Parallel Direct Methods for 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 679

October 11, 1995

Contents

1 Introduction 1

1.1 Block-Diagonal-Bordered Power System Matrices 1
1.2 Block-Diagonal-Bordered Direct Linear Solvers 2
1.3 Active Messages 3
1.4 Embedded Software Applications 3
1.5 Overview 4

2 Power System Applications 4

2.1 Load-Flow Analysis 4
2.2 Transient Stability Analysis 5
2.3 Power System Network Matrices 6
2.4 Direct Methods and Applications 9

3 Direct Linear Solvers 9

3.1 LU Factorization 9
3.2 Choleski Factorization 11
3.3 Ordering Sparse Matrices 13
3.4 A Survey of the Literature 14

4 Available Parallelism in Block-Diagonal-Bordered Form Matrices 16

5 The Three-Step Preprocessing Phase 21

5.1 Ordering 22
5.2 Pseudo-factorization 23
5.3 Load Balancing 24

6 Node-tearing Nodal Analysis 25

6.1 The Node-tearing Algorithm 25
6.2 The Node-tearing Implementation 28

7 Sparse Matrix Solver Implementations 29

7.1 The Hierarchical Data Structure 30
7.2 The Parallel Blocked-Diagonal-Bordered LU Factorization Algorithm 33
7.3 Forward Reduction and Backward Substitution Algorithms 40

8 Empirical Results 44

8.1 Empirical Results — Ordering Power Systems Network Matrices into Block-Diagonal-Bordered Form 47
8.2 Empirical Results — Parallel Direct Sparse Solver Performance 66
8.2.1 Selecting Partitioned Matrices with *Best* Parallel Solver Performance 67
8.2.2 Comparing Timing Performance for Direct Solver Implementations 69
8.2.3 Examining Speedup 72

8.2.4 Analyzing Algorithm Component Performance 75
8.2.5 Comparing Communications Paradigms 81
8.3 Empirical Results — Conclusions 84
8.3.1 Algorithm Performance on an IBM SP1 and SP2 84
8.3.2 Algorithm Performance on Future SPP Architectures 84

9 Conclusions **87**

A Minimum-Degree Ordering **94**

B A Node-tearing Example **96**

Abstract

This paper presents research into parallel direct methods for block-diagonal-bordered sparse matrices — LU factorization and Choleski factorization algorithms developed with special consideration for irregular sparse matrices from the electrical power systems community. Direct block-diagonal-bordered sparse linear solvers exhibit distinct advantages when compared to general direct parallel sparse algorithms for irregular matrices. 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. 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. This ordering relates to more general sparse direct solver algorithms that use elimination trees — however, this algorithm develops an elimination tree with only two-levels of supernodes. Matrix orderings are performed using a diakoptic technique based on node-tearing-nodal analysis, with load balancing to optimize performance when factoring the diagonal blocks and borders, the lower layer of supernodes in the elimination tree. Empirical performance measurements for real power system networks are presented for implementations of a parallel block-diagonal-bordered LU algorithm and a similar Choleski algorithm run on a distributed memory Thinking Machines CM-5 multi-processor. The algorithms presented here requires active message remote procedure calls in order to minimize communications overhead and obtain good relative speedup. The paradigm used with active messages greatly simplified the implementation of these sparse matrix algorithms.

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 [20]. 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. This additional parallelism is often described by *elimination trees* [14, 15, 16, 20, 37, 38, 39, 40, 41, 45], graphs that illustrate the dependencies in the calculations. 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 many real-world sparse matrices 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 [8, 18, 32].

We have focused on developing parallel linear solvers optimized for sparse matrices from the power systems community — in particular, we have examined linear solvers for matrices resulting from power distribution system networks. These matrices are some of the most sparse matrices encountered in real-life applications, and these matrices also are irregular. Recently, [18, 22] have reported scalable Choleski solvers, but they are for matrices that have more row/columns, that have more nonzero elements per row/column, and that are more regular than power systems matrices. When scalability of sparse linear solvers is examined using real, irregular sparse matrices, the available parallelism in the sparse matrix and load-imbalance overhead can be as much the reason for poor parallel efficiency as the parallel algorithm or implementation [26, 32].

1.1 Block-Diagonal-Bordered Power System Matrices

Power system distribution networks are generally hierarchical with limited numbers of high-voltage lines transmitting electricity to connected local networks that eventually distribute power to customers. In order to ensure reliability, highly interconnected local networks are fed electricity from multiple high-voltage sources. Electrical power grids have graph representations which in turn can be expressed as matrices — electrical buses are graph nodes and matrix diagonal elements, while electrical transmission lines are graph edges which can be represented as non-zero off-diagonal matrix elements. We show that it is possible to identify the hierarchical structure within a power system matrix using only the knowledge of the interconnection pattern by *tearing* the matrix into partitions and coupling equations that yield a block-diagonal-bordered matrix. Node-tearing-based partitioning identifies the basic network structure that provides parallelism for the majority of calculations within the direct solution of a linear system.

In this paper we examine the applicability of parallel direct block-diagonal-bordered sparse solvers for real power system applications that require either the solution of symmetric positive definite

sparse matrices or location symmetric sparse matrices that result from solving problems relating to power systems networks. Variations of this technique could be used to solve other power system sparse linear systems such as those that result from solving linearized differential-algebraic equations that result from transient stability analysis or small-signal stability assessments. The implementations we describe in this paper work directly with the equations resulting from the power systems network, the smallest class of power system matrix.

The implementations we developed can be used to solve symmetric positive definite load flow analysis Jacobian matrices or position symmetric network matrices from transient stability analysis. In spite of only examining direct linear solver implementations that solve relatively small network-related matrices, we have been able to obtain good parallel speedups. We expect that even better performances would be possible for parallel implementations designed to solve a single system of linear equations that represent a combination of the generator dynamical equations and network equations in a linearized form of the differential-algebraic equations from transient stability analysis or small-signal analysis. For these problems, there is additional parallel calculations with no additional parallel communications overhead.

1.2 Block-Diagonal-Bordered Direct Linear Solvers

Block-diagonal-bordered sparse matrix algorithms require modifications to the normal preprocessing phase described in numerous papers on parallel Choleski factorization [14, 15, 16, 20, 37, 38, 39, 40, 41, 45]. 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.

Parallel block-diagonal-bordered sparse linear solvers offer the potential for regularity often absent from other parallel sparse solvers [23, 24, 25, 27]. 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, and load-balancing is sufficiently effective that relative speedups greater than ten have been observed in empirical performance measurements for 32 processors on a Thinking Machines CM-5 multi-processor.

In addition to the promising speedup encountered for only the parallel direct linear solver, other dimensions exist in electrical power system applications that can be exploited to efficiently make use of multi-processors with greater than 32 processors. We believe that this research also has utility for other irregular sparse matrix applications where the data is hierarchical, very sparse, and irregular. Other sources of hierarchical matrices exist, for example, electrical circuits, that have the potential for larger numbers of equations than power system matrices.

1.3 Active Messages

The algorithms we developed require active message remote procedure calls in order to minimize communications overhead and obtain good relative speedup. Active message remote procedure calls (RPCs) provide protocol-less access to the transport layer of the communication network on the CM-5. The user must assume responsibilities for all aspects of communications — as a result, active message RPCs provide very low-latency communications. Active messages provide extremely fast interprocessor communications, and also permit a parallel-code development paradigm that greatly simplifies the implementation of these sparse matrix algorithms. Empirical data has been collected on both active message-based implementations and more traditional cooperative buffer-based message passing commands in order to illustrate the need for low latency communications when solving matrices that are as sparse and irregular as power systems matrices.

Active messages are an example of a distributed-memory multiprocessor message-passing paradigm that has been developed by the supercomputer community to provide extremely low latency communications. All other CM-5 message-passing software is built upon this low latency protocol-less remote procedure call. The empirical data presented in this paper clearly shows that low-latency communications are required for direct linear solvers for power systems network matrices. This research has been performed on a small version (only 32 processors) of a massively parallel processing (MPP) architecture, the Thinking Machines CM-5. We believe that scalable parallel processing (SPP) architectures, like the IBM SP2, may eventually provide similar low-latency communications for short messages, in addition to expanded network bandwidth, because there are many parallel algorithms that can only be implemented efficiently with this type of interprocessor communications support. SPP hardware developers recognize that low-latency communications increase the utility of their computer and, consequently, improve market potential.

1.4 Embedded Software Applications

Our research has examined the performance of a block-diagonal-bordered direct solvers, with implementations of both Choleski and LU factorization, 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 factorization algorithms. Due to the reduced amount of calculations in the triangular solution phases of solving a system of factored linear equations, these algorithms are often ignored when parallel Choleski or LU factorization algorithms are presented in the literature.

In our research, we have found that the development of parallel factorization algorithms must consider forward reduction and backward substitution, because the choice of the order of calculations in factorization can greatly influence the performance of the parallel triangular solutions. Data structures are dependent on the order of calculations, in order to ensure cache coherency, and the amount of communications in parallel forward reduction and backward substitution is dependent on the data layout. We have found that the results of additional communications overhead can eliminate any potential speedup for parallel forward reduction with column oriented data storage. This communications overhead cannot be eliminated for Choleski factorization, where either forward reduction or backward substitution must be performed with an implicit transpose of the factored

matrix. Fortunately, the LU factorization algorithm can be implemented in a manner to eliminate this communications overhead problem.

1.5 Overview

This paper is organized as follows. In section 2, we describe the electrical power system applications that are the basis for this work. In section 3, we briefly review direct solution techniques for factorization and forward reduction/backward substitution, and we review the literature concerning general parallel LU and Choleski factorization algorithms. This is followed by a theoretical derivation of the available parallelism in both the 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 the process of 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 introduce 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 LU and Choleski algorithms that has been implemented on the CM-5. Analysis of the performance of these ordering techniques are presented in section 8 for actual power system network matrices from the Boeing-Harwell series, the Electrical Power Research Institute (EPRI), and an electrical utility, the Niagara Mohawk Power Corporation. We present our conclusions concerning parallel block-diagonal-bordered direct linear 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, 44]. 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 multi-processor 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.

2.1 Load-Flow Analysis

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 [44]. 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 with 1,723, 5,300, 6,692, 1,766, and 9,430 nodes.

2.2 Transient Stability Analysis

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 electro-mechanical equations of motion of the interconnected generators [2]. Transient stability analysis can be used to perform selective detailed analyses of generator commitment stability, 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 in 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.

It is also possible to solve the above equations by decoupling the generator equations from the network equations. For decoupled transient stability analysis, the transient stability differential-algebraic equation matrix is partitioned into the four submatrices. The generator equations are solved independently of the network equations, then the sparse admittance matrix is modified by the matrix coefficients in the sparse borders. Instead of the common practice of decoupling the generator and network calculations in a transient stability simulation, we hope to continue this research and eventually examine using more powerful differential-algebraic equation solvers for transient stability analysis that do not decouple the generator and network equations. The fully-coupled differential-algebraic equations will offer more potential for good parallel performance 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.

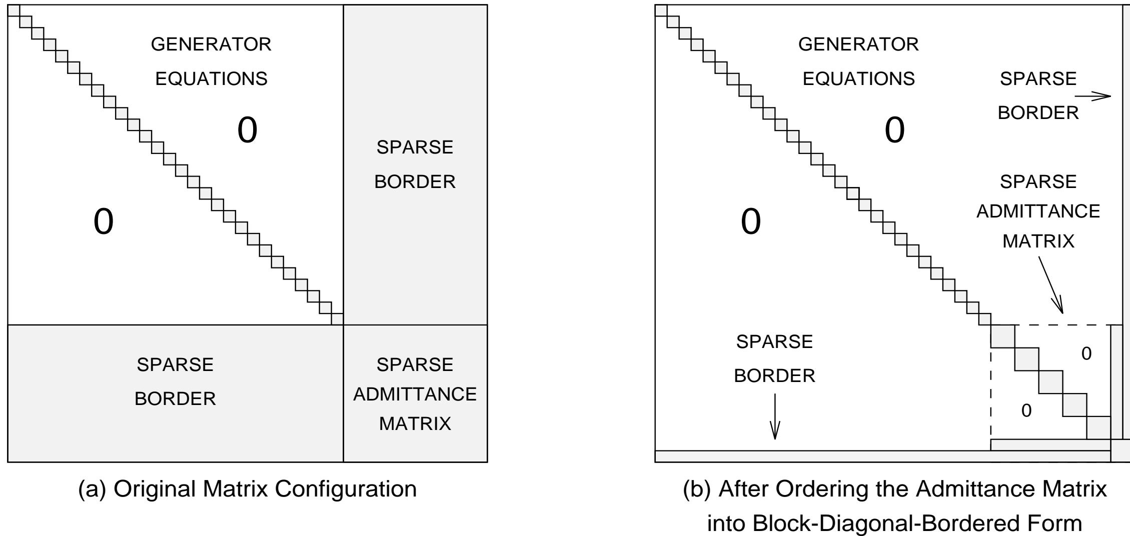

Figure 1: Ordering the Admittance Sub-Matrix in the Transient Stability Differential-Algebraic Equations

The amount of work available will be greater and the effects of load-balance overhead will be minimized, while the amount of communications overhead will remain the same as solving for the decoupled transient stability equations.

2.3 Power System Network Matrices

Power system distribution networks are generally hierarchical with limited numbers of high-voltage lines transmitting electricity to connected local networks that eventually distribute power to customers. In order to ensure reliability, highly interconnected local networks are fed electricity from multiple high-voltage sources. Electrical power grids have graph representations which in turn can be expressed as matrices — electrical buses are graph nodes and matrix diagonal elements, while electrical transmission lines are graph edges which can be represented as non-zero off-diagonal matrix elements.

Matrices representing power system networks are some of the most sparse matrices encountered throughout the academic or industrial community. Figure 2 illustrates the proportion of graph nodes with a particular number of graph edges or the number of non-zero values in a matrix row or column for five separate power system matrices:

- Boeing-Harwell matrix BCSPWR09 — 1,723 nodes and 2,394 graph edges [10],
- Boeing-Harwell matrix BCSPWR10 — 5,300 nodes and 8,271 graph edges [10],
- EPRI matrix EPRI6K matrix — 6,692 nodes and 10,535 graph edges [11],
- Niagara Mohawk Power Corporation operations matrix NiMo-OPS — 1,766 nodes and 2,506 graph edges,

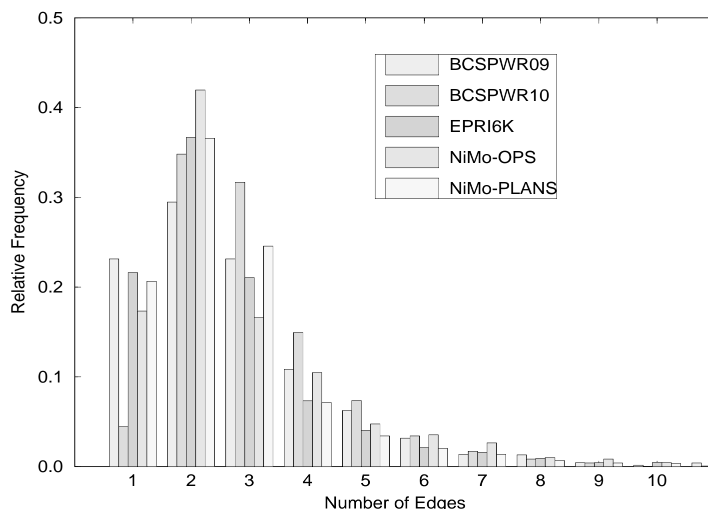

Figure 2: Electrical Power System Networks — Relative Frequency Histogram of Edges per Graph Node

- Niagara Mohawk Power Corporation planning matrix NiMo-PLANS — 9,430 nodes and 14,001 graph edges.

Matrices BCSPWR09 and BCSPWR10 are from the Boeing Harwell series and represent electrical power system networks from the Western and Eastern US respectively. The EPRI-6K matrix is distributed with the Extended Transient-Midterm Stability Program (ETMSP) from EPRI. Matrices NiMo-OPS and NiMo-PLANS have been made available by the Niagara Mohawk Power Corporation, Syracuse, NY.

In this relative frequency histogram, the most frequently occurring number of edges per node is *only 2!* Table 1 provides additional data to illustrate that power system matrices are both relatively small in size and also have the fewest average edges per node of available matrices. In this table, all data except that from EPRI and Niagara Mohawk are from the Boeing-Harwell series [10]. The structural matrices, BCSSTK13 to BCSSTK32, are frequently used in papers to benchmark parallel sparse linear algorithms [14, 15, 16, 18, 20, 22, 31, 32, 37, 38, 39, 40, 41, 45]. For power system matrices, the average number of edges per node is less than two while for many of the structural matrices, the average number of nodes per edge is greater than ten. Also the number of nodes in power system matrices are limited when compared to the Boeing-Harwell structural matrices.

While power systems matrices are extremely sparse, they are also irregular, with the larger matrices having some nodes with greater than twenty edges. The histogram presented in figure 2 has been truncated at ten edges per node to emphasize the high incidence of edges with less than three nodes. As a result of the degree of sparsity and irregularity in these matrices, developing parallel sparse linear solvers for power systems application has proven to be a challenge [36, 4]. Nevertheless, by developing parallel algorithms that actively address the irregular nature of the graphs with explicit load-balancing and by making all necessary communications as balanced, regular, and as asynchronous as possible, we will show in section 8 that our block-diagonal-bordered approach to addressing linear solvers for power system applications can yield respectable speedups even for as many as 32 processors.

Graph Name	Description	Number of Nodes	Number of Edges	Average Edges per Node
BCSPWR09	Western US Power Network	1,723	2,394	1.39
BCSPWR10	Eastern US Power Network	5,300	8,271	1.56
EPRI6K	Power Network	6,692	10,535	1.57
NiMo-OPS	Eastern US Power Network	1,766	2,506	1.41
NiMo-PLANS	Eastern US Power Network	9,430	14,001	1.48
BCSSTK13	Fluid Flow Generalized Eigenvalues	2,003	40,940	20.44
BCSSTK14	Roof of Omni Coliseum, Atlanta	1,806	30,824	17.07
BCSSTK15	Module of an Offshore Platform	3.948	56,934	14.42
BCSSTK16	Corp of Engineers Dam	4.884	142,747	29.23
BCSSTK17	Elevated Pressure Vessel	10,974	208,838	19.03
BCSSTK18	R.E.Ginna Nuclear Power Station	11,948	68,571	5.74
BCSSTK24	Calgary Olympic Saddledome Arena	3.562	78,174	21.95
BCSSTK25	76 Storey Skyscraper	15,439	118,401	7.67
BCSSTK28	Solid Element Model	4,410	107,307	24.33
BCSSTK29	Boeing 767 rear pressure bulkhead	13,992	302,748	21.64
BCSSTK30	Off-Shore Generator Platform	28,924	1,007,284	34.83
BCSSTK31	Automobile Component	35,588	572,914	16.10
BCSSTK32	Automobile Chassis	44,609	985,046	22.08

Table 1: Comparison of Power System Matrices and Boeing-Harwell Structural Matrices

2.4 Direct Methods and Applications

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. Parallel Choleski algorithms have the same amount of interprocessor communications overhead as parallel LU algorithms; meanwhile, there are twice as many floating-point operations available in LU factorization. This means that relative speedup, or the improvement in performance when a problem is solved on multiple processors, will be better for LU factorization because there the additional calculations attenuate the many sources of overhead in the parallel algorithm, especially communications overhead.

The parallel block-diagonal-bordered LU algorithm, also presented later in this paper, would be appropriate to use for solving the position symmetric matrix that occurs in transient stability analysis, when the differential equations representing the generator dynamics are solved decoupled from the linear network equations. The system of linear equations in this application is similar to that encountered in load flow, but with the values at generator buses modified to represent the effects of the dynamic state of the generators. This matrix is position symmetric, with the same structure as a load flow matrix, but values in symmetric locations may not be equal. Decoupled transient stability analysis encounters the same small matrix sizes of load-flow analysis, but there are nearly twice the number of calculations for double precision LU factorization and six times the number of calculations for complex-variate LU factorization.

3 Direct Linear Solvers

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 [9].

3.1 LU Factorization

For a brief review, the sparse matrix A can be numerically factored into a lower triangular matrix L and an upper triangular matrix U as in equation 2, where all values on the diagonal of either L or U must equal 1 — $L_{k,k} = 1$ or $U_{k,k} = 1$. Equation 2 is solved by setting $Ux = y$, and substituting y for Ux. 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 $Ux = y$. Triangular

```
for k = 1 to N /* for all elements along the diagonal */
    for each i ∈ [k, N]
        for each j ∈ [1, k − 1] such that A_{i,j} ≠ 0 and A_{j,k} ≠ 0
            A_{i,k} ← A_{i,k} − (A_{i,j} * A_{j,k})
        endfor
    endfor
    for each i ∈ [k + 1, N]
        A_{k,j} ← (A_{k,j}/A_{k,k})
    endfor
    for each j ∈ [k + 1, N]
        for each i ∈ [1, k − 1] such that A_{k,i} ≠ 0 and A_{i,j} ≠ 0
            A_{k,j} ← A_{k,j} − (A_{k,i} * A_{i,j})
        endfor
    endfor
endfor
```

Figure 3: Sparse LU Factorization - Doolittle Algorithm

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. Additional discussions on the state of the literature for LU factorization are presented below.

Sparse LU factorization can mirror any similar dense 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 factorization by *ordering* the matrix before performing the factorization if there is no requirement for pivoting to ensure numerical stability of the calculations [9]. There are many ordering techniques for position symmetric matrices, with one of the most common being *minimum degree ordering*. If pivoting is required to ensure numerical stability, a Markowitz ordering/pivoting strategy can be employed, and fillin determined during the solution of the matrix. The Markowitz ordering strategy selects pivots with the added constraint of minimizing fillin [9]. Additional discussions on the state of the literature for LU factorization are presented below.

As we continue the review of LU factorization, we present a general sequential sparse factorization algorithm, in figure 3, based upon the factorization algorithms commonly attributed to Doolittle for matrices that do not require pivoting. In Doolittle factorization, all values on the diagonal of L equal 1 — $L_{k,k} = 1$. We also present general sequential sparse forward reduction and backward substitution algorithms in figures 4 and 5 respectively that would be used in conjunction with the Doolittle-based algorithm to solve for x in $Ax = b$.

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

Figure 4: Sparse Forward Reduction for Doolittle Factorization

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

Figure 5: Sparse Backward Substitution for Doolittle Factorization

3.2 Choleski Factorization

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 [9]. A symmetric positive definite sparse matrix A can be numerically factored into a single lower triangular matrix L:

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

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 $L^T x = y$. Our analysis of the available parallelism in block-diagonal-bordered LU factorization, presented in section 4, can be extended to an analysis of available parallelism in block-diagonal-bordered Choleski factorization by simply substituting $L^T x$ for U. Additional discussions on the state of the literature for Choleski factorization are presented below.

We present a general sequential sparse factorization algorithm based upon the column Choleski factorization algorithm [20], which is similar to the factorization algorithms commonly attributed to Crout and Doolittle, and similar to the LU algorithm presented in figure 3. A sequential sparse factorization algorithm is presented in figure 6, and we present sequential sparse forward reduction and backward substitution algorithms for Choleski factorization in figures 7 and 8 respectively. In the backward substitution algorithm, the calculations are performed by implicitly transposing L.

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 6: Sparse Choleski Factorization

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 7: Sparse Forward Reduction for Choleski Factorization

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 $L_{j,k} \neq 0$

    $L_{j,k} \leftarrow L_{j,k} - (x_k * U_{j,k})$

  endfor

endfor

Figure 8: Sparse Backward Substitution for Choleski Factorization

3.3 Ordering Sparse Matrices

Position symmetric sparse matrices can be represented by graphs with elements in equations corresponding to undirected edges in the graph [9, 20]. The motivating applications for this research have position symmetric or have position symmetric submatrices that are derived from power system networks that have graph representations. 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 numerical stability and the only effect of modifying the order of calculations might result from changes in round-off errors [20]. Diagonally dominant matrices also can be factored with little concern for pivoting, and there are many applications where time constraints are critical, so in order to speedup sparse LU factorization, pivoting is ignored. If there is no pivoting, ordering can be performed a priori and static data structures can be used for the most efficient sequential algorithm.

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 9 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

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

where:

f_k is the number of fillin when factoring node k.

ν_k is the number of edges at node k.

