COMPUTATION-FREE PRECONDITIONERS FOR THE PARALLEL SOLUTION OF POWER SYSTEM PROBLEMS

Hasan Dağ Fernando L. Alvarado
Department of Electrical and Computer Engineering
The University of Wisconsin – Madison

Abstract

Solution of a set of linear equations $A\mathbf{x} = \mathbf{b}$ is a recurrent problem in power system analysis. Because of computational dependencies, direct methods have proven of limited value in both parallel and highly vectorized computing environments. The preconditioned conjugate gradient method has been suggested as a better alternative to direct methods. The preconditioning step itself is not particularly well suited to parallel processing. Partitioned inverse representations of A are better suited to high performance computation. However, obtaining the partitioned inverse matrices can be expensive. This paper describes two techniques for preconditioning based on the partitioned inverses where the preconditioner matrix is obtained directly from an incomplete factorization without the need for additional numerical computation. Experiments indicate a 50% reduction in solution time in a parallel environment.

Keywords Iterative methods, conjugate gradient, partitioned inverse, parallel processing.

1 Introduction

Solution of a set of large sparse linear equations of the form:

$$A\mathbf{x} = \mathbf{b}, \qquad (1)$$

where matrix A is symmetric and positive definite, is an integral part of many power system algorithms. The problem arises as part of the solution of power flow equations by the fast decoupled load flow method, the state estimation problem, the security analysis problem, and also during transient stability and electro-magnetic transient analysis.

The time-honored method to solve sparse linear equations in power systems since about 1967 has been the use of a direct method based on ordered sparse elimination [24]. As the dimension of the system grows, direct methods become increasingly impractical. For power systems problems, solution times for direct methods grow faster than linearly (but usually less than quadratically) with matrix dimension [2].

96 WM 327-7 PWRS A paper recommended and approved by the IEEE Power System Engineering Committee of the IEEE Power Engineering Society for presentation at the 1996 IEEE/PES Winter Meeting, January 21-25, 1996, Baltimore, MD. Manuscript submitted July 31, 1995; made available for printing December 15, 1995.

Iterative methods, such as Preconditioned Conjugate Gradient (PCG) methods, offer an alternative to direct methods for certain problems, particularly in parallel or highly vectorized environments, where data dependencies inherent in direct methods cannot be tolerated. The PCG algorithm [17] is given below.

```
Initialize
    Select $\mathbf{x}^0$
    Let $\mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0$
    $\boxed{Solve\ M\,\tilde{\mathbf{r}}^0 \leftarrow \mathbf{r}^0}$
    Let $\mathbf{p}^0 \leftarrow \tilde{\mathbf{r}}^0$
while $(\tilde{\mathbf{r}}^k, \mathbf{r}^k) \geq \epsilon$ do
    $\alpha_k \leftarrow -(\tilde{\mathbf{r}}^k, \mathbf{r}^k)/(\mathbf{p}^k, A\,\mathbf{p}^k)$
    $\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k - \alpha_k \mathbf{p}^k$
    $\mathbf{r}^{k+1} \leftarrow \mathbf{r}^k + \alpha_k A\,\mathbf{p}^k$
    $\boxed{Solve\ M\,\tilde{\mathbf{r}}^{k+1} = \mathbf{r}^{k+1}}$
    $\beta_k \leftarrow (\tilde{\mathbf{r}}^{k+1}, \mathbf{r}^{k+1})/(\tilde{\mathbf{r}}^k, \mathbf{r}^k)$
    $\mathbf{p}^{k+1} \leftarrow \tilde{\mathbf{r}}^{k+1} + \beta_k \mathbf{p}^k$
    $k \leftarrow k + 1$
End
```

The expression $(\mathbf{x}, \mathbf{y})$ defines an inner (dot) product, i.e., $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\,\mathbf{y}$. Bold face lowercase letters represent vectors, upper case letters represent matrices and Greek letters represent scalars.

The preconditioning steps are indicated. Matrix M is *a preconditioner matrix* that approximates matrix A. Matrix M is chosen such that the condition number of $M^{-1}A$ is improved relative to that of A. The simplest choice for matrix M is a diagonal matrix whose entries are the diagonal elements of A, but this choice does not improve the condition number of $M^{-1}A$ enough. A widely used preconditioner is described in appendix A.

The PCG method is dominated by matrix vector products. The number of iterations required for obtaining a solution to some tolerance is a function of both the condition number [17, 21] and of the quality of the preconditioner.

