Parallel Choleski Factorization of Block-Diagonal-Bordered Sparse Matrices

D. P. Koester, S. Ranka, and G. C. Fox
School of Computer and Information Science and
The Northeast Parallel Architectures Center (NPAC)
Syracuse University
Syracuse, NY 13244-4100
dpk@npac.syr.edu, ranka@top.cis.syr.edu, gcf@npac.syr.edu

NPAC Technical Report — SCCS 604

22 January 1994

Abstract

This paper presents research into parallel block-diagonal-bordered sparse Choleski factorization algorithms developed with special consideration to irregular sparse matrices originating in the electrical power systems community. Direct block-diagonal-bordered sparse Choleski algorithms exhibit distinct advantages when compared to general direct parallel sparse Choleski algorithms. Task assignments for numerical factorization on distributed-memory multi-processors depend only on the assignment of data to blocks, and data communications are significantly reduced with uniform and structured communications. Choleski factorization algorithms for block-diagonal-bordered form matrices require a specialized ordering step coupled to an explicit load balancing step in order to generate this matrix form and to uniformly distribute the computational workload for an irregular matrix throughout a distributed-memory multi-processor. Matrix orderings are performed using a diakoptic technique based on node-tearing-nodal analysis, which permits load balancing on either the number of calculations in the factorization step or the number of calculations in the forward reduction and backward substitution phase. Empirical performance measurements for real power system load-flow matrices are presented for an implementation of a parallel block-diagonal-bordered Choleski algorithm run on a distributed memory Thinking Machines CM-5 multi-processor.

1 Introduction

Solving sparse linear systems practically dominates scientific computing, but the performance of direct sparse matrix solvers have tended to trail behind their dense matrix counterparts [14]. Parallel sparse matrix solver performance generally is less than similar dense matrix solvers even though there is more inherent parallelism in sparse matrix algorithms than dense matrix algorithms. Parallel sparse linear solvers can simultaneously factor entire groups of mutually independent contiguous blocks of columns or rows without communications; meanwhile, dense linear solvers can only update blocks of contiguous columns or rows each pipelined communication cycle. The limited success with efficient sparse matrix solvers is not surprising, because general sparse linear solvers require more complicated data structures and algorithms that must contend with irregular memory reference patterns. The irregular nature of these problems has aggravated the task of implementing scalable sparse matrix solvers on vector or parallel architectures: efficient scalable algorithms for these classes of machines require regularity in available data vector lengths and in interprocessor communications patterns [3]. Nevertheless, when scalability of sparse linear solvers is examined using real irregular sparse matrices, the available parallelism in the sparse matrix can be as much the reason for poor parallel efficiency as the parallel algorithm or implementation [15].

In this paper we examine the applicability of parallel direct block-diagonal-bordered sparse solvers for real power system load-flow applications that require the solution of symmetric positive definite sparse matrices. Parallel block-diagonal-bordered sparse linear solvers offer the potential for regularity often absent from other parallel sparse solvers. Load flow analysis entails the solution of non-linear systems of simultaneous equations, which are performed by repeatedly solving sparse linear equations. For power system load-flow applications, however, the limited size of the matrices and load imbalance due to limited parallelism in the matrix structure significantly limits the number of processors that can be used efficiently for a single parallel Choleski solver. This fact will be evident in the empirical data collected on the CM-5. Our research into specialized matrix ordering techniques has shown that it is possible to order actual power system matrices readily into block-diagonal-bordered form, but load imbalance becomes excessive beyond four processors, limiting potential parallelism for a single parallel Choleski solver within an application. Nevertheless, other dimensions exist in electrical power system applications that can be exploited to efficiently make use of large-scale multi-processors. We believe that this research also has utility for other irregular sparse matrix applications where the data is hierarchical. Other sources of hierarchical matrices exist, for example, electrical circuits, that have the potential for larger numbers of equations than power system matrices.

In this paper, we examine the performance of a block-diagonal-bordered Choleski solver to be incorporated within electrical power system applications. Because we are considering software to be embedded within a more extensive application, we examine efficient parallel forward reduction and backward substitution algorithms in addition to parallel Choleski factorization algorithms. Due to the reduced amount of calculations in the triangular solution phases of solving a system of symmetric positive definite equations, these algorithms are often ignored when parallel Choleski algorithms are presented in the literature. We not only include a discussion of these algorithms, we also include an analysis of load balancing as a function of either solution phase: sparse block-diagonal-bordered Choleski factorization and forward reduction/backward substitution. Interprocessor communications costs would be too high to redistribute the data from an optimal load balance data/processor assignment for parallel Choleski factorization to an optimal load balance data/processor assignment for parallel forward reduction and backward substitution, so performance is examined for both factorization and triangular solutions with each load balance data/processor assignment.

Block-diagonal-bordered sparse matrix algorithms require modifications to the normal preprocessing phase described in numerous papers on parallel Choleski factorization [9, 10, 11, 14, 22, 23, 24, 25, 26, 30]. Each of the numerous papers referenced above use the paradigm to *order* the sparse matrix and then perform *symbolic factorization* in order to determine the locations of all fillin values so that static data structures can be utilized for maximum efficiency when performing numerical factorization. We modify this commonly used sparse matrix preprocessing phase to include an explicit *load balancing* step coupled to the *ordering* step so that the workload is uniformly distributed throughout a distributed-memory multi-processor and parallel algorithms make efficient use of the computational resources.

This paper is organized as follows. In section 2, we introduce the electrical power system applications that are the basis for this work. In section 3, we briefly review Choleski factorization and forward reduction/backward substitution and present a review of the literature concerning general parallel Choleski factorization algorithms. This is followed by a theoretical derivation of the available parallelism in both the Choleski factorization and forward reduction/backward substitution phases when solving block-diagonal-bordered form sparse matrices. Paramount to exploiting the advantages of this parallel linear solver is ordering the irregular sparse power system matrices into this form in a manner that balances the workload among multi-processors. In section 5, we describe the three-step preprocessing phase used to generate matrix ordering for block-diagonal-bordered matrices with uniformly distributed processing load. In this section, we present pseudo-factorization and we review minimum degree ordering and pigeon-hole load balancing algorithms. We present the node-tearing algorithm developed to order matrices into block-diagonal-bordered

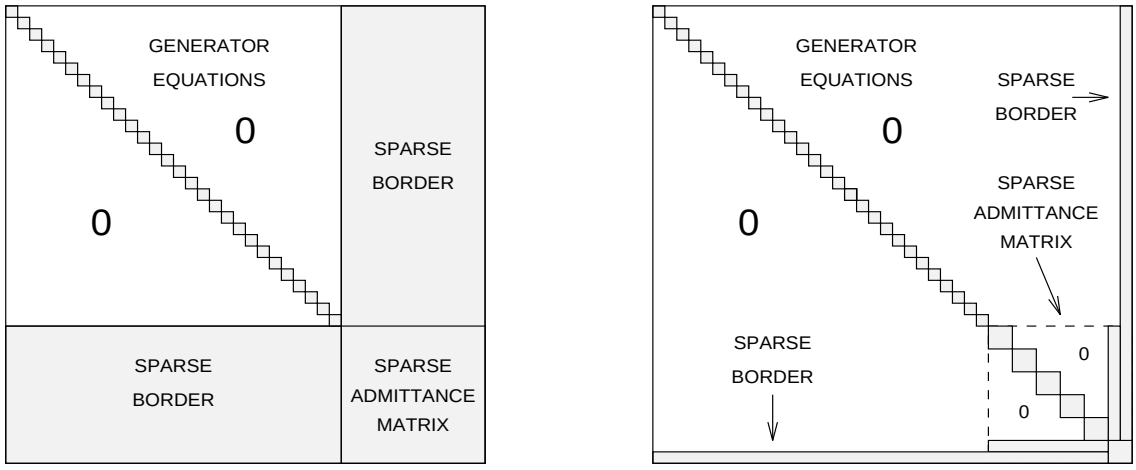
form in section 6. In section 7, we describe our block-diagonal-bordered sparse Choleski algorithm that has been implemented on the CM-5. Analysis of the performance of these ordering techniques for actual power system load flow matrices from the Boeing-Harwell series are presented in section 8. Lastly, we present our conclusions concerning block-diagonal-bordered Choleski solvers for electrical power system applications in section 9.

2 Power System Applications

The underlying impetuous for our research is to improve the performance of electrical power system applications to provide real-time power system control and real-time support for proactive decision making. Our research has focused on load-flow and transient stability applications [2, 29]. Sparse linear solvers are employed in both applications and linear solvers account for the majority of floating point operations encountered. Scalability, or the ability to apply more processors to larger problems, is desired when developing multiprocessor implementations because load-flow and transient stability applications have the potential to be utilized across different sized geographical areas, from single electrical power utilities to regional power authorities.

Load-flow analysis examines steady-state equations based on the positive definite network admittance matrix that represents the power system distribution network. Load-flow analysis is used for identifying potential network problems in contingency analyses, for examining steady-state operations in network planning and optimization, and also for determining initial system state in transient stability calculations [29]. Load flow analysis entails the solution of non-linear systems of simultaneous equations, which are performed by repeatedly solving sparse linear equations. Load flow is calculated using the network admittance matrices, which are symmetric positive definite and have sparsity defined by the power system distribution network. The size of these matrices is limited because individual power systems generally use networks with less than 2,000 sparse complex equations in their operations centers, while regional power authority operations centers would also be limited to sparse load-flow matrices with less than 10,000 sparse complex equations. Power systems planning studies often incorporate larger networks as lower voltage distribution lines are included in these studies. Sparse matrices employed in planning studies can have from 10,000 to 50,000 sparse equations. This paper presents data for power system networks of 1,723, 5,300, and 9,430 nodes. The last network is from a power system planning study.

Transient stability analysis is a detailed simulation of the power system, that models the dynamic behavior of the electrical distribution networks, electrical loads, and the electromechanical equations of motion of the interconnected generators [2]. Transient stability analysis can be used to perform selective detailed analyses of generator commitment stabil-



(a) Original Matrix Configuration

(b) After Ordering the Admittance Matrix into Block-Diagonal-Bordered Form

Figure 1: Ordering the Admittance Sub-Matrix in the Transient Stability Differential-Algebraic Equations

ity, and to support crisis decision-making during network recovery. The transient stability problem is modeled by differential algebraic equations (DAEs) with differential equations representing the generators and non-linear algebraic equations representing the power system network that interconnects the generators. The DAEs are in natural non-symmetric block-diagonal-bordered form, with diagonal blocks of generator equations coupled by the power system distribution network. In this representation, there are as many coupling equations as the entire sparse admittance matrix. However, it it possible to order the admittance matrix to block-diagonal-bordered form to order to increase available parallelism. This is illustrated in figure 1. The size of the sparse matrices representing the DAEs have as many as 10,000 complex equations for an individual power system, while regional power authorities could have as many as 50,000 sparse complex equations in the matrix formed from the DAEs.

The parallel block-diagonal-bordered Choleski algorithm, presented in this paper, addresses the most difficult of these application to implement on multi-processors. Load-flow has the smallest matrices and the fewest calculations due to symmetry and lack of requirements for pivoting to ensure numerical stability. Load-flow calculations are included in decoupled solutions to transient stability differential-algebraic equations. Instead of the common practice of decoupling the generator and network calculations in a transient stability simulation, we will examine using more powerful differential-algebraic equation solvers for transient stability analysis that do not decouple the generator and network equations.

The differential-algebraic equations will offer more potential for good load balancing and offer substantially more calculations because

- the matrices are larger,
- a large portion of these matrices are non-symmetric and require calculations in both the upper and lower triangular portions of the diagonal blocks,
- pivoting will be required in the diagonal blocks containing the generator equations to ensure numerical stability.

3 Choleski Factorization

We are considering the direct solution of the linear system

$$Ax = b, \tag{1}$$

where A is an $N \times N$ sparse matrix. The sparse matrix A can be numerically factored into two separate triangular matrices, one sparse matrix being lower triangular, L, and the other sparse matrix being upper triangular, U:

$$Ax = LUx = b, \tag{2}$$

A lower triangular matrix, L, has all zeros above the diagonal and an upper triangular matrix, U, has all zeros below the diagonal [4].

If the matrix A is an $N \times N$ symmetric positive definite sparse matrix, then a special form of LU factorization can be used that exploits the symmetry and inherently numerical stable characteristics of this matrix form [4]. Our analysis of the available parallelism in block-diagonal-bordered Choleski factorization, presented in section 4, is an extension of the analysis of available parallelism in block-diagonal-bordered LU factorization. Additional discussions on the state of the literature for Choleski factorization are presented below.

For a brief review, the symmetric positive definite sparse matrix A can be numerically factored into a single lower triangular matrix L:

$$Ax = LL^T x = b, \tag{3}$$

A lower triangular matrix, L, has all zeros above the diagonal. Equation 3 is solved by setting $L^T x = y$, and substituting y for $L^T x$. The numerical solution for $Ly = b$ is found by forward reduction, and the numerical solution for x is calculated by backward substitution in the equation $L^T x = y$. Triangular linear systems can be readily solved numerically by solving for the first value in the triangular linear system and substituting that value into subsequent equations.

for $k = 1$ **to** N /* for all elements along the diagonal */
    for each $i \in [k, N]$
        for each $j \in [1, k-1]$ such that $A_{i,j} \neq 0$ **and** $A_{j,k} \neq 0$
            $A_{i,k} \leftarrow A_{i,k} - (A_{i,j} * A_{j,k})$
        endfor
    endfor
    $A_{k,k} \leftarrow \sqrt{A_{k,k}}$
    for each $i \in [k+1, N]$
        $A_{k,j} \leftarrow (A_{k,j}/A_{k,k})$
    endfor
endfor

Figure 2: Sparse Choleski Factorization

Sparse Choleski factorization can mirror any dense Choleski factorization algorithm, although generally a sparse matrix algorithm has only one explicit **for** loop, which can be for any single index in the dense case. The remaining indices are examined only for non-zero values in the original matrix or for non-zero values that will occur from fillin in the matrix. Sparse matrix fillin occurs when a value that formally was zero becomes non-zero in the process of factoring the matrix. Fillin can be controlled in sparse Choleski factorization by *ordering* the matrix before performing the factorization [4].

As we continue the review of Choleski factorization, we present a general sequential sparse factorization algorithm based upon the column Choleski factorization algorithm [14], which is similar to the factorization algorithms commonly attributed to Crout and Doolittle. This sequential sparse factorization algorithm is presented in figure 2. In addition, we present general sequential sparse forward reduction and backward substitution algorithms in figures 3 and 4 respectively. In the backward substitution algorithm, the calculations are performed by implicitly transposing L.

3.1 Ordering Sparse Matrices

Symmetric sparse matrices can be represented by graphs with elements in equations corresponding to undirected edges in the graph [4, 14]. Ordering a symmetric sparse matrix is actually little more than changing the labels associated with nodes in an undirected graph, however, this simple task can drastically effect the amount of calculations involved when factoring a sparse matrix. For symmetric positive definite matrices, there is much latitude in the order to perform the calculations, because there is no requirement for pivoting for

for $k = 1$ **to** N /* for all elements along the diagonal */
    $y_k \leftarrow (b_k / L_{k,k})$
    for each $i \in [k+1, N]$ such that $L_{i,k} \neq 0$
        $L_{i,k} \leftarrow L_{i,k} - (y_k * L_{i,k})$
    endfor
endfor

Figure 3: Sparse Forward Reduction

for $k = N$ **to** 1 **by** -1 /* for all elements along the diagonal */
    $x_k \leftarrow (y_k / L_{k,k})$
    for each $j \in [1, k-1]$ such that $L_{j,k} \neq 0$
        $L_{j,k} \leftarrow L_{j,k} - (x_k * A_{j,k})$
    endfor
endfor

Figure 4: Sparse Backward Substitution

numerical stability and the only effect of modifying the order of calculations might result from changes in round-off errors [14]. There is a graph-theoretical interpretation for fillin; factoring a node is equivalent to removing the node from the graph, however, any path through the factored node to adjacent edges must remain and must now be explicitly listed. This phenomenon is illustrated in figure 5 for a segment of a graph. In this example, the node with the least number of edges is selected for factoring, and two of three possible new edges are created. Only two new edges are created because there is an existing edge already connecting a pair of nodes. Fillin causes the number of edges in the remaining nodes to increase, often drastically increasing the number of calculations. The amount of fillin generated when any node is factored is bounded by the binomial coefficient of *the number of edges at a node choose 2* or

$$f_k \leq \left(\begin{array}{c} \nu_k \\ 2 \end{array} \right) = \frac{\nu_k!}{2 \times (\nu_k - 2)!} = \frac{(\nu_k \times (\nu_k - 1))}{2}, \tag{4}$$

where:

f_k is the number of fillin when factoring node k.

