) takes the form:

$$\mathbf{x} = \widehat{W}_1^T \cdots \widehat{W}_{m-1}^T\, \widehat{W}_m^T\, D^{-1}\, \widehat{W}_m\, \widehat{W}_{m-1} \cdots \widehat{W}_1\, \mathbf{b}, \tag{14}$$

where $\mathbf{x}$ is computed by matrix-vector product from right to left.

Proposal 1: Computation of each $\widehat{W}_i$ can be done independently.
Proof : It is clear from the definition of $\widehat{W}_i$,

$$\widehat{W}_i = W_j\, W_{j+1} \cdots W_{j+k} = L_{j+k}^{-1} \cdots L_{j+1}^{-1}\, L_j^{-1} \quad \square.$$

Proposal 2: Assume that number of available processors is greater than number of partitions. Then, the computational complexity of partitioned inverses is determined by the dimension of the largest partition. Hence, it is bounded above by $\mathcal{O}(l^3)$ where l is the dimension of the largest partition $\widehat{W}_j$.
Proof : It is the result of proposal 1 $\square$.

This paper deals with both the unpartitioned version of W, which generally has many fills beyond those of L, defined by (10) and partitioned

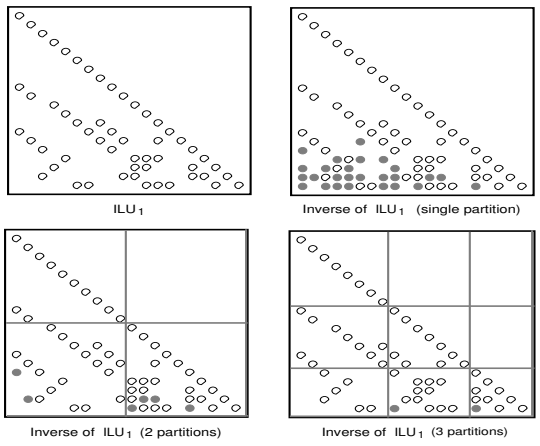

Figure 1. The relationship between inversion fills and number of partitions used when computing the inverse of ILU_1. The solid dots show the inversion fills.

versions of W as expressed by (14). The advantage of dealing with unpartitioned W is that only two serial steps are required to perform the parallel repeat solution phase of a problem. The disadvantage is that the number of inversion fills is very high and it is computationally expensive. The advantage of dealing with the partitioned W is that number of inversion fills decreases substantially and the issue of load distribution becomes easier. The disadvantage is that the number of serial steps increases twice the number of partitions. The relationship between inversion fills and number of partitions is illustrated in Fig. 1 for a 20×20 power network matrix.

Fig. 1 shows that the unpartitioned inversion of ILU_1 has 26 inversion fills. Number of fills decreases to 6 with two partitions, and it decreases to 2 with three partitions.

Studies [14, 18] show the clear advantages of using partitioned inverse (W-matrix) method based triangular solvers instead of substitution based solvers on parallel environments. Higham and Pothen [19] conclude that the partitioned triangular solver is a stable method so long as the matrix L is well conditioned. Two more studies [20, 21] are also suggesting approximate inverses as preconditioners. Both studies obtain approximate inverses by optimization. Two ways of enhancing the sparsity of partitioned inverses (as well as partitioning) are explored. Each of these involves approximation, in the same sense that using the ILU method involves approximations within the L. An explanation as to why partitioned inverse preconditioners work well as preconditioners is provided in the appendix.

3.1. Sparsified Partitioned Inverse Preconditioners (SW^xILU_i)

Sparsification is a very simple concept: small numeric entries in the W matrix are discarded. Empirical observation of numerical values in numerous W matrices has led us to the observation that many of the numeric entries, particularly those in positions that were zero in L, have numerical values that are quite small. As in the case of ILU factorization, discarding of these small numeric entries can take place a-posteriori (after all entries in W have been computed), or can take place during the computation. Clearly, a-posteriori discarding of numeric entries negates many of the computational advantages of sparsification, since W is often quite dense. However, it is more straightforward and issues of selection of discarding criteria are much simpler. This is the approach considered in this subsection. Once W has been obtained, its sparsified version $\widetilde{W}$ can be used for the iterative solution of the original problem. Furthermore, the same matrix can be used for many different right hand side vectors $\mathbf{b}$, thereby compensating for the initial inefficiency in the computation of $\widetilde{W}$. This type of preconditioner is denoted by SW^xILU_i where x is used as the criterion for discarding entries of W, and ILU_i denotes the factors from which the inverse factors are obtained, e.g., $SW^{0.05}ILU_0$ means inverse factors obtained by inverting zero level ILU. Any nonzero inverse entry with a value of 0.05 or smaller is discarded after the computation of W. An algorithm for SW^xILU_i can be given as:

Algorithm 1 Sparsified Partitioned Inverses (SW^xILU_i):

- Compute the level i incomplete LU factors of matrix A,
- Invert these incomplete LU factors (using one or multiple partitions),
- Discard nonzero entries in these inverted (partitioned) incomplete LU factors with

absolute value x or smaller.

Experimental indication that discarding small entries in W does not have a significant effect on the solution vector was presented in [22]. This paper further confirms these earlier results.

3.2. Partitioned Incomplete Inverse Preconditioners (IW_jILU_i)

A second possibility for sparsification of the W matrices can be based on the concept of "fill level," as in the case of ordinary ILU preconditioners. Here, fills do not refer to fills that occur during the factorization process but to fills that occur during the inversion of L. Since the determination of which entries of the W matrix to retain is based on topological considerations alone, this type of preconditioner can be called a (topological) partitioned incomplete inverse preconditioner. This preconditioner will be denoted by IW_jILU_i. Usually first, second, and maybe third level fills are allowed in ILU preconditioners. As in the case of the ILU preconditioners, it is possible to allow no inversion fill at all, to allow inversion fills on existing elements, or to allow inversion fills on both existing elements and first level inversion fills, and so on. The IW_jILU_i denotes many possible ways of obtaining (topological) partitioned incomplete inverse preconditioners, where j denotes the level of inversion fills for partitioned incomplete inverse preconditioners, and i denotes the level of factorization fills for ILU. For example, IW_2ILU_1 indicates level one ILU and level two inversion fills. In the extreme case IW_nILU_n means full factorization and full inversion. An algorithm for IW_jILU_i can be given as:

Algorithm 2 Partitioned Incomplete Inverses (IW_jILU_i):

- Compute level i incomplete LU factors of matrix A,
- Invert these incomplete LU factors (using one or multiple partitions) allowing up to j levels of fills in the inversion process.

3.3. Sparsified Partitioned Incomplete Preconditioners ($SW_j^xILU_i$)

A third possibility for sparsification is a combination of 3.1 and 3.2. This variant can be denoted by ($SW_j^xILU_i$). Algorithm 1 is a special case of this variant. That is, when $j = n$, SW^xILUi can be obtained. The following algorithm is suggested for this variant.