Iterative methods have been used by others to solve linear equations associated with power system analysis in serial environments. Galiana et al. [11] consider the application of the conjugate gradient method to power flow analysis. They use the iterative method for security analysis (DC load flow) and the load flow prob-

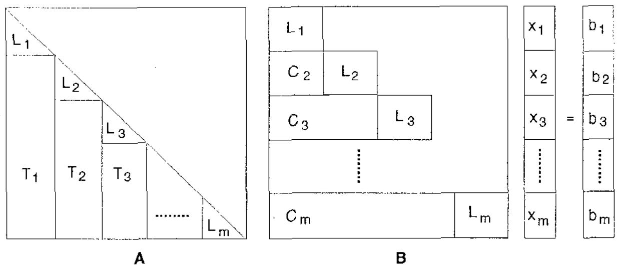

Fig. 1: *Two ways of partitioning a lower triangular matrix for parallel processing.*

lem using fast decoupled load flow (FDLF) method [22]. The preconditioned conjugate gradient method has also been used by Decker at al. in power system dynamic simulation [9] and by Mori et al. in small signal stability analysis [16]. Pai et al. used another iterative method (GMRES), which is applicable to both symmetric and unsymmetric matrices, for the dynamic simulation of power systems [19].

Iterative methods are highly parallel within each iteration. Unfortunately, this advantage is degraded by the need to use preconditioners. Preconditioners such as incomplete LU (ILU) (appendix A) are not well suited to parallel processing due to dependencies in the forward and backward (F/B) substitutions. One way to do F/B substitution in parallel is to use either row or column blocking, as illustrated in Figure 1. Blocking results in the need to solve smaller triangular problems. For repeated solutions, it is often desirable to invert the individual block matrices L_i explicitly. The speed gain of blocking in parallel environments is often modest [6].

A second way of doing F/B substitution is to use a method by Abur [1]. The method treats the right hand side vector as a sum of singleton vectors. It performs sparse F/B substitution for each singleton. The method then uses superposition to find the solution. Because n (dimension of A) singleton solutions are required, the efficiency of the method decreases quickly with matrix dimension [1].

A third alternative is to use partitioned inverses to solve the preconditioning equations. Explicit inverse factors of A make good parallel preconditioners. However, because obtaining these preconditioning matrices can itself be an expensive proposition, this paper proposes two "computation-free" variants of the method, where no additional numerical computation is required to obtain either exact or approximate partitioned inverse factors. By computation-free this paper understands that no *additional numeric* computation beyond that required by the underlying method is necessary to construct M. The paper continues as follows. Section 2 describes the proposed preconditioner. Section 3 presents and discusses test results using a shared memory parallel environment. Concluding remarks based on these tests are given in section 4.

2 Computation-Free Preconditioners

The conjugate gradient method is inherently parallel. Its parallelism degrades significantly with the use of ILU preconditioners. This can be prevented by a highly parallel preconditioner, such as an approximate partitioned inverse method [3]. However, the required computation of approximate inverse factors can be expensive. This paper proposes a new preconditioner which is also based on partitioned inverse factors of matrix A, but such that no extra computation is required to obtain the preconditioning matrices. (Of course, application of the preconditioner does require computation).

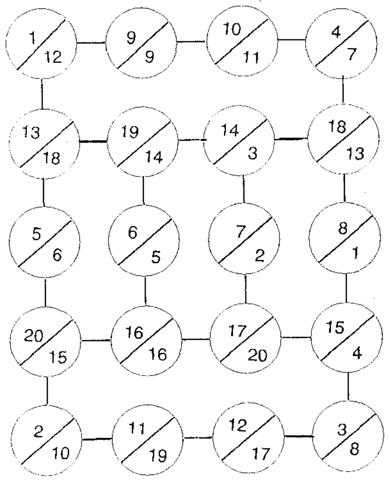

Fig. 2: *The topology of the 20-bus example network. The first numbers show the original node numbering and the second numbers show the numbering according to the* **mlmd** *ordering algorithm [7].*

2.1 Exact Computation Free Partitioned Inverse (CFPI) Preconditioners

This section provides necessary definitions and theoretical background.

Definition 1 *A real square matrix E_i is said to be an* elementary matrix *if it is a unit lower (or upper) triangular matrix with off diagonal non-zero entries only in the i-th column (or row).*

A fundamental fact about elementary matrices is that their inverses are obtained simply by negating the off diagonal non-zero entries.