There are several notable techniques to minimize fillin, with one of the commonly used techniques being minimum-degree ordering. This ordering technique is used for position symmetric matrices and attempts to minimize fillin by choosing that node for the next elimination which has the lowest degree or least number of connected edges. Minimum-degree ordering is closely related to Markowitz ordering [9]. Like minimum-degree ordering, Markowitz ordering attempts to select the next row/column to eliminate that has the least row/column elements. 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

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

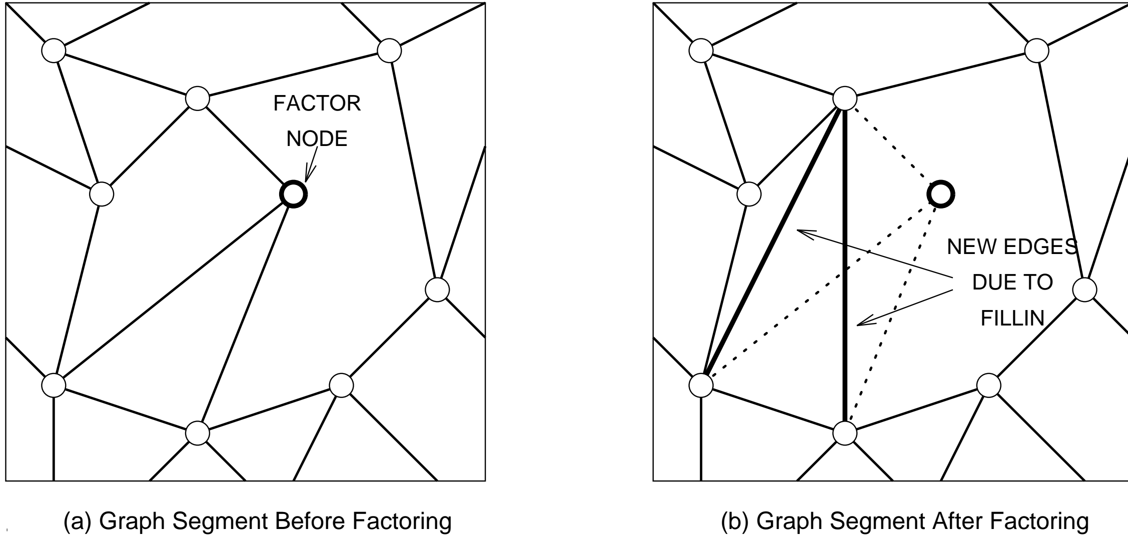

Figure 9: Graph Theoretical Explanation of Fillin

that is solved by factoring PAP^T into LU factors $\bar{L}$ and $\bar{U}$ in $\bar{L}\bar{U}$ or 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. For LU factorization, these steps would require the solutions of:

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

For Choleski factorization, simply substitute $\bar{L}^T$ for $\bar{U}$ in equation 6. Also for Choleski factorization, as long as a symmetric positive definite matrix A is ordered with the permutation matrix P to PAP^T, the resultant matrix after ordering remains symmetric positive definite.

3.4 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 [8, 42, 43], 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 [42, 43], 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 $O(N^{2.0})$, and actual power system sparse matrices used in this work have order of complexities

less than $O(N^{1.5})$. These orders of complexity are consistent with matrices from circuit analysis applications that have complexities ranging from $O(N^{1.2})$ to $O(N^{1.5})$ [33]. 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 [14, 15, 16, 20]. 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 [31, 38, 39, 40, 41].

Some of the most celebrated recent work has revived research into parallel sparse multifrontal Choleski techniques [18, 22]. Multifrontal techniques identify parallelism within the matrix structure in a manner similar to [14, 15, 16, 20], but then create multiple small, dense matrices from independent rows/columns of data, and update each frontal matrix with dense techniques. Parallel sparse multifrontal algorithms have shown scalable performance for very-large, extremely regular sparse structural matrices. There has been some work on solving less-regular problems. Research has recently been published in [32] that describes load balancing techniques to support the work in [31]. Also, research has been ongoing to examine techniques that can efficiently factor irregular matrices using multifrontal techniques [5, 6, 7].

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 [29]. 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 [14, 15, 16, 20, 38, 39, 40, 41] 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. The pseudo-factorization step is similar to the symbolic factorization step, although we require that the number of calculations in matrix partitions be calculated so that we can perform explicit load-balancing on the majority of the sparse matrix. 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 LU 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. Provided that a matrix can be ordered into block-diagonal-bordered form, then the parallel sparse LU 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 LU solvers [13].

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 10 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. The description of parallelism presented here is closely related to the concept of elimination graphs and supernodes described in [20]. A block-diagonal-bordered form sparse matrix can be represented by an elimination tree with supernodes at only two levels. Supernodes form the elimination tree leaves, with another supernode as the root of the $N_{procs} ary$ tree. By simply restructuring the graph presented in figure 10, it is possible to represent the same concept as a tree. An elimination tree for a block-diagonal-bordered form matrix with four supernodes as leaves and a single supernode as the tree's root is presented in figure 11.

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

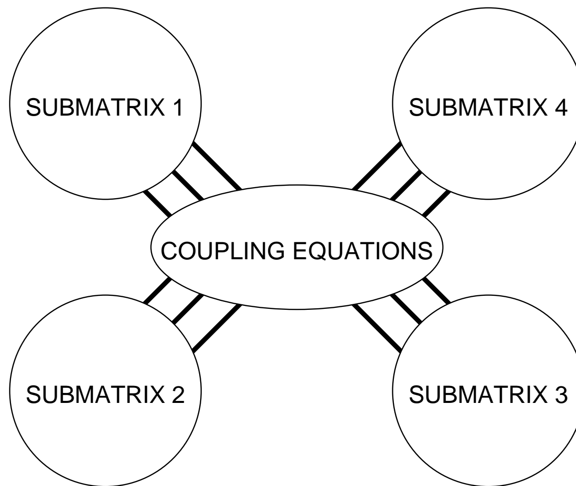

Figure 10: Graph with Four Independent Sub-Matrices

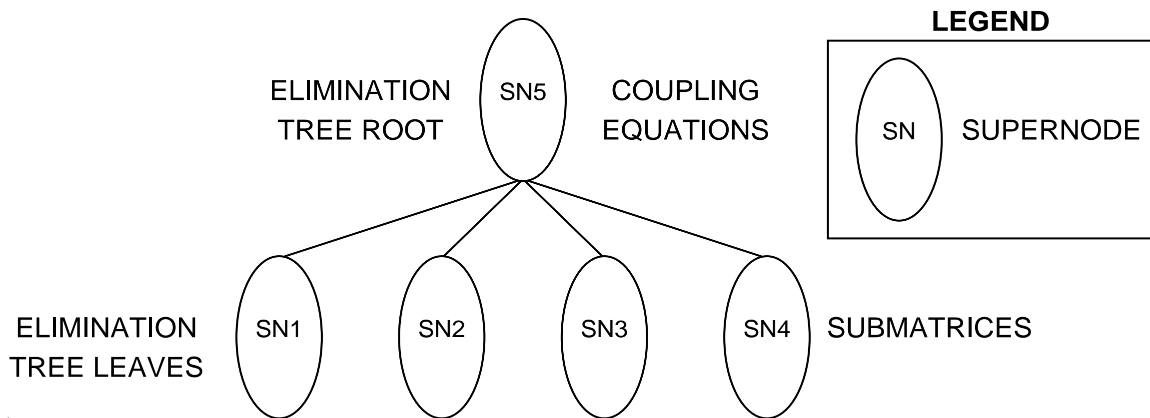

Figure 11: Elimination Tree with Four Supernode Leaves

and calculating the Shur complement [9], it is possible to identify available parallelism by proving a theorem that states the LU factors of a block-diagonal-bordered matrix are also in block-diagonal-bordered form. A supporting lemma stating that the LU 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 Choleski factorization.

Define a partition of $\mathbf{A} = \mathbf{LU}$ as

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

where:

$\mathcal{A}_{1,1}$, $\mathcal{L}_{1,1}$, and $\mathcal{U}_{1,1}$ are of size $n_1 \times n_1$

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

$\mathcal{A}_{1,2}$ and $\mathcal{U}_{1,2}$ are of size $n_1 \times n_2$

$\mathcal{A}_{2,2}$, $\mathcal{L}_{2,2}$, and $\mathcal{U}_{2,2}$ 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 $\mathbf{LU}$ partitions which yields:

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

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

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

Before we can proceed and prove the theorem that the $\mathbf{LU}$ factors of a block-diagonal-bordered (BDB) position 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 $\mathbf{LU}$ 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:

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

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

$$\mathbf{U}_{BDB} = \begin{pmatrix} \mathcal{U}_{1,1} & \mathcal{U}_{1,2} \\ \mathbf{0} & \mathcal{U}_{2,2} \end{pmatrix} = \begin{pmatrix} U_{1,1} & & & \mathbf{0} & U_{1,m} \\ & U_{2,2} & & & U_{2,m} \\ \mathbf{0} & & \ddots & & \vdots \\ & & & U_{m-1,m-1} & U_{m-1,m} \\ & & & & U_{m,m} \end{pmatrix} \tag{12}$$

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

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

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

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

Lemma — *The* **LU** *factors of a block-diagonal matrix are also in block-diagonal form*

Proof:

Let:

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

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

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

and

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

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

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

This lemma can be applied recursively to a block-diagonal matrix with any number of diagonal blocks to prove that the **LU** *factorization of a block-diagonal matrix preserves the block structure.*

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

Proof:

First the matrix partitions $\mathcal{A}_{2,1}$ *and* $\mathcal{A}_{1,2}$ *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,* $\mathbf{A}_{BDB} = \mathbf{L}_{BDB}\mathbf{U}_{BDB}$.

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: $\mathcal{A}_{1,1} = \mathcal{L}_{1,1}\mathcal{U}_{1,1} \Rightarrow \begin{cases} A_{1,1} = L_{1,1}U_{1,1} \\ A_{2,2} = L_{2,2}U_{2,2} \\ \vdots \end{cases}$

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

3. Upper Border: $\mathcal{A}_{1,2} = \mathcal{U}_{1,2}\mathcal{L}_{1,1} \Rightarrow \begin{cases} A_{1,m} = L_{1,1}U_{1,m} \\ A_{2,m} = L_{2,2}U_{2,m} \\ \vdots \end{cases}$

4. Last Block:

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

If the matrix blocks $A_{i,i}$, $A_{m,i}$, 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 LU 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 LU 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 $A_{m,i}$ $(1 \leq i \leq (m-1))$. 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 12 illustrates both the LU 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 $\mathbf{U}$. This figure depicts four

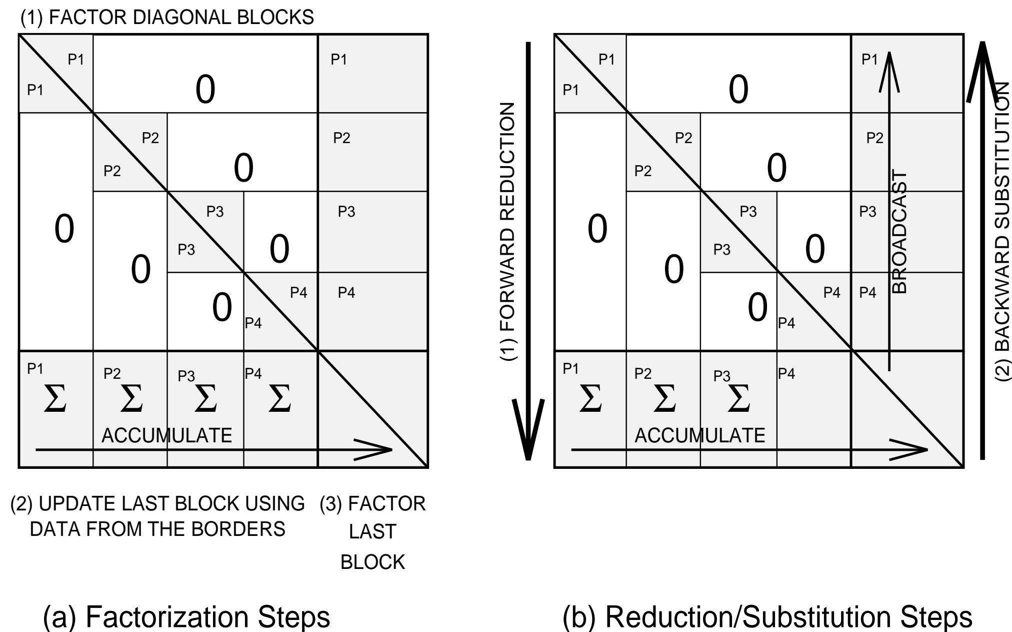

Figure 12: Block Bordered Diagonal Form Sparse Matrix Solution Steps

diagonal blocks, and processor assignments (P1, P2, P3, and P4) are listed with the data block. This figure would represent the block-diagonal-bordered form matrix and data distribution for the data represented in figures 10 and 11.

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 $O(N^3)$, and the computational complexity for factoring entire sparse matrices used in later parallel algorithm performance studies varies is substantially less than $O(N^{1.5})$. 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 [20]. Recursive spectral bisection can be used to partition irregular matrices [1, 30, 35], 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 [26]. A third method to order a sparse matrix into block-diagonal-bordered form is referred to as node tearing [9, 34], which is a specialized form of diakoptics [19]. 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 LU 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 [16, 20, 38].

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 **LU** 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 kji-saxpy LU algorithm to factor the last diagonal block [43]. 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 [26]. 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 [10]. The implementation of the parallel block-diagonal-bordered LU 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 [19] that was developed especially for power system network analysis [34]. 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 network graph, For load-flow analysis, 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 [34] 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 $\mathcal{S}\}$ [34].

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 [34] 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 13: Sample Contour Tableau for the kth 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 8.1. This optimization problem belongs to the family of NP-complete problems [34]. 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 [34], by developing a contour tableau to identify independent sub-graphs. A contour-tableau consists of three columns as illustrated in figure 13 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

$$c_{(i+1)}^k = |\mathcal{A}_{(i+1)}^k|. \tag{26}$$

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

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

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 $\mathcal{I}_i^k$ that has a local minimum value of c_i^k, $i \leq max_{DB}$. This selection criteria is obvious because at any location in the contour tableau, three disjoint sets are specified:

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

In this representation, no node in $\mathcal{I}_i^k$ is adjacent to a node in $\mathcal{Z}_i^k$, and $\mathcal{A}_i^k$ represents the coupling equations between the two sets. The number of elements in the set $\mathcal{A}_i^k$ varies as a function of i. One constraint in this optimization problem is to minimize the number of coupling equations, $\mid \mathcal{N}_2 \mid$, so a greedy algorithm that uses the heuristic for building the kth independent partition, $\mathcal{N}_1^k$, by minimizing the cardinality of $\mathcal{A}_i^k$ should yield an acceptable solution in a polynomial algorithm. Moreover, when a partition is selected, nodes remaining in $\mathcal{A}_i^k$ are placed directly into the set $\mathcal{N}_2$,

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

because $\mathcal{A}_i^k$ represents the nodes adjacent to but not included within the set $\mathcal{I}_i^k$. 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 $(\alpha \times max_{DB}) \leq i \leq max_{DB}$, $0 < \alpha < 1$. When an independent sub-matrix is found, this iterating set is moved into a set $\mathcal{N}_1^k$, where $\mid \mathcal{N}_1^k \mid = i$. After the sets $\mathcal{N}_1 = \{\mathcal{N}_1^1, \ldots, \mathcal{N}_1^m\}$ and $\mathcal{N}_2$ are determined, the equations corresponding to the sets $\mathcal{N}_1^1, \ldots, \mathcal{N}_1^m$ and $\mathcal{N}_2$ are further ordered using the minimum-degree ordering algorithm.

Figure 14 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

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

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 $\max_{\forall\, i} |\mathcal{A}_i^k|$ 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.

$\mathcal{G} \leftarrow$ *the 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 14: The Node-Tearing Algorithm

7 Sparse Matrix Solver Implementations

Implementations of a block-diagonal-bordered sparse LU solver and a similar Choleski solver have been developed in the C programming language for the Thinking Machines CM-5 multi-computer using message passing and a host-node paradigm. 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 network matrices. Results based on empirical data collected in benchmarking trials are presented in the next section. Our block-diagonal-bordered sparse solvers have the following distinct sections where blocks are defined in section 4:

1. LU factorization
    - factor the mutually independent diagonal blocks and associated portions of the border — $A_{i,i} = L_{i,i}U_{i,i}$, $A_{m,i} = L_{m,i}U_{i,i}$, and $A_{i,m} = L_{i,i}U_{i,m}$ for $(1 \leq i \leq (m-1))$
    - update the last diagonal block using the data in the borders — $A_{m,m} = A_{m,m} - \sum_{i=1}^{(m-1)} L_{m,i}U_{i,m}$
    - factor the last diagonal block — $A_{m,m} = L_{m,m}U_{m,m}$
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 Choleski factorization algorithm is similar to LU factorization, with the block-diagonal-bordered Choleski algorithm having the same distinct sections as described above with the exception of $L_{i,i}^T$ and $L_{m,i}^T$ being substituted for $U_{i,i}$ and $U_{i,m}$ respectively.

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.

7.1 The Hierarchical Data Structure

This block-diagonal-bordered sparse LU solver uses implicit hierarchical data structures based on vectors of C programming language structures to efficiently store and retrieve data for a 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. For this work, we are assuming that there is no requirement for pivoting and, consequently, we can using static data structures. In the static data structures, we use explicit pointers to subsequent data locations 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 eight 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. Eight 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 15 graphically illustrates the hierarchical nature of the data structure, where the distinct C structure elements presented in that figure are:

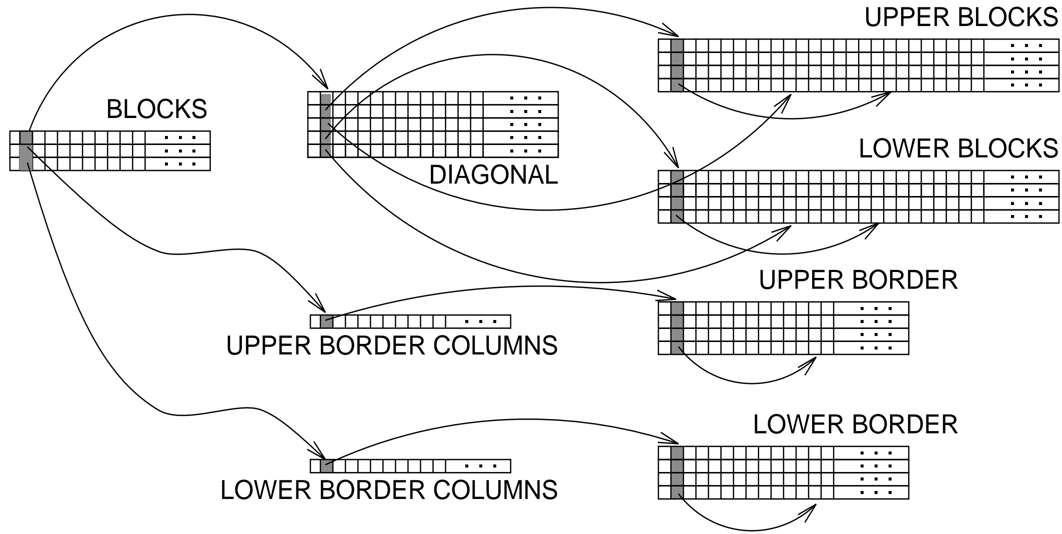

Figure 15: The Hierarchical Data Structure

1. diagonal block identifiers,
2. matrix diagonal elements,
3. non-zero values in the lower triangular diagonal matrix blocks (arranged by rows),
4. non-zero values in the upper triangular diagonal matrix blocks (arranged by columns),
5. non-zero row identifiers in the lower border,
6. non-zero column identifiers in the upper border,
7. non-zero values in the lower border (arranged by rows),
8. non-zero values in the upper border (arranged by columns).

At the top of the hierarchical data structure is the information on the storage locations of independent diagonal blocks, and both the lower and upper borders. The next layer in the data structure hierarchy has the matrix diagonal and the identifiers of non-zero border rows and columns. 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; likewise, data in the strictly upper triangular portion of the matrix is stored as sparse columns vectors. This data storage scheme minimizes the effort to find non-zero $A_{i,k}$ — $A_{k,j}$ pairs used to modify $A_{i,j}$ by consecutively storing values in lower triangular rows and upper triangular columns. However, Crout and Doolittle-based LU factorization algorithms require access to the next non-zero value in the same column or row for lower/upper triangular matrices, 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 upper triangular column, and to the first non-zero elements in the lower and upper 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.

The remaining portions of the hierarchical data structure efficiently store the non-zero values in the borders. Because entire lower border rows or upper border columns may be sparse in a block, two layers are required to store this data in an efficient manner. The next level in this portion of the hierarchy stores the location of the first non-zero value in the row or column. The corresponding row and column identifiers can be found by referencing the structure that the pointer references. The non-zero values in the lower and upper borders are stored with the same format as data in the diagonal blocks.

Conventional compressed data formats require less storage than this data structure; however, additional memory has been traded for reduced indexing overhead. Two reasons exist that justify the use of additional memory: large available memories are available with state-of-the-art distributed-memory multi-processors and these algorithms have been designed with the expressed intention to examine support real-time applications. 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

$$S_h = (\lambda_{fp} + (3 \times \lambda_{int})) \times \eta(A) + (\lambda_{int} \times n) + ((3 \times \lambda_{int}) \times N_{blocks}) + ((2 \times \lambda_{int}) \times N_{border}) \tag{31}$$

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 and column data is available as sparse vectors for ready cache-coherent access when updating values and subsequent column or row values 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 even for the sequential version of this sparse block-diagonal-bordered LU factorization algorithm.

While Crout and Doolittle factorization algorithms permit partial pivoting [21], this static hierarchical data structure assumes that no pivoting is required to maintain numerical stability in the calculations. Traditional numerical pivoting can be difficult in a general sparse matrix due to the sparsity structure and concerns for fillin, so considerations are made to relax the normal numerical pivoting rules in Markowitz pivoting when the matrix is neither positive definite nor diagonally dominate [9]. Block-diagonal-bordered sparse matrices offer the potential for an additional relaxed pivoting rule that limits pivoting choices to within a diagonal matrix block. Numerical pivoting choices could be further limited to a small neighborhood of an equation when sparse matrices are ordered into recursive block-diagonal-bordered form. For the present research, it is assumed that numerical pivoting will not be required, because the matrices for power systems distribution networks will be derived from matrices that are diagonally dominate or even positive definite.

7.2 The Parallel Blocked-Diagonal-Bordered LU Factorization Algorithm

Implementations for both parallel block-diagonal-bordered sparse LU and Choleski factorization have been developed in the C programming language for the Thinking Machines CM-5 multi-processor using a host-node paradigm with message passing. Two versions of each parallel block-diagonal-bordered algorithm have been implemented: one implementation uses low latency active messages to update the last diagonal block using data in the borders and the second implementation uses conventional high(er) latency asynchronous buffered non-blocking interprocessor communications. The communications paradigms for these two implementations differed significantly. The communications paradigm we used with active messages, is to calculate a vector $\times$ vector product and immediately send the value to the processor holding the corresponding value of $A_{m,m}$. The communications paradigm we used with asynchronous, buffered communications was to perform the vector $\times$ vector products, store them in a buffer, and the have each processor send buffers to all other processors. The active message communications paradigm greatly simplified development of the algorithm, and the empirical results, presented in the next section, show that the active message-based implementation is significantly faster.

The block-diagonal-bordered LU factorization algorithm can be broken into three component parts as defined in the derivation on available parallelism in section 4. Pseudo-code representations of each parallel algorithm section are presented separately in figures 16 through 19. In particular, each of these figures correspond to the following figure numbers:

1. factor the diagonal blocks and border — figure 16,

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
        for each $i \in [k, N_l]$
            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
        for each $i \in [k+1, N_l]$
            $A_{k,j} \leftarrow (A_{k,j} / A_{k,k})$
        endfor
        for each $j \in [k+1, N_l]$
            for each $i \in [1, k-1]$ such that $A_{k,i} \neq 0$ **and** $A_{i,j} \neq 0$
                $A_{k,j} \leftarrow A_{k,j} - (A_{k,i} * A_{i,j})$
            endfor
        endfor
    endfor
endfor

Figure 16: Parallel Block-diagonal-Bordered Sparse LU Factorization Algorithm — Diagonal Blocks and Border

2. update the last diagonal block,
    - active message communications paradigm — figure 17
    - asynchronous buffered communications paradigm — figure 18
3. factor the last diagonal block — figure 19.

In the actual parallel implementation, these program sections required a processor synchronization before starting any part of the algorithm. To better understand the algorithm, timing statistics were collected for each portion of the algorithm, and empirical data are reported for the performance of each algorithm portion in section 8.

The LU factorization algorithm for the diagonal blocks and border follows a Doolittle's form and is a *fan-in* type algorithm. Conversely, the LU factorization algorithm for the last diagonal block follows a *fan-out* algorithm requiring rank-1 submatrix updates as each row or block of rows are factored.