Algorithm 3 Sparsified Partitioned Incomplete Inverses ($SW_j^xILU_i$):

- Compute level i incomplete LU factors of matrix A,
- Invert these incomplete LU factors (with or without multiple partitions) allowing up to j levels of fills in the inversion process,
- Discard nonzero entries in these inverted (partitioned) incomplete LU factors with absolute value x or smaller.

There are two kinds of parallelism available in the proposed preconditioners. The first parallelism is in the computation of partitioned incomplete inverse preconditioners which is not exploited in this paper. The second parallelism is in the solution phase. The parallelism available in the solution phase of all the proposed partitioned incomplete inverse preconditioners depends on the number of partitions used. In the case of a single partition defined by (10) the parallelism is quite extreme: only two serial steps are required. In the multiple partitions case, defined by (14), number of serial steps in solving the linear system is twice the number of partitions. The optimal number of partitions will depend on the dimension of matrix, parallel environment and the ordering method, if any, is used.

The storage requirement for the proposed preconditioners is bounded by a full triangular matrix. As the number of partitions increases the storage requirement decreases due to lesser number of inversion fills. In the computation process of the preconditioners two auxiliary single-dimension arrays of length n are used. One of the array is used to keep inversion level information. That is, array stores zero for every entry initially and increases by one each time an inversion fill occurs. Next time when an inversion fill occurs the level information is checked to see whether new fill is due to earlier fills or due to original nonzeros. The second array is used for fast access to nonzero entries in the linked-list structure.

Ordering issues are not studied in this paper. It is true, however, that ordering has big effect on number of iterations with ILU precondition-

Table 1
Statics for test matrices.

Matrix	Size	NNZ	AvgNNZ	$\kappa(A)$	F-Fils	Inv-Fills
s1084	1084	3966	3.76	10	2124	44282
s1993	1993	7443	3.73	11	5318	145868
Bp1390	1209	4317	3.57	28594	2084	72596
Bp4403	3790	13862	3.73	19670	10234	335902
Bp8235	7060	25134	3.56	14428	15188	655898
f40c01	1000	3750	3.75	5570	14454	85198
f40c04	1104	3786	3.43	2391	12146	70272
im3k	3516	24828	7.06	1.4E6	33174	178216
im25k	26607	176435	6.63		121880	†

† : Not enough room for explicit inversion due to excessive amount of inversion fills.

ers and consequently on the proposed preconditioners [13, 23]. It also has effect on the number of optimal no-inversion fills partitions for inverse computation [14]. The secondary ordering also changes number of optimal partitions for inverse factors. In this paper partitions is done by almost equally dividing the matrix by number of processors used choosing last partitions slightly smaller than the rest of partitions. This is due to high concentration of nonzeros at the right-lower corner of the matrix.

4. Discussion of Test Results

In this section test results are presented following the description of both the testing environment and the data. The results for the unpartitioned case defined by (10) and the partitioned case defined by (14) for each of the proposed algorithm are presented separately. Many tests for both single and multiple partitions incomplete inverse preconditioners are conducted, but only a few are presented in the paper due to space limitations. The conclusions, however, are based on all the tests conducted.

4.1. Description of environment and test matrices

The test environment was a shared memory machine, *a Sequent Symmetry*, with 15 processors. Each 386-based Weitech processor with 16 MB memory. The maximum number of processors that can be allocated is 14. The environment is multi-user and time-sharing. Therefore, each test is run 10 times except im25k, which is run only once, and the average cpu times are reported.

The programming language is C and a data structure is made up of doubly linked lists. Both row and column numbers of nonzero entries are accessible.

Table 1 shows the statistics for the test matrices, where NNZ denotes number of nonzeros, AvgNNZ denotes average number of nonzeros per row, $\kappa(A)$ denotes the condition number of A, F-Fills denotes the number of factorization fills and Inv-Fills denotes total number of inversion fills. The condition number, factorization fills and inversions fills are obtained after an ordering of the matrices by multiple-minimum (**mmd**) degree of Liu [24].

Matrices s1084, s1993, Bp1390, Bp4403 and Bp8235 are power network matrices. Matrices s1084 and s1993 represent power network connections and are strictly diagonally dominant. Hence, they are very well-conditioned as it is apparent from Table 1. Matrices Bp1390, Bp4403 and Bp8235 are Jacobian matrices containing the power delivery information. They are also diagonally dominant and their condition number varies depending primarily to the length of transmission/distribution lines. Matrices f40c01 and f40c04 are finite element matrices taken from boeing collation [25]. f40 stands for file number in the boeing collection and c0i correspond to the i^{th} item in the file. Matrices im3k and im25k are related to animal models obtained from [26].

The most time consuming part of PCG is the preconditioner solver and matrix-vector product at each iteration. The matrix-vector product is done by row-interleaving. That is, processor 1 gets row one, processor 2 gets row 2, processor p gets row p assuming p processors are used. The next row for processor 1 is (p+1) and for processors 2 is (p+2) etc. By interleaving a balanced

Table 2
Comparison of $ILUi$ and Sparsified partitioned inverse preconditioners (1 Partition).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_1	$SW^{.01}ILU_0$	$SW^{.01}ILU_1$	$SW^{.05}ILU_0$	$SW^{.05}ILU_1$	$SW^{.1}ILU_0$	$SW^{.1}ILU_1$
s1084	7/1.1	**4/0.8**	7/1.4	4/2.4	7/1.4	6/2.5	7/1.4	7/2.4
s1993	6/1.8	**4/1.6**	7/2.4	4/4.1	7/2.4	6/4.2	7/2.4	7/4.2
Bp1390	57/7.9	**18/3.1**	57/4.9	18/4.6	58/4.7	18/4.4	60/4.7	18/4.3
Bp4403	33/15.5	20/11.3	32/9.0	20/10.0	33/8.9	20/9.6	**32/8.5**	23/9.8
Bp8235	34/29.3	22/22.5	34/17.1	22/19.3	35/16.8	22/18.7	**35/16.3**	24/18.8
f40c01	100/11.8	31/5.6	100/5.0	31/6.9	101/5.0	33/6.7	**103/4.9**	36/6.7
f40c04	40/4.9	20/4.1	40/2.7	20/8.6	**40/2.6**	29/8.4	44/2.7	29/8.4
im3k	46/35.3	**23/23.4**	47/37.6	24/102	50/33.5	36/96.2	56/33.2	58/98.6

load distribution is accomplished. As Figs. 2-4 show the matrix multiplication part of the code is scalable and achieves near ideal speed-up as long as enough data is available. The saxpy and norm computation is done by splitting vectors into p contiguous parts and assigning one part to each of p processors. That is, processor 1 gets the first chunk of a vector, processor 2 gets the second chunk of a vector, etc. This kind of load distribution prevents cache-misses.

The initial guess of PCG is zero and the stopping criterion is $5 \cdot 10^{-6}$. That is, the algorithm stops when $\|\tilde{r}\| < 5 \cdot 10^{-6}$. Even though the residual for the preconditioned system is used in testing for stopping, the convergence of the actual system is also checked at the end of convergence. In all cases tested, the use of the residual of the preconditioned system for stopping the algorithm proved to be adequate.