The partitioned inverse (or W-matrix) method for solving sparse linear equations [4] is now reviewed. A sparse set of linear equations (1) is solved by factoring A into the product of a unit lower triangular matrix L, a diagonal matrix D and a unit upper triangular matrix U, followed by forward substitution, diagonal scaling and back substitution as:

$$L\,\mathbf{y} = \mathbf{b}, \quad D\,\mathbf{z} = \mathbf{y}, \quad U\,\mathbf{x} = \mathbf{z}. \tag{2}$$

Factorization is preceded by an ordering to minimize fills. From here on we assume that matrix A is symmetric and positive definite. Hence the PCG method can be used to solve the set of linear equations (1). If we define $W = L^{-1}$ and compute the inverse explicitly, the forward and back substitution steps in (2) can be replaced with simple matrix-vector products as:

$$\mathbf{y} = W\,\mathbf{b}, \qquad D\,\mathbf{z} = \mathbf{y}, \quad \mathbf{x} = W^T\,\mathbf{z} \tag{3}$$

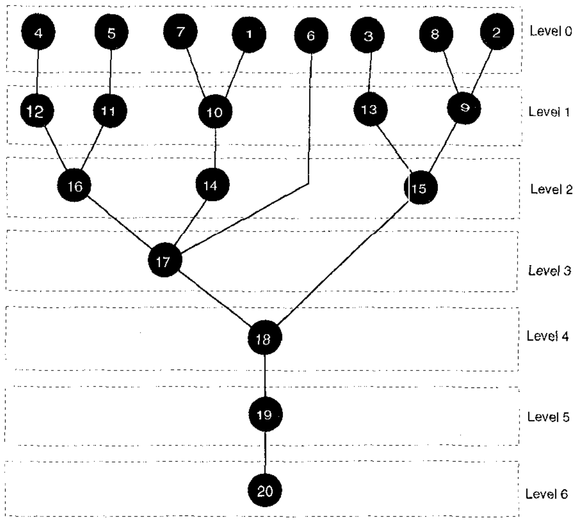

Fig. 3: *The elimination tree of 20-bus network matrix after ordering according to the* **mlmd** *algorithm.*

which are quite amenable to parallel processing. The inverse of L is never computed explicitly. The matrix L can also be expressed as:

$$L = L_1 L_2 \cdots L_n = \prod_{k=1}^{n} L_k. \tag{4}$$

The inverses of these elementary matrices may then be grouped into m groups as follows:

$$W = \prod_{k=m}^{1} \widehat{W_k}, \tag{5}$$

where each $\widehat{W_k}$ is the aggregate of several elementary inverse factors of L. The inverse factors can be aggregated so that the combined sparsity structure of all m inverse factors is the same as the structure of L itself. Furthermore, with suitable ordering and partitioning algorithms, it is usually possible to have $m \ll n$ [4, 5, 20].

How several inverse factors L_i^{-1} combine into one group is the key to the understanding this paper. The 20-bus example network of [23] is shown in Figure 2. Its elimination tree after ordering according to the **mlmd** algorithm [7] is shown in Figure 3. The nodes in the elimination tree indicate column numbers for forward substitution and row numbers for back substitution. The last node, 20, is the root of the tree.

One way to combine several inverse factors so that no-inversion fill occur for the example network using the associated elimination tree can be given as: {1,2,3,4,5,6,7,8}, {9,10,11,12,13}, {14,15,16}, {17}, {18}, {19}, and {20}. That is, all the nodes within each level are included in the same partition. This partitioning guarantees that no inversion fills are produced.

Proposition 1 *The computation of partitioned inverses based on grouping according to elimination tree levels involves nothing but a sign change of the off diagonal non-zero entries.*

Proof 1 *Let the list of nodes $\{1, 2, \cdots, \ell\}$ be in the same level of elimination tree. The inverse of this partition is formed by $L_\ell^{-1} \cdots L_2^{-1} L_1^{-1}$. There is no inversion fill since none of the nodes in the list is the ancestor of another node. Furthermore, the first non-zero off diagonal entry must be in the rows numbered from $\ell + 1$ to n. Otherwise, the nodes would not be in the same level by the rule of tree construction. Now consider the product of $L_j^{-1} L_i^{-1}$ where $1 \le i < j \le \ell$ and the first non-zero in the k-th row, where $j < k \le n$. Let $\widehat{L_j}$ and $\widehat{L_i}$ hold the non-zero off diagonal entries of L_j^{-1} and L_i^{-1} respectively. Then,*

$$L_j^{-1} L_i^{-1} = (I + \widehat{L_j})(I + \widehat{L_i}) = I + \widehat{L_j} + \widehat{L_i} \tag{6}$$

since L_j^{-1} and L_i^{-1} do not have non-zeros until the k-th row (see Figure 4) the product $\widehat{L_j}\,\widehat{L_i}$ is zero. □