ν_k is the number of edges at node k.

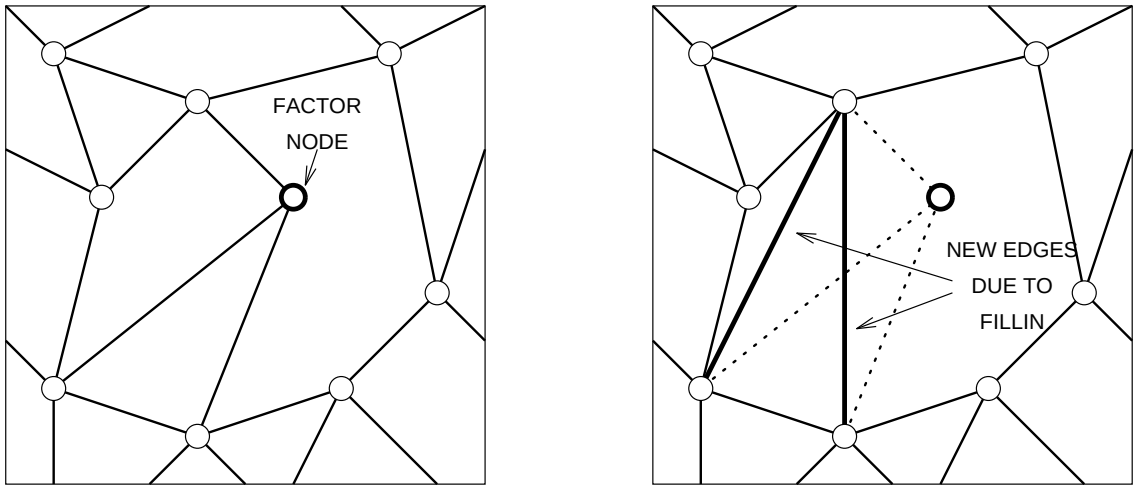


Figure 5: Graph Theoretical Explanation of Fillin

There are several notable techniques to minimize fillin, with one of the commonly used techniques being minimum-degree ordering. Minimum degree ordering is used in conjunction with the node-tearing-based ordering technique to generate block-diagonal-bordered form sparse matrices. Additional detail on minimum degree-based sparse matrix ordering is presented in appendix A.

Modifying the ordering of a sparse matrix is simple to perform using a permutation matrix P of all zeros and ones that simply generates elementary row and column exchanges. Applying the permutation matrix P to the original linear system in equation 1 yields the linear system

$$(PAP^T)(Px) = (Pb), \tag{5}$$

that is solved by factoring PAP^T into the Choleski factor $\bar{L}$ in $\bar{L}\bar{L}^T$ and then performing forward reduction, backward substitution, and undoing the permutation on the x vector. These steps would require the solutions of:

$$\bar{L} = Pb, \bar{L}^T z = y, x = P^T z. \tag{6}$$

As long as the symmetric matrix A is ordered with the permutation matrix P to PAP^T, the resultant matrix after ordering remains symmetric positive definite.

3.2 A Survey of the Literature

Significant research effort has been expended to examine parallel matrix solvers — for both dense and sparse matrices. Numerous papers have documented research on parallel dense

matrix solvers [3, 27, 28], and these articles illustrate that good efficiency is possible when solving dense matrices on multi-processor computers. The calculation time complexity of dense matrix LU factorization is $O(N^3)$, and there are sufficient, regular calculations for good parallel algorithm performance. Some implementations are better than others [27, 28], nevertheless, performance is deterministic for:

- the algorithm,
- the multi-processor architecture,
- the number of processors,
- the matrix size.

Direct sparse matrix solvers, on the other hand, have computational complexity significantly less than $O(N^2)$, and actual power system sparse matrices used in this work have order of complexity ranging from $O(N^{1.3})$ to $O(N^{1.5})$. These orders of complexity are consistent with matrices from circuit analysis applications that have complexities ranging from $O(N^{1.2})$ to $O(N^{1.5})$ [18]. With significantly less calculations than dense direct solvers, and lacking uniform, organized communications patterns, direct parallel sparse matrix solvers often require detailed knowledge of the application to permit efficient implementations.

The bulk of recent research into parallel direct sparse matrix techniques has centered around symmetric positive definite matrices, and implementations of Choleski factorization. A significant number of papers concerning parallel Choleski factorization for symmetric positive definite matrices have been published recently [9, 10, 11, 14]. These papers have thoroughly examined many aspects of the parallel direct sparse matrix solver implementations, symbolic factorization, and appropriate data structures. Techniques to improve interprocessor communications using block partitioning methods have been examined in [23, 24, 25, 26]. Techniques for sparse Choleski factorization have even been developed for single-instruction-multiple-data (SIMD) computers like the Thinking Machines CM-1 and the MasPar MPP [16]. This discussion is by no means an exhaustive literature survey, although it does represent a significant portion of the direct sparse matrix research performed for vector and multi-processor computers.

References [9, 10, 11, 14, 23, 24, 25, 26] have kept with a general two step preprocessing paradigm for parallel sparse Choleski factorization:

1. order the matrix to minimize fillin,
2. symbolic factorization to identify fillin and set up static data structures,

In this paper, we break from this two step pre-processing paradigm and introduce a new three-step preprocessing phase that includes ordering, pseudo-factorization, and explicit load balancing. Our three-step preprocessing phase is described in section 5.

4 Available Parallelism in Block-Diagonal-Bordered Form Matrices

The most significant aspect of parallel sparse Choleski factorization is that the sparsity structure can be exploited to offer more parallelism than is available with dense matrix solvers. Parallelism in dense matrix factorization is achieved by distributing the data in a manner that the calculations in one of the **for** loops can be performed in parallel. Sparse factorization algorithms have inadequate calculations in any row or column for efficient parallelism; however, sparse matrices offer additional parallelism as a result of the nature of the data and the precedence rules governing the order of calculations. Instead of just parallelizing a single **for** loop as in parallel dense matrix factorization, entire independent portions of a sparse matrix can be factored in parallel — especially when the sparse matrix has been ordered into block-diagonal-bordered form. The description of parallelism presented here has some things in common with elimination graphs and super-nodes described in [14]; however, this work adds a two-dimensional flavor to these concepts. Provided that the matrix can be ordered into block-diagonal-bordered form, then the parallel sparse Choleski algorithm can reap additional benefits, such as the elimination of task graphs for distributed-memory multi-processor implementations. Minimizing or eliminating task graphs is significant because the task graph can contain as much information as the representation of the sparse matrix for more conventional parallel sparse Choleski solvers [8].

There are several distinct ways to examine the available parallelism in block-diagonal-bordered form matrices. The first way to consider available parallelism in a block-diagonal-bordered sparse matrix is to consider the graph of the matrix. Figure 6 represents the form of a graph with four mutually independent sub-matrices (subgraphs) interconnected by shared coupling equations. No graph node in a subgraph has an interconnection to another subgraph except through the coupling equations. It should be intuitive that data in columns associated with nodes in subgraphs can be factored independently up to the point where the coupling equations are factored. While this graph offers intuition into the available parallelism in block-diagonal-bordered sparse matrices, it is possible to examine the theoretical mathematics of matrix partitioning to clearly identify available parallelism in this sparse matrix form. By partitioning the block-diagonal-bordered matrix into:

- a block-diagonal matrix

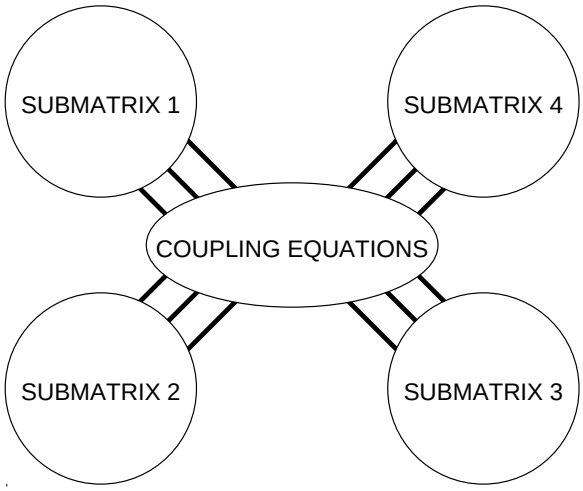


Figure 6: Graph with Four Independent Sub-Matrices

- an upper border
- a lower border
- a last block

and calculating the Shur complement [4], it is possible to identify available parallelism by proving a theorem that states the Choleski factors of a block-diagonal-bordered matrix are also in block-diagonal-bordered form. A supporting lemma stating that the Choleski factors of a block-diagonal matrix are also block-diagonal form is required to complete the proof of the theorem. A similar version of this derivation can be used to identify the parallelism in general LU factorization.

Define a partition of $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ as

$$\mathbf{A} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{2,1}^T \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathcal{L}_{2,1} & \mathcal{L}_{2,2} \end{pmatrix} \begin{pmatrix} \mathcal{L}_{1,1}^T & \mathcal{L}_{2,1}^T \\ \mathbf{0} & \mathcal{L}_{2,2}^T \end{pmatrix} = \mathbf{L}\mathbf{L}^T \tag{7}$$

where:

$\mathcal{A}_{1,1}$ and $\mathcal{L}_{1,1}$ are of size $n_1 \times n_1$
$\mathcal{A}_{2,1}$ and $\mathcal{L}_{2,1}$ are of size $n_2 \times n_1$
$\mathcal{A}_{2,1}^T$ and $\mathcal{L}_{2,1}^T$ are of size $n_1 \times n_2$
$\mathcal{A}_{2,2}$ and $\mathcal{L}_{2,2}$ are of size $n_2 \times n_2$.

The Shur complement of the partitioned matrices in equation 7 can be calculated by simply

performing the matrix multiplication on the $\mathbf{LL}^T$ partitions which yields:

$$\mathbf{A} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{2,1}^T \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1}\mathcal{L}_{1,1}^T & \mathcal{L}_{1,1}\mathcal{L}_{2,1}^T \\ \mathcal{L}_{2,1}\mathcal{L}_{1,1}^T & \mathcal{L}_{2,1}\mathcal{L}_{2,1}^T + \mathcal{L}_{2,2}\mathcal{L}_{2,2}^T \end{pmatrix} \tag{8}$$

By equating blocks in equation 8, we can easily identify how to solve for the partitions:

$$\begin{aligned} \mathcal{A}_{1,1} = \mathcal{L}_{1,1}\mathcal{L}_{1,1}^T \quad &\Rightarrow \quad \mathcal{L}_{1,1}\mathcal{L}_{1,1}^T = \mathcal{A}_{1,1} \\ \mathcal{A}_{2,1}^T = \mathcal{L}_{1,1}\mathcal{L}_{2,1}^T \quad &\Rightarrow \quad \mathcal{L}^{\mathcal{T}}{}_{2,1} = \mathcal{L}_{1,1}^{-1}\mathcal{A}_{2,1}^T \\ \mathcal{A}_{2,1} = \mathcal{L}_{2,1}\mathcal{L}_{1,1}^T \quad &\Rightarrow \quad \mathcal{L}_{2,1} = \mathcal{A}_{2,1}(\mathcal{L}_{1,1}^T)^{-1} \\ \mathcal{A}_{2,2} = \mathcal{L}_{2,1}\mathcal{L}_{2,1}^T + \mathcal{L}_{2,2}\mathcal{L}_{2,2}^T \quad &\Rightarrow \quad \mathcal{L}_{2,2}\mathcal{L}_{2,2}^T = \mathcal{A}_{2,2} - \mathcal{L}_{2,1}\mathcal{L}_{2,1}^T \end{aligned} \tag{9}$$

Before we can proceed and prove the theorem that the $\mathbf{LL}^T$ factors of a block-diagonal-bordered (BDB) symmetric sparse matrix are also in block-diagonal-bordered form, we must define additional matrix partitions in the desired form and prove a Lemma that the $\mathbf{LL}^T$ factors of a block-diagonal (BD) matrix are also in block-diagonal form. At this point, we must define additional partitions of $\mathbf{A}$ that represent the block-diagonal-bordered nature of the original $\mathbf{A}$ matrix:

$$\mathbf{A}_{BDB} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{2,1}^T \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & & \mathbf{0} & & A_{m,1}^T \\ & A_{2,2} & & & A_{m,2}^T \\ \mathbf{0} & & \ddots & & \vdots \\ & & & A_{m-1,m-1} & A_{m,m-1}^T \\ A_{m,1} & A_{m,2} & \cdots & A_{m,m-1} & A_{m,m} \end{pmatrix} \tag{10}$$

$$\mathbf{L}_{BDB} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathcal{L}_{2,1} & \mathcal{L}_{2,2} \end{pmatrix} = \begin{pmatrix} L_{1,1} & & & & \\ & L_{2,2} & & & \mathbf{0} \\ \mathbf{0} & & \ddots & & \\ & & & L_{m-1,m-1} & \\ L_{m,1} & L_{m,2} & \cdots & L_{m,m-1} & L_{m,m} \end{pmatrix} \tag{11}$$

$$\mathbf{L}_{BDB}^T = \begin{pmatrix} \mathcal{L}_{1,1}^T & \mathcal{L}_{2,1}^T \\ \mathbf{0} & \mathcal{L}_{2,2}^T \end{pmatrix} = \begin{pmatrix} L_{1,1}^T & & & \mathbf{0} & L_{m,1}^T \\ & L_{2,2}^T & & & L_{m,2}^T \\ \mathbf{0} & & \ddots & & \vdots \\ & & & L_{m-1,m-1}^T & L_{m,m-1}^T \\ & & & & L_{m,m}^T \end{pmatrix} \tag{12}$$

$$\mathcal{A}_{1,1} = \mathbf{A}_{BD} = \begin{pmatrix} A_{1,1} & & \mathbf{0} & \\ & A_{2,2} & & \\ \mathbf{0} & & \ddots & \\ & & & A_{m-1,m-1} \end{pmatrix} \tag{13}$$

$$\mathcal{A}_{2,1}^T = \begin{pmatrix} A_{m,1}^T \\ A_{m,2}^T \\ \vdots \\ A_{m,m-1}^T \end{pmatrix} \tag{14}$$

$$\mathcal{A}_{2,1} = \begin{pmatrix} A_{m,1} & A_{m,2} & \cdots & A_{m,m-1} \end{pmatrix} \tag{15}$$

$$\mathcal{A}_{2,2} = A_{m,m} \tag{16}$$

Lemma — *The* $\mathbf{LL}^T$ *factors of a block-diagonal matrix are also in block-diagonal form*

Proof:

Let:

$$\mathbf{A}_{BD} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathcal{L}_{2,1} & \mathcal{L}_{2,2} \end{pmatrix} \begin{pmatrix} \mathcal{L}_{1,1}^T & \mathcal{L}_{2,1}^T \\ \mathbf{0} & \mathcal{L}_{2,2}^T \end{pmatrix} = \mathbf{L}_{BD}\mathbf{L}_{BD}^T \tag{17}$$

By applying the Shur complement to equation 17, we obtain:

$$\mathcal{A}_{2,1}^T = \mathcal{L}_{1,1}\mathcal{L}_{2,1}^T = \mathbf{0} \Rightarrow \mathcal{L}_{2,1}^T = \mathcal{L}_{1,1}^{-1}\mathbf{0} = \mathbf{0} \tag{18}$$

and

$$\mathcal{A}_{2,1} = \mathcal{L}_{2,1}\mathcal{L}_{1,1}^T = \mathbf{0} \Rightarrow \mathcal{L}_{2,1} = \mathbf{0}(\mathcal{L}_{1,1}^T)^{-1} = \mathbf{0} \tag{19}$$

If $\mathbf{A}_{BD}$ *is non-singular and has a numerical factor, then* $\mathcal{L}_{1,1}^{-1}$ *and* $(\mathcal{L}_{1,1}^T)^{-1}$ *must exist and be non-zero: thus*

$$\mathbf{A}_{BD} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{2,1}^T \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathcal{L}_{2,2} \end{pmatrix} \begin{pmatrix} \mathcal{L}_{1,1}^T & \mathbf{0} \\ \mathbf{0} & \mathcal{L}_{2,2}^T \end{pmatrix} = \mathbf{L}_{BD}\mathbf{L}_{BD}^T \tag{20}$$

This lemma can be applied recursively to a block-diagonal matrix with any number of diagonal blocks to prove that the $\mathbf{LL}^T$ *factorization of a block-diagonal matrix preserves the block structure.*

Theorem — *The* $\mathbf{LL}^T$ *factors of a block-diagonal-bordered matrix are also in block-diagonal-bordered form. To restate this theorem, we must show that* $\mathbf{A}_{BDB} = \mathbf{L}_{BDB}\mathbf{L}_{BDB}^T$.