The reported cpu(second) time in Tables 2–8 include solution time of the PCG method excluding I/O time and is obtained using 14 processors. The cpu time does not include the computation time for ILU preconditioners. They are assumed to be available. The computation time of the partitioned incomplete inverse preconditioners, however, is included in the reported cpu time. ILU (also called Incomplete Cholesky) preconditioner is used in the square-root free form, i.e. $L\,D\,L^T$, in our implementation. In the solution phase of PCG only the diagonal scaling part is parallelized. The forward and backward substitutions do not lend themselves to parallelization. In order to have a fair comparison between ILU preconditioners and the proposed preconditioners, even though they are amenable to parallel processing, the computation of partitioned incomplete inverse preconditioners is also not parallelized.

4.2. Sparsified partitioned inverse preconditioners

In this section, sparsified partitioned inverse preconditioners are compared to traditional ILU preconditioners. Tables 2-4 present some of the test results. The cpu numbers are in seconds and are the average of 10 runs except im25k test case. Total of 14 processors are used in obtaining these cpu times.

Tables 2 and 3 use a pre-specified sparsification criterion while Table 4 uses a row norm based sparsification. That is, each nonzero entry on a row is compared to the given percentage of two norm of the row and the necessary action is taken. In Table 4, $SW^{fi}ILU_i$ refers to sparsified partitioned inverses where fi indicates the two norm-based sparsification and i indicates the percentage of the norm. Results indicate that norm-based sparsification is not as good as a prespecified criterion based sparsification. In both cases however, the proposed preconditioners perform better than ILU preconditioner in almost all the test cases when 4 partitions are used. For small well-conditioned matrices such s1084, s1993 the proposed preconditioners perform better with either a single partition or with two partitions. The reduction in cpu time is as big as 50% when the proposed preconditioners are used. In the case of im25k with 4 processors the cpu time for the proposed preconditioners is much bigger than that of ILU. This is due to both the lack of processors to use in the solution phase of PCG and the use of a single processor for computing the partitioned inverse preconditioner. The complete inverse computation is expensive. Note that if a bigger sparsification criterion is used the number

Table 3
Comparison of $ILUi$ and Sparsified partitioned inverse preconditioners (4 Partitions).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_1	$SW^{.01}ILU_0$	$SW^{.01}ILU_1$	$SW^{.05}ILU_0$	$SW^{.05}ILU_1$	$SW^{.1}ILU_0$	$SW^{.1}ILU_1$
s1084	7/1.1	**4/0.8**	7/1.3	4/1.2	7/1.3	5/1.3	7/1.3	7/1.4
s1993	6/1.8	4/1.6	6/1.9	4/2.1	7/2.0	**6/1.2**	7/1.9	7/2.2
Bp1390	57/7.9	18/3.1	57/4.3	18/2.4	58/4.3	**18/2.3**	58/4.3	**18/2.3**
Bp4403	33/15.5	20/11.3	32/8.1	20/6.8	32/8.0	**20/6.6**	33/8.0	21/6.7
Bp8235	34/29.3	22/22.5	34/15.4	22/13.2	35/15.6	**22/12.8**	35/15.3	23/12.8
f40c01	100/11.8	31/5.6	100/5.4	**31/3.1**	101/5.3	**33/3.1**	103/5.3	39/3.2
f40c04	40/4.9	20/4.1	40/3.0	20/4.0	**40/3.0**	23/3.9	43/3.1	29/4.0
im3k	46/35.3	23/23.4	47/18.4	23/32.6	47/17.0	28/32.6	**49/16.8**	38/34.4
im25k	21/124.3	**11/85.3**	21/188.4	12/8137	22/187.0	14/840.5	22/185.5	15/859.0

Table 4
Comparison of $ILUi$ and Sparsified (wrt row norm) partitioned inverse preconditioners (4 Partitions).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_1	$SW^{f1}ILU_0$	$SW^{f1}ILU_1$	$SW^{f2}ILU_0$	$SW^{f2}ILU_1$	$SW^{f5}ILU_0$	$SW^{f5}ILU_1$
s1084	7/1.1	**4/0.8**	7/1.3	4/1.3	7/1.3	4/1.2	7/1.3	5/1.3
s1993	6/1.8	**4/1.6**	6/1.9	4/2.1	6/1.9	4/2.0	6/1.9	5/2.1
Bp1390	57/7.9	18/3.1	57/4.3	**18/2.3**	57/4.3	18/2.4	58/4.3	23/2.6
Bp4403	33/15.5	20/11.3	32/8.1	20/6.9	32/8.1	20/6.8	33/8.2	**20/6.7**
Bp8235	34/29.3	22/22.5	34/15.7	22/13.8	34/15.5	22/13.2	35/15.6	**23/12.9**
f40c01	100/11.8	31/5.6	113/5.8	**66/4.6**	118/5.9	82/5.2	151/7.1	155/8.1
f40c04	40/4.9	20/4.1	40/3.0	20/4.1	**40/3.0**	21/4.0	40/3.0	22/3.9
im25k	21/124.3	**11/85.3**	23/211.2	16/1846	25/213	19/1912	34/225	33/1839

Table 5
Comparison of $ILUi$ and partitioned incomplete inverse preconditioners (1 Partition).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_2	IW_2ILU_0	IW_2ILU_1	IW_4ILU_0	IW_4ILU_1	IW_nILU_0	IW_nILU_1
s1084	7/1.1	**4/0.8**	7/1.0	6/1.1	7/1.1	6/1.4	7/1.1	4/1.8
s1993	6/1.8	**4/1.6**	7/1.7	6/2.1	6/1.9	6/2.5	6/1.9	4/3.3
Bp1390	57/7.9	**18/3.1**	236/12.8	308/17.6	117/7.6	285/18.3	57/4.7	18/4.4
Bp4403	33/15.5	20/11.3	110/19.3	168/30.6	70/14.0	142/29.2	33/8.9	**20/9.4**
Bp8235	34/29.3	22/22.5	101/33.3	302/97.6	29/19.7	147/55.4	**34/16.6**	22/18.5
f40c01	100/11.8	31/5.6	112/5.3	129/7.5	**101/5.1**	86/6.6	100/5.3	31/7.8
f40c04	40/4.9	20/4.1	**41/2.3**	34/3.2	40/2.4	33/3.8	40/2.4	20/8.7
im3k	46/35.3	23/23.4	**75/20.7**	99/31.6	69/24.3	99/37.1	46/42.8	23/105.4

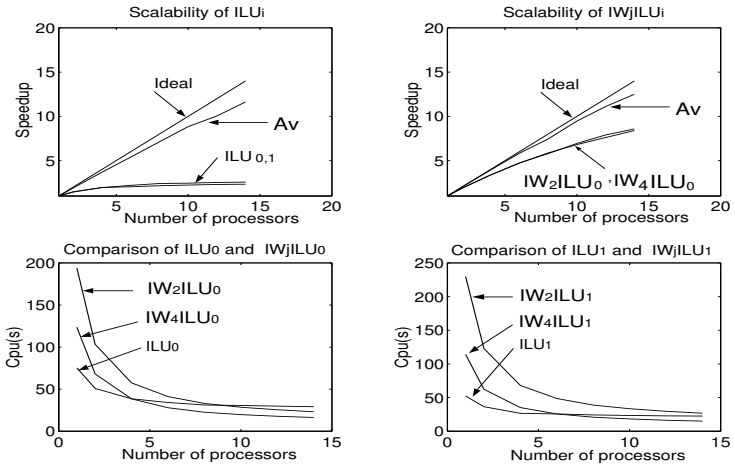

Figure 2. Comparisons of traditional ILU and the proposed partitioned incomplete inverse preconditioners (IW_jILU_i) in terms of both scalability and cpu time for matrix Bp8235. Four partitions are used for computing partitioned inverses. Av refers to matrix vector product.