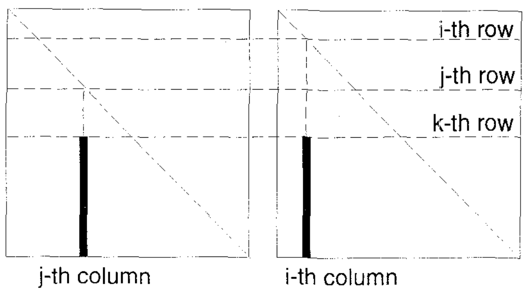

Fig. 4: *The matrix-matrix product of two elementary matrices, where $1 \le i < j < k \le n$. Solid lines show possible non-zero locations in the* i-th *and* j-th *columns.*

Proposition 1 implies that the product of the inverses of the elementary matrices above can be found from:

$$L_\ell^{-1} \cdots L_2^{-1} L_1^{-1} = I - \sum_{k=1}^{\ell} L_k \backslash L_k^{kk}, \tag{7}$$

where $\backslash L_k^{kk}$ means "excluding diagonal" entries. Thus, the product of elementary matrices from the same level of the tree requires no computation. Partitioning by levels is a sufficient condition for both no-inversion fills and no-computation for the inverse factors.

Reference [18] partitions L using an elimination tree until a predetermined level (called break-in-depth) is reached. Then assigns all remaining levels to one partition. Inverse factors corresponding to all partitions except the last one require no computation. Hence, the inverse of a large matrix L can be obtained by an explicit inversion of a much smaller matrix.

Elimination trees are defined for complete factorizations. In the incomplete factorization case, such as ILU, the notion of tree becomes meaningless. The tree becomes a forest if the algorithm in [23] is used. The proper graph is a directed acyclic graph (DAG) if all dependencies are shown in the graph. The notion of level (or depth) of elimination trees can also be used for forest. Figure 5 shows the topology of level 1 ILU (ILU_1) and Figure 6 shows the forest of level 1 ILU (see appendix A for a definition of level i ILU).

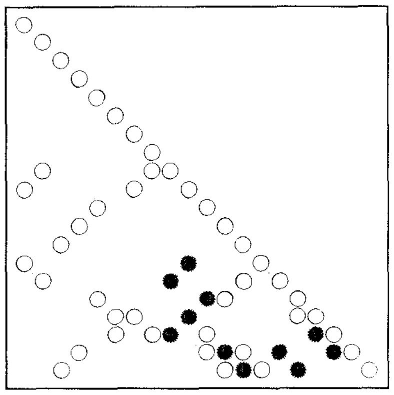

Fig. 5: *The topology of level 1 ILU of 20-bus network matrix after ordering according to the* **mlmd** *algorithm. Solid dots show ILU fills.*

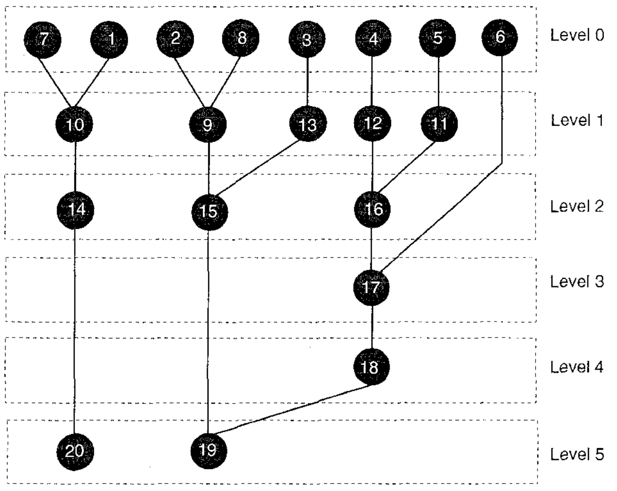

Fig. 6: *The forest graph of level 1 ILU factors (corresponding to the matrix in Figure 5).*

A no inversion-fill computation-free partitioning of the forest in Figure 6 obtained by applying the tree construction algorithm [23] to level 1 ILU factors is : {1,2,3,4,5,6,7,8}, {10,11,12,13}, {14,15,16}, {17}, {18}, and {19,20}. The above partitioning is not unique since node 20 can be included in level 3, level 4 or level 5 as is done here. This should be apparent from Figure 6. An algorithm for CFPI can be given as:

CFPI Algorithm

- *Order the matrix to reduce both the factorization fills and the height of the elimination tree,*
- *Perform level i incomplete LU factorization of A,*
- *Partition the $ILUi$ factors according to levels,*
- *Negate the non-zero off-diagonal entries in $ILUi$.*

The last step can be done during matrix multiplication. The acronym $CFPI_{ILUi}$ is used for a CFPI obtained from a level i ILU.