Proof:

First the matrix partitions $\mathcal{A}_{2,1}$ *and* $\mathcal{A}_{2,1}^T$ *have simply been further partitioned to match the sizes of the diagonal blocks. Meanwhile, the matrix partition* $\mathcal{A}_{2,2}$ *has been left unchanged. In the lemma, we proved that the factors of* $\mathcal{A}_{1,1}$ *are block-diagonal if* $\mathcal{A}_{1,1}$ *is block-diagonal. Consequently,* $\mathbf{A}_{BDB} = \mathbf{L}_{BDB}\mathbf{L}_{BDB}^T$.

As a result of this theorem, it is relatively straight forward to identify available parallelism by simply performing the matrix multiplication in a manner similar to the Shur complement. As a result we obtain:

1. Diagonal Blocks: $\mathcal{A}_{1,1} = \mathcal{L}_{1,1}\mathcal{L}_{1,1}^T \Rightarrow \begin{cases} A_{1,1} = L_{1,1}L_{1,1}^T \\ A_{2,2} = L_{2,2}L_{2,2}^T \\ \vdots \end{cases}$

2. Lower Border: $\mathcal{A}_{2,1} = \mathcal{L}_{2,1}\mathcal{L}_{1,1}^T \Rightarrow \begin{cases} A_{m,1} = L_{m,1}L_{1,1}^T \\ A_{m,2} = L_{m,2}L_{2,2}^T \\ \vdots \end{cases}$

3. Last Block:

$$\mathcal{A}_{2,2} - \mathcal{L}_{2,1}\mathcal{L}_{2,1}^T = \mathcal{L}_{2,2}\mathcal{L}_{2,2}^T \Rightarrow \left\{ A_{m,m} - \sum_{i=1}^{(m-1)} L_{m,i}L_{m,i}^T = L_{m,m}L_{m,m}^T \right.$$

If the matrix blocks $A_{i,i}$ and $A_{i,m}$ $(1 \leq i \leq (m-1))$ are assigned to the same processor, then there are *no* communications until the last block is factored. At that time, only sums of sparse matrix $\times$ sparse matrix products are sent to the processors that hold the appropriate data in the last block. This data-assignment to processors is similar to column-oriented sparse Choleski algorithms, although a significant difference exists with block-diagonal-bordered form matrices. Data in block-diagonal-bordered form sparse matrices have a two-dimensional blocked nature that groups calculations and should permit efficient parallel operations.

This derivation identifies the parallelism in the Choleski factorization step of a block-bordered-diagonal sparse matrix. The parallelism in the forward reduction and backward substitution steps also benefits from the aforementioned data/processor distribution. By assigning data in a matrix block and its associated border section to the same processor, no communications would be required in the forward reduction phase until the last block of the factored matrix, $\mathbf{L}$, is updated by the product of a dense vector partition y_m $\times$ the sparse matrix $A_{i,m}$ $(1 \leq i \leq (m-1))$. No communications is required in the backward substitution phase after the values of x_m are broadcast to all processors holding the matrix blocks $A_{i,i}$ and $A_{i,m}$ $(1 \leq i \leq (m-1))$.

Figure 7 illustrates both the Choleski factorization steps and the reduction/substitution steps for a block-diagonal-bordered sparse matrix. In this figure, the strictly lower diagonal portion of the matrix is $\mathbf{L}$, and the strictly upper diagonal portion of the matrix is $\mathbf{L}^T$. This figure depicts four diagonal blocks, and processor assignments (P1, P2, P3, and P4) are listed with the data block.

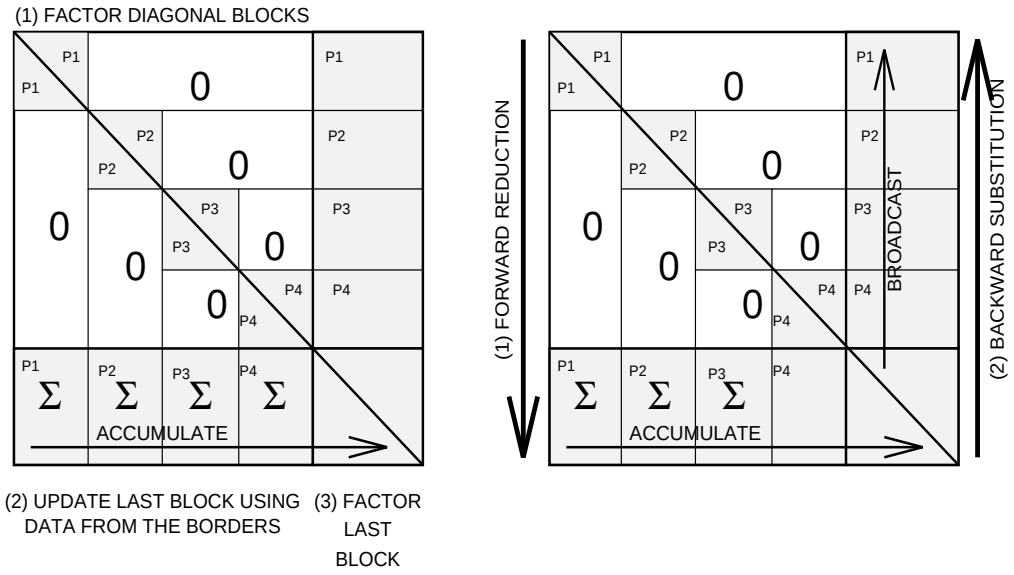


Figure 7: Block Bordered Diagonal Form Sparse Matrix Solution Steps

5 The Three-Step Preprocessing Phase

For parallel sparse block-diagonal-bordered matrix algorithms to be efficient when factoring irregular sparse matrices, the following three step preprocessing phase must be performed:

- *order* the matrix into block-diagonal-bordered form while minimizing the number of calculations,
- *pseudo-factorization* to identify both fillin and the number of calculations for all diagonal blocks and corresponding portions of the borders, and
- *load balance* to uniformly distribute the calculations among processors.

The first step determines the block-diagonal-bordered form and the ordering of nodes within diagonal blocks to minimize calculations; the second step determines the locations of fillin values for static data structures and also determines the number of calculations in independent blocks for the load balancing step; and the third step determines a mapping of data to processors for efficient implementation of the algorithm for the user specified data. These three steps may be incorporated into an optimization framework that uses the three-step preprocessing phase to produce matrix orderings with optimal overall performance for a particular version of the block-diagonal-bordered sparse matrix factorization algorithm. For this paper, the optimization was performed by hand — various values of input parameters

for the node-tearing routine were examined and the block-diagonal-bordered form sparse matrix with the best load balance and least numbers of operations were chosen for collecting performance benchmarks.

The metric for load balancing or the metric for an optimization routine to determine the most efficient overall ordering technique must be based on the actual workload required by the individual processors. This number may differ substantially from the number of equations assigned to processors because the number of calculations in an independent sub-block is a function of the number and location of non-zero matrix elements in that block — not the number of equations in a block. For dense matrices, the computational complexity of factorization is $O(N^3)$, however, the computational complexity for factoring entire sparse matrices used in later parallel algorithm performance studies varies from $O(N^{1.30})$ to $O(N^{1.55})$. Determining the actual workload requires a detailed simulation of all processing for the factorization and triangular solution phases, which we refer to as pseudo-factorization.

5.1 Ordering

The ordering portion in the preprocessing phase must identify diagonal matrix blocks while also attempting to minimize the amount of fillin during factorization. Few matrices can be readily ordered into block-diagonal-bordered form with equal workload in each block. The exception to this rule are highly regular matrices from the structural analysis community, where the nested dissection ordering algorithm can produce balanced block-diagonal-bordered matrices on some regular matrices [14]. Recursive spectral bisection can be used to partition irregular matrices [1, 17, 20], and subsequently, the coupling equations can be extracted. Unfortunately, this technique, as well as nested dissection, relies on dividing the matrix into m equal sized partitions, without considering the coupling equations or considering the number of calculations in each diagonal block. Load-imbalance limits the potential for using recursive spectral bisection, because the number of calculations for factorization or forward reduction/backward substitution are higher than linear order complexity, even for sparse matrices [15]. A third method to order a sparse matrix into block-diagonal-bordered form is referred to as node tearing [4, 19], which is a specialized form of diakoptics [13]. This technique attempts to extract the natural structure in the matrix or graph, and generally produces many irregularly sized blocks, while minimizing the number of coupling equations or the size of the lower border and last diagonal block. Load balancing techniques must be used after the node tearing matrix ordering step to uniformly distribute the processing load onto a multi-processor. As proven in section 4, diagonal blocks can be assigned to any processor without requirements for interprocessor communications to factor the diagonal

block and associated portion of the lower border.

There are several notable techniques to minimize fillin when factoring a sparse matrix, with one of the commonly used techniques being minimum-degree ordering. Minimum degree ordering is used in conjunction with the node-tearing-based ordering technique to generate block-diagonal-bordered form sparse matrices. Additional detail on minimum degree-based sparse matrix ordering is presented in appendix A.

We are looking for an ordering technique that limits the number of coupling equations for irregular problems and also limits fillin while ordering matrices into block-diagonal-bordered form. Minimizing the number of coupling equations minimizes the number of calculations and also minimizes the size of the nearly dense last block in a parallel block-diagonal-bordered sparse matrix solver; however, the amount of potential parallelism may suffer if the workload for factoring the diagonal blocks cannot be distributed uniformly throughout a multi-processor. Moreover, the distribution of workload between diagonal blocks and the last block must be considered. The last block will be nearly dense, and if the size of the last block of the matrix can be adequately constrained, the number of calculations can be drastically reduced. When determining the optimal ordering for a sparse matrix, the minimum total number of calculations may be traded for the optimal ordering that yields the most parallelism. The node-tearing ordering algorithm has the ability to adjust the characteristics of the ordering by varying an input parameter. A sample of the variety of matrix ordering possible with real irregular sparse admittance matrices is presented later in section 8. Nevertheless, the final measure of merit for matrix ordering is the performance of the parallel Choleski solver.

5.2 Pseudo-factorization

As stated above, the metric for performing load balancing or for comparing the performance of ordering techniques must be based on the actual workload required by the processors in a distributed-memory multi-computer. Consequently, more information is required than just the locations of fillin as in previous work that used symbolic factorization to identify fillin for static data structures [11, 14, 23].

To accomplish the two-fold requirement for both identifying the location of fillin and determining the amount of calculations in each independent block, we utilize a pseudo-factorization step as part of the preprocessing phase. Pseudo-factorization is merely a replication of the numerical factorization process without actually performing the calculations. Counters are used to tally the numbers of calculations to factor the independent data blocks and the numbers of calculations to update the last block using data from the borders. In addition, while performing the pseudo-factorization step, it is also simple to keep

track of the number of operations that would be required when performing the triangular solutions. By collecting this data, it provides an option to order the matrix to optimize the number of calculations per processor in the factorization step, or to optimize the number of calculations in the triangular solution steps. Often the $\mathbf{LL}^T$ matrix calculated by factorization is utilized multiple times in *dishonest* iterative numerical solutions. As a result, some applications may require special attention to maximum efficiency in the forward reduction and backward substitution steps.

There is no way to avoid the computational expense of this preprocessing step, because the computational workload in factorization is not correlated with the number of equations in an independent block. The number of calculations when factoring an independent sub-block is a function of the number and location of non-zero matrix elements in that block — not necessarily the number of equations in the block. Efficient parallel sparse matrix solvers require that any disparities in processor workloads be minimized in order to minimize load imbalance overhead, and consequently, to maximize processor utilization.

5.3 Load Balancing

The load balancing step of the preprocessing phase can be performed with a simple pigeon-hole type algorithm that uses one of several metrics based on the numbers of calculations determined in the pseudo-factorization step. There are three distinct steps in the proposed block-diagonal-bordered matrix solver:

- factor independent blocks,
- update the last block using data from the borders,
- factor the last block.

Load balancing as implemented for factorization of the diagonal blocks and the lower border emphasizes the uniform distribution of the processing workload in the first two steps. The parallel calculations in the last diagonal block are load balanced separately, which is simple, because we are using a parallel blocked kij-saxpy Choleski algorithm to factor the last diagonal block [28]. Our research into this area has emphasized uniformly distributing the workload to separate processors based on the number of calculations when factoring both the independent blocks and calculating the updates of the last block from data in the borders [15]. The second factorization step, updating the last block using data in the borders, requires that partial sums be accumulated from multiple processors and sent to the processor that holds the data for an element in the last block. However, the independent nature of calculations in the diagonal blocks and the border permit a processor to start the second phase as soon as that processor has completed factoring the independent

blocks. No processor synchronization is required between these steps and it is assumed that communications will occur independent of the calculations. Consequently, the sum total of all calculations in the diagonal blocks and corresponding border sub-matrices can be used when load balancing for factoring.

When load balancing for the triangular solutions, we chose as a metric the number of non-zero elements (including fillin) in all rows except the last diagonal block. This effectively emulates the total number of calculations, although the forward reduction of the last block requires processor synchronization. As a result, there may be some room for variation in this load-balancing metric.

Regardless of which metric is used for load-balancing, there is an important point to note. These metrics do not consider indexing overhead, which can be rather extensive when sparse matrices are stored in an implicit form. The data structure used in our solver has explicit links between non-zero values in a column and stores the data in any row as a sparse vector. This data structure should minimize indexing overhead at the cost of additional memory required to store the sparse matrix when compared with other sparse data storage methods [5]. The implementation of the parallel block-diagonal-bordered Choleski solver is discussed in greater detail in section 7.

The load-balancing algorithm is a simple greedy assignment algorithm that distributes objective function values to multiple pigeon-holes in a manner that minimizes the disparity between the sums of objective function values in each pigeon-hole. This is performed in a three-step process. First the objective function values for each of the independent blocks are placed into descending order. Second, the N_{procs} greatest values are distributed to a pigeon-hole for each processor, where N_{procs} is the number of processors in a distributed-memory multi-computer. Then the remaining objective function values are selected in descending order and placed in the pigeon-hole with the least aggregate workload. This algorithm is straightforward and minimizes the disparity in aggregate workloads between processors. This algorithm finds an optimal distribution for workload to processors, however, actual disparity in processor workload is dependent on the irregular sparse matrix structure. This algorithm works best when there are minimal disparities in the workloads for independent blocks or when there are significantly more independent blocks than processors. In this instance, the workloads in multiple small blocks can sum to equal the workload in a single block with more computational workload.