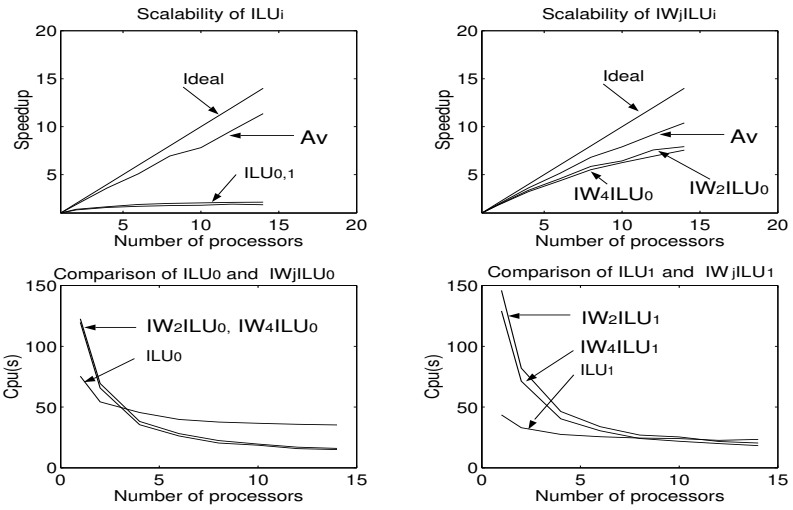

Figure 3. Comparisons of traditional ILU and the proposed partitioned incomplete inverse preconditioners (IW_jILU_i) in terms of both scalability and cpu time for matrix im3k. Four partitions are used for computing partitioned inverses. Av refers to matrix vector product.

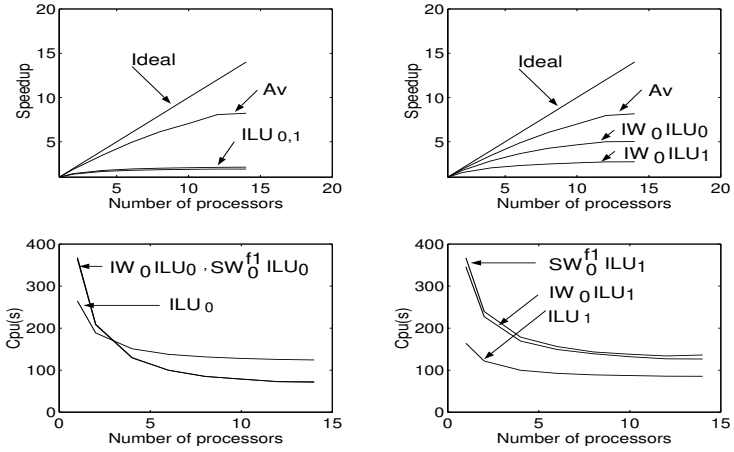

Figure 4. Comparisons of traditional ILU and the proposed partitioned and sparsified incomplete inverse preconditioners (IW_jILU_i and $SW_j^xILU_i$) in terms of both scalability and cpu time for matrix im25k. Four partitions are used for computing the partitioned inverses. Av refers to matrix vector product.

of iterations becomes higher, but cpu time does not necessarily increase in all cases. Also note that obtaining the proposed preconditioner from higher level of ILU does not mean smaller number of iterations. For example, obtaining $SW^{f1}ILU_i$ from ILU_0 has smaller number of iteration than that of obtaining from ILU_1, see Table 4. This is because ILU_0 has less number of inversion fills than ILU_1. Thus, the difference (in norm for example) between $W_2^{f1}ILU_0$ and full inverse of ILU_0 is smaller than that of $W_2^{f1}ILU_1$ and full inverse of ILU_1.

4.3. Partitioned incomplete inverse preconditioners

In this section the partitioned incomplete inverse preconditioner are compared to traditional incomplete LU preconditioners. For ILU preconditioners level zero and level one are used.

For the partitioned incomplete inverse preconditioners inversion fill levels from zero to four and a full inversion are used. Tables 5 and 6 present some results for these comparisons. For single partition case the proposed preconditioners are comparable to ILU preconditioners. For multiple partitions case, however, the proposed preconditioners perform better than ILU in terms of cpu time for all test cases. This is because as the number of partitions increases the amount of inversion fills decreases. Hence, incomplete inversion approximates full inversion better. If higher level of inversion fills is allowed in addition to multiple partitions, then the incomplete inverse approximates complete inverse more closely. This is a valid statement for all the proposed partitioned incomplete inverse preconditioners. For very large matrices multiple partitions must be used. Otherwise the amount of inversion fills will nullify any advantage of the proposed preconditioners. That is why, im25k is not tested with one partition. The im25k row in Table 6 has the following columns for the proposed preconditioners. The columns are in order of IW_0ILU_0, IW_0ILU_1, IW_1ILU_0, IW_1ILU_1, IW_2ILU_0, and IW_2ILU_1. Note that the number of iterations for IW_nILU_i is the same as that of corresponding ILU_i as expected.

4.4. Sparsified partitioned incomplete inverse preconditioners

In this section test results of sparsified partitioned incomplete inverse preconditioners are presented. The sparsification is based on norm of each row. Sparsification is done after computing the partitioned incomplete inverse preconditioners. $SW_j^{f1}ILU_i$ refers to sparsified partitioned incomplete inverse preconditioner that is obtained from ILU_i by allowing up to j level of inversion fills and using norm based sparsification. Any nonzero entry in a row which is less than 1% of the two norm of the row is discarded. All diagonal entries and the largest nonzero entry in each