2.2 Computation Free Approximate Partitioned Inverse (CFAPI) Preconditioners

The exact CFPI algorithm produces a high number of partitions, which implies increased communication in parallel environments [25]. In order to reduce global communication, it is necessary to reduce the number of partitions. One way to do this is to allow some fills in each partition [5]. This, however, requires computation. Because ILU preconditioners work remarkably well as preconditioners, their approximate partitioned inverses are also expected to work well since the numeric values of inversion fills are often quite small. Approximate inverses can be obtained with the following algorithm.

CFAPI Algorithm

- *Order the matrix to reduce both the factorization fills and the height of the elimination tree,*
- *Perform level i incomplete LU factorization of A,*
- *Partition the $ILUi$ factors*
  1. *Either use the partitioning algorithm of [18]*
  2. *Or pretend that ℓ fills are to be allowed in each partition to reduce the number of partitions but, do not add the actual fills*
- *Negate the non-zero off-diagonal entries in $ILUi$.*

No numeric computation is required. The last step can be done during matrix multiplication. The acronym $CFAPI^{\ell}_{ILUi}$ is used for a CFAPI obtained from level i ILU as if ℓ fills were to be allowed in each partition.

3 Experimental Results

In this section test results are reported in two parts. In the first part, the comparison of traditional ILU and CFPI preconditioners is presented. In the second part, the test results for comparison of ILU and CFAPI preconditioners are presented. All B' and B'' matrices are obtained directly from common format data files. The *gain matrices* are obtained from the Weighted Least Squares (WLS) power system state estimation formulation. The statistics for the gain matrices are shown in Table 1.

Table 1: Statistics for the gain matrices ($G = H^T H$).

Bus number	57	68	118	300	414
Dimension (G)	113	135	235	599	827
Rows (H)	113	135	235	599	827
Columns (H)	289	343	634	1518	2020
Non-zeros (H)	1347	1545	3283	7273	9631
Non-zeros (G)	1903	2517	5029	11387	13867

The **mlmd** algorithm [7] is used to order the matrices. This ordering is good both in terms of reducing the factorization fills and lowering the height of the elimination tree. Ordering specifically intended to reduce the height of elimination trees (such as those of [12, 13]) are not tested in this paper. Different orderings may lead to different numbers of PCG iterations [8].

A zero initial guess vector is used for the PCG method. The convergence tolerance is 0.000005. The testing environment is a shared memory machine, *a Sequent Symmetry*, with fourteen 386-based Weitech processors.

Table 2: ILU_i vs. $CFPI_{ILUi}$ iterations (partitions)

	Number of iterations for PCG			
System	ILU_0	ILU_1	$CFPI_{ILU0}$	$CFPI_{ILU1}$
414 B'	57	19	57 (6)	19 (11)
Bus B''	29	11	29 (7)	11 (12)
1390 B'	118	53	118 (22)	53 (25)
Bus B''	56	28	56 (12)	28 (18)
4403 B'	164	83	164 (60)	83 (118)
Bus B''	53	20	53 (24)	20 (36)
8235 B'	223	122	223 (56)	122 (131)
Bus B''	42	20	42 (20)	20 (27)
G57	29	13	29 (36)	13 (39)
G68	32	10	32 (43)	10 (46)
G118	79	14	79 (55)	14 (56)
G300	324	48	333 (66)	48 (67)
G414	116	10	116 (54)	10 (56)

Table 3: ILU_i vs. CFPI cpu time

	Average cpu(second) time for PCG			
System	ILU_0	ILU_1	$CFPI_{ILU0}$	$CFPI_{ILU1}$
414 B'	2.6	1.1	1.5	0.6
Bus B''	1.1	0.6	0.7	0.3
1390 B'	20.6	11.3	11.0	5.8
Bus B''	8.2	5.1	4.2	2.5
4403 B'	106.5	71.5	63.0	48.5
Bus B''	25.8	12.4	12.7	6.1
8235 B'	257.0	183.0	134	106.2
Bus B''	37.8	22.7	17.7	10.1
G57	1.5	1.0	1.5	1.0
G68	2.2	1.1	2.2	1.2
G118	11.0	3.1	9.7	2.9
G300	104.7	22.5	77.6	18.1
G414	46.7	5.5	28.5	3.8