The pseudo-factorization step incurs significantly more computational cost than symbolic factorization in previous sparse matrix solvers. Additionally, the ordering phase is more costly than minimum degree ordering, and load balancing is often not explicitly considered. Consequently, block-diagonal-bordered sparse matrix solvers have significantly more overhead in the preprocessing phase, and consequently, the use of this technique will be

limited to problems that have static matrix structures that can reuse the ordered matrix and load balanced processor assignments multiple times in order to amortize the cost of the preprocessing phase over numerous matrix factorizations.

6 Node-tearing Nodal Analysis

Node-tearing nodal analysis partitions a graph into independent subgraphs and a coupling network, which corresponds to determining the diagonal blocks and lower border in a block-diagonal-bordered form matrix. We have selected node-tearing nodal analysis because this algorithm examines the natural structure in the matrix while providing the means to minimize the number of coupling equations. Tearing here refers to breaking the original problem into smaller sub-problems whose partial solutions can be combined to give the solution of the original problem. Node-tearing nodal analysis is a specialized form of diakoptic analysis [13] that was developed especially for power system network analysis [19]. In general, node-tearing analysis is superior to conventional diakoptic analysis because node-tearing simply orders the network graph and does not generate new nodes in the power distribution graph, The corresponding ordered admittance matrices retain their symmetry and positive definite nature. For this analysis, we are also interested in node-tearing because this algorithm identifies independent diagonal blocks in the matrix to generate block-diagonal-bordered form matrices, Examples in reference [19] illustrate that the technique also has validity for general structural analysis matrices.

6.1 The Node-tearing Algorithm

To describe node-tearing in rigorous mathematical terms, let the set $\mathcal{N}$ denote the nodes of a graph $\mathcal{G}$ and let $\mathcal{E}$ denote the edges in $\mathcal{G}$, or $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Partition the node set $\mathcal{N}$ into two arbitrary subsets $\mathcal{N}_1$ and $\mathcal{N}_2$, and partition the edge set $\mathcal{E}$ into two subsets $\mathcal{E}_1$ and $\mathcal{E}_2$ such that:

1. $\mathcal{E}_1$ contains all edges in $\mathcal{E}$ that touch nodes in $\mathcal{N}_1$,
2. $\mathcal{E}_2$ contains all other edges of $\mathcal{E}$.

Two conditions exist to ensure that the partitioned graph is suitable for tearing. The topological condition specifies the form into which we partition the graph, and the edge-coupling condition specifies limits on the connectivity of edges in graph partitions. Before defining the topological condition concerning the connected nature of the graph, we introduce the concept of a section graph.

Definition — Section Graph *Given a graph $\mathcal{G}$, let $\mathcal{S} \subset \mathcal{G}$, then a section graph is defined as:*

$$\mathcal{G}(\mathcal{S}) \equiv (\mathcal{S}, E_{\mathcal{S}}), \tag{21}$$

where: $\mathcal{E}_{\mathcal{S}} \equiv \{\varepsilon \in \mathcal{E} \mid \varepsilon$ is incident only with $\mathcal{S}\}$ [19].

The topological condition for graph connectivity requires that the section graph $\mathcal{G}_{\mathcal{N}_1}$ can be partitioned into m, $(m > 1)$ disconnected sub-graphs such that:

$$\begin{aligned} \mathcal{G}_1^1 &\equiv (\mathcal{N}_1^1, \mathcal{E}_1^1) \\ \mathcal{G}_1^2 &\equiv (\mathcal{N}_1^2, \mathcal{E}_1^2) \\ &\vdots \\ \mathcal{G}_1^m &\equiv (\mathcal{N}_1^m, \mathcal{E}_1^m). \end{aligned} \tag{22}$$

Given the topological condition, we can define the two partitions of the node set N:

$$\begin{aligned} \mathcal{N}_1 &\equiv \cup_{i=1}^{m} \mathcal{N}_1^i \\ \mathcal{N}_2 &\equiv \mathcal{N} - \mathcal{N}_1 \end{aligned} \tag{23}$$

where:

$\mathcal{N}_1$ is the set of nodes in the mutually independent sub-blocks

$\mathcal{N}_2$ is the set of nodes in the coupling equations

In the case of block-diagonal-bordered form matrices, $\mathcal{N}_1$ equates to the diagonal blocks, and $\mathcal{N}_2$ equates to those block-diagonal-bordered matrix rows in the lower border and the last diagonal block.

The edge-coupling condition simply requires that the edges in $\mathcal{E}_1^i$ are not connected to edges in $\mathcal{E}_1^j$ $\forall\, i \neq j$ and $i, j = 1, 2, \ldots, m$. Consequently, $\mathcal{G}_1^i$ has no edges in common with $\mathcal{G}_1^j$, $\forall i \neq j$, and there are no edges directly interconnecting any nodes in $\mathcal{N}_1^i$ and $\mathcal{N}_1^j$, $\forall i \neq j$. Connectivity between $\mathcal{G}_1^i$ and $\mathcal{G}_1^j$, $\forall i \neq j$, is not direct and must go through nodes in $\mathcal{N}_2$. Reference [19] contains the straight forward proof that these conditions yield a block-diagonal-bordered form matrix when the corresponding graph $\mathcal{G}$ is ordered by node-tearing.

In addition to ordering matrices into block-diagonal-bordered form using node-tearing, we require that the number of coupling equations, $\mid \mathcal{N}_2 \mid$, is minimized over all distinct partitions $\{\mathcal{N}_1, \mathcal{N}_2\}$ of $\mathcal{G}$. The tearing optimization problem attempts to minimize $\mid \mathcal{N}_2 \mid$ given that:

1. the topological condition holds,

2. the edge coupling condition holds,

3. $|\mathcal{N}_1^k| \leq max_{DB}$, $k = 1, 2, \ldots, m$.

Iterating Sets	Adjacency Sets	Contour Number
$\mathcal{I}_1^k$	$\mathcal{A}_1^k$	c_1^k
$\mathcal{I}_2^k$	$\mathcal{A}_2^k$	c_2^k
$\mathcal{I}_3^k$	$\mathcal{A}_3^k$	c_3^k
$\vdots$	$\vdots$	$\vdots$

Figure 8: Sample Contour Tableau for the k^{th} Diagonal Block

The last constraint for the tearing optimization problem permits some control of the maximum size of diagonal blocks, max_{DB}, which can prove quite useful when tearing a graph for factoring on multi-processors. By modifying this parameter, control can be exercised over the shape of the ordered sparse matrix — yielding narrow bandwidth blocked-diagonal-bordered form matrices when max_{DB} is small and limiting the size of the borders in a block-diagonal-bordered matrix when max_{DB} is large. The effects of varying the value of max_{DB} is illustrated in section 8.2. This optimization problem belongs to the family of NP-complete problems [19]. We expect to apply node-tearing to order large sparse matrices into block-diagonal-bordered form, so the use of an exact exponentially-bounded-complexity algorithm is not feasible, and the following efficient heuristic algorithm has been developed,

The technique chosen to solve the graph optimization problem is based on examining the contour of the graph [19], by developing a contour tableau to identify independent sub-graphs. A contour-tableau consists of three columns as illustrated in figure 8 and a separate contour-tableau is developed for each diagonal block. The leftmost column contains the iterating sets or the potential elements of a set of nodes in the sub-graph $\mathcal{N}_1^k$. The middle column contains the adjacency set, which contains the set of nodes adjacent to, but not including any elements in the corresponding iterating set. The remaining column contains the contour number or the cardinality of the corresponding adjacency set.

The contour tableau is constructed by selecting the initial iterating set element ν_1 and placing ν_1 in $\mathcal{I}_1^k$. Next, all nodes adjacent to ν_1, $\Lambda(\nu_1)$, are stored in $\mathcal{A}_1^k$: then $|\mathcal{A}_1^k|$ is placed in c_1^k. The next iterating set is constructed by forming the union of the previous iterating set and the next iterating node:

$$\mathcal{I}_{(i+1)}^k = \mathcal{I}_i^k \cup \{\nu_{(i+1)}\}. \tag{24}$$

The adjacency set is updated by the formula:

$$\mathcal{A}_{(i+1)}^k = \mathcal{A}_i^k \cup \Lambda(\nu_{(i+1)}) - \{\nu_{(i+1)}\}, \tag{25}$$

and

$$c^k_{(i+1)} = |\mathcal{A}^k_{(i+1)}|. \tag{26}$$

What remains to be described are the methods to select an initial node, select the next node, and to select an independent graph partition from the contour tableau. Because the algorithm is attempting to minimize $|\mathcal{N}_2|$, it can be shown that the selection of both the initial node and the next node should always be the node with the smallest number of adjacent nodes or select $\nu_{(i+1)}$ such that

$$\Lambda(\nu_{(i+1)}) = \min_{\forall\, v \in \mathcal{N} - \mathcal{I}^k_i} \Lambda(v) \tag{27}$$

If there are ties, then a node is selected arbitrarily. Lastly, the criteria to select an independent graph partition from the contour tableau requires identifying the iterating set $\mathcal{I}^k_i$ that has a local minimum value of c^k_i, $i \leq max_{DB}$. This selection criteria is obvious because at any location in the contour tableau, three disjoint sets are specified:

1. $\mathcal{I}^k_i$ — the iterating set,
2. $\mathcal{A}^k_i$ — the adjacency set,
3. $\mathcal{Z}^k_i = \mathcal{N} - \mathcal{I}^k_i - \mathcal{A}^k_i$ — the remaining nodes in $\mathcal{G}$.

In this representation, no node in $\mathcal{I}^k_i$ is adjacent to a node in $\mathcal{Z}^k_i$, and $\mathcal{A}^k_i$ represents the coupling equations between the two sets. The number of elements in the set $\mathcal{A}^k_i$ varies as a function of i. One constraint in this optimization problem is to minimize the number of coupling equations, $|\,\mathcal{N}_2\,|$, so a greedy algorithm that uses the heuristic for building the k^{th} independent partition, $\mathcal{N}^k_1$, by minimizing the cardinality of $\mathcal{A}^k_i$ should yield an acceptable solution in a polynomial algorithm. Moreover, when a partition is selected, nodes remaining in $\mathcal{A}^k_i$ are placed directly into the set $\mathcal{N}_2$,

$$\mathcal{N}_2 = \mathcal{N}_2 \cup \mathcal{A}^k_i \tag{28}$$

because $\mathcal{A}^k_i$ represents the nodes adjacent to but not included within the set $\mathcal{I}^k_i$. According to the topological condition, these nodes must be part of the coupling equations.

An example illustrating the node-tearing technique is presented in appendix B.

6.2 The Node-tearing Implementation

The software implementation to perform node-tearing nodal analysis utilizes the basic concept of building a contour tableau to identify independent sub-matrices and the coupling equations in an undirected graph representing a sparse matrix. In our implementation, the search for the local minimum of the contour number is limited to within the range

$\mathcal{G} \leftarrow$ *the symmetric graph representing the sparse matrix*
while $\mathcal{G} \neq \phi$ **do**
    while $i \leq max_{DB}$ **do**
        select $\nu_i \in \mathcal{A}^k_{(i-1)}$ such that $\Lambda(\nu_i) = \min_{\upsilon \in \mathcal{I}^k_{(i-1)}} \Lambda(\upsilon)$
        $\mathcal{I}^k_i \leftarrow \mathcal{I}^k_{(i-1)} \cup \{\nu\}$
        $\mathcal{A}^k_i \leftarrow \mathcal{A}^k_{(i-1)} \cup \Lambda(\nu_i) - \{\nu_i\}$
        if $(\alpha \times max_{DB}) \leq i \leq max_{DB}$
            determine the location of the local minimum ψ
        endif
    endwhile
    $\mathcal{N}^k_1 \leftarrow \mathcal{I}^k_\psi$
    $\mathcal{N}_2 \leftarrow \mathcal{N}_2 \cup \mathcal{A}^k_\psi$
    $\mathcal{G} \leftarrow \mathcal{G} - \mathcal{N}^k_1 - \mathcal{N}_2$
    minimum-degree order $\mathcal{N}^k_1$
end while
minimum-degree order $\mathcal{N}_2$

Figure 9: The Node-Tearing Algorithm

$(\alpha \times max_{DB}) \leq i \leq max_{DB}$, $0 < \alpha < 1$. When an independent sub-matrix is found, this iterating set is moved into a set $\mathcal{N}^k_1$, where $\mid \mathcal{N}^k_1 \mid = i$. After the sets $\mathcal{N}_1 = \{\mathcal{N}^1_1, \ldots, \mathcal{N}^m_1\}$ and $\mathcal{N}_2$ are determined, the equations corresponding to the sets $\mathcal{N}^1_1, \ldots, \mathcal{N}^m_1$ and $\mathcal{N}_2$ are further ordered using the minimum-degree ordering algorithm.

Figure 9 illustrates the major steps in the node-tearing ordering algorithm that produces block-bordered-diagonal form matrices with minimized fillin. The algorithm examines all nodes essentially once, where the size of the independent sub-blocks are limited to max_{DB}. The computational complexity of this algorithm is

$$O(\max_{\forall i} |\mathcal{A}^k_i| \times n) \tag{29}$$

due to the fact that all nodes in the graph must be examined, and for each element in the contour tableau — all elements of the adjacency set must be examined for the next node. The value of $\max_{\forall i} |\mathcal{A}^k_i|$ must be less than n, and because the graphs will be sparse, the maximum number in the adjacency set will be substantially less than n.

7 Sparse Matrix Solver Implementations

Implementations of a block-diagonal-bordered sparse Choleski solver have been developed primarily in the C programming language for the Thinking Machines CM-5 multi-computer using message passing and a host-node paradigm. Portions of our implementation use existing FORTRAN routines to factor and solve the last diagonal block of the matrix. A version of the software is available that runs on a single processor on the CM-5 to provide empirical speed-up data to quantify multi-processor performance. Empirical performance data has been gathered for a range of numbers of processors and real power systems sparse load-flow matrices. This empirical data is presented in the next section. Our block-diagonal-bordered sparse Choleski solver can be viewed as having the following distinct sections where blocks are defined in section 4:

1. Choleski factorization
   - factor the mutually independent diagonal blocks and associated portions of the border — $A_{i,i} = L_{i,i}L_{i,i}^T$ and $A_{m,i} = L_{m,i}L_{i,i}^T$ for $(1 \leq i \leq (m-1))$
   - update the last diagonal block using the data in the borders — $A_{m,m} = A_{m,m} - \sum_{i=1}^{(m-1)} L_{m,i}L_{m,i}^T$
   - factor the last diagonal block — $A_{m,m} = L_{m,m}L_{m,m}^T$
2. forward reduction
   - calculate the y vector partition corresponding to the mutually independent diagonal blocks — y_i for $(1 \leq i \leq (m-1))$
   - update the b vector partition corresponding to the last diagonal block — $b_m = b_m - \sum_{i=1}^{(m-1)} y_i L_{m,i}$
   - calculate the y vector partition corresponding to the last diagonal block — y_m
3. backward substitution
   - calculate the x vector partition corresponding to the last diagonal block — x_m
   - calculate the x vector partition corresponding to the mutually independent diagonal blocks — x_i for $((m-1) \geq i \geq 1)$

The parallel implementation presented in this section has been developed as an instrumented proof-of-concept to examine the efficiency of each section of the code described above. The host processor is used to gather and tabulate statistics on the multi-processor calculations. Statistics are gathered in a manner that do not impact the total empirical measures of performance for factorization, forward reduction, or backward substitution.