By distributing the factorization of the last block to all active node processors, interprocessor communications is required in the second step of this algorithm. The algorithm section that updates the last diagonal block calculates a sparse matrix $\times$ sparse matrix product by calculating individual

```
Node Program
/* calculate updates to the last block */
for those independent blocks l assigned to this processor
    for all non-zero columns j in the upper border of this block
        for all non-zero rows i in the lower border
            σ = 0
            for each k such that L_{i,k} and U_{k,j} ≠ 0
                σ ← σ + (L_{i,k} * U_{k,j})
            endfor
        endfor
        p ← P(i, j) /* the function P() maps the (i, j) tuple to the processor location p */
        Send an active message RPC to the handler function Update_FAC_LB(σ, i, j) on processor p
        Poll for active message RPCs
    endfor
endfor

/* update the last block with active message RPC handler function Update_FAC_LB */
Function Update_FAC_LB(σ, i, j)
        A_{i,j} ← A_{i,j} − σ
End Update_FAC_LB
```

Figure 17: Parallel Block-diagonal-Bordered Sparse LU Factorization Algorithm — Update the Last Diagonal Block — Active Message Communications Paradigm

Node Program

/* calculate updates to the last block */

$\sigma_{*,*,*} = 0$

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

  for all non-zero columns j **in the upper border**

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

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

      for each k **such that** $L_{i,k}$ **and** $U_{k,j} \neq 0$

        $\sigma_{i,j,p} \leftarrow \sigma_{i,j,p} + (L_{i,k} * U_{k,j})$

      endfor

    endfor

  endfor

endfor

/* update the last diagonal block on this processor using the data calculated on this processor */

for all data in the buffer

  $A_{i,j} \leftarrow A_{i,j} - \sigma_{i,j}$

endfor

/* update the last diagonal block on this processor using the data calculated on other processors */

$p_{send} \leftarrow p_{receive} \leftarrow local_proc_num$

$p_{send} \leftarrow (p_{send} + 1) \bmod N_{proc}$

$p_{receive} \leftarrow (p_{receive} + N_{proc} - 1) \bmod N_{proc}$

for all processors around a conceptual ring

  Asynchronously send $\sigma_{*,*,p_{send}}$ *to processor* p_{send}

  Asynchronously receive $\tilde{\sigma}_{*,*}$ *from processor* $p_{receive}$

  /* update the last diagonal block on this processor using the data from processor $p_{receive}$ */

  for all data in the received buffer

    $A_{i,j} \leftarrow A_{i,j} - \tilde{\sigma}_{i,j}$

  endfor

  $p_{send} \leftarrow (p_{send} + 1) \bmod N_{proc}$

  $p_{receive} \leftarrow (p_{receive} + N_{proc} - 1) \bmod N_{proc}$

endfor

Figure 18: Parallel Block-diagonal-Bordered Sparse LU Factorization Algorithm — Update the Last Diagonal Block — Asynchronous Buffered Communications Paradigm

Node Program

/* factor the last block — all communications are sent around a conceptual ring */

$p_{factor} \leftarrow p_{start} \leftarrow 0$

if $p_{factor} = local_proc_num$

  factor the first block

  send the panel of values to processor 1

end if

for all blocks on this processor

  if $p_{factor} \neq local_proc_num$

    receive the next panel

    if $p_{start} \neq local_proc_num$

      $p_{send} \leftarrow (local_proc_num + 1) \bmod N_{proc}$

      send the next panel of values to processor p_{send}

    end if

  end if

  $p_{factor} \leftarrow (p_{factor} + 1) \bmod N_{proc}$

  if $p_{factor} = local_proc_num$

    update only the next block with the next panel

    factor the next block

    if *not* **the last block**

      $p_{send} \leftarrow (local_proc_num + 1) \bmod N_{proc}$

      send the panel of values to processor p_{send}

      $p_{start} \leftarrow p_{send}$

    end if

  end if

  update all remaining blocks with the next panel

endfor

Figure 19: Parallel Block-diagonal-Bordered Sparse LU Factorization Algorithm — Last Diagonal Block

sparse vector × sparse vector products for lower border rows and upper border columns and these partial sums must be distributed to the proper processor holding data in the last diagonal block. Separate sparse vector products are performed for each block of data on a processor. Only nonzero rows in the lower border and non-zero columns in the upper border are utilized when calculating vector × vector products to generate the required partial sum values to update the last diagonal block. Examining only non-zero values significantly limits the amounts of calculations in this phase. In the process of developing an implementation with optimal performance, it was discovered that any attempt to consolidate updates to a value in the last diagonal block caused more overhead than was encountered by sending multiple update values to the same processor. There is a higher order of work required to sum update data than to calculate the sparse vector products. Likewise, there has been no attempt at parallel reduction of the partial sums of updates from the borders.

This block-diagonal-bordered LU factorization implementation solves the last block using a sparse blocked kji-saxpy LU algorithm based on the dense kij-saxpy algorithm described in [43]. Our examinations of power systems network matrices showed that after partitioning these matrices into block-diagonal-bordered form, there is little additional available parallelism in the last diagonal block — insufficient parallelism to be exploited in a load-balanced manner. The algorithm to factor the last diagonal block maintains the blocked format and pipelines nature of the dense kij-saxpy algorithm; however, special changes were made to the algorithm in order to minimize calculations and to minimize overhead when performing communications. The three significant changes to the kij-saxpy algorithm are:

- the order of calculations in each *fan-out* update has been changed,
- sparse data structures,
- separate data structures are used to store the values on the diagonal and values in the strictly lower and strictly upper triangular matrices.

While changing the order of calculations in each rank-1 update of the partially-factored submatrix has little effect on the factorization algorithm, it is equivalent to exchanging the order of **for** loops, this seemingly small modification contributed significantly to improving the performance of the forward reduction and backward substitution steps. This small modification reduced the amount of communications greatly during both forward reduction and backward substitution by allowing the broadcast of only calculated values of y_m and x_m and not also requiring the broadcast of partial sums encountered when updating values.

As would be expected with a sparse matrix, data is stored as sparse vectors with explicit row and column identifiers. To optimize performance for a kji algorithm, data is stored in sparse vectors corresponding to rows in the matrix. These sparse vectors of row data are stored as three separate data structures:

1. values on the diagonal,
2. values in the strictly lower triangular matrix,
3. values in the strictly upper triangular matrix.

The use of three data structures greatly simplifies parallel rank-1 updates in this *fan-out* algorithm. In parallel LU factorization, only the data in either the strictly lower triangular or strictly upper triangular matrix must be broadcast to all processors when updating the submatrix. By storing the data in the triangular matrices in separate data structures, the data in a block can be accessed directly by the buffered communications software. With irregular sparse matrices, this storage technique is required to eliminate a memory-to-memory copy step required if data was stored in single sparse row or column vectors. The data is irregular and no regular strides can be used when forming the communications buffer if all data is stored contiguously for a row.

Parallel block-diagonal-bordered Choleski factorization algorithms are similar to the LU factorization algorithms presented in figures 16 through 19 — with modifications to account for the symmetric nature of the matrices used in Choleski factorization. Choleski factorization has only about half of the calculations of LU factorization. Block-diagonal-bordered Choleski factorization has half the number of updates to the last diagonal block. The parallel Choleski algorithm using the active message communications paradigm would see reduced communications when compared to LU factorization; however, there would be no reduction in the number of messages required in a buffered communications update of the last diagonal block. While the buffers would be shorter, there is still the requirement for each processor to communicate with all other processors. In most instances, the message start-up latency dominates, not the per-word transport costs. There would also be no reduction in the amount of communications when factoring the last block.

In other words, parallel Choleski factorization is a more difficult algorithm to implement than LU factorization, because there is a significant reduction in the amount of calculations, without a similar reduction in communications overhead. Consequently, the results in section 8 will compare the performance of LU factorization and Choleski factorization to illustrate performance as a function of the amount of floating point operations versus the communications overhead. To get a better understanding of this trend, a version of the parallel block-diagonal-bordered LU factorization algorithm has been implemented for complex data. Complex math has four floating point multiplies and two subtracts/adds when compared to double precision floating point calculations. With these three implementations, we will be able to clearly illustrate the need for high-speed low-latency communications for algorithms to solve power systems network linear equations. These matrices are sufficiently sparse, that by increasing the number of floating-point operations by two or six times that of Choleski factorization, there is a marked increase in the relative parallel speedup of these algorithms. Communications overhead remains constant and only the number of floating point operations increases. By considering the capabilities of the target parallel architecture, namely the communications to computations ratio or granularity, you can identify what communications capabilities are required in your target parallel architecture.

The same implicit hierarchical data structure used in the parallel implementation can be readily used in a sequential implementation, where the entire matrix is placed on a single node processor on the CM5. Single-processor performance is measured in this manner, so that relative speedup can be calculated to examine parallel algorithm performance.

```
Node Program
/* reduce the independent blocks */
for all independent blocks l assigned to this processor
    for all elements k along the diagonal in block l
        y_k ← b_k
        for each i ∈ [k + 1, N_l] such that L_{i,k} ≠ 0
            b_i ← b_i − (y_k * L_{i,k})
        endfor
    endfor
endfor
```

Figure 20: Parallel Block-diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Diagonal Blocks and Border

7.3 Forward Reduction and Backward Substitution Algorithms

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 [23]. The combination of these techniques is utilized to minimize communications times when solving for the last diagonal block. The forward reduction algorithm to operate with the parallel block-diagonal-bordered LU factorization algorithm can be broken into three component parts, similar to LU factorization. Pseudo-code representations of each parallel algorithm section are presented separately in figures 20 through 23. In particular, each of these figures correspond to the following figure numbers:

1. forward reduce the diagonal blocks and border — figure 20,
2. update the last diagonal block,
    - active message communications paradigm — figure 21
    - asynchronous buffered communications paradigm — figure 22
3. forward reduce the last diagonal block — figure 23.

As with the LU factorization algorithms, in the actual parallel implementation, these program sections required a processor synchronization before starting any part of the algorithm. To better understand the algorithm, timing statistics were collected for each portion of the algorithm, and are presented in section 8.

The backward substitution algorithm to operate with the parallel block-diagonal-bordered LU factorization algorithm can be broken into two component parts, back substitute the last diagonal block then back substitute the upper triangular matrix. The only interprocessor communications required occurs when solving for x_m in the last diagonal block. The solution for x_m in this portion

```
Node Program
/* 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
        σ = 0
        for each j such that L_{i,j} ≠ 0
            σ ← σ + (y_j * L_{i,j})
        endfor
        p ← P(i) /* the function P() maps the row value (i) to the processor location p */
        Send an active message RPC to the handler function Update_FR_LB(σ, i) on processor p
    endfor
endfor

/* update the last block with active message RPC handler function Update_FR_LB */
Function Update_FR_LB(σ, i)
        b_i ← b_i − σ
End Update_FR_LB
```

Figure 21: Parallel Block-diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Update the Last Diagonal Block — Active Message Communications Paradigm

Node Program

/* 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**

    $p \leftarrow \mathcal{P}(i)$ /* the function $\mathcal{P}()$ maps the row value (i) to the processor location p */

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

      $\sigma_{i,p} \leftarrow \sigma_{i,p} + (y_j * L_{i,j})$

    endfor

  endfor

endfor

/* update the last diagonal block on this processor using the data calculated on this processor */

for all data in the buffer

  $b_i \leftarrow b_i - \sigma_i$

endfor

/* update the last diagonal block on this processor using the data calculated on other processors */

$p_{send} \leftarrow p_{receive} \leftarrow local_proc_num$

$p_{send} \leftarrow (p_{send} + 1) \bmod N_{proc}$

$p_{receive} \leftarrow (p_{receive} + N_{proc} - 1) \bmod N_{proc}$

for all processors around a conceptual ring

  Asynchronously send $\sigma_{*,p_{send}}$ *to processor* p_{send}

  Asynchronously receive $\tilde{\sigma}_*$ *from processor* $p_{receive}$

  /* update the last diagonal block on this processor using the data from processor $p_{receive}$ */

  for all data in the received buffer

    $b_i \leftarrow b_i - \tilde{\sigma}_i$

  endfor

  $p_{send} \leftarrow (p_{send} + 1) \bmod N_{proc}$

  $p_{receive} \leftarrow (p_{receive} + N_{proc} - 1) \bmod N_{proc}$

endfor

Figure 22: Parallel Block-diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Update the Last Diagonal Block — Asynchronous Buffered Communications Paradigm

Node Program

/* reduce the last block */

$p_{reduce} \leftarrow p_{start} \leftarrow 0$

if $p_{reduce} = local_proc_num$

  forward reduce the first block

  send the values of y from this block to processor 1

end if

for all blocks on this processor

  if $p_{reduce} \neq local_proc_num$

    receive the next values of y

    if $p_{start} \neq local_proc_num$

      $p_{send} \leftarrow (local_proc_num + 1) \bmod N_{proc}$

      send the next values of y on to processor p_{send}

    end if

  end if

  $p_{reduce} \leftarrow (p_{reduce} + 1) \bmod N_{proc}$

  if $p_{reduce} = local_proc_num$

    forward reduce only the next block with the next values of y

    if *not* **the last block**

      $p_{send} \leftarrow (local_proc_num + 1) \bmod N_{proc}$

      send the values of y from this block to processor p_{send}

      $p_{start} \leftarrow p_{send}$

    end if

  end if

  forward reduce all remaining blocks with the next values of y

endfor

Figure 23: Parallel Block-diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Last Diagonal Block

of the matrix broadcasts the values of x_m to all processors, so those values are available to the next step, solving for x_1 to x_{m-1} in the remaining diagonal blocks. Pseudo-code representations of each parallel algorithm section are presented separately in figures 24 and 25, respectively for backward substitution of the last diagonal block and backward substitution of the diagonal blocks and border.

Forward reduction and backward substitution algorithms for Choleski factorization are similar to those for LU factorization, with one major difference. The factorization process generates only a single lower triangular matrix, L. For the last diagonal block, one triangular solution step must occur in a manner that requires more communications than an optimally implemented triangular solution for LU factorization. The selection of kji factorization in the last diagonal block for LU factorization was to maximize performance for both forward reduction and backward substitution — as a result communications was minimized. Meanwhile, for Choleski factorization, the optimal direct solver algorithm must use a column distribution for the data in the last block, require additional communications and calculations be incurred in the forward reduction of the last diagonal block, and then backward substitute the last diagonal block using an implicit transpose of L. The final step ensures that all x_m values are broadcast to all processors, eliminating an extra, costly communications step. This combination of data distributions and algorithm specifics ensures the least number of calculations and the minimum amount of communications are performed and should offer the best opportunity for good parallel performance.

8 Empirical Results

A stated goal of this block-diagonal-bordered LU solver is to simplify the task organization of the parallel LU algorithm and have interprocessor communications significantly reduced and regular. The performance of this block-diagonal-bordered LU solver is dependent on the ability to order the real power systems sparse matrices into the appropriate form with both uniformly distributed data in the diagonal blocks and a minimum number of equations on the lower border. In section 8.1, we illustrate the ordering capabilities of the node-tearing nodal analysis by presenting pseudo-images of selected sparse power systems network matrices after we have applied our node-tearing algorithm to partition the matrices into block-diagonal-bordered form and also have applied the pigeon-hole load-balancing algorithm. We provide additional information as to the overall performance of the three-step preprocessing phase, with special note to the amount of fillin in the matrices after ordering and to the total number of floating operations required to factor the matrices. We then report on the performance of the block-diagonal-bordered sparse LU and Choleski solvers in section 8.2. Performance of these parallel block-diagonal-bordered direct linear solvers is dependent on both the ability of the node-tearing algorithm and the performance of the parallel implementations. The real performance test of the node-tearing algorithm occurs when the performance of the block-diagonal-bordered sparse LU solver is examined for real power system network matrices in section 8.2. In section 8.3, we present our conclusions concerning the performance of our parallel implementations — and add projections of how the parallel algorithms would perform when ported to near-term future scalable parallel processing (SPP) architectures.

Node Program

/* backward substitute the last block */

$p_{sub} \leftarrow p_{start} \leftarrow p_{last_block}$

if $p_{sub} = local_proc_num$

  backward substitute the last block

  $p_{send} \leftarrow (local_proc_num + N_{proc} - 1) \bmod N_{proc}$

  send the values of x *from this block to processor* p_{send}

end if

for all blocks on this processor

  if $p_{sub} \neq local_proc_num$

    receive the next values of x

    if $p_{start} \neq local_proc_num$

      $p_{send} \leftarrow (local_proc_num + N_{proc} - 1) \bmod N_{proc}$

      send the next values of x *on to processor* p_{send}

    end if

  end if

  $p_{sub} \leftarrow (p_{sub} + N_{proc} - 1) \bmod N_{proc}$

  if $p_{sub} = local_proc_num$

    backward substitute only the next block with the next values of x

    if *not* **the first block**

      $p_{send} \leftarrow (local_proc_num + N_{proc} - 1) \bmod N_{proc}$

      send the values of x *from this block to processor* p_{send}

      $p_{start} \leftarrow p_{send}$

    end if

  end if

  backward substitute all remaining blocks with the next values of x

endfor

Figure 24: Parallel Block-diagonal-Bordered Sparse Backward Substitution Algorithm — LU Factorization — Last Diagonal Block

Node Program
/* backward substitute in the independent blocks and the border */
for all independent blocks l **in descending order**

/* backward substitute the border in this block */
for all elements k **along the diagonal in block** l **in descending order**
for all j **in the upper border such that** $U_{k,j} \neq 0$
$y_k \leftarrow y_k - x_j * U_{k,j})$
endfor
endfor

/* backward substitute the triangular section of this block */
for all elements k **along the diagonal in block** l **in descending order**
$x_k \leftarrow (y_k / U_{k,k})$
for each $i \in [1, i-1]$ such that $U_{i,k} \neq 0$
$y_i \leftarrow y_i - (x_k * U_{i,k})$
endfor
endfor
endfor

Figure 25: Parallel Block-diagonal-Bordered Sparse Backward Substitution Algorithm — LU Factorization — Diagonal Blocks and Border

8.1 Empirical Results — Ordering Power Systems Network Matrices into Block-Diagonal-Bordered Form

Performance of our parallel block-diagonal-bordered LU and Choleski solvers will be examined with five separate power systems network matrices:

- Boeing-Harwell matrix BCSPWR09 — 1,723 nodes and 2,394 graph edges [10],
- Boeing-Harwell matrix BCSPWR10 — 5,300 nodes and 8,271 graph edges [10],
- EPRI matrix EPRI6K matrix — 6,692 nodes and 10,535 graph edges [11],
- Niagara Mohawk Power Corporation operations matrix NiMo-OPS — 1,766 nodes and 2,506 graph edges,
- Niagara Mohawk Power Corporation planning matrix NiMo-PLANS — 9,430 nodes and 14,001 graph edges.

Matrices BCSPWR09 and BCSPWR10 are from the Boeing Harwell series and represent electrical power system networks from the Western and Eastern US respectively. The EPRI-6K matrix is distributed with the Extended Transient-Midterm Stability Program (ETMSP) from EPRI. Matrices NiMo-OPS and NiMo-PLANS have been made available by the Niagara Mohawk Power Corporation, Syracuse, NY.

To illustrate the performance of the graph partitioning algorithm, we will present pseudo-images that illustrate the sparsity of the matrices that represent the power systems networks. These pseudo-images illustrate the locations of the non-zero values in the matrices, both the original non-zero values and those that would become non-zero due to fillin during factorization. In the following pseudo images, original non-zero values are represented as black and fillin values are represented by a lighter grey color. A bounding box has been placed around the sparse matrix.

A detailed ordering analysis of graph partitioning is presented here for the BCSPWR09 power systems network data to illustrate the ability of the node-tearing ordering algorithm described in section 6.1. In order to provide a baseline with which to illustrate the performance of the node-tearing algorithm, we provide a representation of the original matrix in figure 26 and a representation of the sparse matrix after ordering with the minimum degree algorithm in figure 27. The original matrix has no fillin and is presented with the graph node identifiers as supplied in the Boeing-Harwell data distribution — without ordering. The minimum degree ordered 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.

Our parallel block-diagonal-bordered direct solver algorithms require that the power systems network 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 by the node-tearing algorithm. Examples of applying the node-tearing algorithm to the BCSPWR09 matrix are presented in figures 28 through 31, respectively for maximum diagonal block sizes of 16, 32, 64, and 96 nodes. Statistics are presented in table 2 for

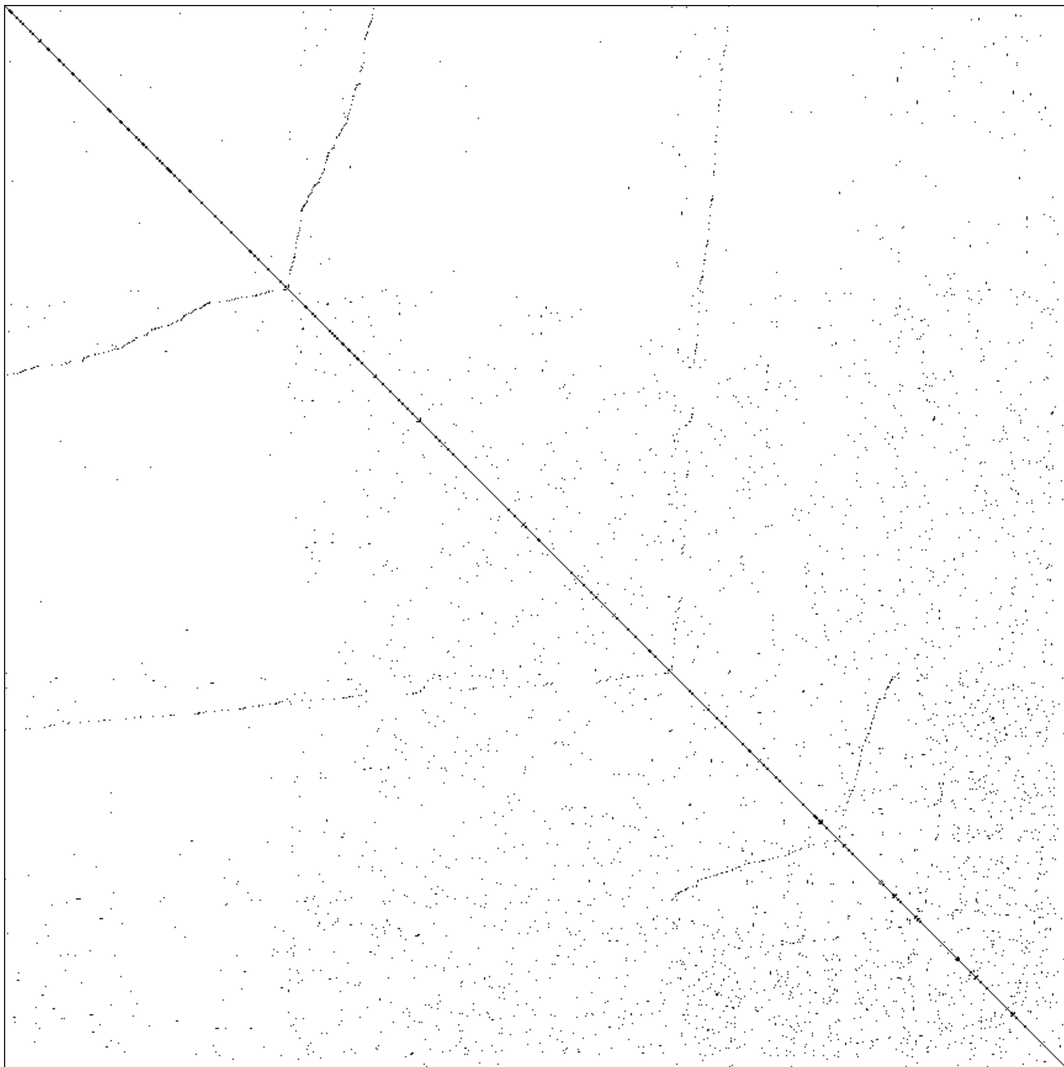

Figure 26: BCSPWR09 — Original Matrix

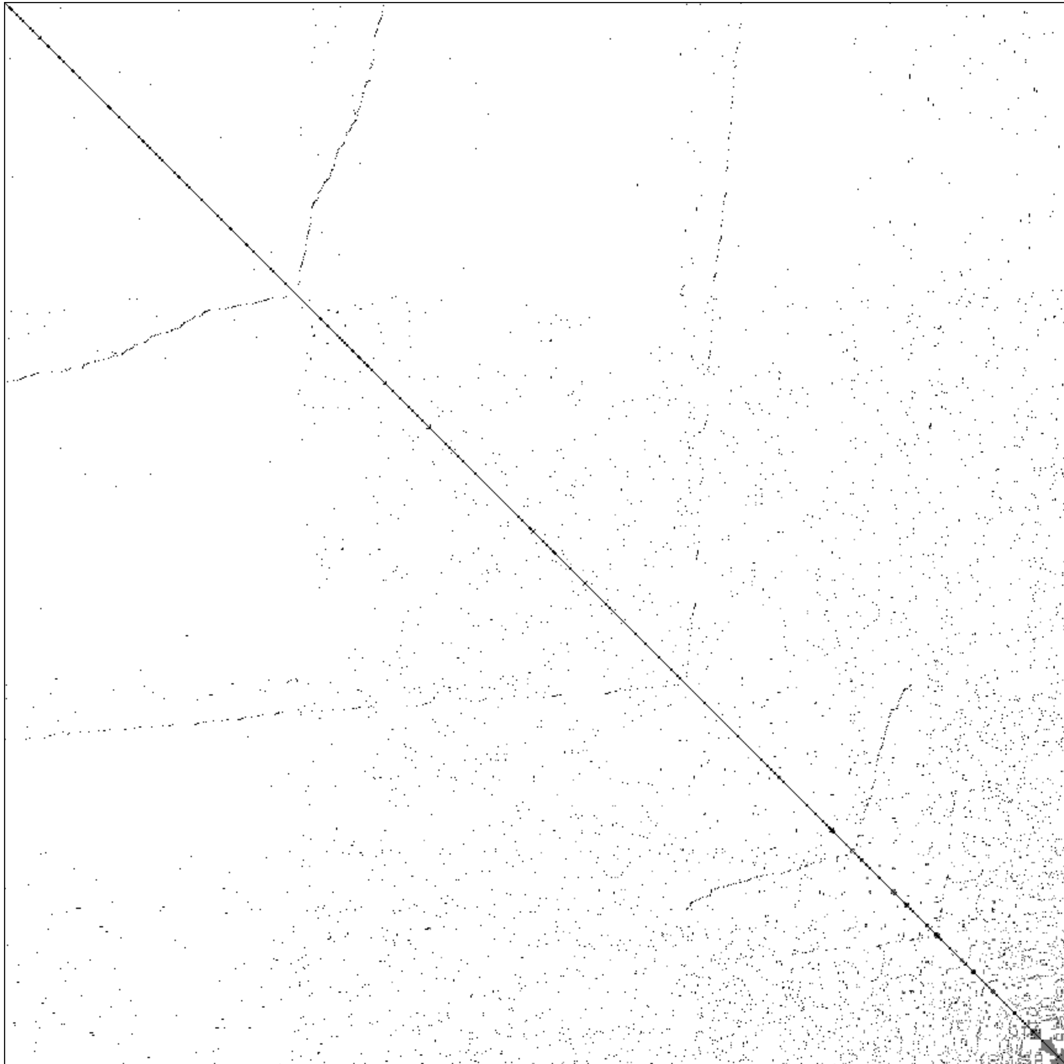

Figure 27: BCSPWR09 — Minimum Degree Ordering

Table 2: **BCSPWR09 — LU and Choleski Factorization Ordering Statistics**

number of nodes	1723
number of edges	2394
fillin for minimum degree ordering	2168

Maximum Partition Size	**Lower Triangular Matrix and Border**						
				Factor		**Fr or Bs**	
	N_{LB}	N_{FILLIN}	**Non Zeros**	**Total Ops**	**% Ops in Partitions**	**Total Ops**	**% Ops in Partitions**
16	277	3109	7226	18958	36.5%	5503	58.9%
32†	190	2765	6882	16759	45.7%	5192	67.7%
64	153	3248	7365	23809	37.7%	5642	65,6%
96	131	3266	7383	24906	40.3%	5660	68.1%

† Best parallel direct block-diagonal-bordered sparse linear solver performance

the four matrix orderings. In this table, N_{LB} is the number of rows/columns in the borders and last diagonal block of the ordered matrix and N_{FILLIN} is the number of fillin. This table shows that the ordering with maximum partition size of 32 has the least fillin, the fewest total operations, the largest percentage of operations in the mutually independent matrix partitions, and the best parallel direct linear solver performance.