Table 2 compares ILU_i and $CFPI_{ILUi}$ in terms of iterations. Numbers in parenthesis show the number of partitions. Because $CFPI_{ILUi}$ preconditioner is an exact partitioned inverse of ILU_i the number of PCG iterations is the same for both preconditioners. Table 3 shows the average cpu time for 10 runs with 10 processors with the given linear system and the ILU factors of the coefficient matrix. The cpu times for the PCG method with the proposed preconditioner is consistently smaller than that of the corresponding ILU and does not include the time for obtaining the ILU preconditioners. Moreover, there is no computation time for obtaining the proposed preconditioner since it only involves a sign change. The $CFPI_{ILUi}$ preconditioner does not change the condition number of $M^{-1}A$ beyond

Table 4: ILU_i vs. $CFAPI^{\ell}_{ILUi}$ iterations (partitions)

	Number of iterations for PCG			
System	ILU_0	ILU_1	$CFAPI^{\ell}_{ILU0}$	$CFAPI^{\ell}_{ILU1}$
414 B'	57	19	57 (5)	19 (9)
Bus B''	29	11	28 (3)	11 (10)
1390 B'	118	53	149 (12)	91 (16)
Bus B''	56	28	63 (8)	31 (13)
4403 B'	164	83	187 (19)	113 (44)
Bus B''	53	20	59 (11)	22 (16)
8235 B'	223	122	281 (10)	207 (32)
Bus B''	42	20	45 (7)	24 (12)
G57	29	13	32 (17)	21 (20)
G68	32	10	32 (31)	10 (46)
G118	79	14	110 (40)	56 (28)
G300	324	48	479 (41)	140 (39)
G414	116	10	209 (38)	49 (35)

Table 5: ILU_i vs. $CFAPI^{\ell}_{ILUi}$ cpu time

	Average cpu(second) time for PCG			
System	ILU_0	ILU_1	$CFAPI^{\ell}_{ILU0}$	$CFAPI^{\ell}_{ILU1}$
414 B'	2.6	1.1	1.5	0.6
Bus B''	1.1	0.6	0.7	0.3
1390 B'	20.6	11.3	12.8	9.1
Bus B''	8.2	5.1	4.5	2.6
4403 B'	106.5	71.5	58.7	50.8
Bus B''	25.8	12.4	13.2	6.0
8235 B'	257.0	183.0	142.7	143.5
Bus B''	37.8	22.7	18.0	11.4
G57	1.5	1.0	1.1	1.0
G68	2.2	1.1	1.8	1.2
G118	11.0	3.1	11.5	7.2
G300	104.7	22.5	99.0	42.7
G414	46.7	5.5	46.3	13.2

the one that is obtained by ILU_i since $CFPI_{ILUi}$ is just an alternative way of doing F/B substitutions. The $CFAPI_{ILUi}$ preconditioner, however, usually worsens the condition number since it is an approximate inverse of ILU_i. Thus, it tends to increase the number of PCG iterations.

F/B substitution is not completely parallelized. Only the diagonal scaling part is done in parallel. Hence, the cpu time for ILU can be smaller if F/B also implemented in parallel with a very small number of processors. But with a high number of processors the scalability of F/B is worse than that of the CFPI, see Figures 7 and 8.

Tables 4 and 5 compare ILU_i and $CFAPI^{\ell}_{ILUi}$ in terms of both number of PCG iterations and average cpu time respectively. The number of iterations increases with the $CFAPI^{\ell}_{ILUi}$ preconditioners. However, the decrease in the number of partitions somehow balances the total cpu time. The increase in the number of iterations of PCG when applied to *gain matrices* is bigger than that of B' and B'' matrices. This is expected since the gain matrices are ill-conditioned. The ℓ in the CFAPI algorithm is chosen as .1% of the matrix dimension, and $\ell \geq 1$. The actual fills are not added, as explained above.

Our experience indicate that level 0 ILU (ILU_0) preconditioners result in a high number of PCG iterations. Level 1 ILU (ILU_1) preconditioners reduce the number of PCG iterations substantially while producing acceptable factorization fills. Going beyond ILU_1 does not pay of since floating-point operations due to increased fills preclude any gain made by reductions in the number of iterations.