The last diagonal block can be factored in various manners, using either parallel dense or parallel sparse codes. Because the last block is generally relatively dense, this parallel implementation uses a parallel dense pipelined blocked kij-saxpy-based Choleski factorization algorithm. This algorithm uses both LAPACK and generalized BLAS routines for this portion of the algorithm [6]. When this algorithm is run on a single processor, a dense Choleski factorization algorithm from the LAPACK library and non-blocked forward reduction and backward substitution are utilized.

7.1 The Hierarchical Data Structure

This block-diagonal-bordered sparse Choleski solver uses implicit hierarchical data structures based on vectors of C programming language structures to efficiently store and retrieve data for a symmetric block-diagonal-bordered sparse matrix. These data structures provide good cache coherence, because non-zero data values and row and column location indicators are stored in adjacent physical memory locations. This data structure is static, consequently, the locations of all fillin must be determined before memory is allocated for the data structures. There is no requirement for pivoting in Choleski factorization algorithms because the sparse matrices are by definition positive definite and numerically stable. Due to the static nature of the data structure, explicit pointers to subsequent data locations have been used in order to reduce indexing overhead. Row location indicators are explicitly stored as are pointers to subsequent values in columns that are required when updating values in the matrix. The use of additional memory in the data structures is traded for reduced indexing overhead. Modern distributed memory multi-processors are available with substantial amounts of random access memory at each node, so this research examines data structures that are designed to optimize processing speed at the cost of increased memory usage when compared to other compressed storage formats. We compare the memory requirements for these data structures to the memory requirements for the more conventional compressed data structures below.

The hierarchical data structure is composed of five separate parts that implicitly store a block-diagonal-bordered sparse matrix. The hierarchical nature of these structures store only non-zero values, especially in the borders where entire rows may be zero. Five separate C language structures are employed to store the data in a manner that can efficiently be accessed with minimal indexing overhead. Static vectors of each structure type are used to store the block-diagonal-bordered sparse matrix. Figure 10 graphically illustrates the hierarchical nature of the data structure, where the five separate C structure elements presented in that figure are:

1. diagonal block identifiers,

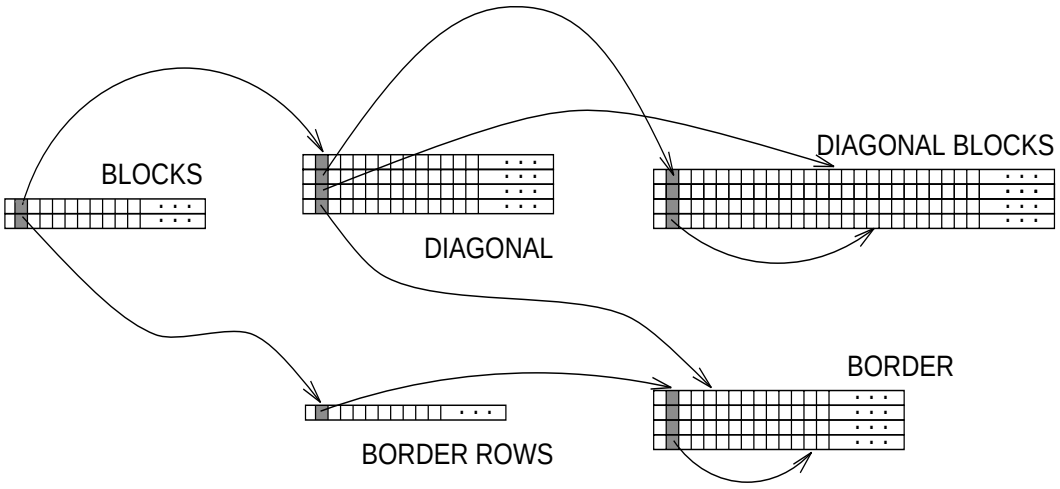


Figure 10: The Hierarchical Data Structure

2. matrix diagonal elements,

3. non-zero values in the lower triangular diagonal matrix blocks (arranged by rows),

4. non-zero row identifiers in the lower border,

5. non-zero values in the lower border (arranged by rows),

At the top of the hierarchical data structure is the information on the storage locations of independent diagonal blocks, and the lower borders. The next layer in the data structure hierarchy are the matrix diagonal and the identifiers of non-zero border rows. Data values on the original matrix diagonal are stored in the diagonal portion of the data structure, however, most of the remaining information stored along with each diagonal element are pointers so that data in related columns or rows can be rapidly accessed.

Data in the strictly lower triangular portion of the matrix is stored as sparse row vectors. This data storage scheme minimizes the effort to find non-zero $A_{i,k}$ — $A_{j,k}$ pairs used to modify $A_{i,j}$ by consecutively storing values in lower triangular rows. However, column-oriented Choleski factorization algorithms require access to the next non-zero value in the same column, so pointers are used to permit direct access to those values without requiring searching for the data as is required in compressed storage formats. This data structure provides the benefits of a doubly linked data structure in order to minimize indexing overhead. The value corresponding to any diagonal element has pointers to the first non-zero element in the lower triangular row and to the first non-zero element in the lower border. This data structure trades memory utilization for speed by storing indicators to all non-zero column values. In addition, the combination of adjacent storage of non-zero row values and the

explicit storage of column identifiers, greatly simplify the forward reduction and backward substitution steps.

Conventional compressed data formats require less storage than this data structure; however, additional memory has been traded for reduced indexing overhead. The compressed data format requires

$$S_c = (\lambda_{fp} + \lambda_{int}) \times \eta(A) + (\lambda_{int} \times n) \tag{30}$$

bytes to store the A matrix implicitly. Likewise, the hierarchical data structure used in this implementation requires

$$S_h = (\lambda_{fp} + (3 \times \lambda_{int})) \times \eta(A) + (\lambda_{int} \times n) + ((3 \times \lambda_{int}) \times N_{blocks}) + (\lambda_{int} \times N_{border}) \tag{31}$$

bytes to store the same matrix implicitly.

where:

S_c is the storage requirements in bytes for the compressed data structure.
S_h is the storage requirements in bytes for the hierarchical data structure.
λ_{fp} is the length if a floating point data type.
λ_{int} is the length if an integer data type.
$\eta(A)$ is the number of non-zero values in the matrix.
n is the order of the matrix.
N_{blocks} is the number of independent blocks.
N_{border} is the number of non-zero row and column segments in the borders.

For double precision floating point or single precision complex representations of the actual data values and single word integer representations of all pointers, the hierarchical data structure takes approximately twice the data storage of the compressed data structure. By doubling the storage requirements, row data is available in sparse vectors for ready access when updating a column value and subsequent values in the column are directly addressable. When using conventional compressed data structures, indexing information is stored only on a single dimension and values along the other dimension must be found by searching through the data structures to find the next values to update. To find a value in a row or column, the average number of operations in the search will be one-half the average number of values in the row or column. Given that this costly search must be performed for nearly every non-zero value in the matrix, substantial indexing overhead is required when using the implicit compressed storage format. By using this data structure and doubling the storage, there is a significant decrease in indexing overhead.

7.2 The Parallel Algorithm

Implementation of the parallel block-diagonal-bordered sparse matrix solver has been developed primarily in the C programming language for the Thinking Machines CM-5 multiprocessor using a host-node paradigm with message passing. A pseudo-code representation of the Choleski factorization algorithm is presented in figure 11. The same implicit hierarchical data structure used in the parallel implementation can be readily used in a sequential implementation, although the sparse matrix data is no longer physically allocated to the distributed memory located with the various processors. This block-diagonal-bordered Choleski factorization implementation solves the last block using a dense blocked kij-saxpy Choleski algorithm [28]. Efforts have been relatively successful to minimize the size of the last diagonal block; consequently, this block is relatively dense — often 20% to 35% dense. The reduced indexing overhead and regularity in data access justify the use of parallel dense Choleski algorithms for this relatively small partition of the overall matrix.

By distributing the factorization of the last block to all active node processors, partial sums of updates for values in the borders in the second phase of the factorization step must also be distributed to all processors. Only those nonzero rows in the last block are examined when calculating the required partial sum values, which significantly limits the amounts of calculations in this phase. In order to minimize communications times in this factorization step, care is taken to minimize the length of the interprocessor communications messages; and separate vectors containing only information for a particular processor are sent to each machine. Because of the sparsity of the rows in the border, there has been no attempt at parallel reduction of the partial sums of updates from the borders.

The remaining steps in the parallel algorithm are forward reduction and backward substitution. The parallel version of these algorithms take advantage of the fact that calculations can be performed in one of two distinct orders that preserve the precedence relation in the calculations. The combination of these techniques is utilized to minimize communications times when solving for the last diagonal block. Pseudo-code versions of the algorithms for parallel forward reduction and parallel backward substitution are presented in figures 12 and 13 respectively.

8 Empirical Results

A stated goal of this block-diagonal-bordered Choleski solver is to simplify the task organization of the parallel Choleski algorithm and have interprocessor communications significantly reduced and regular. The performance of this block-diagonal-bordered Choleski solver is dependent on the ability to order the real power systems sparse matrices into the appropriate

Node Program

/* factor the independent blocks and corresponding borders */

for those independent blocks l **assigned to this processor**

  for all elements k **along the diagonal in block** l

    Update the k^{th} *column in the* l^{th} *lower diagonal block*

    Update the k^{th} *column in the* l^{th} *lower border section*

  endfor

endfor

/* calculate updates to the last block */

for those independent blocks l **assigned to this processor**

  for all non-zero rows i_2

    for all non-zero rows i_1 **such that** $i_1 \geq i_2$

      for each j **such that** $L_{i_1,j}$ **and** $L_{i_2,j} \neq 0$

        $p \leftarrow \mathcal{P}(i_1, i_2)$ /* the function $\mathcal{P}()$ maps the (i_1, i_2) tuple to the processor location p */

        $\Sigma_{i_1,i_2,p} \leftarrow \Sigma_{i_1,i_2,p} + (L_{i_1,j} * L_{i_2,j})$

      endfor

    endfor

  endfor

endfor

/* update the last block */

for all processors i_p

  if $(i_p == me)$

    for all processors j_p **starting with** $(i_p + 1)$ **around a ring in ascending order**

      Send the sparse vector of partial sums

    endfor

    Update the data in the last diagonal block on this processor

  end if

  Receive the sparse vector of partial sums

  Update the data in the last diagonal block on this processor

endfor

/* factor the last block */

Factor the last diagonal block using a parallel blocked kij-saxpy dense Choleski algorithm

Figure 11: Parallel Block-diagonal-Bordered Sparse Choleski Factorization

Node Program

/* reduce the independent blocks */

for all independent blocks l **assigned to this processor**

  for all rows i **in block** l

    $y_i \leftarrow (b_i / L_{i,i})$

    Update b_i *using* y_j *and* $L_{i,j}$ *values in the independent diagonal blocks*

  endfor

endfor

/* calculate updates to the last block */

for all independent blocks l **assigned to this processor**

  for all non-zero rows i **in the lower border of this block**

    for each j such that $L_{i,j} \neq 0$

      $p \leftarrow \mathcal{P}(i_1, i_2)$ /* the function $\mathcal{P}()$ maps the (i_1, i_2) tuple to the processor location p */

      $\Sigma_{i,p} \leftarrow \Sigma_{i,p} + (y_i * L_{i,j})$

    endfor

  endfor

endfor

/* update the last block */

for all processors i_p

  if $(i_p == me)$

    for all processors j_p **starting with** $(i_p + 1)$ **around a ring in ascending order**

      Send the sparse vector of partial sums

    endfor

    Update the data in the last diagonal block on this processor

  end if

  Receive the sparse vector of partial sums

  Update the data in the last diagonal block on this processor

endfor

/* reduce the last block */

Forward reduce the last block using a parallel blocked algorithm

Figure 12: Parallel Sparse Forward Reduction

Node Program

/* backward substitute the last block */

Backward substitute the last block using a parallel blocked algorithm

Broadcast the x-values to the node processors as blocks are substituted

/* backward substitute in the independent blocks and the border */

for all independent blocks l **in descending order**

  for all rows i **in block** l

    Update y_j *using* x_j *and* $L_{j,i}$ *values in the border*

    $x_i \leftarrow (y_i / L_{i,i})$

    Update y_j *using* x_i *and* $U_{j,i}$ *values in the independent diagonal blocks*

  endfor

endfor

Figure 13: Parallel Sparse Backward Substitution

form with both uniformly distributed data in the diagonal blocks and a minimum number of equations on the lower border. In this section of the paper, we first examine the performance of the Choleski solver for generated test data that has perfect load balance, in order to understand the performance potential of the block-diagonal-bordered Choleski solver. We then report on the performance of the node-tearing nodal analysis and the performance of the block-diagonal-bordered sparse Choleski solver. In section 8.2, we illustrate the ordering capabilities of the node-tearing nodal analysis by presenting both pseudo-images of selected sparse load-flow matrices and data collected on the load imbalance as a function of the number of processors. But the real performance test of the node-tearing algorithm will occur when the performance of the block-diagonal-bordered sparse Choleski solver is examined for real power system load-flow matrices in section 8.3.

8.1 Empirical Results — Sparse Choleski Solver with Machine Generated Test Data

The first step in understanding the performance potential of this block-diagonal-bordered Choleski solver is to examine the parallel algorithm performance with machine generated test data that has perfect load balance for all processors. This data has equal numbers of calculations in diagonal-blocks to eliminate requirements for load balancing. The pattern of the matrices is a recursive block-diagonal-bordered form as illustrated in figure 14. This figure depicts the sparsity structure in the test matrix, where non-zero values are black

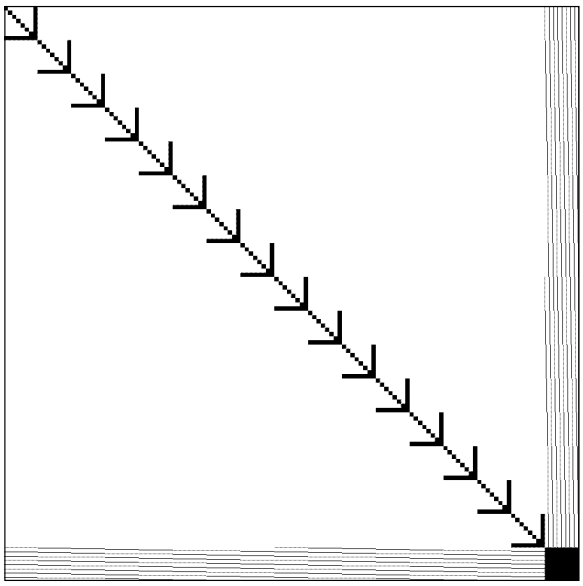


Figure 14: Machine Generated Test Matrix

and the remainder of the matrix is comprised of zero-values. A bounding box has also been placed around the example matrix. This matrix has been used in tests to examine speedup and sources of parallel processing overhead. The block-diagonal-bordered Choleski solver has been run on this machine generated matrix using one to sixteen processors on the Thinking Machines CM-5. This matrix has sixteen separate major diagonal blocks, each with 128 diagonal elements. The last diagonal block also has 128 diagonal elements, with eight columns per diagonal block in the lower border. The size of this matrix was chosen to be similar to the BCSPWR09 test matrix, although it has a substantially higher computational complexity than the BCSPWR09 matrix, $O(N^{1.85})$ versus $O(N^{1.38})$

Performance data have been collected for distinct operations in the factorization and triangular solution of block-diagonal-bordered matrices to examine the speedup of each portion of the algorithm. Data was collected in such a manner as not impact the overall measures of performance. Performance of the multi-processor algorithms are illustrated using graphs plotting relative speedup versus the number of processors (one to sixteen).

Definition — Relative Speedup *Given a single problem with a sequential algorithm running on one processor and a concurrent algorithm running on p independent processors, relative speedup is defined as*

$$S_p \equiv \frac{T_1}{T_p}, \tag{32}$$

where T_1 is the time to run the sequential algorithm as a single process and T_p is the time to run the concurrent algorithm on p processors.