Figures 28 through 31 illustrate that the size of the borders and last diagonal block can be manipulated by varying the value of the single input parameter to the partitioning algorithm, the maximum partition size. The number of rows/columns in the borders and last diagonal block of these ordered matrices vary from 277 to 131 for maximum partition size of 16 and 96 respectively. Each of these four figures has been load-balanced for eight processors. Figure 29 includes additional markings to illustrate how this matrix would be distributed to the eight processors — P1 through P8. Load-balancing is a function of the number of operations and not the number of columns assigned to a processor. The load balancing step is simply another permutation of the matrix that keeps rows/columns within partitions together in the same order. As the matrix is load-balanced for various numbers of processors, there is no change in the number of fillin or the total number of operations.

Figure 32 illustrates the relationship between maximum partition size and size of the borders and last diagonal block for each of the five power systems networks used in this analysis. The partitioning results for the BCSPWR09 network is very similar to the data for the Niagara Mohawk operations data. These matrices are similar in size and have similar numbers of edges per node. The larger matrices have significantly larger numbers of rows in the last diagonal block, and there is significantly larger variation between the number of rows in the last diagonal block for a maximum

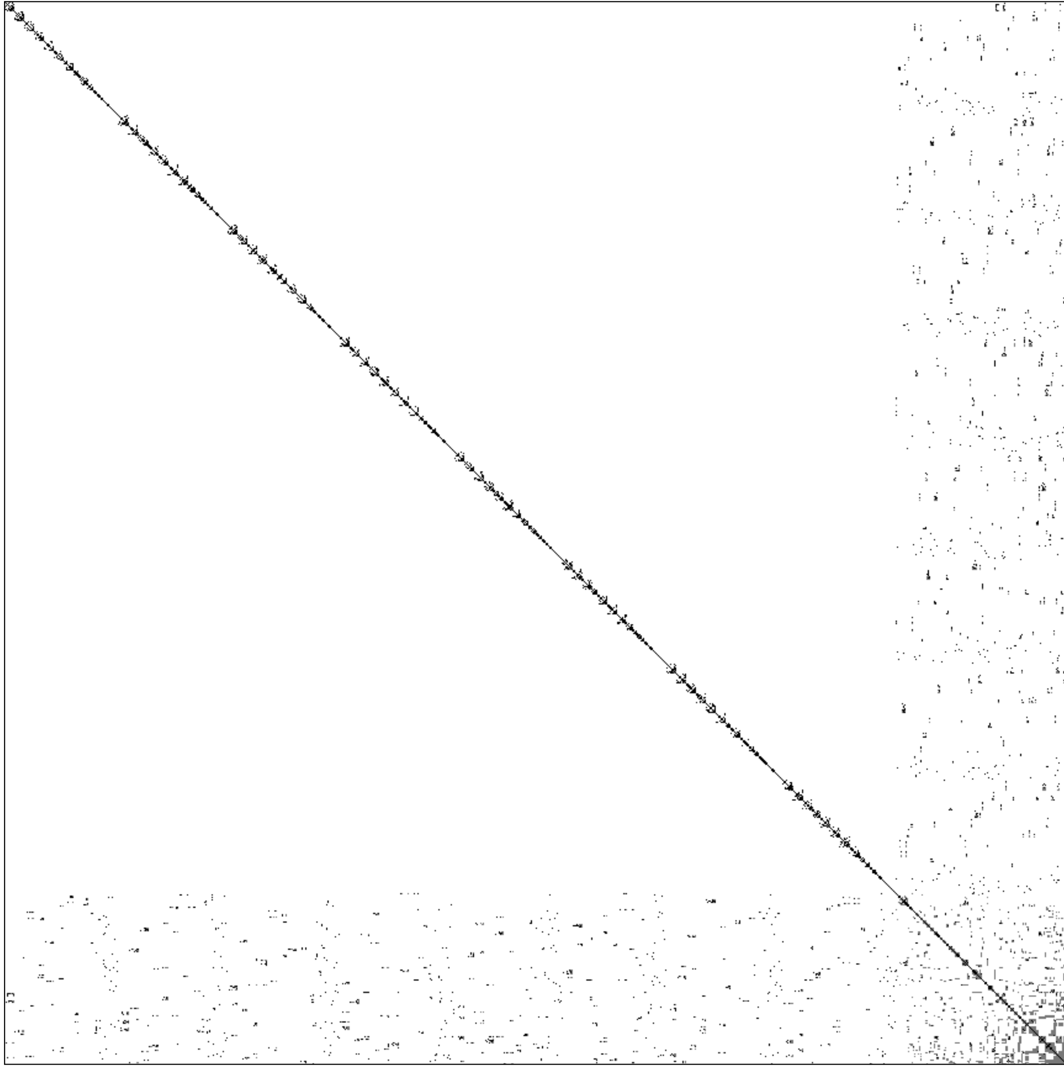

Figure 28: BCSPWR09 — Block-Diagonal-Bordered Form — Maximum Partition Size = 16 — Load Balanced for 8 Processors

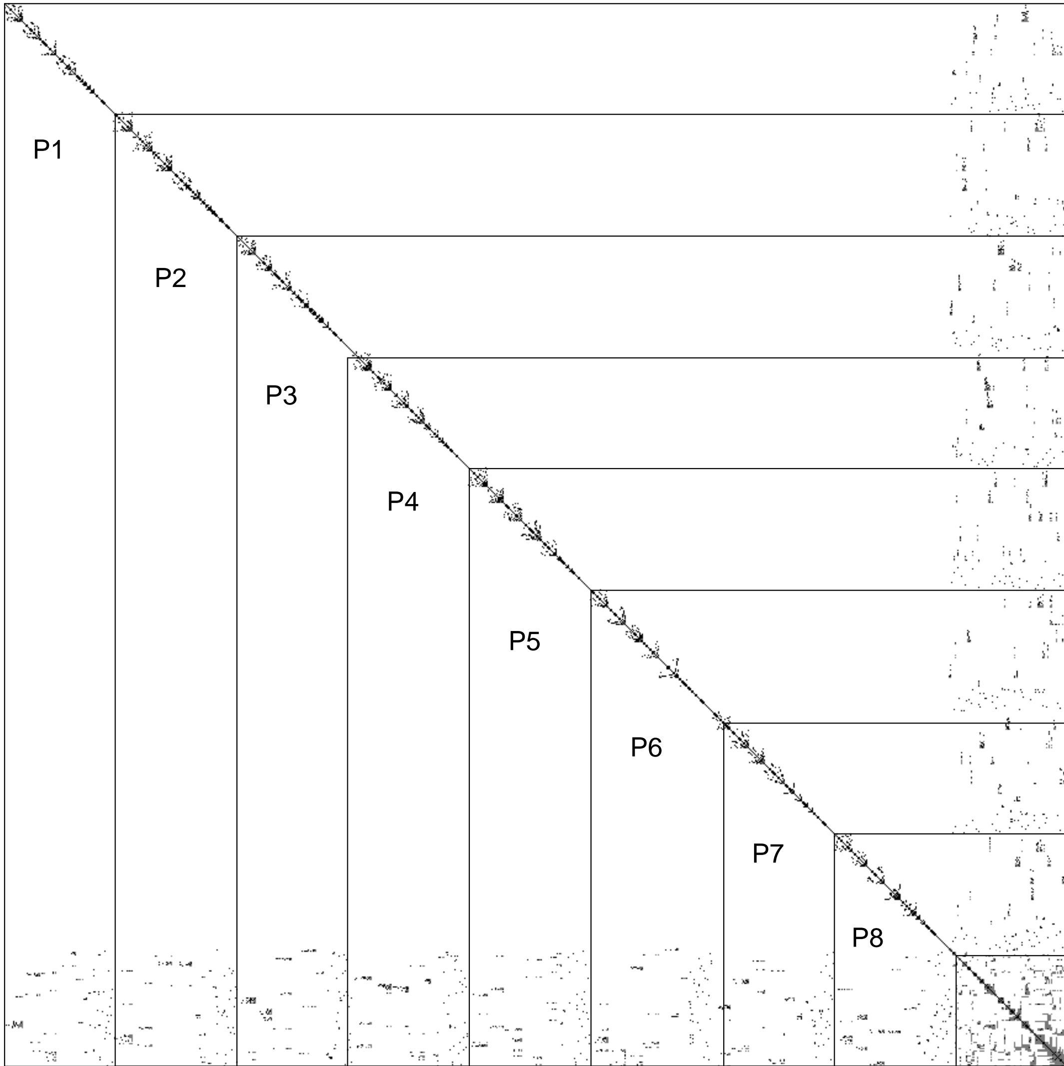

Figure 29: BCSPWR09 — Block-Diagonal-Bordered Form — Maximum Partition Size = 32 — Load Balanced for 8 Processors

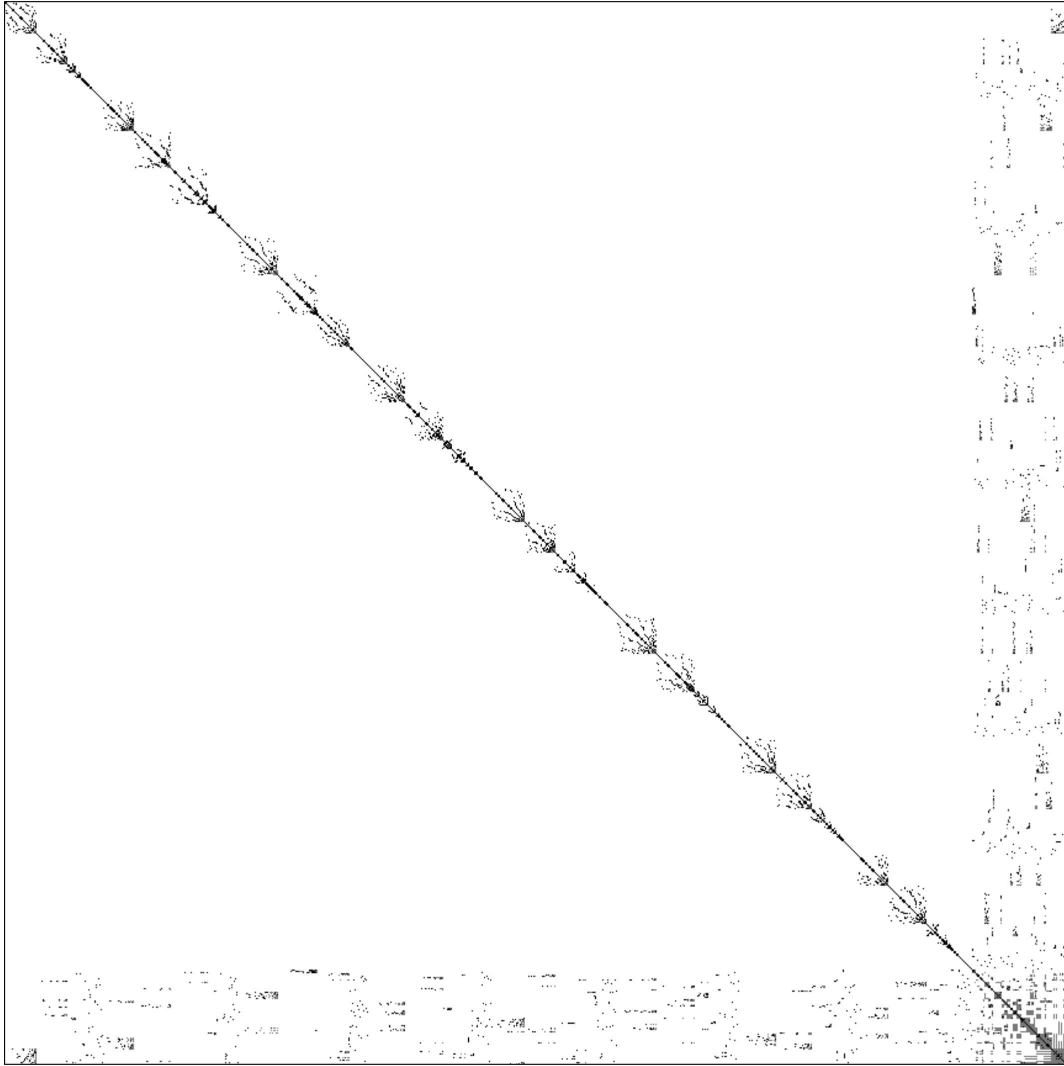

Figure 30: BCSPWR09 — Block-Diagonal-Bordered Form — Maximum Partition Size = 64 — Load Balanced for 8 Processors

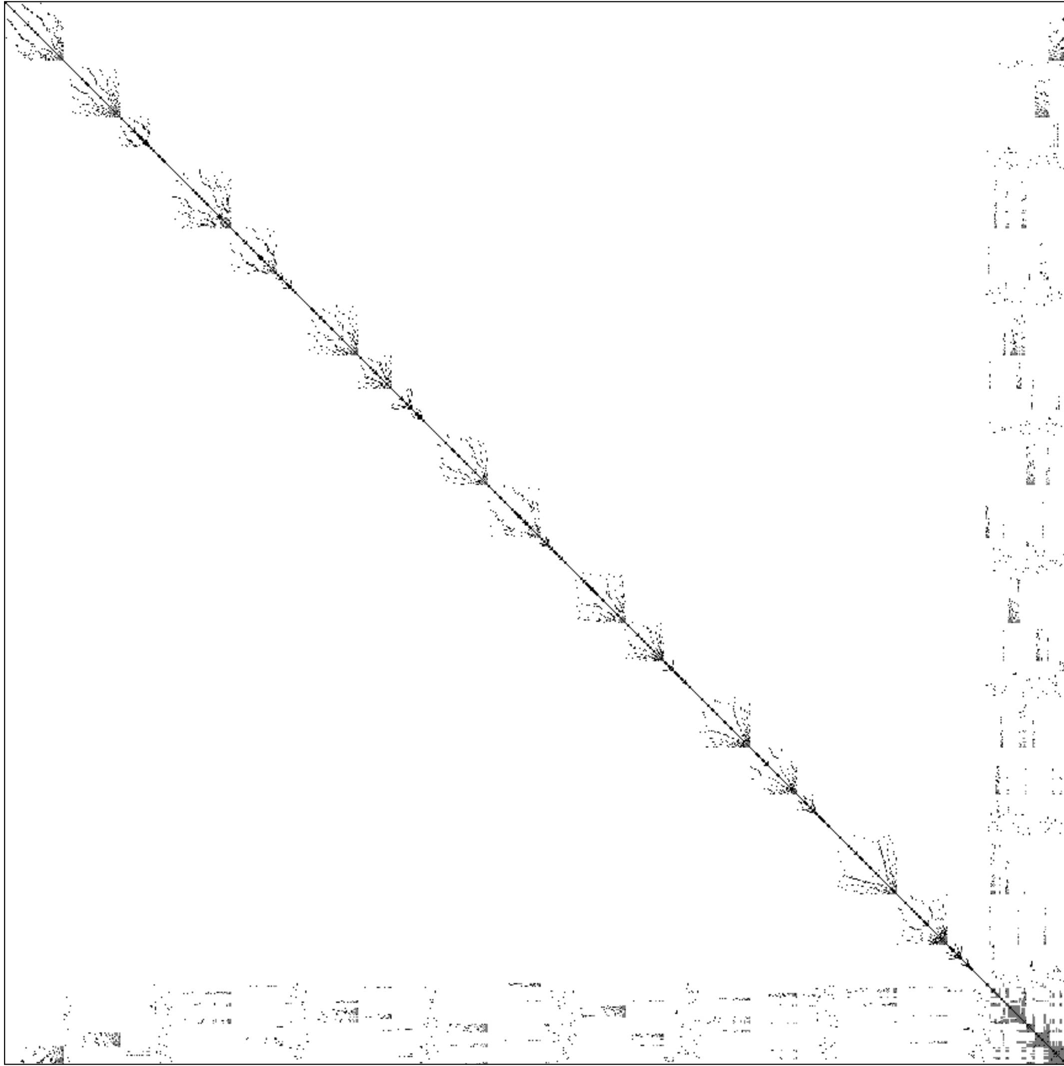

Figure 31: BCSPWR09 — Block-Diagonal-Bordered Form — Maximum Partition Size = 96 — Load Balanced for 8 Processors

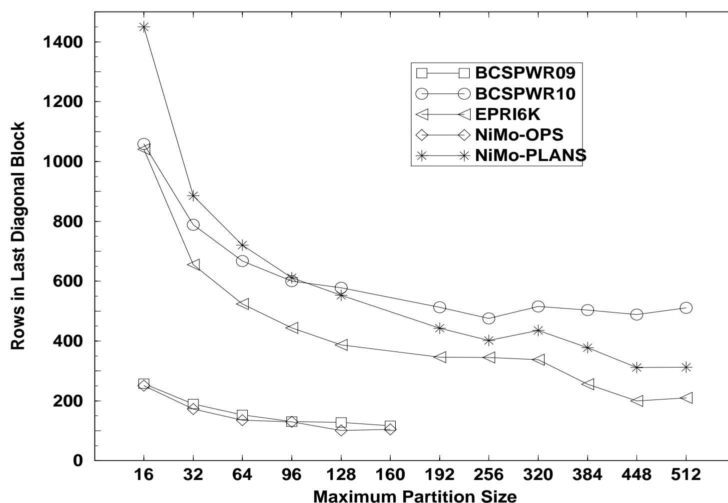

Figure 32: Last Diagonal Block Size for Power Systems Matrices after Partitioning with the Node-Tearing Algorithm

partition size of sixteen than for larger maximum partition sizes. This is empirical evidence that there are variations between data from operational analysis networks and larger planning networks. Additional evidence of differences are discussed below, both when we present example of the ordering for these matrices and when we discuss the performance of the parallel direct linear solvers.

Note that the maximum size of the diagonal blocks is inversely related to the size of the last diagonal block in figure 32. 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 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. The empirical performance data for the parallel solvers show that the best parallel performance is closely correlated with the minimum numbers of operations. In tables 3 through 6, we present summary statistics for the remaining power systems networks used in this analysis. In each table, the maximum partition size that yielded the best parallel performance is marked.

In figures 33 through 40, we provide an accompanying visual reference to the partitioning performance data presented in tables 3 through 6. We present two figures for each power systems network: the original matrix before ordering, and the matrix after partitioning and load-balancing for 8 processors. Partitioned graphs presented here have values of the maximum partition size that yielded the best empirical parallel block-diagonal-bordered direct linear solver performance.