Table 6
Comparison of $ILUi$ and partitioned incomplete inverse preconditioners (4 Partitions).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_1	IW_2ILU_0	IW_2ILU_1	IW_4ILU_0	IW_4ILU_1	IW_nILU_0	IW_nILU_1
s1084	7/1.1	**4/0.8**	7/0.9	**4/0.8**	7/1.0	4/0.9	7/1.0	4/0.9
s1993	6/1.8	4/1.6	6/1.5	5/1.6	**6/1.5**	4/1.6	6/1.5	4/1.5
Bp1390	57/7.9	18/3.1	63/4.4	40/3.2	57/4.09	**20/2.0**	57/4.1	18/2.1
Bp4403	33/15.5	20/11.3	54/11.2	47/10.9	36/8.3	24/6.8	33/7.9	**20/6.5**
Bp8235	34/29.3	22/22.5	62/23.1	67/26.9	39/16.4	31/15.1	34/14.9	**22/12.4**
f40c01	100/11.8	31/5.6	100/5.4	**38/3.1**	100/5.4	36/3.3	100/5.4	31/3.3
f40c04	40/4.9	20/4.1	40/2.6	**23/2.5**	40/2.6	22/2.7	40/2.6	20/4.0
im3k	46/35.3	23/23.4	**51/15.1**	46/18.3	49/16.1	46/20.3	46/20.1	23/33.8
im25k	21/124.3	11/85.3	**22/71.3**	14/127.6	21/103	13/138	21/162	13/142

Table 7
Comparison of ILU_i and sparsified partitioned incomplete inverse preconditioners (1 Partition).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_1	$SW_2^{f1}ILU_0$	$SW_2^{f1}ILU_1$	$SW_3^{f1}ILU_0$	$SW_3^{f1}ILU_1$	$SW_4^{f1}ILU_0$	$SW_4^{f1}ILU_1$
s1084	7/1.1	**4/0.8**	7/1.3	6/1.5	7/1.4	6/1.7	1/1.5	6/1.8
s1993	6/1.8	**4/1.6**	7/2.2	6/2.6	7/2.3	6/2.8	6/2.3	6/3.1
Bp1390	57/7.9	**18/3.1**	225/12.3	307/17.3	181/10.7	176/11.1	116/7.5	284/17.7
Bp4403	33/15.5	**20/11.3**	109/19.6	172/31.3	102/19.1	170/35.4	70/14.4	146/29.2
Bp8235	34/29.3	**22/22.5**	101/34.0	311/99.7	75/27.4	176/62.2	51/21.0	147/ 54.8
f40c01	100/11.8	31/5.6	120/5.5	185/8.9	116/5.4	144/7.7	**111/5.2**	137/7.7
f40c04	40/4.9	20/4.1	41/2.7	34/3.6	**41/2.7**	33/3.8	40/2.7	33/4.2
im3k	46/35.3	23/23.4	**73/20.3**	99/31.2	73/21.9	97/33.0	69/22.9	99/35.1

Table 8
Comparison of ILU_i and sparsified partitioned incomplete inverse preconditioners (4 Partitions).

	Number of iterations/cpu(s) with preconditioner:							
Matrix	ILU_0	ILU_1	$SW_2^{f1}ILU_0$	$SW_2^{f1}ILU_1$	$SW_3^{f1}ILU_0$	$SW_3^{f1}ILU_1$	$SW_4^{f1}ILU_0$	$SW_4^{f1}ILU_1$
s1084	7/1.1	**4/0.8**	7/1.3	4/1.2	7/1.3	4/1.2	7/1.3	4/1.2
s1993	6/1.8	**4/1.6**	6/1.9	5/2.0	6/1.9	4/2.0	6/1.9	4/2.0
Bp1390	57/7.9	bf 18/3.1	85/5.7	86/6.0	63/4.6	40/3.4	62/4.6	**33/3.1**
Bp4403	33/15.5	20/11.3	55/11.9	47/11.3	47/10.6	35/9.2	36/8.7	**24/7.3**
Bp8235	34/29.3	22/22.5	62/24.1	66/27.2	47/19.5	45/20.7	39/16.9	**31/15.8**
f40c01	100/11.8	31/5.6	135/5.8	**70/4.5**	113/5.8	70/4.5	113/5.8	70/4.6
f40c04	40/4.9	20/4.1	40/3.0	**23/2.9**	40/3.0	22/2.9	40/3.0	22/3.1
im3k	46/35.3	23/23.4	**51/15.4**	46/18.7	50/15.6	49/20.2	50/16.0	48/20.6
im25k	21/124.3	11/85.3	**23/71.1**	17/133	23//95	16/144	23/137	16/152

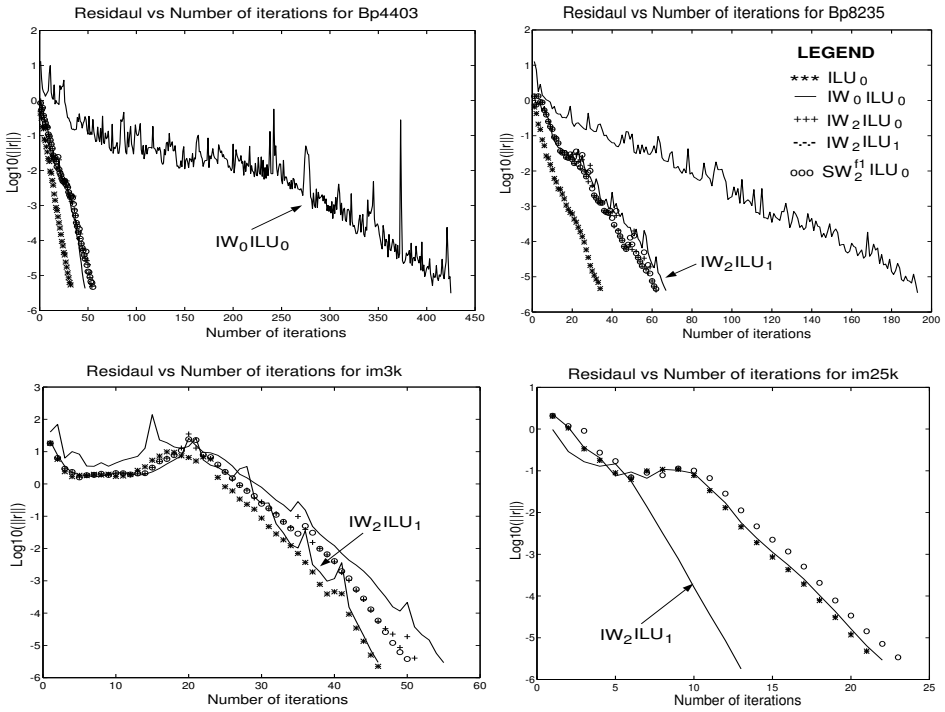

Figure 5. Residual of preconditioned linear system vs number of iteration for different test matrices.

row are kept. Tables 7-8 show the test results for these experiments. In the single partitioned case sparsified partitioned incomplete inverse preconditioners perform worse than ILU preconditioner for most of the test cases. This is due to both incomplete inversion and the following sparsification which makes approximation to full inverse worse.