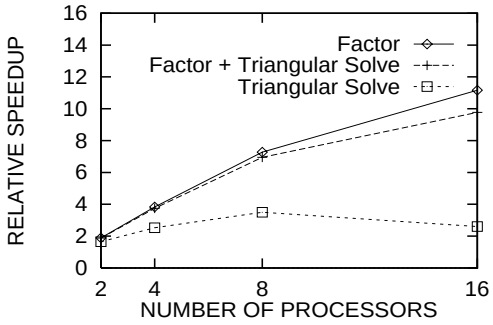


Figure 15: Speedup for Generated Test Data — 2, 4, 8, and 16 Processors

A graph of speedup calculated from empirical performance data for the test matrix is provided in figure 15. This figure has a family of three curves that show speedup for:

1. Choleski factorization
2. forward reduction and backward substitution (triangular solve)
3. a combination of factorization and a single forward reduction and backward substitution

In general this speedup graph shows that the block-diagonal-bordered Choleski solver has significant potential for efficient operations, because speedups of over eleven were obtained with sixteen processors. This equates to a processor utilization efficiency of 70%. When examining the runtime timing data for each processor, near perfect speedup (speedup equal to the number of processors) is not achieved because of overhead that occurs during those portion of the algorithm that use asynchronous pipelined communications. The following sections of the block-diagonal-bordered Choleski solver use asynchronous pipelined communications:

- the update of data in the last block using data from the lower border
- the dense blocked kij-saxpy based Choleski factorization of the last block
- the update of data in the b-vector partition using data from the lower border
- the forward reduction of the last block
- the backward substitution of the last block

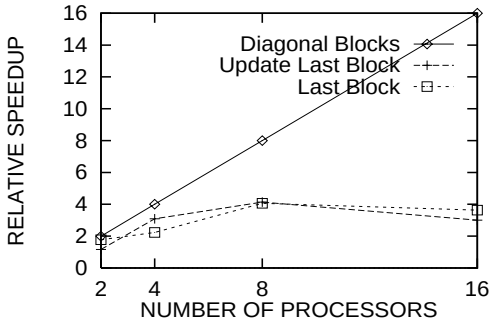


Figure 16: Speedup for Choleski Factorization Algorithm Components — 2, 4, 8, and 16 Processors

As a result of the regularity in this test matrix, it is possible to examine the performance of each portion of the parallel block-diagonal-bordered Choleski factorization algorithm. A graph illustrating speedup calculated from the empirical performance data is presented in figure 16. In this graph, the speedup for factorization of the diagonal blocks show perfect speedup, which is not unexpected because there is no communications in this step. However, the measured speedup for the two sections of the Choleski factorization algorithm that require communications, is not as impressive, due to communications overhead. As we use more processors, there is more communications and less work for each processor. It takes longer to fill communications pipelines and there are less calculations per processor to help mitigate the cost of communications. Consequently, there are performance limits to this algorithm. These limits are not a function of a sequential component of the parallel calculations (Amdahl's Law) [7], but rather the limits are a function of what percentage of the calculations are perfectly parallel and what percentage of the calculations have asymptotic performance limits due to communications overhead.

Parallel block-diagonal-bordered Choleski factorization algorithm performance can be viewed as a linear combination of the individual performance of these three portions of the algorithm.

$$S_p = (\lambda_{DB} \times S_{DB,p} + (\lambda_U \times S_{U,p}) + (\lambda_{LB} \times S_{LB,p}) \tag{33}$$

where:

S_p is the p processor speedup for block-diagonal-bordered Choleski factorization.

$S_{DB,p}$ is the p processor speedup for factoring of the diagonal blocks.

$S_{U,p}$ is the p processor speedup for updating the data in the last block using data in the border.

$S_{LB,p}$ is the p processor speedup for factoring the last block.

λ_{DB} is the fraction of time spent factoring of the diagonal blocks.

λ_U is the fraction of time spent updating the data in the last block using data in the border.

λ_{LB} is the fraction of time spent factoring the last block.

$\lambda_{DB} + \lambda_U + \lambda_{LB} = 1$

Our empirical data shows that $S_{p_{DB}} \approx p$ because this step has no communications, so the performance of this factorization algorithm on a problem using other block-diagonal-bordered sparse matrices can be estimated as a function of the percentage of calculations in the diagonal blocks. Similar analyses can be performed for forward reduction and backward substitution. Parallel performance of those sections of the triangular solution algorithms that require pipelined communications will have less speedup potential than the triangular solution of data in the diagonal blocks. Unfortunately, the calculations in the diagonal blocks for the triangular solution will be of a lesser computational complexity than the calculations in the factorization step. Consequently, poorer parallel performance, lower speedup, would be expected in forward reduction and backward substitution than can be obtained in block-diagonal-bordered factorization.

There are two basic forms of communications used in this algorithm.

1. Pipelined broadcasts for those portions of the algorithm that factor or solve the last diagonal block

2. Pipelined, asynchronous, everyone-to-everyone communications in both factorization and forward reduction when sums of matrix $\times$ matrix products or vector $\times$ matrix products from data in the lower border are used to update the last diagonal block or the b-vector partition associated with the last diagonal block.

In both instances, the pipelined communications appears to *data-starve* and not hide communications behind the calculations. This is not surprising in the forward reduction of the last block, but this phenomenon also occurs in the block factorization step. Experiments with adjusting the block size offered little improvement. While the communications have been made regular in the other communications steps, it appears that communications times are so large relative to calculations in the same processing step, that little benefit can be made of hiding communications behind calculations. Further research into other asynchronous communications techniques such as active messages [21] may be able to significantly improve performance in this area.

This analysis has been performed on data that had no load imbalance overhead. Additional sources of overhead would degrade potential performance of the algorithm [7]. We have discussed the effects of communications overhead, nevertheless, other sources of parallel processing overhead are possible. They include indexing overhead and load imbalance

overhead. Indexing overhead is the extra computational cycles required to set up loops with multiple processors. Care has been exercised in the development of this algorithm to minimize indexing overhead. While it is not possible to eliminate the cost of repeating the process of setting up loops on each processor, the amount of calculations required to keep track of the values as a function of individual multi-processors can be minimized. Many of the sources of indexing overhead have been accounted for in the preprocessing stage that prepares a matrix form for processing by this algorithm. The computational cost of the preprocessing stage is expected to be amortized over multiple uses of the block-diagonal-bordered Choleski, thus the costs incurred in this stage are not addressed here. Load imbalance overhead is also addressed in the preprocessing stage, although the sources of load imbalance overhead are a function of the interconnections in the real data sets. In the preprocessing phase, we use load-balancing to attempt to order the data as to minimize this source of overhead in our parallel algorithm. We will show in section 8.3 that even with attempts at load balancing, this source of overhead cannot be entirely removed.

8.2 Empirical Results — Ordering

Performance of our block-diagonal-bordered Choleski solver will be analyzed with three separate power systems matrices:

- Boeing-Harwell matrix BCSPWR09 — 1723 nodes and 4117 edges in the graph [5]
- Boeing-Harwell matrix BCSPWR10 — 5300 nodes and 13571 edges in the graph [5]
- data obtained from Niagara Mohawk Power Corporation — 9430 nodes and 14001 edges in the graph

Matrices BCSPWR09 and BCSPWR10 are from the Boeing Harwell series and represent real electrical power system networks from the Western and Eastern US respectively. These matrices have an edge-to-node-ratio of approximately 2.5. The data obtained from the Niagara Mohawk Power Corporation is substantially sparser, with an edge-to-node-ratio of only 1.5. While this data has more nodes than either of the Boeing-Harwell matrices, the reduced sparsity equates to lesser calculations than the BCSPWR10 matrix.

A detailed ordering analysis has been performed on the BCSPWR09 data to illustrate the ability of the node-tearing ordering algorithm. To contrast the performance of this ordering technique, figure 17 illustrates a minimum degree ordering of the BCSPWR09 matrix without ordering. Note that the matrix is the most sparse in the upper left-hand corner, while the matrix is less sparse in the lower right-hand corner. When factoring this matrix, the number of zero values that become non-zero while factoring the matrix, is 2,168. Original nonzero values are represented in this figure in black, fillin locations are

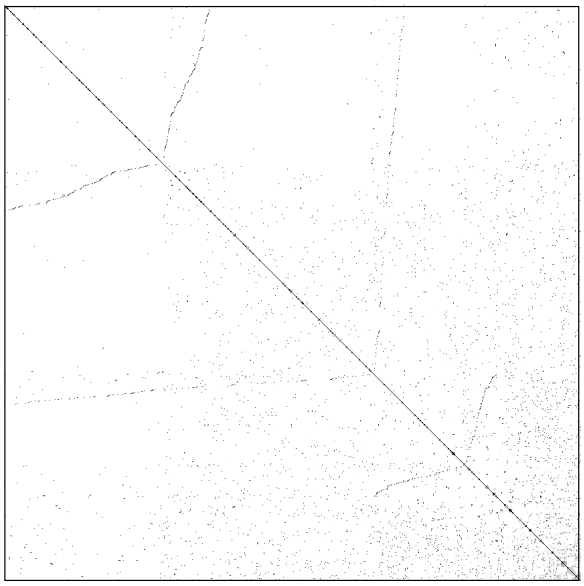


Figure 17: Minimum Degree Ordering — BCSPWR09

represented in gray, and all remaining zero values are white. A bounding box has been placed around the sparse matrix. Our block-diagonal-bordered Choleski solver requires that the BCSPWR09 matrix be ordered into block-diagonal-bordered form with uniformly distributed workload at each of the processors. A single specified input parameter, the maximum partition size, defines the shape of the matrix after ordering using the node-tearing algorithm. Examples of applying the node-tearing algorithm to the BCSPWR09 matrix are presented in figures 18, 19, and 20 respectively for maximum diagonal block sizes (max_{DB}) of 32, 64, and 128 nodes. The number of fillin for these block-diagonal-bordered form matrices is 2,765, 3,248, and 3,838 respectively, with a substantial portion of the fillin occurring in the lower right-hand corner, or the last diagonal-block.

The general performance of node-tearing-based ordering is dependent on the maximum number of nodes in a diagonal block, and the intended number of processors to which the mutually independent diagonal blocks will be distributed. The number of coupling equations and the size of the last block is dependent only on the maximum number of nodes in a diagonal block, which is illustrated in figure 21. Note that the maximum size of the diagonal blocks is inversely related to the size of the last diagonal block. This is intuitive, because as diagonal matrix blocks are permitted to grow larger, multiple smaller blocks can be incorporated into a single block. Not only will the two blocks be consolidated into the single block, but in addition, any elements in the coupling equations that are unique to those network partitions could also be moved into the larger block. Another interesting

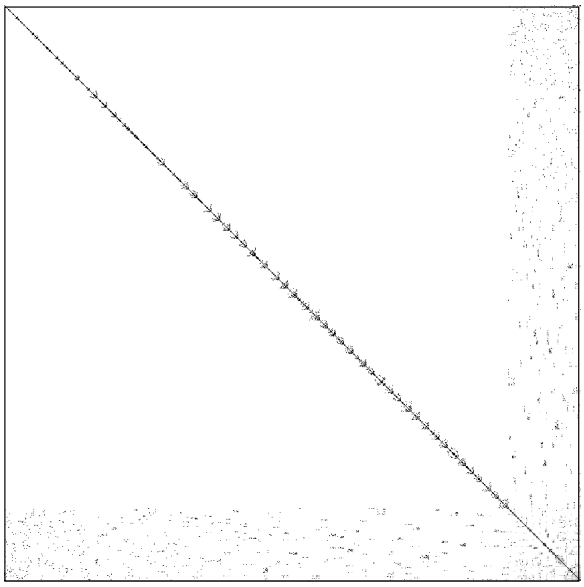


Figure 18: Node-Tearing-Based Ordering — BCSPWR09 — $max_{DB} = 32$

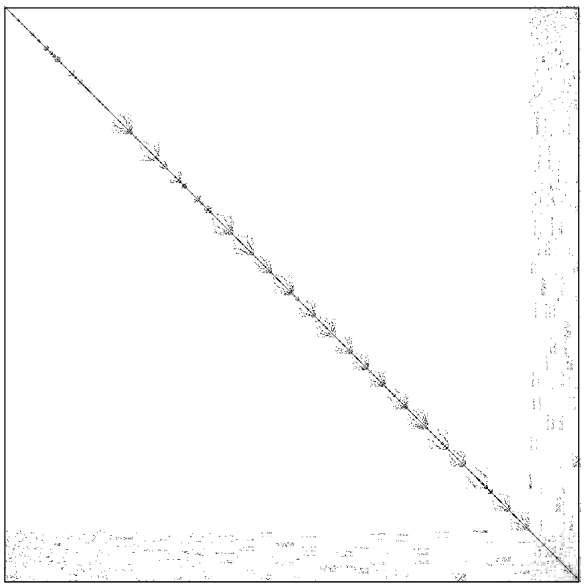


Figure 19: Node-Tearing-Based Ordering — BCSPWR09 — $max_{DB} = 64$

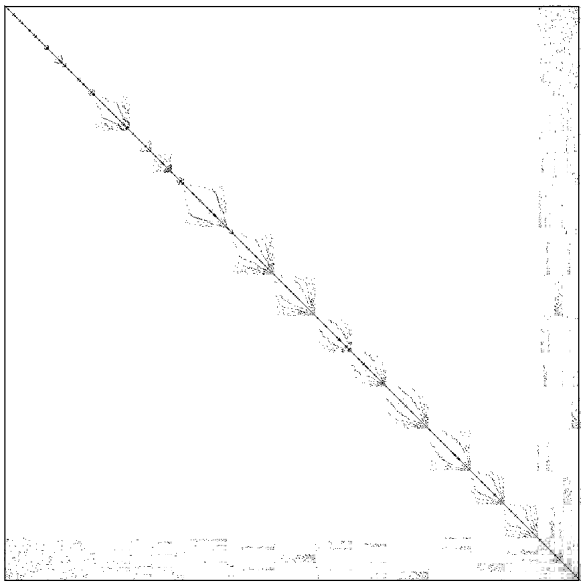


Figure 20: Node-Tearing-Based Ordering — BCSPWR09 — $max_{DB} = 128$

point with the relationship between maximum size of the diagonal block and the size of the last block, is that the percentage of non-zeros and fillin in the last diagonal block increases significantly as the size of the last block decreases. This makes the use of parallel dense Choleski factorization techniques even more desirable for the last diagonal block when the selection of the maximum size of the diagonal blocks is chosen to minimize the size of the last diagonal block.

It is desirable to minimize the size of the last block, but this can cause load imbalance as the number of processors increases. Load imbalance is defined as the ratio of the difference of the maximum and minimum numbers of floating point operations divided by the maximum number of floating point operations per processor and illustrates the percentage of time that the processor with the least amount of work is inactive. Families of curves illustrating load imbalance as a function of the number of processors is presented in figure 22. The load imbalance data presented in this figure are calculated only for the number of operations required to calculate the Choleski factor of the matrix. For all ordering with different size diagonal blocks, there is perfect load-balancing for four processors, however, as the number of processors increases, so does load imbalance.