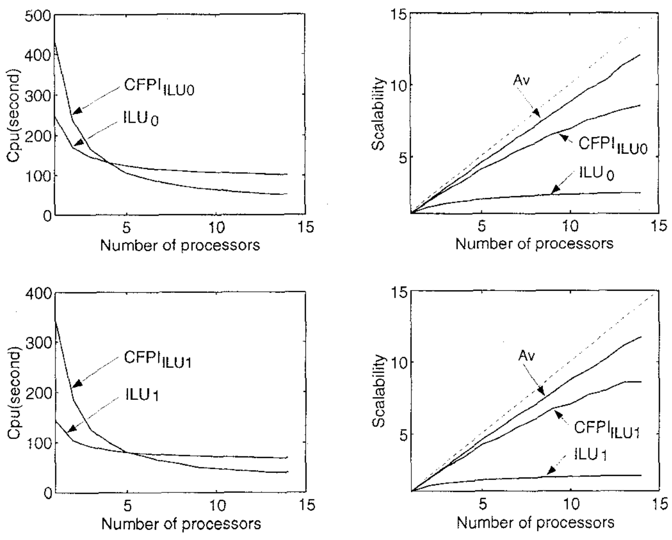

Fig. 7: *Comparison of ILU_i and $CFPI_{ILUi}$ preconditioners used by PCG when applied to B' of 4403 bus. Av refers to sparse matrix-vector product. Dotted line represents ideal scalability.*

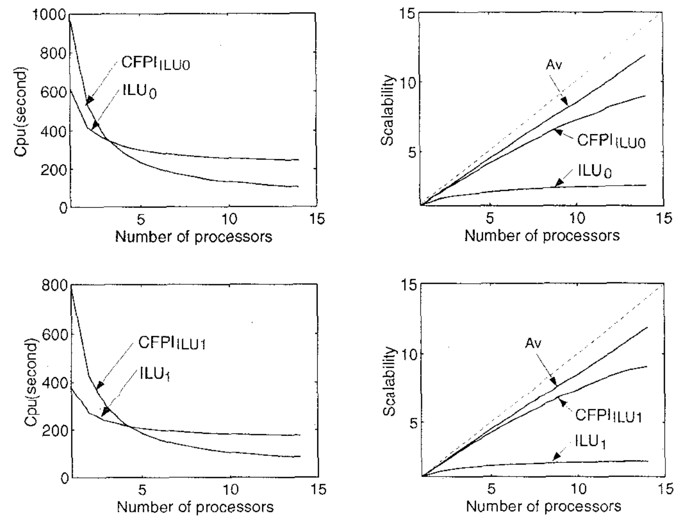

Fig. 8: *Comparison of ILU_i and $CFPI_{ILUi}$ preconditioners used by PCG when applied to B' of 8235 bus. Av refers to sparse matrix-vector product. Dotted line represents ideal scalability.*

4 Conclusions

The use of computation-free (approximate) partitioned inverses as preconditioners for the CG method in a parallel environment can reduce computing time typically by about 50%, even though the number of iterations increases.

Orderings that reduce the height of the elimination tree are essential to reduce the number of partitions (hence the number of global communications steps in distributed-memory parallel environments).

Acknowledgements

The authors acknowledge support from NSF grant ECS-9216308 and thank Dr. Harmohan Singh for his assistance with data cases for the gain matrices.

REFERENCES

[1] A. Abur. A parallel scheme for the forward/backward substitutions in solving sparse linear equations. *IEEE Transactions on Power Systems*, PWRS-3(4):1471–1478, November 1988.

[2] F. L. Alvarado. Computational complexity in power systems. *IEEE Transactions on Power Apparatus and Systems*, 95(4):1028–1037, July/August 1976.

[3] F. L. Alvarado and H. Dağ. Incomplete Partitioned Inverse Preconditioners. Submitted to Parallel Computing, June 1994.

[4] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific and Statistical Computing*, pages 446–460, March 1993.

[5] F. L. Alvarado, D. C. Yu, and R. Betancourt. Partitioned sparse A^{-1} methods. *IEEE Transactions on Power Systems*, 5(2):452–459, May 1990.

[6] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1:73–95, 1989.

[7] R. Betancourt. An efficient heuristic ordering algorithm for partial matrix refactorization. *IEEE Transactions on Power Systems*, 3(3):1181–1187, August 1988.