Finaly, Table 9 shows the comparisons of the cpu times to obtain some of the proposed preconditioners to total solution time of PCG. The preconditioners are obtained using 4 partitions and a single processor is used for computing the inverses but 14 processors for the rest of the solution including the sparsification part of the preconditioners. Expectedly as the dimension of matrix becomes bigger cpu time gets bigger but not as $\mathcal{O}(n^3)$. If preconditioner is obtained from zero level ILU with no inversion fills the cpu time for obtaining the preconditioner is less than a third of total solution time for the largest matrix tested. If inversion process is also implemented in parallel this cpu time will be a much smaller fraction of total solution time. The number in the parenthesis for the sparsified preconditioners in Table 9 shows sparsification time. For the largest matrix tested the sparsification is about a third of total cpu time for obtaining the preconditioner. It can be substantially reduced with higher number of processors. In our case the maximum number of processor is 14.

Figs. 2-4 show the scalability of both ILU and the propsed preconditioners. As expeceted, ILU preconditioners do not lend themselves to parallelization and their performance get worse as number of processors increases. The proposed preconditioners, on the other hand, are scalable. The scalability is affected by the sparsity of preconditioner. Fig. 5 shows the residual behavior of the preconditioned system with respect to iterations and it suggests that zero level inversion of ILU_0 is not a good preconditioner for power system jacobian matrices, but it is quite good for animal model matrices.

5. Conclusions

Two new kinds of preconditioners based on sparse approximations to partitioned representa-

Table 9
Cpu (second) time to obtain some of the proposed preconditioners

	cpu(s) for preconditioner/Total solution time(s)					
Matrix	Size	IW_0ILU_0	IW_2ILU_0	IW_2ILU_1	$SW_2^{f1}ILU_0$	$SW^{.01}ILU_0$
Bp4403	3790	1.1/72.4	1.3/11.4	1.5/11.1	1.9(.6)/12.2	2.1(.7)/8.3
Bp8235	7060	2.3/6.0	2.6/23.3	2.8/27.2	4(1.4)/24.6	3.9(.9)/15.8
im3k	3516	1.5/14.7	1.8/15.2	4.8/18.4	2.9(1.1)/15.3	5.3(1.8)/18.5
im25k	26607	21.3/71.8	63.6/161	106/141.8	94.3(30.4)/138.1	155.8(54.86)/307.2

tions of A^{-1} are introduced and compared against well established traditional ILU preconditioners. The proposed preconditioners perform as good as ILU preconditioners when a single partition is used in computing the inverse of ILU preconditioner. They perform better than ILU preconditioners in terms of both cpu time and scalability when multiple partitions are used in computing the inverse of ILU.

The scalability of the proposed preconditioners are much better than that of ILU. The number of iterations does not increase with the dimension of matrix.

6. Acknowledgement

We would like to thank Ingacy Misztal for providing us with the animal model test data. We also like to thank the anonymous reviewers for their constructive comments and suggestions for improving the paper. The support from the National Science Foundation under grant ECS-9216308 is gratefully acknowledged.

References

[1] M. R. Hestenes and E. Stiefel, J. Res. Nat. Bur. Stand. **49**, 409 (1952).
[2] O. Axelsson, Iterative Solution Methods (Cambridge University Press, 1994).
[3] R. Freund, G. Golub, and N. Nachtigal, Iterative solution of linear systems (Cambridge University Press, 1992).
[4] G. H. Golub and C. F. Van Loan, Matrix Computations (Johns Hopkins University Press, Baltimore, 1989).
[5] J. M. Ortega, Introduction to Parallel and Vector Solution of Linear Systems (Plenum Press, 1988).
[6] D. M. Young, Iterative Solution of Large Linear Systems (Academic Press., 1971).
[7] O. Axelsson and T. A. Lindskog, Numerische Mathematik **48** 499 (1986).
[8] A. Jennings, Journal of the Institute of Mathematics and Applications **20**, 61 (1977).
[9] A. van der Sluis and H. van der Vorst, Numerische Mathematik **48**, 543 (1986).
[10] R. A. Horn and C. R. Johnson, Matrix Analysis (Cambridge University Press, 1990).
[11] T. Manteuffel, Mathematics of Computation **34**, 473 (1980).
[12] H. Dağ, F. L. Alvarado, and H. Singh, Variations on ILU preconditioners applied to electric network least squaresproblems, Proceedings of the fifth SIAM Conference on Linear Algebra, (Snowbird, Utah, 1994).
[13] I. S. Duff and G. A. Meurant, BIT **29**, 635 (1989).
[14] F. L. Alvarado and R. Schreiber, SIAM Journal on Scientific and Statistical Computing, 446 (1993).
[15] M. K. Enns, W. F. Tinney, and F. L. Alvarado, IEEE Transactions on Power Systems **5**, 466 (1990).
[16] F. L. Alvarado, D. C. Yu, and R. Betancourt, IEEE Transactions on Power Systems **5**, 452 (1990).
[17] A. Pothen and F. L. Alvarado, SIAM Journal on Scientific and Statistical Computing, (1992).
[18] F. L. Alvarado, A. Pothen, and R.Schreiber, Volume in Mathematics and its Applications **56**, 141 (1993).
[19] N. J. Higham and A. Pothen, SIAM Journal on Scientific and Statistical Computing **15**, 139 (1994).
[20] T. Huckle and M. Grote, Technical Report SCCM-94-03, Scientific Computing and Computational Mathematics Program (Computer Science Department Stanford University, Stanford, CA 94305, 1994).
[21] L. Y. Kolotilina and Y. Yeremin, SIAM Journal of Matrix Analysis and Applications **14**, 45 (1993).
[22] H. Dağ and F. L. Alvarado, Propagation of perturbation in entries of power system W-matrices, Proceedings of the North American Power Symposium, (Carbondale, Illinois, 1991).
[23] H. Dağ and F. L. Alvarado, The effect of or-

dering on the preconditioned conjugate gradient method for power system applications, Proceedings of the North American Power Symposium, (Manhattan, KS, 1994).
[24] J. W. H. Liu, ACM Transactions on Mathematical Software **11**, 141 (1985).
[25] I. S. Duff, R. Grimes, and J. Lewis, ACM Transactions on Mathematical Software **15**, 1 (1989).
[26] I. Misztal and M. Perez-Enciso, Journal of Dairy Science **76**, 1479 (1993).
[27] J. W. H. Liu, SIAM Journal of Matrix Analysis and Applications **11**, 134 (1990).
[28] F. L. Alvarado, W. F. Tinney, and M. K. Enns, Control and Dynamic Systems **41**, 207 (1991).
[29] W. F. Tinney, V. Brandwajn, and S. M. Chan, IEEE Transactions on Power Apparatus and Systems **104**, 295 (1985).
[30] J. G. Lewis, B. W. Peyton, and A. Pothen, SIAM Journal on Scientific and Statistical Computing **10**, 1146 (1989).