For the BCSPWR09 matrix, we have selected a maximum diagonal block size of 128 as the trade-off between load balance and minimum size of the last block. Figure 23 depicts a reordered version of the matrix presented in figure 20 with diagonal blocks assigned to a processor ordered to make those blocks contiguous. The ordering of the matrix to reflect

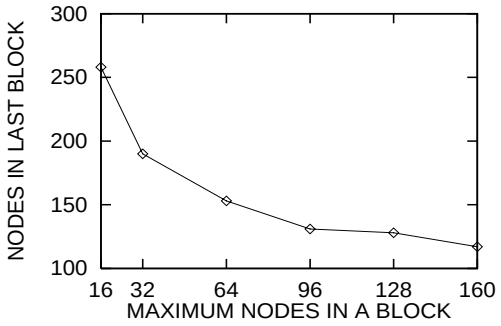


Figure 21: Node-Tearing — Nodes in Last Block for BCSPWR09

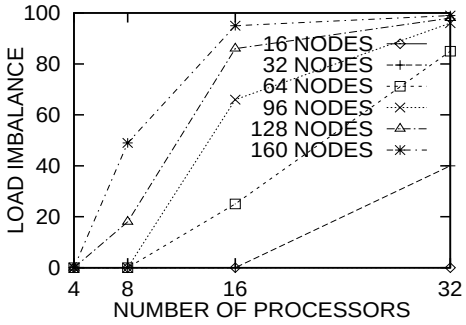


Figure 22: Node-Tearing — Load Imbalance for Factoring BCSPWR09

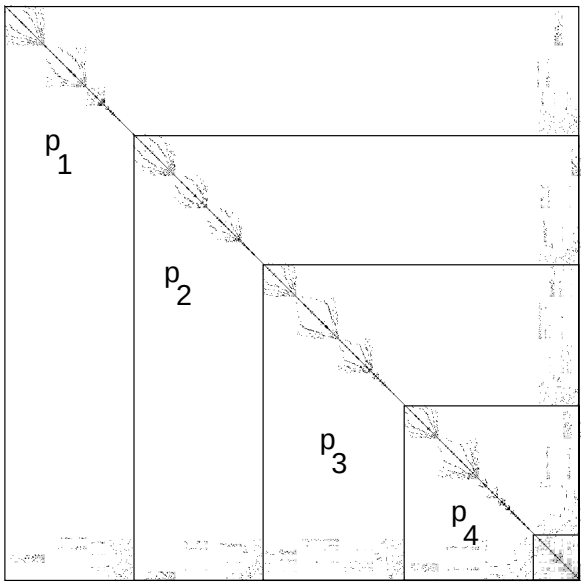


Figure 23: BCSPWR09 — Load Balancing on Factoring — 4 Processors

contiguous diagonal blocks within each processor is performed by simple row and column exchanges, and has no effect on either the positive definite nature of the matrix or on the number of fillin. This is just a modification to the permutation matrix used in equation 5. In this figure, the measure of merit for the pigeon-hole load balancing algorithm is the number of factorization (multiply and addition) operations. In contrast, figure 24 depicts the same matrix after node-tearing with the exception that the load balancing has been performed as a function of the number of multiply and addition operations encountered in forward reduction and backward substitution. Note the obvious differences in block assignments to processors P2 and P4. In figures 23 and 24, the assignments of matrix partitions to processors have been denoted. Figure 25 presents plots of load imbalance verses numbers of processors for the BCSPWR09 matrix ordered with a maximum of 128 nodes. Two load imbalance curves are presented — one for load balancing on the number of calculations in the factorization step and the second curve represents load-imbalance for load balancing on the number of calculations in the triangular solution phase. The number of processors range from two through 32 and it is clear that the load imbalance becomes significant for greater than eight processors. This graph shows that there is only a significant amount of additional load imbalance encountered for eight processors. Load imbalance for other numbers of processors is quite similar.

Similar ordering analyses have been performed for the BCSPWR10 and Niagara Mohawk load flow matrices. Examples of these matrices after applying the node-tearing algorithm

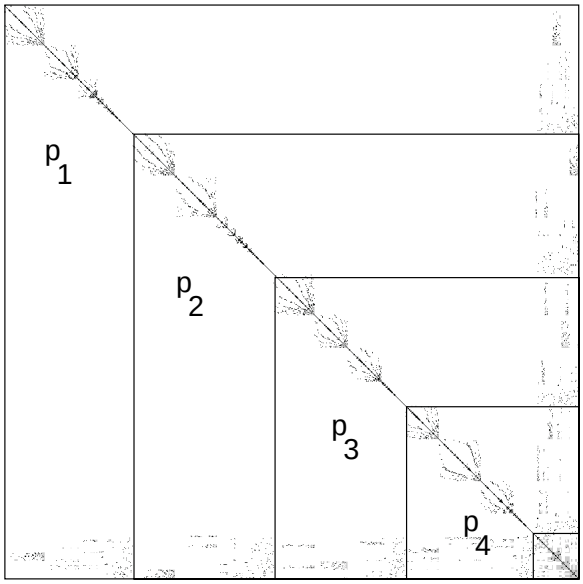


Figure 24: BCSPWR09 — Load Balancing on Triangular Solutions — 4 Processors

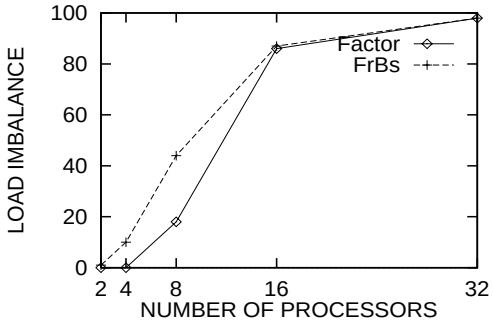


Figure 25: Load Imbalance for Factoring BCSPWR09

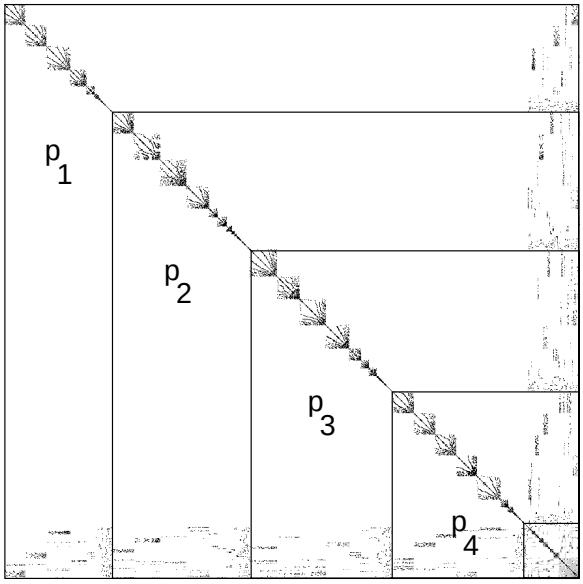


Figure 26: BCSPWR10 — Load Balancing on Factoring — $max_{DB} = 256$ — 4 Processors

and ordering both into block-diagonal-bordered form and for load-balancing are presented in figures 26 through 28. Figure 26 illustrates the block-diagonal-bordered form achievable when ordering the BCSPWR10 matrix with maximum block size of 256, while figures 27 and 28 illustrate the block-diagonal-bordered matrix form achievable when ordering the Niagara Mohawk data for maximum block sizes of 256 and 512 nodes respectively. The number of graph nodes or diagonal elements in the BCSPWR10 and Niagara Mohawk matrices are 5300 and 9430 respectively. The BCSPWR10 matrix in figure 26 has 476 equations in the lower border and last block while the Niagara Mohawk matrices in figures 27 and 28 have 402 and 313 equations in the coupling equations respectively. These graphs illustrate load balancing for four processors based on the number of calculations to perform the factorization. The Niagara Mohawk power system network graphs are similar to the BCSPWR09 power system graph in that they can be readily ordered into block-diagonal-bordered form, there are some interesting and substantial differences. Summary statistics are presented in table 1 for the three matrices and various maximum diagonal block sizes.

The BCSPWR10 matrix has substantially more edges per node than the BCSPWR09 matrix, which results in a significantly greater computational complexity for the factorization of the matrix. Meanwhile, the computational complexity of the triangular solution ($\triangle$) is similar to the BCSPWR09 matrix. Greater computational complexity means more work, and if that work can be distributed uniformly to all processors, then there is the potential for greater parallel implementation efficiency. Our goal is to develop scalable algorithms

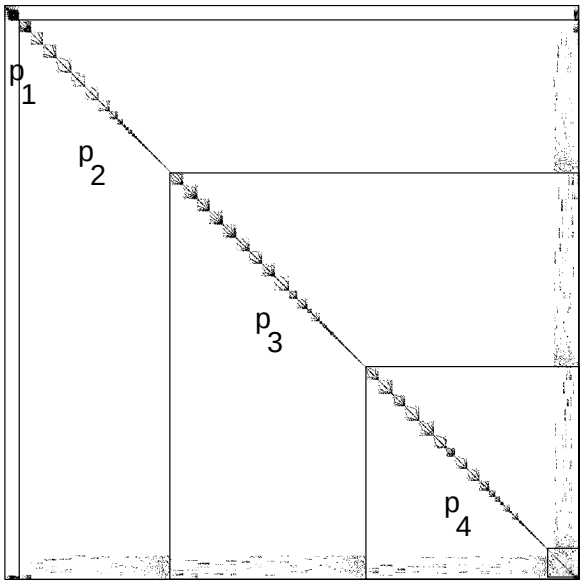


Figure 27: Niagara Mohawk Data — Load Balancing on Factoring — $max_{DB} = 256$ — 4 Processors

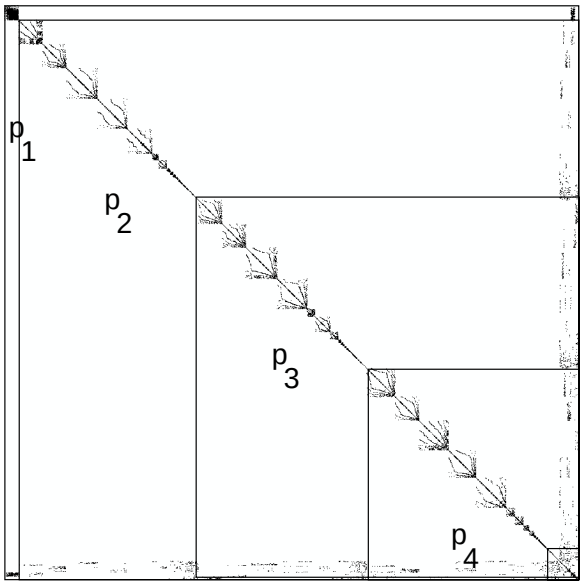


Figure 28: Niagara Mohawk Data — Load Balancing on Factoring — $max_{DB} = 512$ — 4 Processors

Graph	Nodes	Edges	max_{DB}	Fillin	Last Block	Factor	$\triangle$ Solve
BCSPWR09	1,723	4,117	32	2,765	190	$O(N^{1.31})$	$O(N^{1.21})$
			64	3,248	153	$O(N^{1.35})$	$O(N^{1.22})$
			128	3,486	128	$O(N^{1.38})$	$O(N^{1.23})$
BCSPWR10	5,300	20,596	128	26,234	578	$O(N^{1.51})$	$O(N^{1.23})$
			256	29,420	476	$O(N^{1.55})$	$O(N^{1.24})$
			512	38,880	511	$O(N^{1.62})$	$O(N^{1.24})$
NiMo	9,430	14,001	128	20,398	553	$O(N^{1.22})$	$O(N^{1.19})$
			256	20,706	402	$O(N^{1.30})$	$O(N^{1.20})$
			512	21,375	313	$O(N^{1.31})$	$O(N^{1.20})$

Table 1: Summary Statistics

for Choleski factorization; however, parallelism in sparse Choleski algorithms is heavily dependent on the available parallelism in the actual sparse matrix. In section 8.3, it will be shown that the additional computations in BCSPWR10 (when compared to BCSPWR09) permits reasonable efficiencies when factoring this matrix with four processors. Meanwhile, the solver for BCSPWR09 simply does not have adequate calculations to overcome the effects of limited amounts of calculations.

The matrix from Niagara Mohawk, labeled *NiMo* in table 1, has similar computational complexity as BCSPWR09, however, it has nearly 5.5 times as many diagonal elements and 4.7 times as many non-zero elements and fillin. Careful examination of figures 27 and 28 show that for four processors, the load balancing has placed a single diagonal block on one of four processors. This block has only 247 nodes, however, it has 62% of the factorization operations. The computation complexity of this block is $O(N^{2.08})$ for factorization and $O(N^{1.61})$ for the triangular solution phase. It is possible to divide that diagonal matrix block by decreasing the maximum size of the diagonal blocks, Unfortunately, when the maximum size of a diagonal block is reduced from 247 to 128, two situations arise that generate less than optimal solutions. First, the size of the last block increases by 77%, which increases the number of computations in the last block by over 550%. While the last diagonal block is solved in parallel, the pipelined parallel dense Choleski solver used is not sufficiently scalable to be able to overcome such a large increase in number of operations by applying additional processors. Second, even when the size of the diagonal blocks are limited to a size that forces this block in question to be divided into separate matrices, the number of operations in a single block remains 31% of the total number of operations in the

diagonal blocks and borders. Load imbalance would occur at greater than three processors.

Further examination of table 1 shows that the computational complexity of the forward reduction and backward substitution phases are similar in magnitude regardless of the choice of matrix. It is difficult to obtain good speedup on large multi-processors for algorithms with strong precedence in the calculations, especially when the computational complexity is so close to unity.

8.3 Empirical Results — Sparse Choleski Solver

The performance of the block-diagonal-bordered Choleski algorithm was tested with the following maximum diagonal block size for the respective matrices:

- BCSPWR09 — $max_{DB} = 128$
- BCSPWR10 — $max_{DB} = 256$
- Niagara Mohawk data — $max_{DB} = 512$

Examples of each of these block-diagonal-bordered sparse matrices were presented in section 8.2 with load balancing for four processors. These ordered matrices have been used because they offered the best performance for a particular load-flow matrix. Performance data has been collected for individual operations to examine the speedup of each distinct portion of the algorithm. The data was collected in such a manner as to have no impact on the overall measures of performance. Performance data has been collected while running the sparse block-diagonal-bordered solver on from one to sixteen CM-5 node processors. Performance of the multi-processor algorithms are illustrated using graphs plotting relative speedup versus the number of processors. Relative speedup has been defined above in section 8.1.

Graphs of speedup calculated from empirical performance data are provided in figures 29 through 31 for the three matrices. Each figure has a family of three curves that show speedup for:

1. Choleski factorization
2. forward reduction and backward substitution
3. factorization and a single forward reduction and backward substitution

In general these speedup graphs illustrate that for the real power system load-flow matrices from the Boeing-Harwell series and the Niagara Mohawk data, speedup appears to be limited to approximately 2.7 regardless of the number of processors used to solve the problem. Careful analysis of the performance data shows that the following data trends.

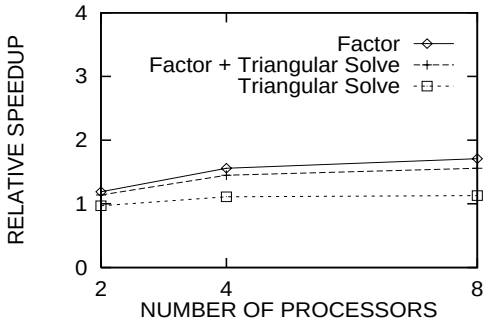


Figure 29: Speedup for BCSPWR09 Data — 2, 4, and 8 processors

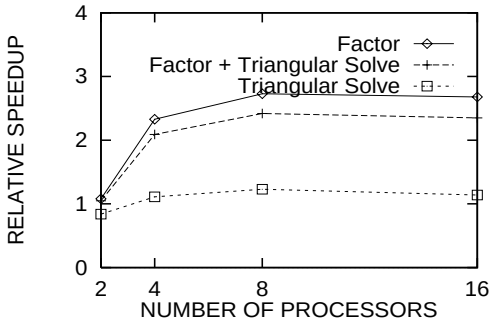


Figure 30: Speedup for BCSPWR10 Data — 2, 4, 8, and 16 processors

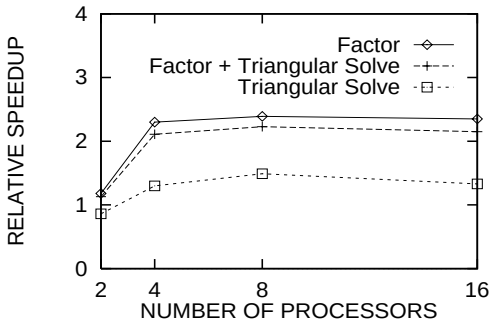


Figure 31: Speedup for Niagara Mohawk Data — 2, 4, 8, and 16 processors

For all three matrices, the performance data showed no significant difference between load balancing on the number of operations in factorization or the number of operations in the triangular solution. The time to factor, forward reduce, or backward substitute the data in the last diagonal block dominates the calculations for these matrices because of the low order of computational complexity in the diagonal block partitions of the matrix. These algorithms use pipelined communications that *data-starve* quickly with the small sizes of the last diagonal blocks. On the other hand, solutions with different values of the maximum diagonal block size yield larger last diagonal blocks. However, while better speedups are achieved, the actual run times are not improved.