When examining the unordered matrices, there appears to be significant differences between the power systems networks from the Niagara Mohawk Power Corporation — they have some block structure, while the Boeing-Harwell matrices and the EPRI matrix appear that they have been ordered with a minimum degree ordering. This can be seen in figures 26 and 27. Except for the fillin, the general pattern appears the same for the original matrix and the minimum degree ordered

Table 3: **BCSPWR10 — LU and Choleski Factorization Ordering Statistics**

number of nodes	5300
number of edges	8271
fillin for minimum degree ordering	14525

Maximum Partition Size	**Lower Triangular Matrix and Border**						
				Factor		**Fr or Bs**	
	N_{LB}	N_{FILLIN}	**Non Zeros**	**Total Ops**	**% Ops in Partitions**	**Total Ops**	**% Ops in Partitions**
16	1059	20040	33611	193384	17.0%	28311	41.5%
32†	789	19080	32651	190445	20.1%	27351	50.7%
64	668	21886	35457	261910	18.5%	30157	49.8%
96	600	22812	36383	295067	18.3%	31983	50.0%

† Best parallel direct block-diagonal-bordered sparse linear solver performance

Table 4: **EPRI6K — LU and Choleski Factorization Ordering Statistics**

number of nodes	6692
number of edges	10535
fillin for minimum degree ordering	10048

Maximum Partition Size	**Lower Triangular Matrix and Border**						
				Factor		**Fr or Bs**	
	N_{LB}	N_{FILLIN}	**Non Zeros**	**Total Ops**	**% Ops in Partitions**	**Total Ops**	**% Ops in Partitions**
16†	1041	15267	32494	207825	18.1%	25802	50.0%
32	655	14923	32150	242716	19.8%	25458	55.1%
64	524	15692	32919	271605	19.7%	26227	58.0%
96	444	15989	33216	309488	20.5%	26524	57.8%

† Best parallel direct block-diagonal-bordered sparse linear solver performance

Table 5: **NiMo-OPS — LU and Choleski Factorization Ordering Statistics**

number of nodes	1766
number of edges	2506
fillin for minimum degree ordering	2227

Maximum Partition Size	Lower Triangular Matrix and Border						
				Factor		Fr or Bs	
	N_{LB}	N_{FILLIN}	Non Zeros	Total Ops	% Ops in Partitions	Total Ops	% Ops in Partitions
16	281	3408	7680	22616	32.9%	5914	58.0%
32†	173	3152	7424	22762	35.7%	5658	65.2%
64	136	2959	7231	21147	40.7%	5465	69.6%
96	130	3259	7531	25727	37.1%	5765	67.7%

† Best parallel direct block-diagonal-bordered sparse linear solver performance

Table 6: **NiMo-PLANS — LU and Choleski Factorization Ordering Statistics**

number of nodes	9430
number of edges	14001
fillin for minimum degree ordering	13637

Maximum Partition Size	Lower Triangular Matrix and Border						
				Factor		Fr or Bs	
	N_{LB}	N_{FILLIN}	Non Zeros	Total Ops	% Ops in Partitions	Total Ops	% Ops in Partitions
16	1450	20300	43731	243945	19.4%	34301	52.2%
32†	886	19172	42603	250301	23.1%	33173	58.9%
64	721	19508	42939	265654	22.8%	33509	60.2%
96	612	19974	43405	287427	27.6%	33975	62.9%

† Best parallel direct block-diagonal-bordered sparse linear solver performance

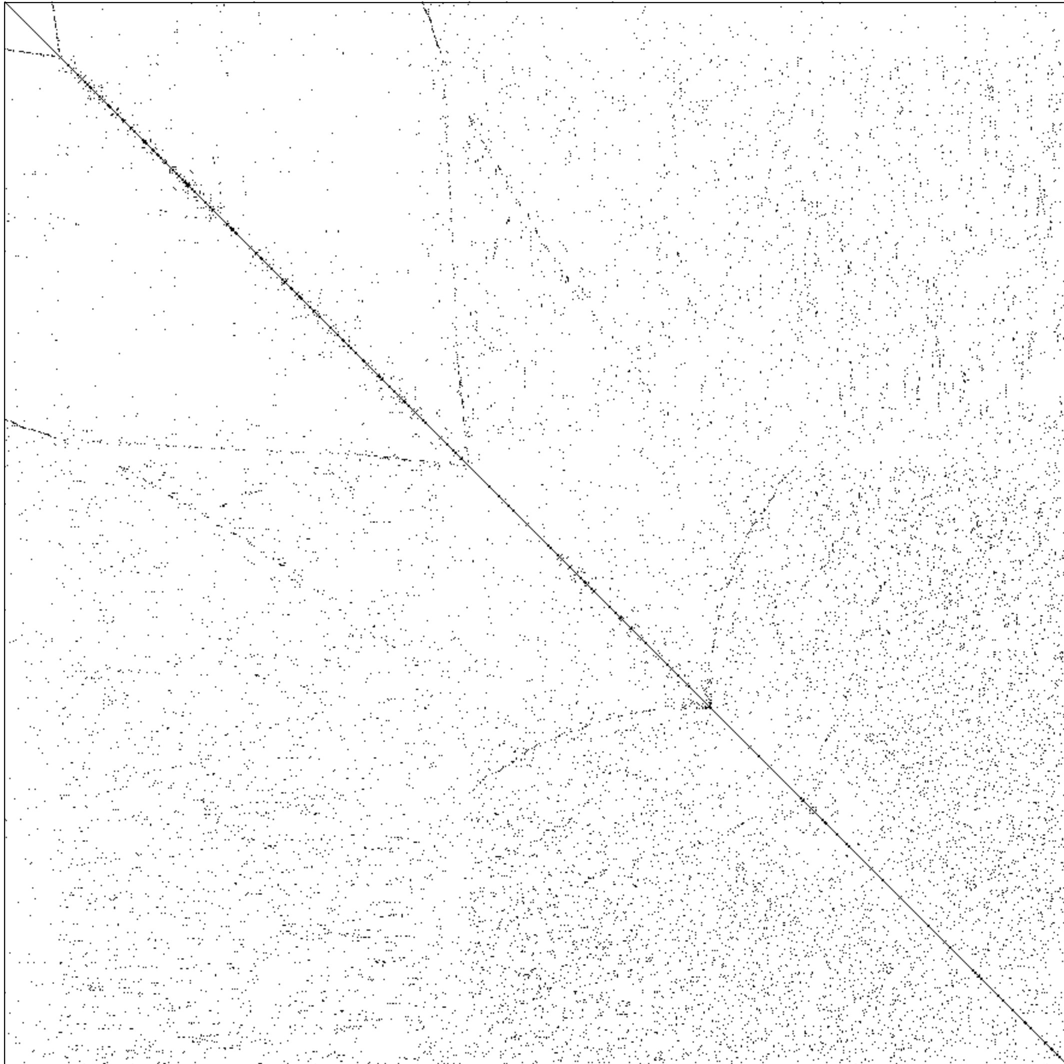

Figure 33: BCSPWR10 — Original Matrix

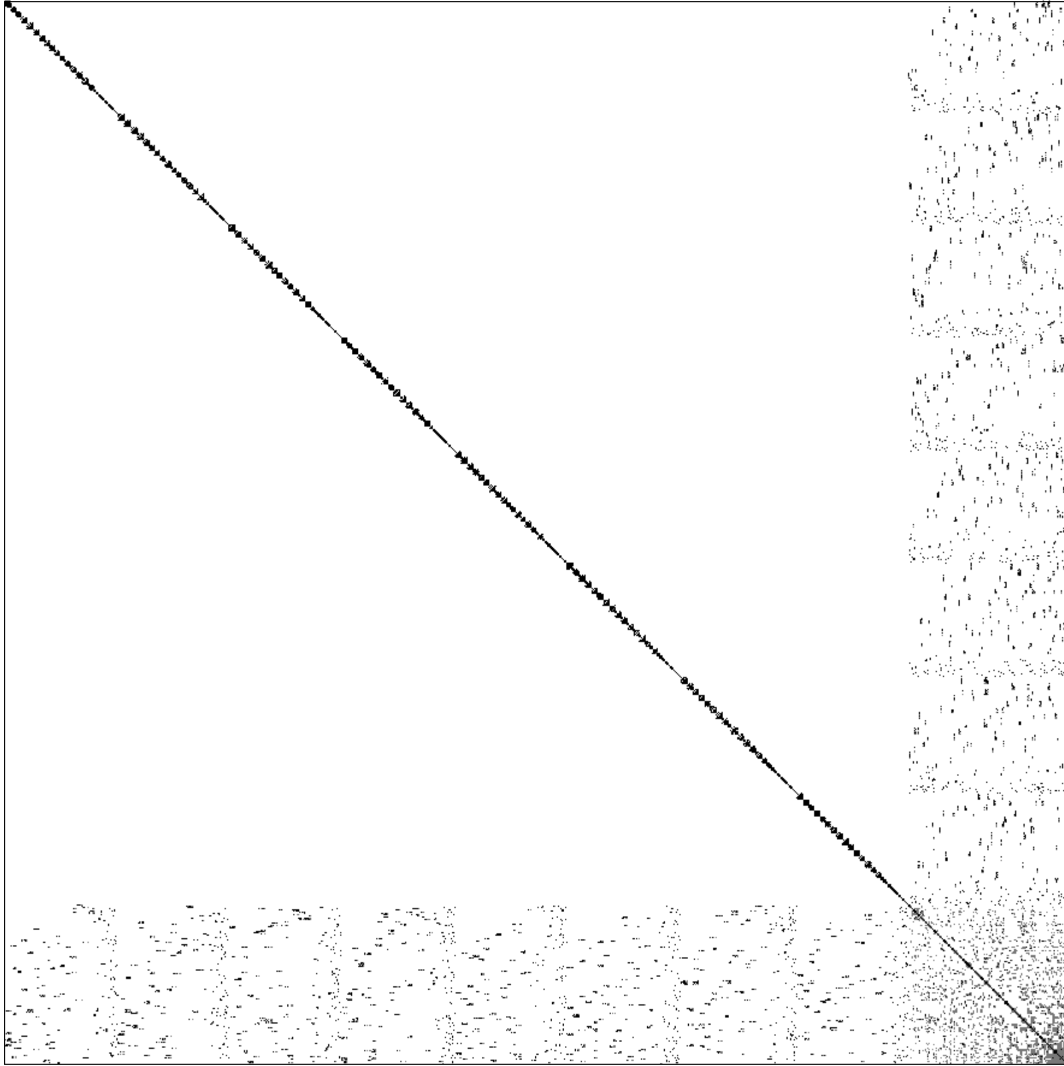

Figure 34: BCSPWR10 — Block-Diagonal-Bordered Form — Maximum Partition Size = 32 — Load Balanced for 8 Processors

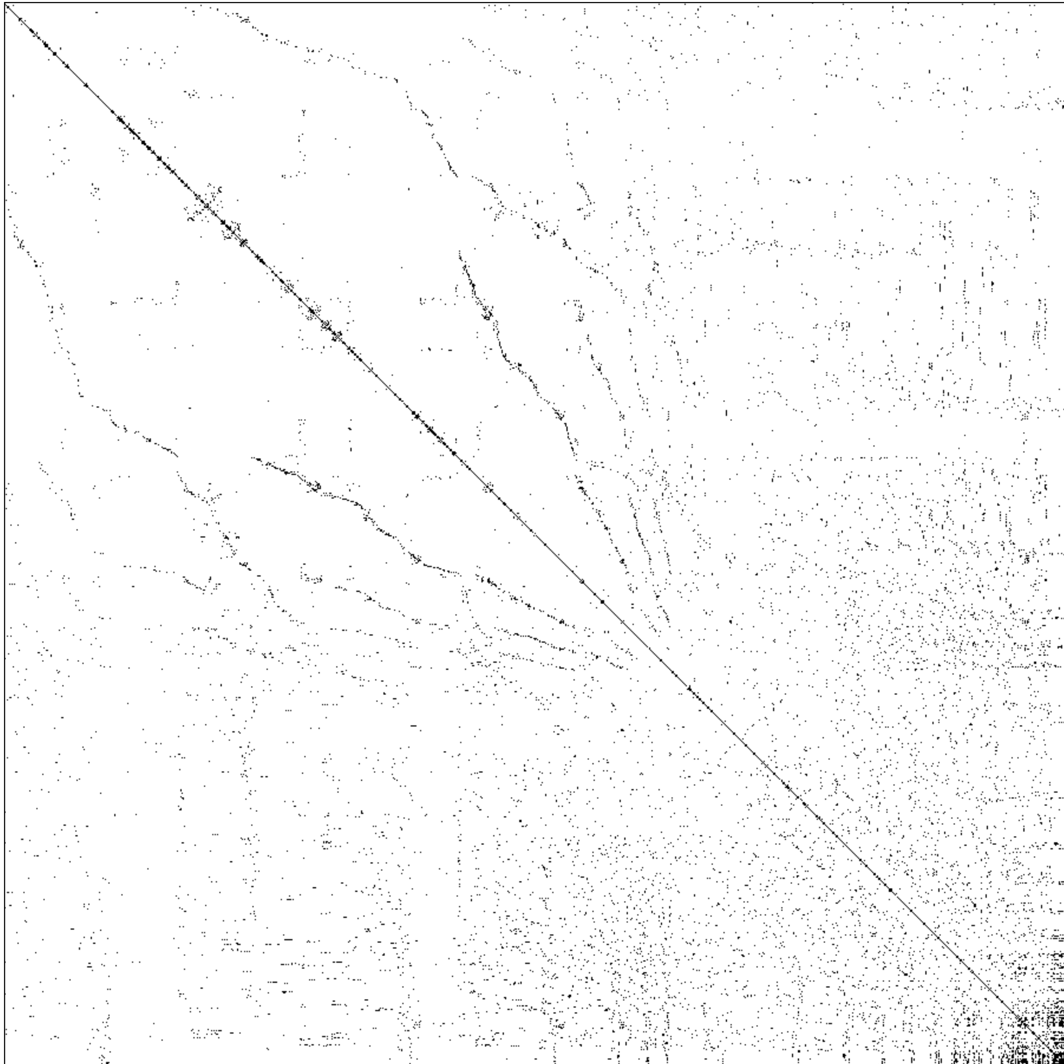

Figure 35: EPRI6K — Original Matrix

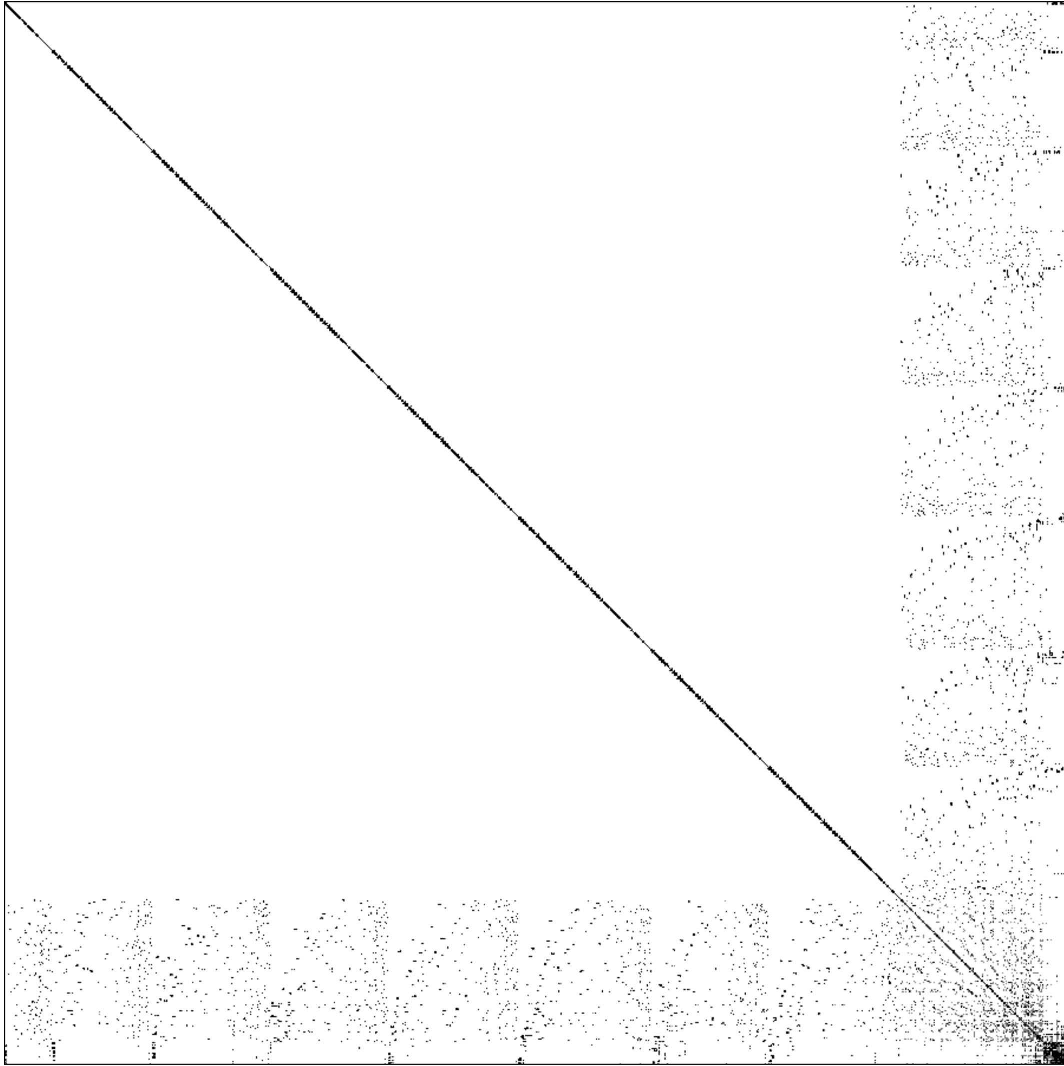

Figure 36: EPRI6K — Block-Diagonal-Bordered Form — Maximum Partition Size = 16 — Load Balanced for 8 Processors

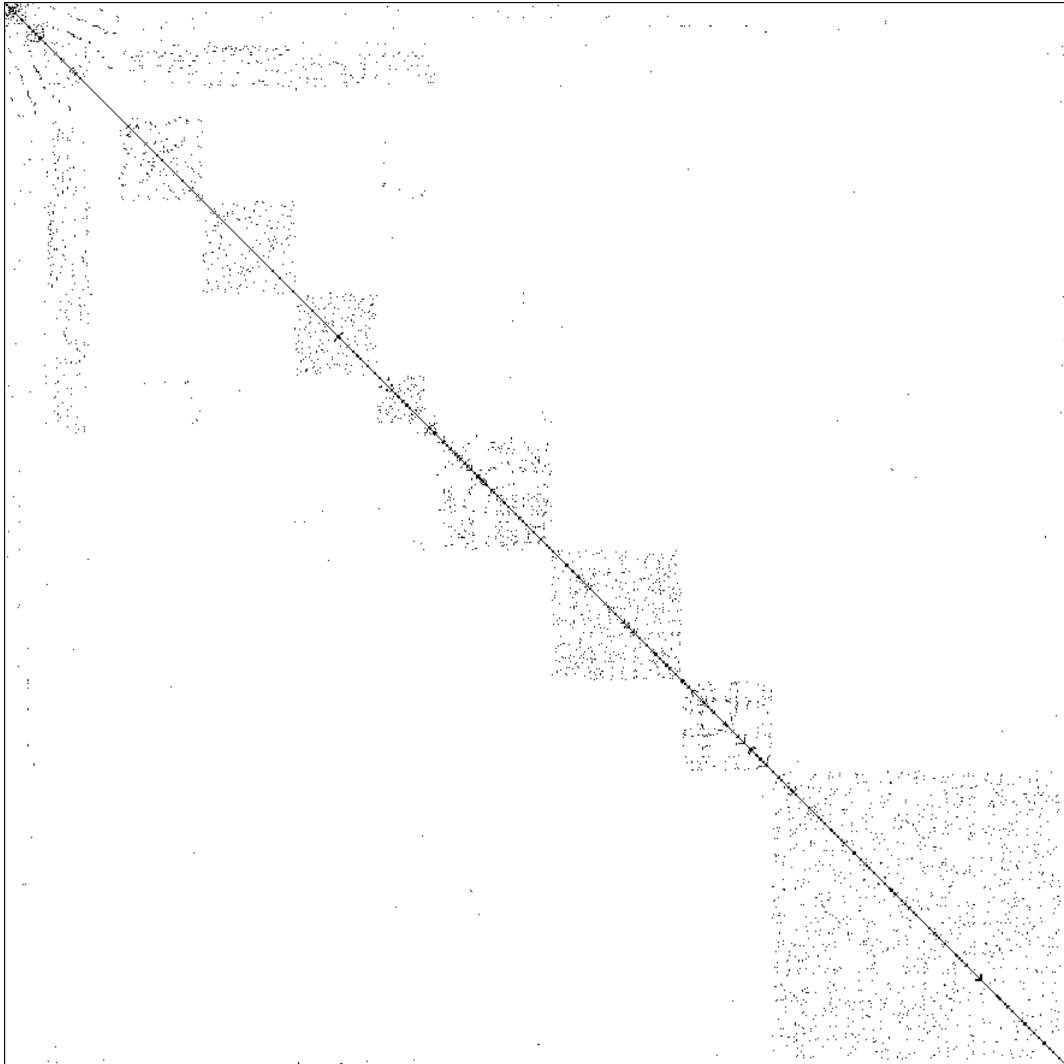

Figure 37: NiMo-OPS — Original Matrix

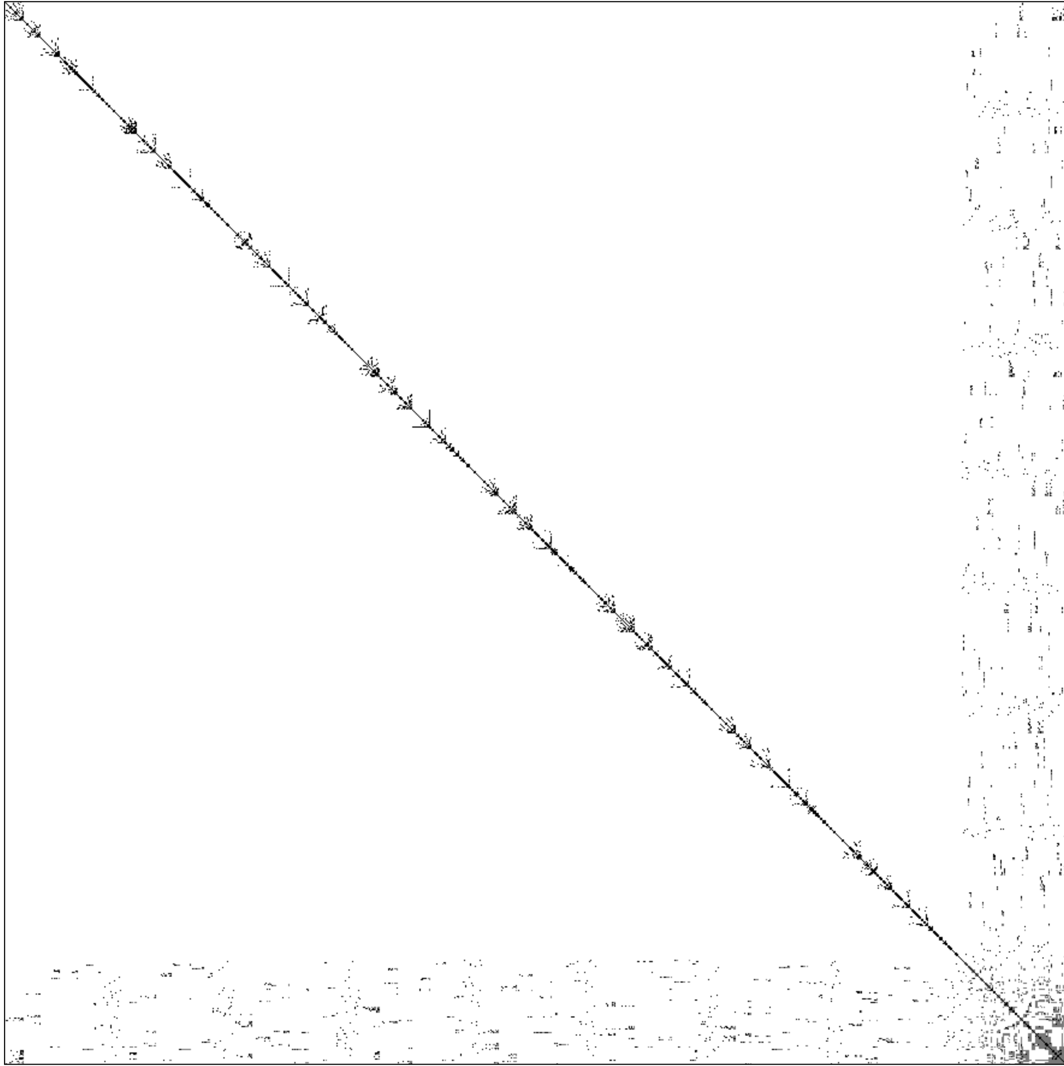

Figure 38: NiMo-OPS — Block-Diagonal-Bordered Form — Maximum Partition Size = 32 — Load Balanced for 8 Processors