Appendix

Theoretical basis for the partitioned inverse preconditioners

Sparsification works quite well in part because the discarded values are numerically small, but also in part because of some more fundamental topological properties of partitioned inverse matrices. A perturbation to an entry of a connected matrix A or to any of its factors propagates to the entire solution vector $\mathbf{x}$, regardless of the vector or which entry has been perturbed. More formally,

$$(\Delta a_{ij} \neq 0) \text{ and } (||\mathbf{b}|| \neq 0) \quad \Rightarrow \quad \Delta x_j \neq 0 \, \forall \, j \tag{15}$$

The numeric values of the perturbations to many of the entries of $\mathbf{x}$ may be quite small, but the effect is nonzero. Likewise, perturbations to arbitrary entries of the L and L^T factors of A propagate to the entire solution vector:

$$(\Delta \ell_{ij} \neq 0) \text{ and } (||\mathbf{b}|| \neq \mathbf{0}) \quad \Rightarrow \quad \mathbf{\Delta x_j \neq 0 \, \forall \, j}$$

$$(\Delta \ell_{ji} \neq 0) \text{ and } (||\mathbf{b}|| \neq \mathbf{0}) \quad \Rightarrow \quad \mathbf{\Delta x_j \neq 0 \, \forall \, j}$$

The basic observation in this section is that perturbations to individual entries of the inverse factors W do not always have an effect on the solution vector, depending on the solution vector itself and on the nature of the elimination tree for the matrix. This requires a review of the concept of elimination trees [27] and factorization paths within elimination trees [28, 29]. Also of interest are definitions of descendants and ancestors in elimination trees [30]. Consider the matrix illustrated in Fig. 6. Its elimination tree is illustrated in Fig. 7.

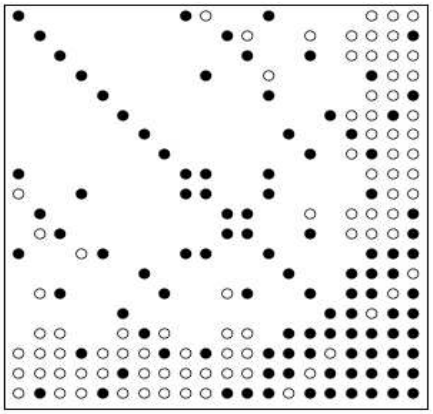

Figure 6. A 20 by 20 perfect elimination matrix. Solid dots denote nonzero entries of L and U factors. Circles denote additional nonzero entries for the factored inverse matrices W and W^T.

Of interest will be both the set of descendants of a node, as illustrated in Fig. 7, and the set of ancestors of a node, as illustrated in Fig. 7.

Consider two cases: perturbations to entries of W and perturbations to entries of W^T. The solution procedure using the partitioned inverse requires two matrix-vector products, as indicated in (10).

Perturbations to W affect the computation of $\mathbf{y}$ and consequently the computation of $\mathbf{x}$. Perturbations to entries of W^T affect the computation of $\mathbf{x}$ directly. Consider first the case of perturbations to entries of W. The perturbed element is w_{ij}, where $i > j$. There are three possible right hand side vectors $\mathbf{b}$, each leading to different conclusions about the effect of the perturbation Δw_{ij}. The possibilities are:

1. The right hand side vector $\mathbf{b}$ is a full vector (that is, $b_j \neq 0 \; \forall \; j$).

2. The right hand side vector $\mathbf{b}$ is a singleton (that is, $\mathbf{b} = \mathbf{e}_k$) and the nonzero position k corresponds to the column of the perturbed

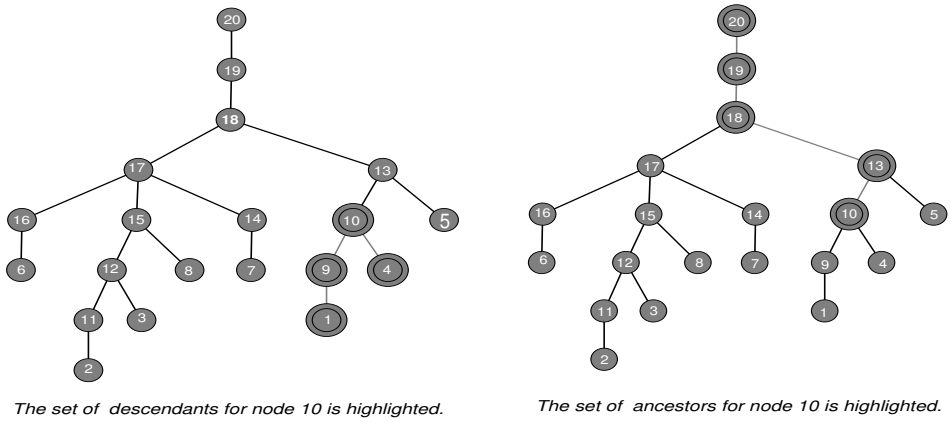

Figure 7. Elimination tree for matrix.

entry (that is, $k = j$).

3. The right hand side vector $\mathbf{b}$ is a singleton $\mathbf{e}_k$, and the singleton position k does not correspond to the column of Δw_{ij} (that is, $j \neq k$).

Observation (For case 1a): Perturbations Δw_{ij} when $\mathbf{b}$ is full, only element y_i is affected by the perturbation to w_{ij}. Because an element in $\mathbf{y}$ has been affected, pre-multiplication by W^T to obtain $\mathbf{x}$ affects all elements in $\mathbf{x}$ in the i^{th} column of W^T. These are the elements in the set of descendants of i. Fig. 8 illustrates this situation for the example at hand.

Observation (For case 2a): A perturbation to an entry Δw_{ij} when $\mathbf{b} = \mathbf{e}_j$ results in a perturbation to the i^{th} element of $\mathbf{y}$, and consequently to entries in $\mathbf{x}$ in the set of descendants for node i. This is illustrated in Fig. 9.

Observation (For case 3a): A perturbation to an entry of W when the right hand side is a singleton vector and $k \neq j$, the $\mathbf{y}$ vector is simply not affected and consequently neither is the $\mathbf{x}$ vector. Thus, a perturbation to the W matrix in this position will not have any effect on the solution *regardless of how large the numeric value of the perturbation is.* This is illustrated in Fig. 10.

Consider next perturbations to W^T. Because these perturbations are applied after the "forward substitution" step, they have no effect on $\mathbf{y}$. Once again, three cases are possible.

Observation (For case 1b): The vector $\mathbf{b}$ is full. In this case, the intermediate vector $\mathbf{y}$ will also be full. A perturbation to position (i, j) of W^T results in a perturbation to the i^{th} position of the

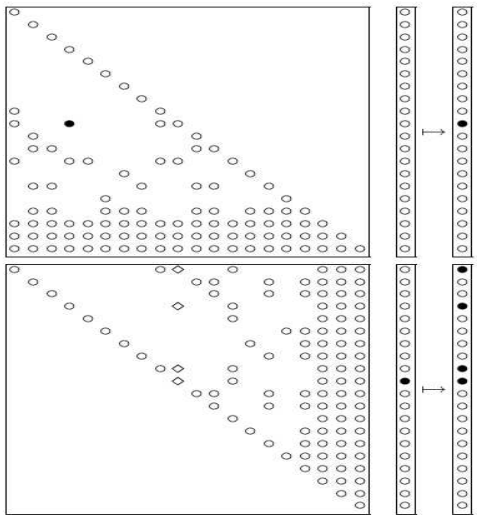

Figure 8. Perturbation to $(10, 4)$ element of W, full right hand side vector. $\circ$ denotes nonzero positions, $\diamond$ denotes nonzero positions of particular relevance to the computation, and $\bullet$ denotes changed values. Top equation is $W\mathbf{b} = \mathbf{y}$, bottom equation is $W^T\mathbf{z} = \mathbf{x}$, where $\mathbf{z} = D^{-1}\mathbf{y}$. Descendants of node 10 are affected.