[8] H. Dağ and F. L. Alvarado. The effect of ordering on the preconditioned conjugate gradient method for power system applications. In *Proceedings of the North American Power Symposium*, Manhattan, KS, September 1994.

[9] I. C. Decker, D. M. Falcão, and E. Kaszkurewicz. Parallel implementation of a power system dynamic simulation methodology using the conjugate gradient method. In *Power Industry Computer Applications Conference (PICA)*, pages 245–252, May 1991.

[10] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.

[11] F. D. Galiana, H. Javidi, and S. McFee. On the applications of a pre-conditioned conjugate gradient algorithm to power network analysis. In *Power Industry Computer Applications Conference (PICA)*, pages 404–410, April 1993.

[12] A. Gómez and L. G. Franquelo. An efficient ordering algorithm to improve sparse vector methods.

IEEE Transactions on Power Systems, 3(4):1538–1544, November 1988.

[13] F. Manne. *Load balancing in Parallel sparse matrix computations.* PhD thesis, Department of Informatics, University of Bergen, Bergen, Norway, 1993.

[14] T. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34:473–497, 1980.

[15] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148–162, January 1977.

[16] H. Mori, J. Kanno, and S. Tsuzuki. A sparsity-oriented technique for power system small signal stability analysis with a precondition conjugate residual method. *IEEE Transactions on Power Systems*, 8(3):1150–1158, August 1993.

[17] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, 1988.

[18] A. Padilha and A. Morelato. A W-matrix methodology for solving sparse network equations on multiprocessor computers. *IEEE Transactions on Power Systems*, 7(3):1023–1030, 1992.

[19] M. A. Pai, P. W. Sauer, and A. Y. Kulkarni. A preconditioned iterative solver for dynamic simulation of power systems. In *International Symposium on Circuits and Systems*, pages 1279–1281, Seattle,WA, April/May 1995.

[20] A. Pothen and F. L. Alvarado. A fast reordering algorithm for parallel sparse triangular solution. *SIAM Journal on Scientific and Statistical Computing*, March 1992.

[21] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

[22] B. Stott and O. Alsaç. Fast decoupled load flow. *IEEE Transactions on Power Apparatus and Systems*, PAS-93(3):859–869, May/June 1974.

[23] W. F. Tinney, V. Brandwajn, and S. M. Chan. Sparse vector methods. *IEEE Transactions on Power Apparatus and Systems*, PAS-104(2):295–301, February 1985.

[24] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, November 1967.

[25] O. Yasar, F. L. Alvarado, and H. Dağ. Partitioned inverse sparse matrix solutions on the Intel iPSC/860 hypercube. In *Proceedings of The Intel Supercomputer User Group Conference*, Dallas, TX, October 4–7 1992.

Appendix A Conventional ILU preconditioner

A popular preconditioner is based on the approximate (or incomplete) $L\,L^T$ or $L\,D\,L^T$ factorization of the matrix A. Due to fill-in, L can be considerably less sparse than A. Instead of using L, an approximation to L (denoted by $\tilde{L}$) can be used:

$$A = \tilde{L}\tilde{L}^T + E \tag{8}$$

where $E \neq 0$ is an error matrix, and $\tilde{L}$ is a lower triangular matrix, which is more sparse than L. This matrix implicitly defines $M = \tilde{L}\tilde{L}^T$ and it is called an *incomplete factorization* of A [17].

One of several possible ways of constructing $\tilde{L}$ 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.

A more efficient way to obtain ILU factors is to perform an ordinary factorization of a matrix, but preclude the creation of any new non-zeros. This simple departure from ordinary LU factorization is the "level 0" ILU algorithm [14, 15].

The numerical performance of an ILU preconditioner can usually be improved upon if some fill-in 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 $\tilde{L}$ 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 E [10].

Biographies

Hasan Dağ (S'90) obtained the BS(Hon) degree in electrical engineering from the Technical University of Istanbul, Turkey, the MS degree from the University of Wisconsin-Madison. He is currently a Ph.D. student at the University of Wisconsin-Madison in the Department of Electrical and Computer Engineering. His main interests are in the application of iterative methods to large sparse power systems problems and parallel solution of these problems.

Fernando L. Alvarado (F'93) obtained the BS degree from the National University of Engineering in Lima, Peru, the MS degree from Clarkson University, and a Ph.D. from the University of Michigan. He is currently a Professor at the University of Wisconsin-Madison in the Department of Electrical and Computer Engineering. His main interests are in computer applications to power systems and sparse matrix problems.