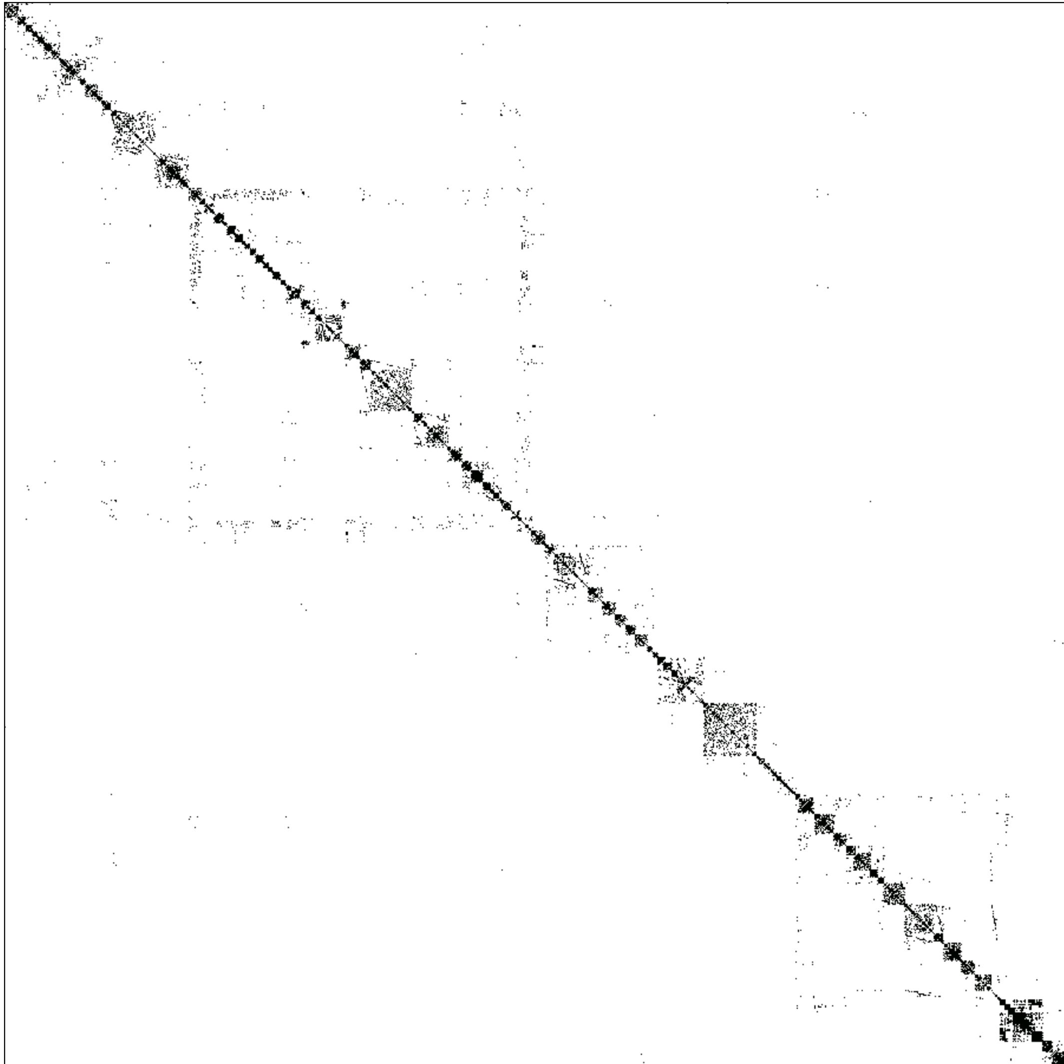

Figure 39: NiMo-PLANS — Original Matrix

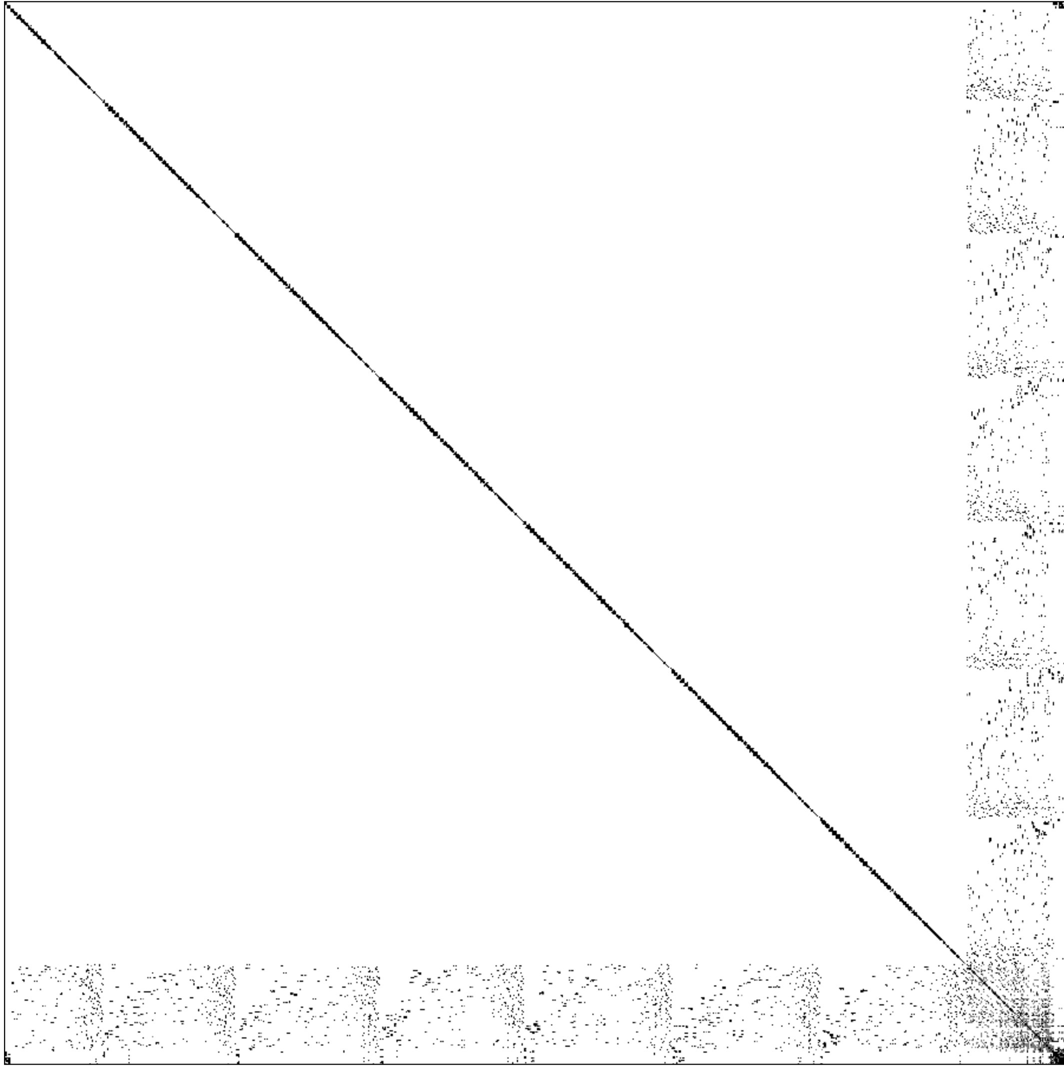

Figure 40: NiMo-PLANS — Block-Diagonal-Bordered Form — Maximum Partition Size = 32 — Load Balanced for 8 Processors

matrix. It is important to note that the block-diagonal-bordered matrices for the BCSPWR09 and NiMo-OPS matrices and the EPRI6K and NiMo-PLANS matrices look similar. The BCSPWR09 and NiMo-OPS matrices are for operational networks that are homogeneous and have very similar voltage distributions throughout. Meanwhile, the EPRI6K and NiMo-PLANS matrices are from planning applications, and one subsection of these networks includes some lower voltage distribution lines. This matrix has enhanced detail in the local area, with less detail in areas around the power utility's area of interest. This causes additional rows/columns in the borders and the last diagonal blocks, but our parallel block-diagonal-bordered direct solvers appear to have little difficult with efficiently solving these matrices. The small, highly connected graph section can be seen at the lower right-hand corner of the matrix in figures 36 and 40.

8.2 Empirical Results — Parallel Direct Sparse Solver Performance

We have collected empirical data for parallel block-diagonal-bordered sparse direct methods on the Thinking Machines CM-5 multi-computer for three solver implementations —

1. Choleski factorization and forward reduction/backward substitution for double precision variables
2. LU factorization and forward reduction/backward substitution for double precision variables
3. LU factorization and forward reduction/backward substitution for complex variables

for each of two communications paradigms —

1. active message based communications
2. asynchronous, non-blocking, buffered communications

for five separate power systems networks —

1. BCSPWR09
2. BCSPWR10
3. EPRI6K
4. NiMo-OPS
5. NiMo-PLANS

for 1, 2, 4, 8, 16, and 32 processors — and for four matrix partitioning — with a maximum of 16, 32, 64, and 96 graph nodes per partition.

As we examine the empirical results, we first describe the selection process to identify the matrix partitioning that yielded the best parallel empirical performance. This reduces the amount of data we must consider when examining the performance of the implementations. For the three solver implementations, there are increasing amounts of floating point calculations in double precision Choleski factorization, double precision LU factorization, and complex LU factorization, with the relative workload on a single processor of approximately 6:2:1, because Choleski algorithms have

only approximately one half the floating point operations of LU algorithms, and complex floating point operations require four separate multiplications and two addition/subtraction operations for a double precision multiply/add operation. While there are differing amounts of calculations in these algorithms, there are equal amounts of communications. We will present timing comparisons that illustrate how sensitive parallel sparse direct solvers for power systems networks are to communications overhead. This sensitivity is not totally unexpected, given the extremely sparse nature of power systems matrices. We next examine speedup for the three solver implementations, and examine speedups for both factorization and a combination of forward reduction and backward substitution. We then examine the performance of the load-balancing step by examining the timing data for each component of the algorithm. Lastly, we discuss the performance improvements achieved by using active message communications and the corresponding simplifications to the algorithm that were possible using this communications paradigm.

8.2.1 Selecting Partitioned Matrices with *Best* Parallel Solver Performance

There are many factors that can effect the performance of parallel direct sparse linear solvers. The primary factors are available parallelism, load balancing, and communication overhead. Our choice to order the power systems matrices into block-diagonal-bordered form provides a means to significantly limit the task graph to factor the matrix and to make all communications regular. We have shown in section 8.1 that the node tearing algorithm can partition the power systems network matrices into block-diagonal-bordered form and offer substantial parallelism in the diagonal blocks and borders.

There is one input parameter to the node-tearing algorithm, the maximum partition size, which when varied, effects the size of the diagonal blocks and the size of the borders and last diagonal block. We also presented tables 2 through 6 to illustrate that the amount of fillin and the amount of floating point operations varied as a function of the maximum partition size. When determining the partitioning with the best parallel direct block-diagonal-bordered sparse linear solver performance, we examined the empirical data collected from algorithm benchmark trials on the Thinking Machines CM-5 multi-computer to make those selections. Graphs presented in figures 41 and 42 illustrate the performance for LU factorization and for the combination of the forward and backward triangular solution steps for the Boeing-Harwell matrices; BCSPWR09 and BCSPWR10. Each graph has timing data for double precision LU factorization and for forward reduction/backward substitution. These graphs are on a log-log scale and show that for each power system network, a maximum of 32 nodes per partition yields the best overall performance for factorization.

We are considering software to be embedded within a more extensive power systems application, so we must examine efficient parallel forward reduction and backward substitution algorithms in addition to parallel factorization algorithms. These figures show that the time to factor the matrix is only approximately an order of magnitude greater than the time required to perform forward reduction and backward substitution on a single processor. This is significant because it illustrates the sparsity of these power system matrices. Due to the reduced amount of calculations in the triangular solution phases of solving a system of factored linear equations, these algorithms are often ignored when parallel Choleski or LU factorization algorithms are presented in the literature. For a

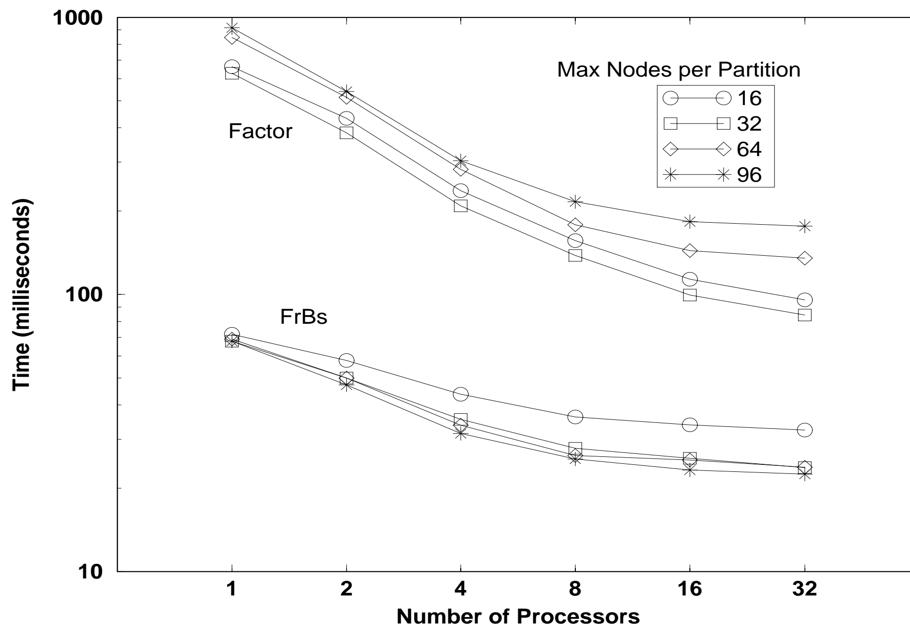

Figure 41: Parallel LU Factorization Timing Data — BCSPWR09 — Double Precision

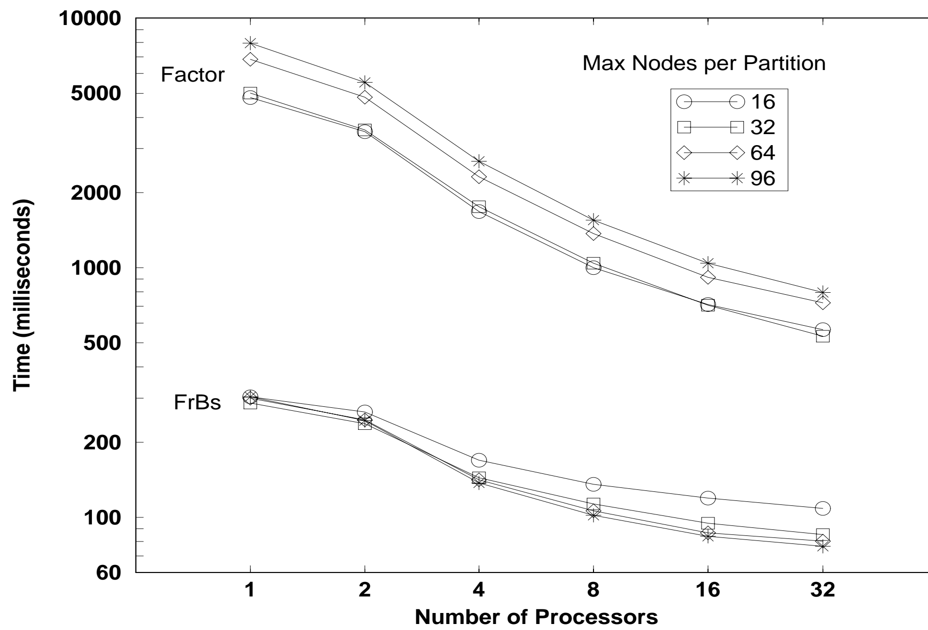

Figure 42: Parallel LU Factorization Timing Data — BCSPWR10 — Double Precision

dense matrix, the number of calculation to factor a matrix is $O(n^3)$ and the number of calculations to triangular solve the matrix is $O(n^2)$. For dense matrices as large as these two matrices, there would be a significant difference in wall-clock time between factorization and triangular solutions, a difference that is not present here. As a result, we must also consider the performance of the triangular solution step, especially if there will be dishonest *(re)use* of a factored matrix as it is repeatedly *(re)used* for multiple triangular solutions. Meanwhile, this order of magnitude difference in performance erodes for large numbers of processors, because it will be shown that there is better relative speedup for the factorization algorithms than for forward reduction and backward substitution.

In figure 42, we must consider the performance of the forward reduction/backward substitution step in selecting the better of the two partitioning; for a maximum of sixteen or 32 nodes per partition. Performance of the factorization algorithm for sixteen and 32 nodes per partition are nearly similar, although the performance of the triangular solution step is significantly better for 32 nodes per partition than 16 nodes per partition.

8.2.2 Comparing Timing Performance for Direct Solver Implementations

For the three solver implementations, there are increasing amounts of floating point calculations in double precision Choleski factorization, double precision LU factorization, and complex LU factorization, with a relative workload of approximately 6:2:1. While there are differing amounts of calculations in these algorithms, there are equal amounts of communications. We present timing comparisons for the three solver implementations in figures 43 through 47 for the five sample power system networks. These graphs each have six curves — three each for factorization and for the triangular solution. These graphs illustrate just how sensitive parallel sparse direct solvers for power systems networks are to the relative amount of communications overhead. This sensitivity is not totally unexpected, given the extremely sparse nature of power systems matrices.

These graphs show the time in milliseconds that it will take to factor and calculate a triangular solution for these matrices. These graphs also show the relative amount of floating point operations between implementations for a single processor. That ratio of time to factor the matrix decreases as additional processors are used when solving the sparse matrix. With constant amounts of communications, communications overhead has proportionally less of an effect when there are more calculations, and it is easier to hide communications behind calculations when there are more floating point operations to perform.

These graphs also illustrate some important information concerning parallel triangular solutions for Choleski factorization. First, while there is only half the calculations in the factorization step, there is no reduction in the number of calculations in the triangular solution step. To calculate the triangular solution, every non-zero coefficient in L is used once during forward reduction and every non-zero coefficient in L^T must also be used once during backward substitution. While it is possible to avoid explicitly performing the matrix transpose, one of the triangular solutions will require additional overhead because of the data must be explicitly or implicitly column oriented. This solution phase will require additional communications, $O(n_{non-zeros})$ as compared to $O(n_{rows})$. The number of non-zeros is greater than the number of row/columns in the matrix. For a single processor, there is the same amount of work when solving the factored equations for double precision LU and

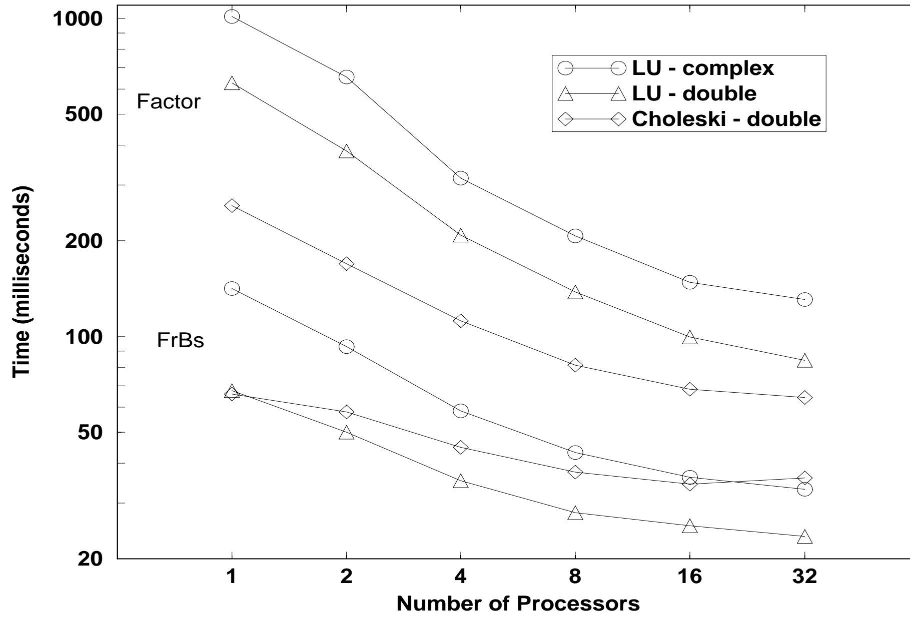

Figure 43: Parallel Choleski and LU Timing Comparisons — BCSPWR09 — Maximum Nodes per Partition = 32

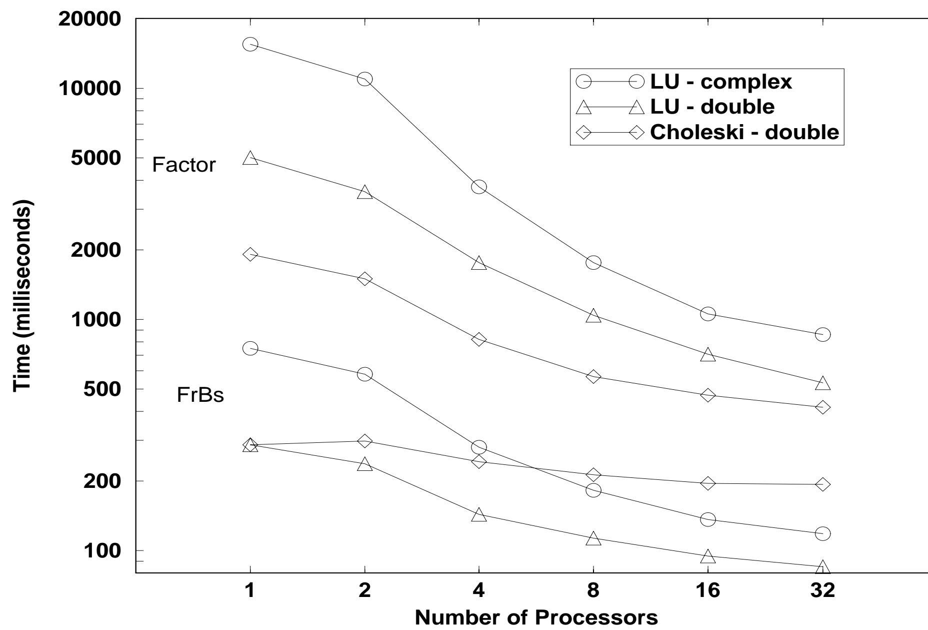

Figure 44: Parallel Choleski and LU Timing Comparisons — BCSPWR10 — Maximum Nodes per Partition = 32

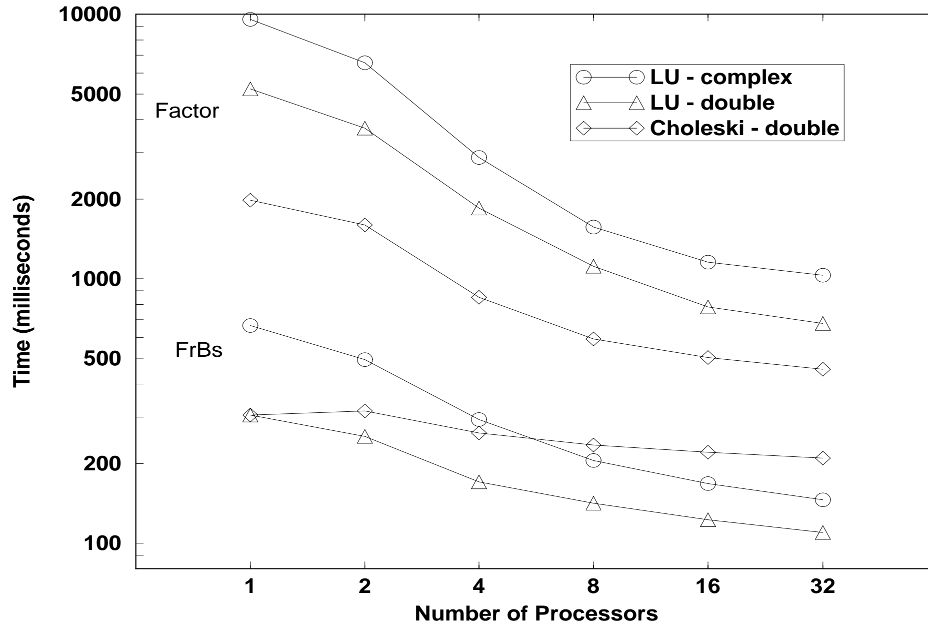

Figure 45: Parallel Choleski and LU Timing Comparisons — EPRI6K — Maximum Nodes per Partition = 16

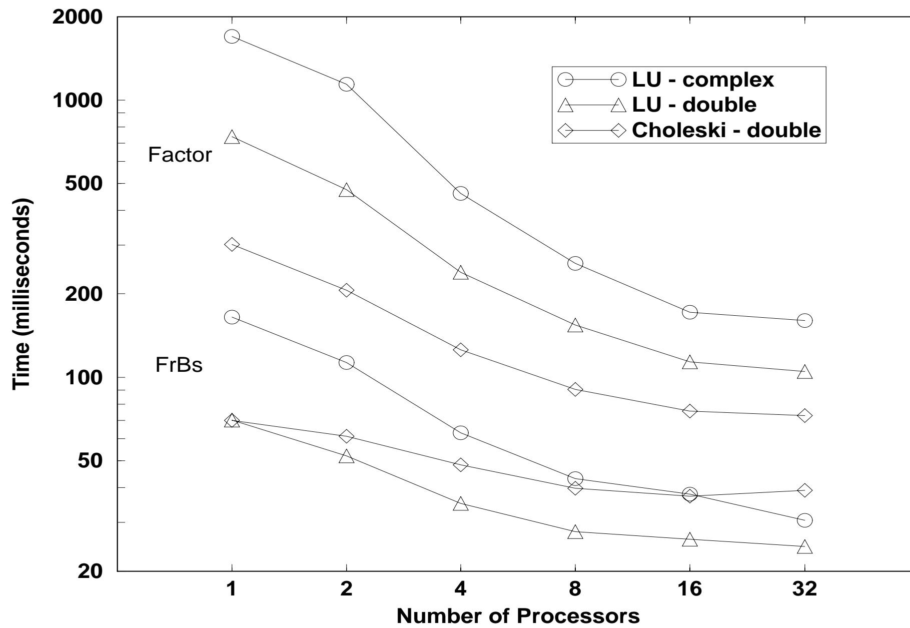

Figure 46: Parallel Choleski and LU Timing Comparisons — NiMo-OPS — Maximum Nodes per Partition = 32

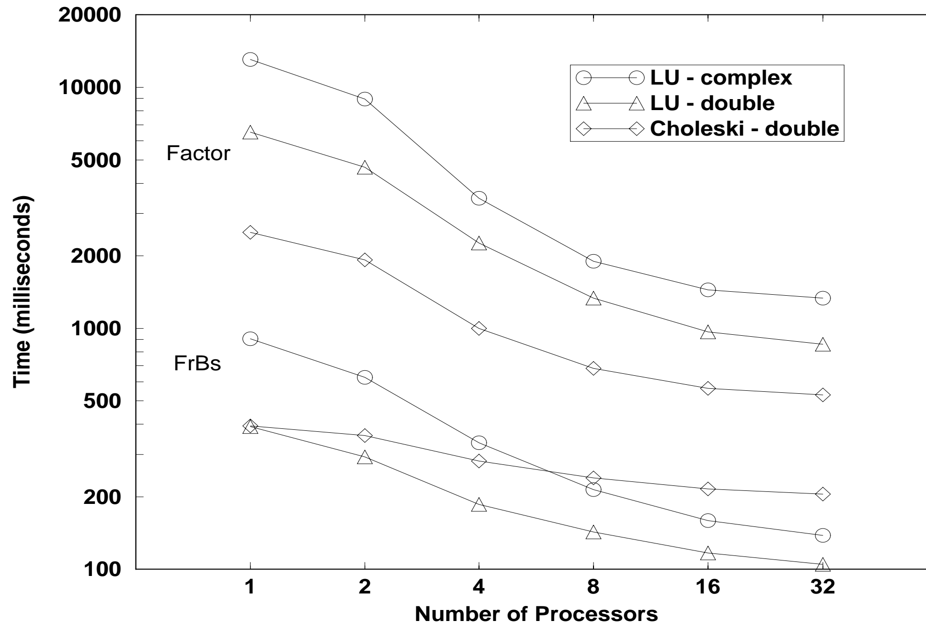

Figure 47: Parallel Choleski and LU Timing Comparisons — NiMo-PLANS — Maximum Nodes per Partition = 32

Choleski. However, the effect of the additional communications overhead has a noticeable effect on the slope of the curve representing the triangular solution for Choleski solvers. This phenomenon can be seen for all five power systems networks.