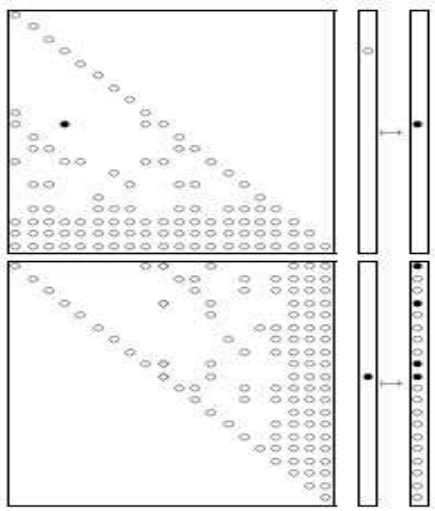

Figure 9. Perturbation to $(10, 4)$ element of W, singleton right hand side vector e_4 $(j = k)$. Descendants of node 10 are affected.

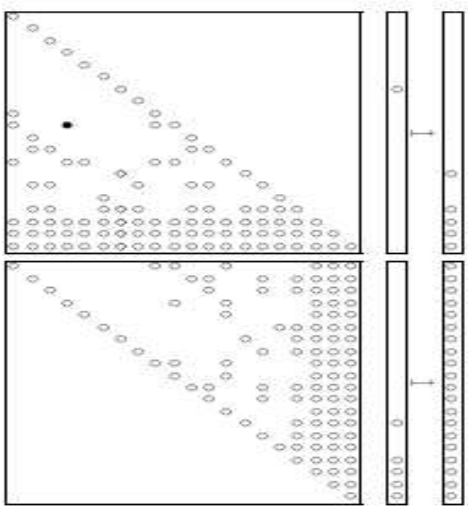

Figure 10. Perturbation to $(10, 4)$ element of W, singleton right hand side vector e_7 $(j \neq k)$. No elements of the solution vector are affected.

solution vector. Thus, a single element of $\mathbf{x}$ is affected. This is illustrated in Fig. 11.

The second case involves perturbations to an entry of W^T when the right hand side vector $\mathbf{b}$ is a singleton.

Observation (For case 2b): If $\mathbf{b} = \mathbf{e}_k$, the only nonzero entries in the vector $\mathbf{y}$ will be those entries in the set of ancestors for node k. The only way in which an element of $\mathbf{x}$ can be affected by a perturbation to position (i, j) of W^T is if $j \in \text{anc}(k)$, that is, if j belongs to the set of ancestors of k. In this case, only the i^{th} position of the solution vector $\mathbf{x}$ is affected. In all other cases, the perturbation has no effect. This is illustrated in Fig. 12.

The final case is when $\mathbf{b}$ is a singleton vector $\mathbf{e}_k$, but $j \notin \text{anc}(k)$.

Observation (For case 3b): In this case the solution vector $\mathbf{x}$ is not affected, regardless of how large the perturbation to W^T is. This is illustrated in Fig. 13.

All these cases are summarized in Table 10.

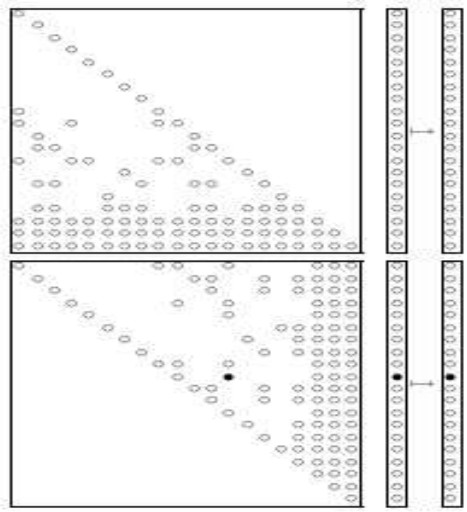

Figure 11. Perturbation to $(10, 13)$ element of W^T, full right hand side vector. Only element 10 of the solution vector is affected.

Table 10
Effect of perturbations to single entries of W.

	ΔW	ΔW^T
	$(\ \Delta w_{ij}, j < i\ )$	$(\ \Delta w_{ij}, j > i\ )$
Full RHS	$\Delta x_m, m \in \text{desc}(i)$	Δx_j
$\mathbf{b} = \mathbf{e}_j, \quad j = k$	$\Delta x_m, m \in \text{desc}(i)$	Δx_j
$\mathbf{b} = \mathbf{e}_j$	$\Delta x_m, m \in \text{desc}(i)$	Δx_j
	$(j \neq k)$	$(j \notin \text{anc}(k))$

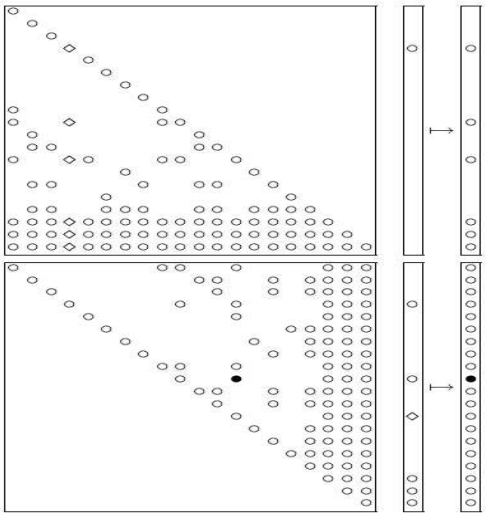

Figure 12. Perturbation to $(10, 13)$ element of W^T, singleton right hand side vector e_4, $j \in \text{anc}(k)$. Only element 10 of the solution vector is affected.

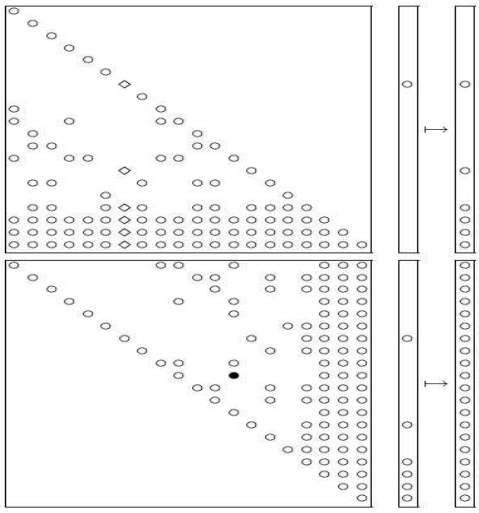

Figure 13. Perturbation to $(10, 13)$ element of W^T, singleton right hand side vector e_7, $j \notin \text{anc}(k)$. No elements of the solution vector are affected.