The performance of the block-diagonal-bordered sparse Choleski solver using the BCSPWR10 matrix shows some improvement when compared to the solver performance on the BCSPWR09 matrix. The additional operations in the BCSPWR10 matrix permit improvements in speedup and efficiencies of 60% for four processors, although speedup remains nearly constant while additional processors are used. The Niagara Mohawk data offered a considerably larger matrix, although the number of calculations were less than the BCSPWR10 matrix and this matrix suffered from load imbalance even for four processors. The time to factor the mutually independent diagonal blocks showed that this portion of the calculations are embarassingly parallel for the two matrices from the Boeing-Harwell series. In the Niagara Mohawk data, the time to factor the the mutually independent blocks stays constant for four or more processors. As discussed above, attempts at selecting the maximum size of the diagonal blocks to minimize the load imbalance caused the size of the last diagonal block to increase so significantly that subsequent execution times were greater than for a matrix ordering that suffered with load imbalance.

9 Conclusions

In general, solving load-flow matrices from real-power systems has proven to be a very difficult challenge for parallel sparse Choleski solvers. As we have illustrated, the computational complexity of the actual data matrices is sufficiently low, even for matrix factorization, that the speedup performance of the parallel block-diagonal-bordered Choleski solver is limited by computational starvation and in some instances, load imbalance.

In this paper we present research into parallel block-diagonal-bordered sparse Choleski factorization algorithms developed with special considerations to irregular sparse matrices originating in the electrical power systems community. Available parallelism in the block-diagonal-bordered matrix structure offers promise for simplified implementation and also offers a simple decomposition of the problem into clearly identifiable subproblems. Parallel block-diagonal-bordered Choleski solvers require a three step preprocessing phase that is

reusable for static matrices. The matrix is ordered into block-diagonal-bordered form, pseudo-factored to identify the location of all fillin and obtain operations counts in the mutually independent diagonal blocks and corresponding portions of the borders, and the load-balanced to uniformly distribute operations (when possible).

We developed an implementation that offered efficiencies of 60% for Choleski factorization with four processors, although the implementation was not efficient beyond four processors with any of the power system load-flow matrices examined. Further examinations into techniques to better solve the last diagonal block could significant improve performance. While a dense Choleski solver was used in this implementation, due to the available parallelism in block-diagonal-bordered form matrices, any technique can be used to solve this sub-matrix, including iterative techniques.

The parallel block-diagonal-bordered Choleski algorithm, presented in this paper, addresses one of the most difficult power systems applications to implement on a multiprocessor. Load-flow has the smallest matrices and the fewest calculations due to symmetry and lack of requirements for pivoting to ensure numerical stability. In the near future, we will investigate applying block-diagonal-bordered LU factorization to transient stability analysis simulations. These matrices have substantially more available parallelism due to the increased number of the diagonal blocks corresponding to the addition of the generator equations to the block-diagonal-bordered network equations.

Acknowledgments

We thank Alvin Leung, Kamala Anupindi, Nancy McCracken, Paul Coddington, and Tony Skjellum for their assistance in this research. This work has been supported in part by Niagara Mohawk Power Corporation, the New York State Science and Technology Foundation, the NSF under co-operative agreement No. CCR-9120008, and ARPA under contract #DABT63-91-K-0005.

References

[1] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. Technical Report RNR-92-033, NASA Ames Research Center, November 1992.

[2] A. R. Bergen. *Power Systems Analysis*. Prentice-Hall, 1986.

[3] J. J. Dongarra, D. C. Sorensen I. S. Duff, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, 1990.

[5] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR/PA/92/86, Boeing Computer Services, October 1992. (available by anonymous ftp at orion.cerfacs.fr).

[6] E. Anderson, et. al. *LAPACK User's Guide*. SIAM, Philadelphia, 1992.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[8] A. George and E. Eg. Some Shared Memory is Desirable in Parallel Sparse Matrix Computation. *SIGNUM Newsletter*, 23(2):9–13, April 1988.

[9] A. George, M. T. Heath, J. Liu, and E. Ng. Solution of Sparse Positive Definite Systems on a Shared-Memory Multiprocessor. *International Journal of Parallel Programming*, 15(4):309–328, August 1986.

[10] A. George, M. T. Heath, J. Liu, and E. Ng. Sparse Cholesky Factorization on a Local-Memory Multiprocessor. *SIAM journal on Scientific and Statistical Computing*, 9(2):327–340, March 1988.

[11] A. George, M. T. Heath, J. Liu, and E. Ng. Solution of Sparse Positive Definite Systems on a Hypercube. *Journal of Computational and Applied Mathematics*, 27:129–156, 1989.

[12] A. George and J. Liu. The Evolution of the Minimum Degree Ordering Algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[13] H. H. Happ. Diakoptics - The Solution of System Problems by Tearing. *Proceedings of the IEEE*, 62(7):930–940, July 1974.

[14] M. T. Heath, E. Ng, and B. W. Peyton. Parallel Algorithms for Sparse Linear Systems. In *Parallel Algorithms for Matrix Computations*, pages 83–124. SIAM, Philadelphia, 1991.

[15] D. P. Koester, S. Ranka, and G. C. Fox. Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Electrical Power System Applications. In A. Skjellum, editor, *Proceeding of the Scalable Parallel Libraries Conference*. IEEE Press, 1994.

[16] V. Pan. Parallel Solution of Sparse Linear and Path Systems. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 14. Morgan Kaufmann, San Mateo, CA, 1993.

[17] A. Pothen, H. Simon, and K.. P. Liou. Partitioning Sparse Matrices with Eigenvalues of Graphs. *SIAM J. Mat. Anal. Appl.*, 11(3):pp. 430–452, 1990.

[18] R. A. Saleh, K. A. Gallivan, M. Chang, I. N. Hajj, D. Smart, and T. N. Trick. Parallel Circuit Simulation on Supercomputers. *Proceedings of the IEEE*, 77(12):1915–1930, December 1989.

[19] A. Sangiovanni-Vincentelli, L. K. Chen, and L. O. Chua. Node-Tearing Nodal Analysis. Technical Report ERL-M582, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, October 1976.

[20] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. Technical Report RNR-91-008, NASA Ames Research Center, February 1991.

[21] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual,: Preliminary Documentation for Version 3.0 Beta*, December 1992.

[22] S. Venugopal and V. K. Naik. Effects of Partitioning and Scheduling Sparse Matrix Factorization on Communications and Load Balance. NASA Contractor Report 189563 ICASE Report No. 91-80, NASA, Langley Research Center, October 1991.

[23] S. Venugopal and V. K. Naik. SHAPE: A Parallelization Tool for Sparse Matrix Computations. Research Report RC 17899 (77448), IBM Research Division, T. J. Watson Research Center Yorktown Heights, NY 10598, January 1992.

[24] S. Venugopal and V. K. Naik. Towards Understanding Block Partitioning for Sparse Cholesky Factorization. Research Report RC 18666 (80517), IBM Research Division, T. J. Watson Research Center Yorktown Heights, NY 10598, October 1992.

[25] S. Venugopal and V. K. Naik. Towards Understanding Block Partitioning for Sparse Cholesky Factorization. pages 792–796, April 1993.

[26] S. Venugopal, V. K. Naik, and J. Saltz. Performance of Distributed Sparse Cholesky Factorization with Pre-scheduling. Research Report RC 18623 (78732), IBM Research Division, T. J. Watson Research Center Yorktown Heights, NY 10598, April 1992.

[27] G. von Laszewski, M. Parashar, A. G. Mohamed, and G. C. Fox. High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures. Technical Report SCS-271, Northeast Parallel Architectures Center, Syracuse University, 1992.

[28] G. von Laszewski, M. Parashar, A. G. Mohamed, and G. C. Fox. On the Parallelization of Blocked LU Factorization Algorithms on Distributed Memory Architectures. Technical Report SCS-271b, Northeast Parallel Architectures Center, Syracuse University, June 1992.

[29] Y. Wallach. *Calculations and Programs for Power System Networks.* Prentice-Hall, 1986.

[30] M. Zubair and M. Ghose. A Performance Study of Sparse Cholesky Factorization on the INTEL iPSC/860. NASA Contractor Report 189634 ICASE Report No. 92-13, NASA, Langley Research Center, March 1992.

A Minimum-Degree Ordering

Minimum-degree ordering has been used in our research in a two-fold manner:

1. to order symmetric power system admittance matrices to provide baseline orderings with which to compare the performance of other ordering techniques
2. to order the independent sub-matrices in recursive spectral bisection and node-tearing ordering techniques

Minimum degree ordering is a greedy algorithm that selects a node with a minimum number of connected edges in the graph for factoring next. This algorithm is not optimal because truly efficient techniques do not exists to resolve *ties* and numerous rows have equal numbers of elements. The minimum-degree ordering algorithm is based on the iterative application of the following equation to solve for i for all rows in a matrix:

$$r_i^{(k)} = \min_t r_t^{(k)}, \tag{34}$$

where:

$r_i^{(k)}$ is the number of variables in row i when factoring the k^{th} row.
$r_t^{(k)}$ is the number of variables in row t when factoring the k^{th} row

When factoring the k^{th} row, the row with the minimum number of variables is selected, moved by elementary row and column exchange rules to the k^{th} row, and then factored. Algorithms to implement this iterative formula are best described using the graph theoretical explanation of fillin presented in figure 5. Let G be an undirected graph and ν a node in G, then let $\Lambda_G(\nu)$ describe the set of nodes adjacent to ν and let $|\Lambda_G(\nu)|$ represent the degree of node ν. The last concept required to develop a concise minimum-degree algorithm is the concept of an *elimination graph* [12]. Given a graph G, the elimination graph G_ν is the resulting graph after the node ν is factored. Elimination graphs get their name because of the close relationship of LU factorization and Gaussian elimination. The rudimentary minimum-degree algorithm used throughout this work is presented in figure 32. The outer loop examines each node in the graph, and the inner loop searches through all remaining nodes in the present graph to select a node with the minimum degree. After a minimum-degree node is selected, the edges at adjacent nodes must be updated to reflect factorization. As illustrated in figure 5, the addition of new edges in the elimination graph G_ν is limited to those nodes in $\Lambda_G(\nu)$. For $\upsilon \in \Lambda_G(\nu)$, then

$$\Lambda_{G_\nu}(\upsilon) = (\Lambda_G(\upsilon) \cup \Lambda_G(\nu)) - \{\nu, \upsilon\}. \tag{35}$$

$G \leftarrow$ *the symmetric graph representing the sparse matrix*
while $G \neq \phi$ **do**
    select a node $\nu \in G$ *with minimum degree*
    order ν *next*
    /* calculate the elimination graph G_ν */
    for all *nodes* $v \in \Lambda_G(\nu)$
        $\Lambda_{G_\nu}(v) \leftarrow (\Lambda_G(v) \cup \Lambda_G(\nu)) - \{\nu, v\}$
    end for
    $G \leftarrow G_\nu$
end while

Figure 32: The Minimum-Degree Algorithm

Given the two nested loops that can examine all nodes in the original sparse graph, the computational order of this algorithm is $O(n^2)$, although a significant portion of the workload is required to calculate the elimination graph G_ν [12]. As stated above, in formula 4, the total amount of calculations in the loop to update the elimination graph G_ν is bounded by the binomial coefficient of *the number of edges at a node choose 2* or $|\Lambda_G(\nu)|$ *chose 2*. See equation 4 for details on calculating the binomial coefficient. It is important to note that the location of all fillin can be determined when using this classical implementation of minimum degree ordering.

This version of the minimum-degree algorithm has been used in our research in a two-fold manner: to order symmetric power systems admittance matrices to provide baseline orderings with which to compare the performance of other ordering techniques, and to order the independent sub-matrices obtained with node-tearing ordering techniques.

B A Node-tearing Example

An example illustrating node-tearing nodal analysis is presented in figures 33 through 36. The example graph, presented in figure 33, has two distinct portions connected at node ν_4. Node ν_1 meets the selection criteria for the first node, and the contour tableau is presented in figure 34. There is a distinct local minimum in the contour number at c_4 which identifies node ν_4 as the node that couples the two mutually independent graph partitions. Figure 35 illustrates the ordered graph, note that only the labels on the modes have changed from figure 33. To illustrate the effect of ordering the matrix, the matrix sparsity structure for the original and ordered graphs are presented in figure 36. In these figures, original data values are represented with + symbols while fillin are denoted with F characters. Within the sub-blocks, the values would be ordered with a minimum-degree ordering algorithm. For this sample matrix, minimum degree ordering for the entire matrix would yield the same results.

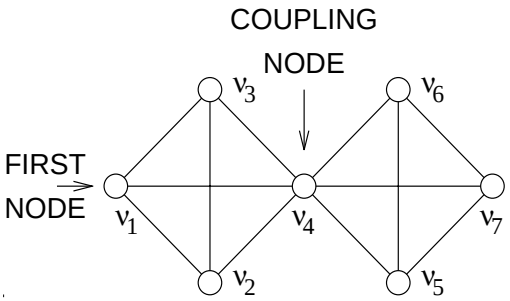


Figure 33: Graph for a Node-Tearing Example

	Iterating Sets	Adjacency Sets	Contour Number
1	$\{\nu_1\}$	$\{\nu_2, \nu_3, \nu_4\}$	3
2	$\{\nu_1, \nu_2\}$	$\{\nu_3, \nu_4\}$	2
3	$\{\nu_1, \nu_2, \nu_3\}$	$\{\nu_4\}$	1
4	$\{\nu_1, \nu_2, \nu_3, \nu_4\}$	$\{\nu_5, \nu_6, \nu_7\}$	3
5	$\{\nu_1, \nu_2, \nu_3, \nu_4, \nu_5\}$	$\{\nu_6, \nu_7\}$	2
6	$\{\nu_1, \nu_2, \nu_3, \nu_4, \nu_5, \nu_6\}$	$\{\nu_7\}$	1
7	$\{\nu_1, \nu_2, \nu_3, \nu_4, \nu_5, \nu_6, \nu_7\}$	$\{\phi\}$	0

Figure 34: Example Contour Tableau

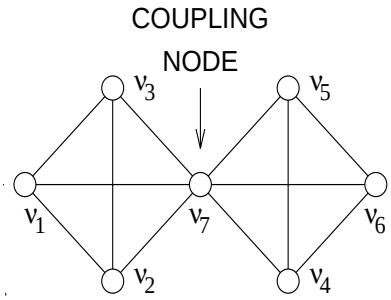


Figure 35: Relabeled Example Graph

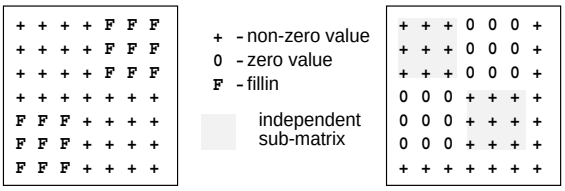


Figure 36: Matrix Representation of the Example Graphs