8.2.3 Examining Speedup

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.

Graphs of relative speedup calculated from empirical performance data are provided in figures 48 through 50 for the three parallel direct solver implementations. Figures 48 and 49 each have two families of speedup curves that show speedup for the five power systems networks examined in this research with separate families of curves for both factorization and the triangular solution. Figure 50 has three families of curves that show speedup for the five power systems networks for factorization, forward reduction, and backward substitution. Each curve plots relative speedup for 2, 4, 8, 16, and 32 processors,

Figure 48 illustrates that parallel performance of the complex LU factorization algorithm can be as much as eighteen on 32 processors for the BCSPWR10 power systems network. Meanwhile, factorization performance for the other data sets range from eleven to nearly eight. The BCSPWR10 matrix has the most calculations, and careful examination of figures 43 through 47 show that the empirical timing data for the BCSPWR10 matrix requires a greater relative increase in time from

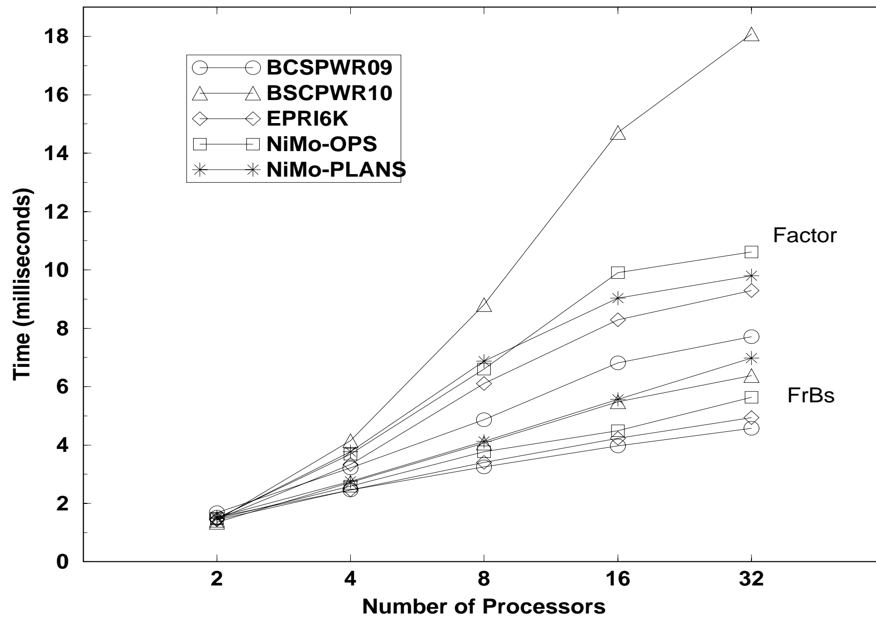

Figure 48: Relative Speedup — Complex Variate LU Factorization

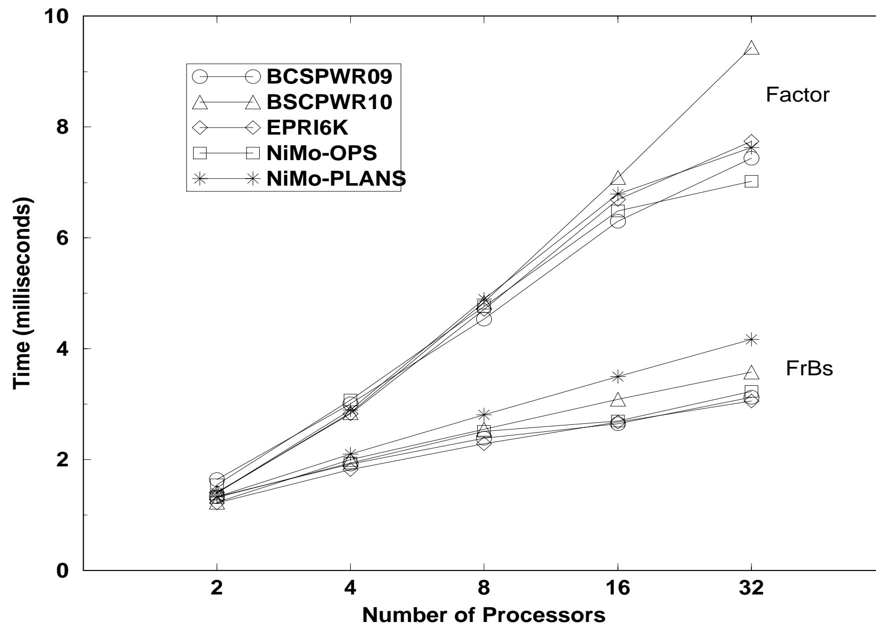

Figure 49: Relative Speedup — Double Precision LU Factorization

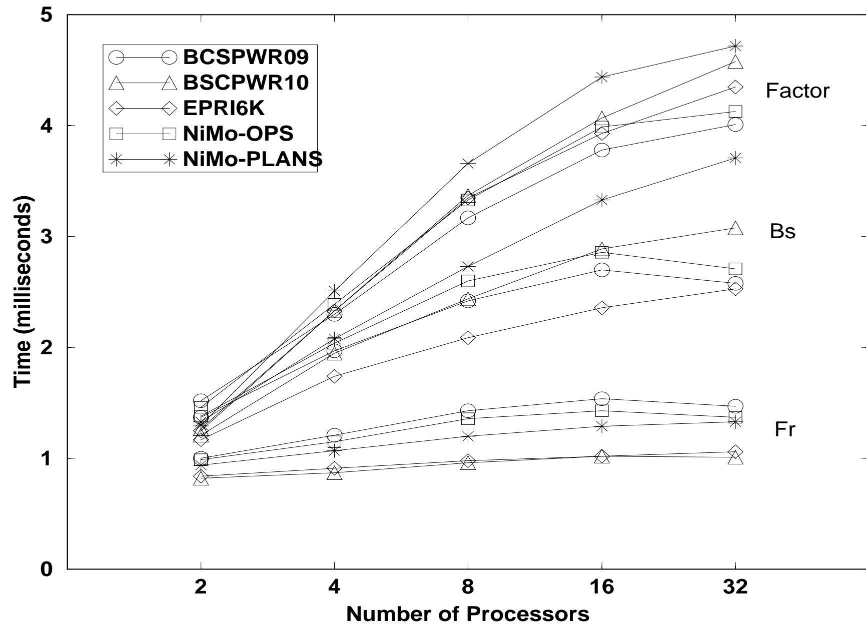

Figure 50: Relative Speedup — Choleski Factorization

double precision LU factorization to complex LU factorization for a single processor than other power systems networks. A significant increase in the time to factor the matrix on a single processor will cause significant increases in speedup. We believe that the unusually good performance of the parallel solver for the BCSPWR10 data is a result of caching effects — when the program is run on one or two processors, there is too much data on each processor to fit into the fast memory cache. When more processors are used, the entire memory can fit concurrently into the fast cache, and the program runs considerably faster with all cache hits. Meanwhile, complex triangular solutions provide speedups ranging between a high of eight and a low of four.

Figure 49 illustrates that parallel performance of the double precision LU factorization algorithm can be nearly ten on 32 processors for the BCSPWR10 power systems network. Factorization performance falls between eight and seven for the other four networks. Likewise, double precision triangular solutions provide speedups ranging from slightly greater than four to a low of three.

Parallel Choleski factorization yields speedups that are less than similar LU algorithms. Empirical data for relative speedup varies between five and four for 32 processors, as illustrated in figure 50. This figure also presents empirical speedup data for forward reduction and backward substitution. Due to the significant differences in the implementation of these triangular solution algorithms, empirical data are presented for both.

Backward substitution is the simplest algorithm with the lowest communications overhead — limited to only the broadcast of recently calculated values in x_m when performing the triangular backward substitution on $L^T_{m,m}$. Empirical relative speedup ranges from a high of 3.5 to 2.5. These speedups are only slightly less than speedups for backward substitution associated with LU factorization. Meanwhile, essentially no speedup has been measured for the forward reduction algorithm, due primarily to additional communications overhead for this implementation than either LU forward reduction or Choleski backward substitution. Additional communications is required to update the

last diagonal using data in the borders, and there are additional communications for reducing the last diagonal block. These additional communications occur because the data distribution forces interprocessor communications of partial updates when calculating values in y_m, rather than broadcasting values in y_m as in the LU-based forward reduction. Interprocessor communications increase from $O(n_{lb})$ to $O(n_{nz_lb})$, or from being proportional to the number of rows/columns in the last diagonal block to being proportional to the number of non-zeros in the last diagonal block. After minimum degree ordering of the last diagonal block, $n_{nz_lb} \gg n_{lb}$. Due to the characteristics of Choleski factorization, it is inevitable that either forward reduction or backward substitution would have to deal with the problem of increased interprocessor communications [20].

The sensitivities of these parallel algorithms to communications overhead is clearly apparent when comparing the relative speedups presented in figures 48, 49, and 50. For the three solver implementations, there are increasing amounts of floating point calculations in double precision Choleski factorization, double precision LU factorization, and complex LU factorization, with a relative workload of approximately 6:2:1; however, there are nearly equal amounts of communications. Communications in block-diagonal-bordered Choleski or LU factorization occurs in two locations — updating the last diagonal block using data in the borders and factoring the last diagonal block. There are twice as many calculations and twice as many values to distribute to processors holding data in the last diagonal block when updating the last diagonal block for LU factorization versus Choleski factorization, because LU factorization requires the update of $L_{m,m}U_{m,m}$ versus only $L_{m,m}$. There are equal amounts of communications for LU and Choleski factorization when factoring the last diagonal block.

When factoring the last diagonal block, the Choleski algorithm requires that data in $L_{m,m}$ be broadcast to all processors in the pipelined algorithm that perform the rank 1 update of the submatrix, However, the parallel algorithm for the last diagonal block requires only that $U_{m,m}$ be broadcast during the parallel rank 1 update. Communications overhead is nearly constant and the 6:2:1 relative workload of floating point operations result in relative speedups of 4:2:1 for the BCSPWR10 power systems network. Consequently, if speedups of 18 were required for a Choleski factorizations algorithm embedded in a real-time application, one way to reach those design goals is to improve the processor/communications performance by a factor of six to cause proportional reductions in the communications overhead. Another way that algorithm speedup could be achieved is by increasing the performance of the floating point capability of the processor, although, the ratio of communications-to-calculations performance ratio must stay equal to that in the CM-5 for these test runs.

8.2.4 Analyzing Algorithm Component Performance

We next present a detailed analysis of the performance of the component parts of the parallel block-diagonal-bordered direct linear solver. We present graphs that show the time in milliseconds to perform each of the component operations of the algorithm:

1. factor
    - diagonal blocks

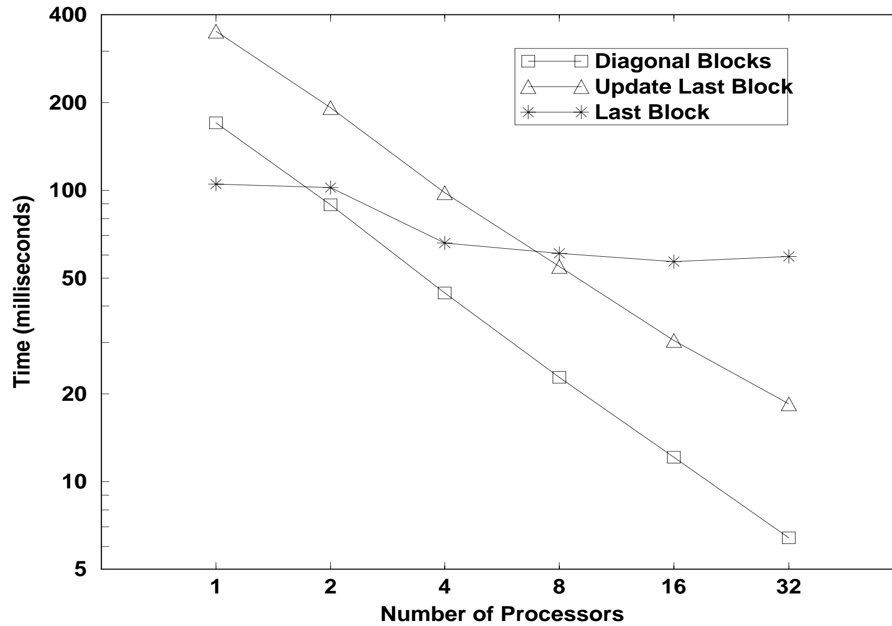

Figure 51: Algorithm Component Timing Data — Double Precision LU Factorization — BCSPWR09

- update last diagonal block
- last diagonal block

2. forward reduction
   - diagonal blocks
   - update last diagonal block
   - last diagonal block
3. backward substitution
   - last diagonal block
   - diagonal blocks

This detailed analysis of the parallel algorithm will demonstrate that the pre-processing phase can effectively load balance the matrix for as many as 32 processors and illustrate some of the limitations of the algorithm for certain classes of data sets. We present the data for two separate power systems networks: BCSPWR09 — a small, 1723 node network, from an operations application; and BCSPWR10 — a larger, 5300 node network, from a planning application. The operations network empirical performance data is presented in figures 51, 52, and 53. The planning network empirical performance data is presented in figures 54, 55, and 56.

For factoring the operations network, figure 51 illustrates that the performance of factoring the diagonal blocks and updating the last diagonal block have no apparent load balancing overhead. Also, communications overhead is minimal when updating the last diagonal block. The curve representing the performance to factor the last block is nearly straight, with a slope that denotes nearly perfect

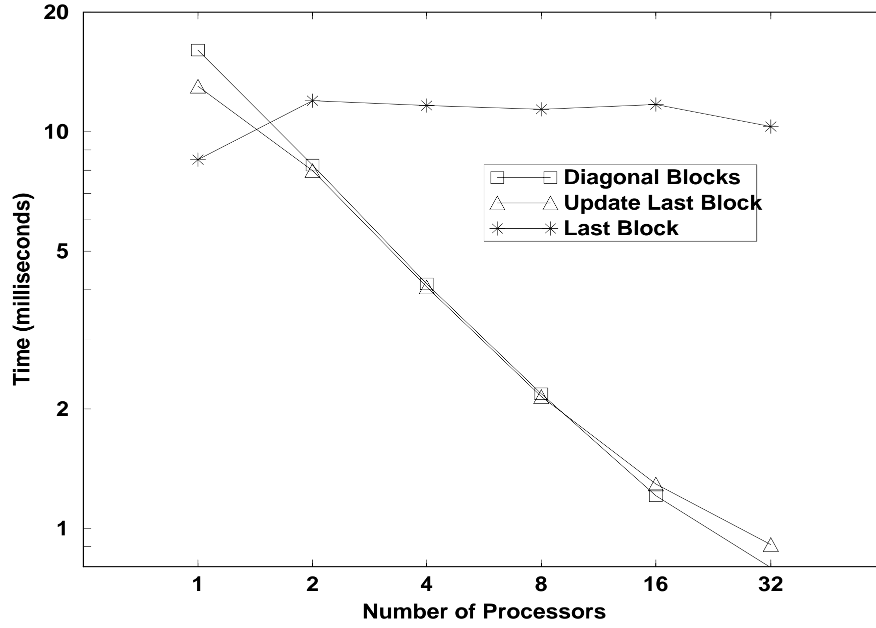

Figure 52: Algorithm Component Timing Data — Double Precision Forward Reduction — BC-SPWR09

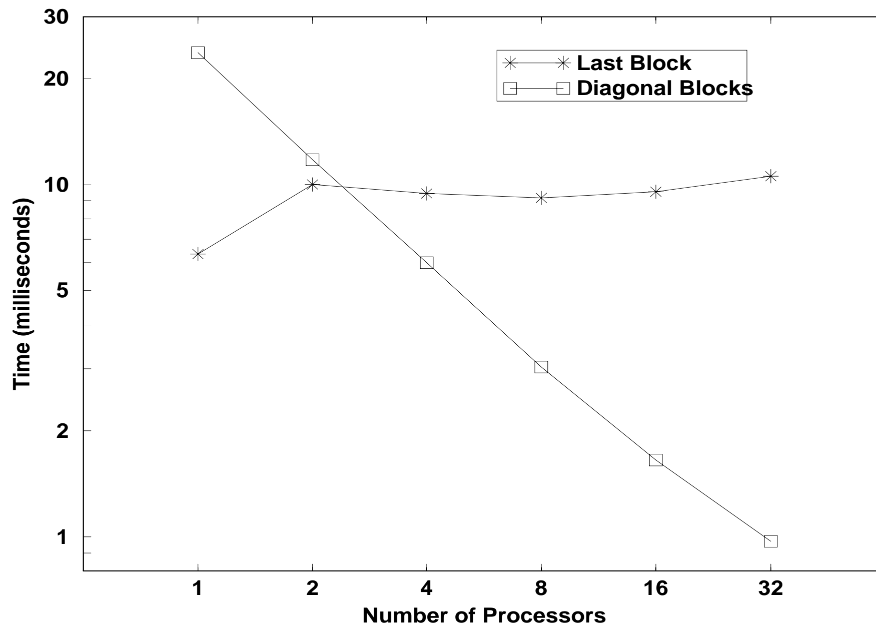

Figure 53: Algorithm Component Timing Data — Double Precision Backward Substitution — BCSPWR09

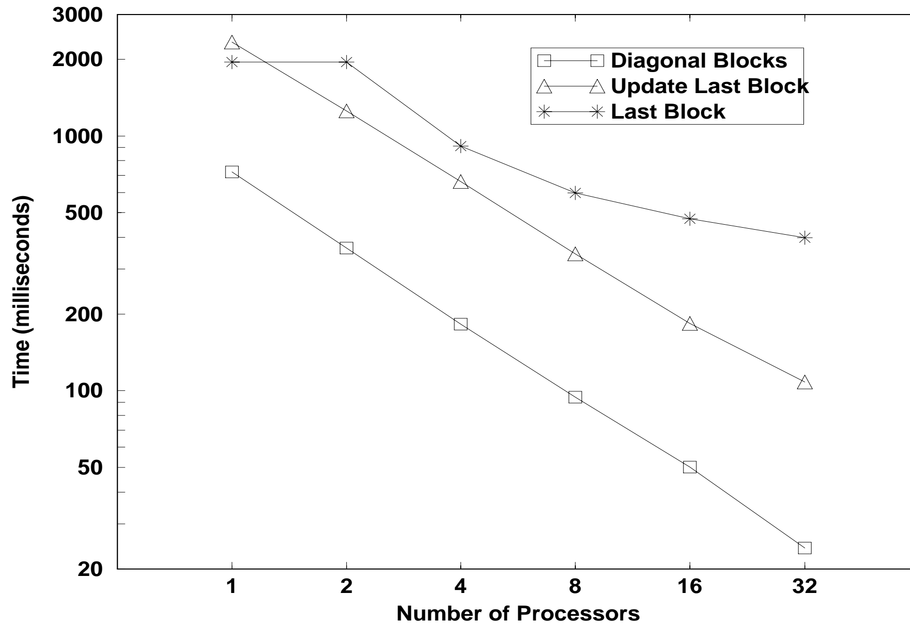

Figure 54: Algorithm Component Timing Data — Double Precision LU Factorization — BC-SPWR10

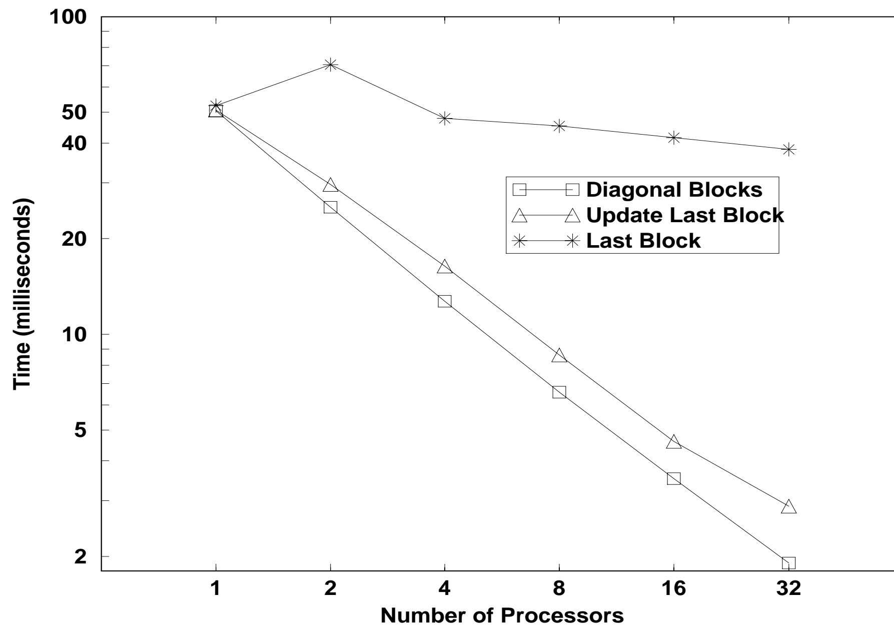

Figure 55: Algorithm Component Timing Data — Double Precision Forward Reduction — BC-SPWR10

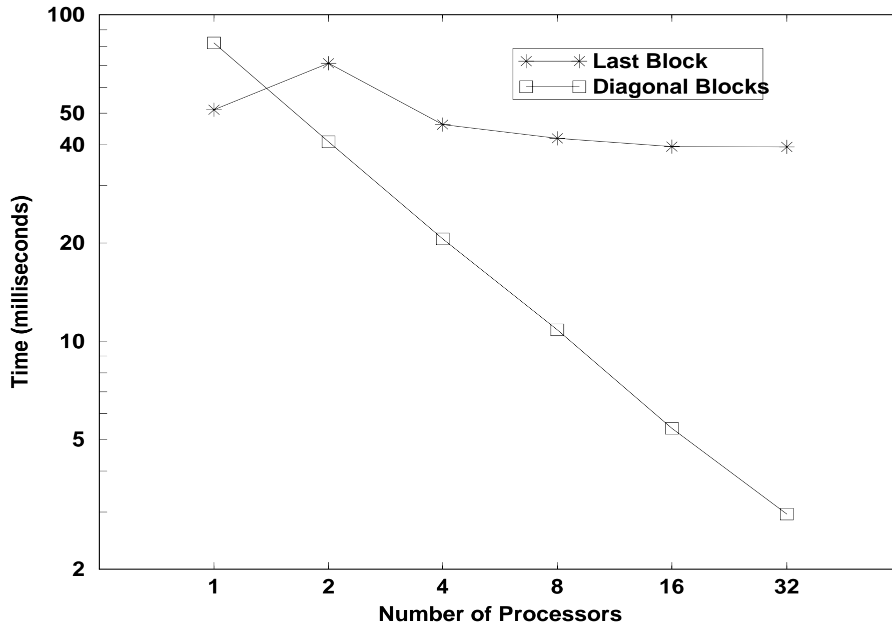

Figure 56: Algorithm Component Timing Data — Double Precision Backward Substitution — BCSPWR10

parallelism — relative speedup at each point is approximately equal to the number of processors. The curve representing the performance to update the last diagonal block is also nearly straight, although the slope of the curve shows that some overhead has occurred. On this log-log chart, the difference in slope is slight, Meanwhile, the curve representing the times to factor the last diagonal block show that this portion of the algorithm has poor performance — speedups are no more that 1.84 for sixteen processors and performance declines for 32 processors. Fortunately, the preprocessing phase was able to partition the network and generate matrices where the number of operations to factor the last diagonal block is significantly less than the number of operations to factor the diagonal blocks or update the last diagonal block. The relative time required to factor the three algorithm components is approximately 4:2:1. Relative speedup of the overall implementation is affected by the limited speedup of the last block, not to the extent of Amdahl's Law [12], but there are performance limits to this algorithm. Amdahl's Law would limit speedup for this algorithm to approximately seven — we have been able to achieve that speedup here, although the doubling of processors from 16 to 32 yields improvements in speedup from 6.30 to only 7.44.

For the triangular solutions, figures 52 and 53 show that we were able to get no speedup when performing the triangular solutions in the last diagonal block. The relative time required to reduce the algorithm components is approximately 4:3:2, so speedup for forward reduction would be limited to approximately 4.5 — a value that the algorithm could not reach with 32 processors due to load imbalance overhead in the reduction of the diagonal blocks and when updating the last diagonal block. The relative time required to back solve for the algorithm components is approximately 4:1, so speedup for backward substitution would be limited to approximately 5.0 — a value that the algorithm also could not attain due to additional overhead. Both triangular solution algorithms suffered load imbalance overhead, which was slight and not unexpected. We distributed the data to processors as a function of balanced computational load for factorization. Dense factorization

has $O(n^3)$ floating point operations, while dense triangular solutions have only $O(n^2)$ floating point operations. The sparse matrices associated with these power systems networks have significantly lower orders of computational complexity for the two components; however, factorization still has more calculations per row than triangular solves. As a result, some load imbalance overhead has been encountered in these algorithms.

We next examine parallel algorithm performance for a larger power systems network, BCSPWR10, that has four times the number of rows/columns and over eleven times the number of floating point operations. Figure 51 illustrates that the performance of factoring the diagonal blocks and updating the last diagonal block have little apparent load balancing overhead and communications overhead when updating the last diagonal block is minimal. Relative speedups are 29.9 for factoring the diagonal blocks on 32 processors and 21.6 for updating the last diagonal block on 32 processors. Performance for factoring the last diagonal block shows great improvement for this planning matrix when compared to the small operations matrix, BCSPWR09. While there is no measurable speedup for two processors due to the pipelined-nature of the algorithm, parallel performance improves respectably for larger numbers of processors. The timing data in figure 51 correspond to speedups of 4.9 for factoring the last diagonal block on 32 processors. The relative workload on a single processor is 2:6:5 for factoring this matrix. The extensive amount of operations to update and factor the last block make it imperative that good speedups have been obtained in these algorithm sections — in spite of the fact that both algorithm sections contain communications. The relative speedup obtained for factoring this matrix is 9.4 for 32 processors.

Performance of the triangular solvers on this larger, planning matrix are more promising than for the operations matrix. Figures 55 and 56 show that we were able to get limited speedup when performing the triangular solutions in the last diagonal block. The relative time required to reduce the algorithm components is approximately 1:1:1, so speedup for forward reduction would be limited to approximate 3 under Amdahl's law — a value that the algorithm was able to exceed, 3.6, with 32 processors. The relative time required to back solve for the algorithm components is approximately 2:1, so backward substitution speedup would be limited to approximately 3.0 — a value that the algorithm also did surpass, although only slightly. Both triangular solution algorithms suffered nearly no load imbalance overhead for this larger power systems network, in spite of the fact that we distributed the data to processors as a function of balanced computational load for factorization.

We have conducted similar detailed examinations into the performance of the algorithm for the three other power systems networks, and have obtained similar results. We draw the following conclusions from this detailed examination of the algorithm components:

- Power systems networks can vary greatly — not only are planning networks larger than operations networks, there are different characteristics to the matrices. Planning matrices are likely to have adequate workload in the last diagonal block that the this portion of the algorithm will yield speedups. Little speedup appeared possible when factoring the last diagonal block in operations matrices: however, this generates minimal cause for concern, because there is very little work involved in factoring that matrix relative to the other algorithm sections.

- The preprocessing stage was successful in generating matrices with block-diagonal-bordered form and balancing the processing load in the diagonal blocks and the update of the last

diagonal block.

- There are limitations to the number of processors that can be used to solve linear systems derived from these power systems networks due to the extreme sparsity of these matrices. We have shown that all algorithm components have good performance for as many as 32 processors, except those algorithm components working with the last diagonal block. There is overhead to fill pipelines, and it is questionable whether or not there are adequate floating point operations to keep the pipeline full for algorithms factoring or solving the last diagonal block. Increasing the size of the last block results in significant performance improvements, but no power system network examined here was large enough that the algorithm could get speedups greater than ten for parallel double precision block-diagonal-bordered LU factorization. Performance improved when the number of floating point operations increased for a complex-variate version of the algorithm and worsened when the number of floating point operations was reduced in a Choleski implementation.

8.2.5 Comparing Communications Paradigms

We have developed two versions of this parallel block-diagonal-bordered sparse linear solver, one version using an active message communications paradigm and the other using an asynchronous non-blocking buffered communications paradigm. These communications paradigms significantly modified the respective algorithms as seen in section 7.2. For all power systems networks examined, the largest relative workload when factoring or forward reducing the matrix is to update the last diagonal block. Increases in communications overhead in this portion of the algorithm could significantly affect parallel algorithm performance. In figure 57 we present a graph of the speedups obtained using active messages on the CM-5 versus asynchronous non-blocking buffered communications for factoring the matrices. Likewise, in figure 58 we present a graph of the speedups obtained using active messages versus conventional communications for forward reduction of the matrices. These graphs show that speedups can be as great as 1.6 for factoring the operations matrices with active messages, but speedups are less for the planning networks. Speedups are also greatest for the smaller operations matrices for updating the last diagonal block during forward reduction.

The speedups reported in figures 57 and 58 are relative to the entire time required to factor or reduce the matrices in parallel for the respective number of processors. While updating the last block has the most relative workload for a single processor, for larger numbers of processors, this relative workload changes significantly and the algorithm section for the last diagonal block assumes a larger relative workload because these algorithm sections are less efficient. To provide a better understanding of the algorithm component speedup, we present speedup graphs for active messages versus buffered communications in figures 59 and 60. Speedups for the factorization algorithm component are as great as 2.8, while speedups for the reduction component are as much as 14.8.

Active message communications have their greatest improvement when there are fewer operations to offset the additional communications overhead of the buffered communications. This is most evident when comparing speedups for factorization versus forward reduction in figures 59 and 60. It has been sufficiently difficult to obtain usable speedups for the triangular solutions with the active message communications paradigm. Performance reductions of 1.2 to 2.0 for buffered com-

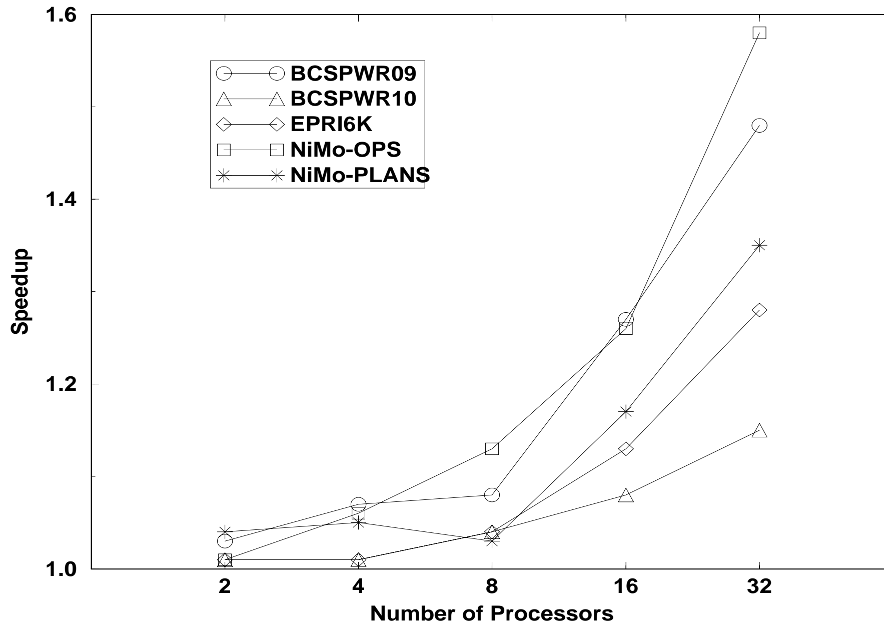

Figure 57: Speedup Active Messages versus Buffered Communications - Double Precision LU Factorization

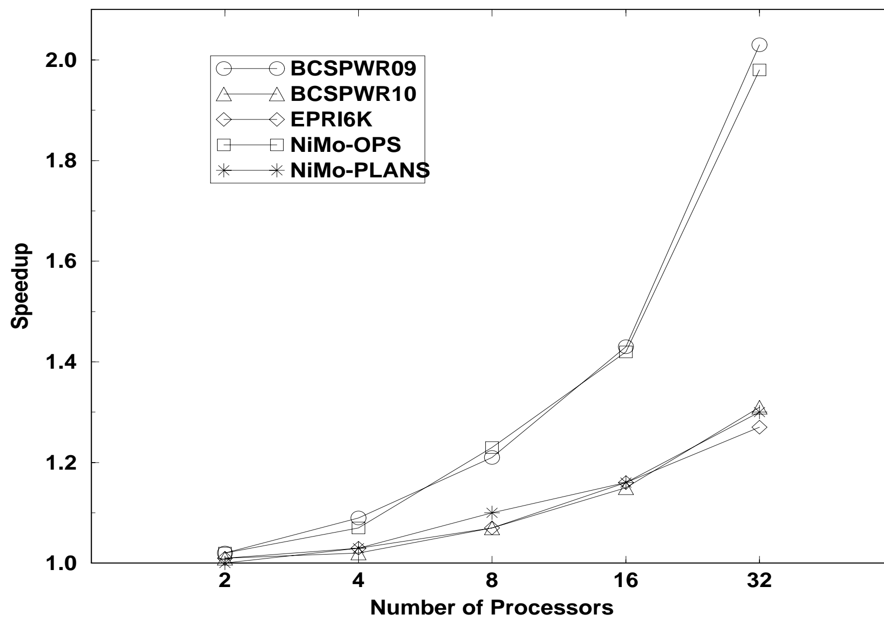

Figure 58: Speedup Active Messages versus Buffered Communications - Double Precision Forward Reduction

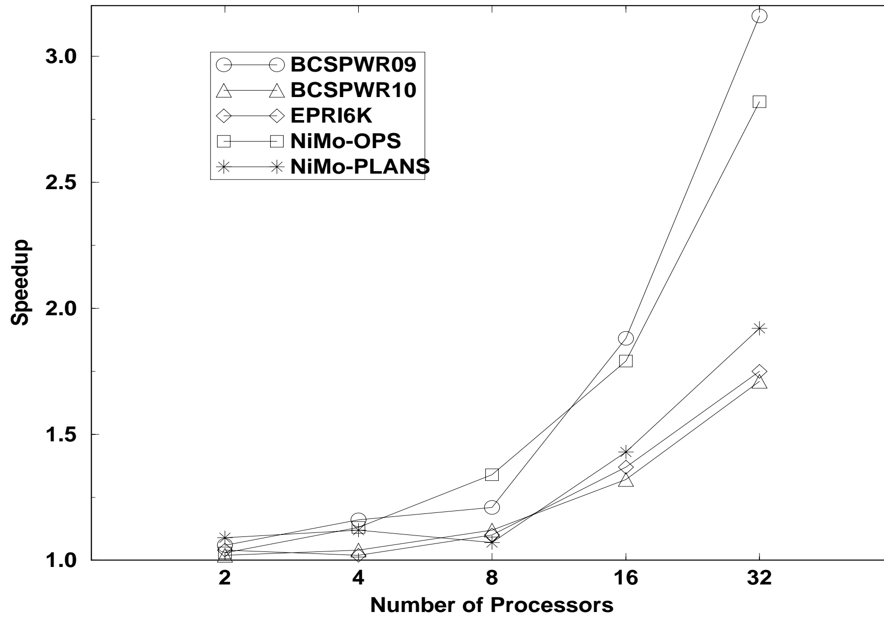

Figure 59: Speedup Active Messages versus Buffered Communications - Double Precision LU Factorization — Update Last Block

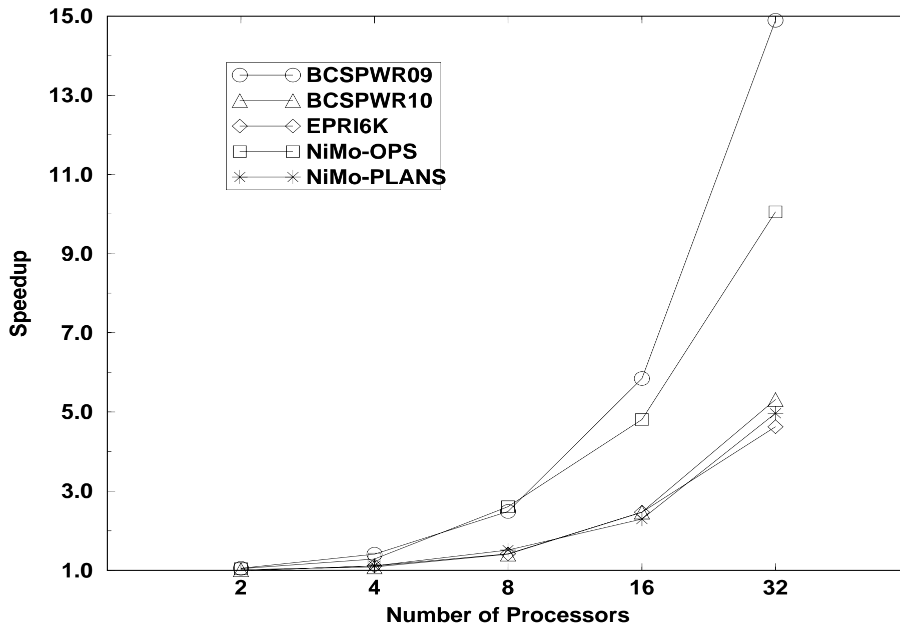

Figure 60: Speedup Active Messages versus Buffered Communications - Double Precision Forward Reduction — Update Last Block

munications would have a significant effect on the usability of this algorithm. For complex-variate implementations of these LU algorithms, the effect was somewhat less than for the double precision versions of the LU algorithms. Conversely, the Choleski implementation saw more pronounced speedups from active messages due to the reduced workload. There are the same number of buffered communications in all algorithms, with less data sent for the Choleski algorithms.

8.3 Empirical Results — Conclusions

We have extensively analyzed the performance of parallel linear solvers for power systems applications on the Thinking Machines CM-5. We have shown that the performance of our parallel block-diagonal-bordered sparse linear solvers can yield good speedups for LU factorization. Power system matrices are so sparse that we were able to show that relative speedups for parallel Choleski factorization and complex-variate LU factorization can differ by factors from two to greater than three. There is a six-fold increase in the number of calculations for complex LU factorization versus Choleski factorization. The sparsity in the matrices has an even greater effect on the triangular solution steps as it does on the factorization. Communications overhead when reducing or substituting in the last diagonal block is so great that there is no available speedup, so the performance of these algorithms becomes limited by Amdahl's law for the Thinking Machines CM-5 architecture and software.

8.3.1 Algorithm Performance on an IBM SP1 and SP2

We have ported this software to the IBM SP1 and SP2. The available communications on the IBM parallel machines required the use of the non-blocking buffered communications paradigm, because active messages are not implemented on this hardware. We chose to use the Message Passing Interface (MPI) for the communications language, because it is being developed as a communications standard for multi-processors with strong emphasis on optimizing message-passing performance. In table 7, we present empirical performance data for the IBM SP1 using MPI and the IBM SP2 using standard Transmission Control Protocol (TCP)/Internet Protocol (IP) based communications through the embedded communications switch. This preliminary data from the IBM scalable parallel processors (SPPs), based on workstation clusters with switched network communications, shows that the processor/communications ratio for MPI applications shows some promise, although there is too much latency in the TCP/IP based communications for the triangular solutions.

8.3.2 Algorithm Performance on Future SPP Architectures

While we design and implement algorithms on existing hardware, it is desirable to predict algorithm performance for future architectures. We can expect future SPPs to be similar to the IBM SPx series with features approaching the Cray T3D massively parallel processor (MPP) [28]. When comparing the single processor performance of the CM-5 (a 33 MHz Sparc microprocessor from Sun Microsystems [3]) with the node of an SP1 or SP2 (a 62.5 MHz IBM RS/6000 model 370 four command superscalar microprocessor), the RS/6000 is 6.6 times faster today when comparing empirical data from our algorithm run on a single processor. In the near-future, it will feasible to get four times the individual processor power that we see today, so it is conceivable that the (near-term)

Table 7: **Empirical Performance Data from the IBM SP1 and SP2 — EPRI6k — Complex Variate LU Solver**

Number of Processors	IBM SP1 using MPI		
	Factorization (seconds)	Forward Reduction (seconds)	Backward Substitution (seconds)
1	1.450000	0.050000	0.050000
2	1.165000	0.050000	0.045000
4	0.827500	0.045000	0.040000
Number of Processors	**IBM SP2 using TCP/IP**		
	Factorization (seconds)	**Forward Reduction (seconds)**	**Backward Substitution (seconds)**
1	1.460000	0.080000	0.080000
2	0.865000	0.165000	0.115000
4	0.607500	0.175000	0.180000
8	0.460000	0.203750	0.212500

next generation of SPP microprocessors will be 25 times as fast as today's microprocessors. Some of this power may come from placing multiple processors per SPP node,

If SPP node processor capability increases by a factor of 25, communications capabilities must improve by at least as much if the performance of parallel sparse direct linear solvers for power systems applications are to have equal or better performance on multiple processors. In other words, the communications-to-calculations ratio or granularity must remain constant. As we analyze communications performance of the parallel sparse direct solver, we must look at the two portions of the factorization algorithm that include communications: updating the last block and factoring the last block.

We implemented two versions of the parallel software on the CM-5 that updated the last diagonal block: with active message remote procedure calls (RPCs), and with non-blocking buffered communications. The active message based communications has latency of 1.6 μsecond for an RPC, which transmits a data payload of four words. The non-blocking buffered communications version of the algorithm utilized the CMMD communications library, which has 86 μsecond latency and 0.12 μsecond per word communications costs [3]. The CM-5 has a multi-tiered communications network with 40 megabytes-per-second bandwidth at the lowest layer [3]. The IBM SP2 has a 30 μsecond latency and 30 megabyte-per-second bandwidth in present configurations. In the near-future, we expect interprocessor communications for SPPs to improve significantly, with latency for buffered communications decreasing to levels that are available in MPPs like the Cray T3D today. We anticipate that buffered communications latency for SPPs, in the near future, will be only one μsecond, with 100 megabytes-per-second bandwidths between individual processors [28]. Per-word communications costs for this architecture should be less than 0.04 μsecond. For the active message communications version of updating the last diagonal block, we can't expect much improvement; however, for buffered communications, most communications messages are short, less than a kilobyte, so we can expect that communications performance in this section of the algorithm would improve by a factor of five. If communications latency decreases as significantly as we anticipate, the version of the algorithm to update the last diagonal block that would yield the best performance would be the buffered communications algorithm, but not keep pace with the performance improvement of individuals SPP processors.

The communications in the section of the CM-5 program that factors the last diagonal block uses active message s-copy commands, which have 23 μsecond latency and 0.12 μsecond per word communications costs [3]. Messages are short, again about a kilobyte, so we can expect that communications performance would improve by a factor of nearly four.

If we combine the three portions of the speedup analysis: improvements of a factor of 25 for the processor speed, and improvements of four or five in the communications speeds, it may not be possible to sustain the parallel speedup that we have obtained in this example program. Performance may be limited for 32 processors, however, strong performance with lessor numbers of processors should be sustainable, because communications overhead is not as great. Consequently, we should be able to obtain strong speedups with a single processor due to increased processor performance, while additional speedup due to parallelism may see less, although, multi-processor speedup should not decline to levels less than those speedups achieved for Choleski factorization.

If communications bandwidths between individual processors for our future machine improved

an order of magnitude, to a gigabyte-per-second, the prognosis for this algorithm would change. For gigabyte-per-second networks, communications to update the last-diagonal block could improve by a factor of 40 and communications to factor the last diagonal block could improve by a factor greater than 25. As a result, the computation-to-communications ratio would be preserved, if not improved, and similar or better parallel speedups could be expected.

This research was inspired by the low latency communications possible using active messages on the CM-5. We believe that SPP architectures, like the IBM SP2, may eventually provide similar low-latency communications for short messages because there are many parallel algorithms that can only be implemented efficiently with this type of interprocessor communications support. SPP hardware developers recognize that low-latency communications increase the utility of their computer and, consequently, improve market potential. The research community also recognizes this fact and university research will keep pressure on hardware developers to provide lower latency communications and higher interconnection bandwidths.

9 Conclusions

In this paper we present research into parallel block-diagonal-bordered sparse direct linear solver algorithms developed with special considerations to solve 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 direct linear 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.

We developed an implementation that offered speedups of nearly ten for double precision LU factorization and even greater speedups for complex variate LU factorization with 32 processors. Speedups for parallel block-diagonal-bordered Choleski factorization were less than for LU factorization, and there are formidable problems implementing forward reduction due to data distribution. We have performed additional research into parallel block-diagonal-bordered sparse Gauss-Seidel algorithms, an iterative linear solution technique [24, 25]. We are able to get substantially better speedups with the parallel Gauss-Seidel algorithm, although the only matrix types that there is assurance of convergence for Gauss-Seidel are diagonally dominant and positive definite matrices. Moreover, Choleski factorization is limited to either symmetric diagonally dominant or symmetric positive definite matrices. Consequently, we have compared the performance of parallel sparse Choleski solvers and parallel sparse Gauss-Seidel algorithms.

Power systems applications use sparse linear solvers in conjunction with either non-linear equation or differential-algebraic equation solvers. Often applications *reuse* a factored matrix numerous times, as a trade-off is made between the computational costs of repeated factorization and additional iterations in the non-linear equation solvers. A new LU factorization is not calculated every iteration – instead, an old LU decomposition is used to solve an approximate linear system. A new factorization is only calculated every few iterations. The cost of multiple linear solutions for *dishon-*

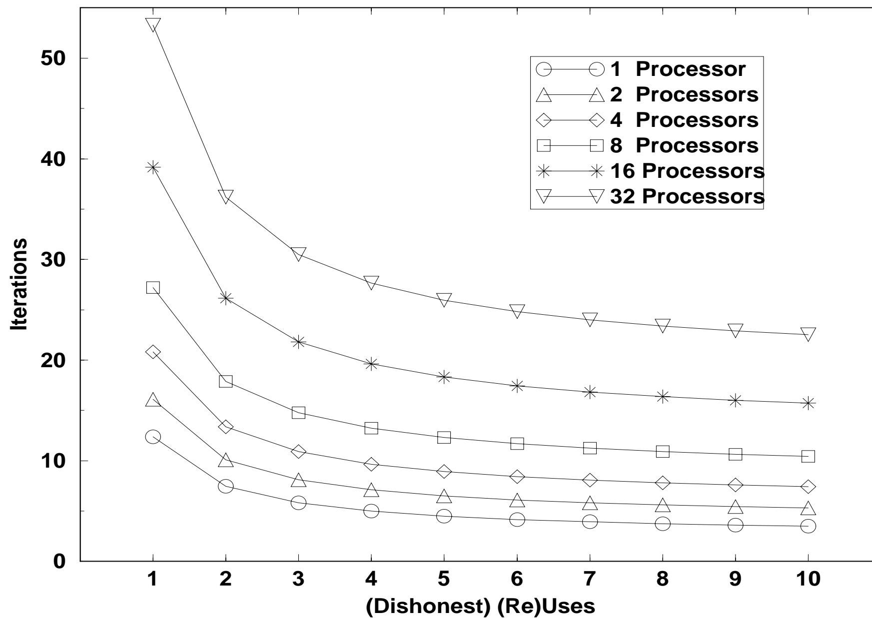

Figure 61: Gauss-Seidel Iterations as a Function of Dishonest Reuses of LU Matrix — BCSPWR09

est reuse would be a linear combination of the cost for factorization plus the cost for the repeated number of factorization *(re)uses*.

We compare the performance of parallel Choleski solvers with parallel iterative Gauss-Seidel solvers by determining the number of iterations for the parallel Gauss-Seidel given a number of *(re)uses*. Families of curves plotting the number of iterations versus the number of *dishonest (re)uses* are presented in figures 61 and 62 for one through ten reuses and one through 32 processors for power systems networks BCSPWR09 and BCSPWR10. The shape of the curves show that the largest number of iterations possible for a constant time solution occur for a single use of the factored matrix. As the factorization is *(re)used*, the cost to factor the matrix is amortized over the additional *(re)uses*. For large numbers of factorization *(re)uses*, the curve becomes asymptotic to $y = T_{Choleski} \div T_{Gauss_Seidel}$.

Figure 61 illustrates that 12 Gauss-Seidel iterations take as much time as a single factorization and triangular solution for the BCSPWR09 operations matrix on a single processor. Meanwhile, only four iterations per solution would equal the time for 10 *dishonest (re)uses*. However, when 32 processors are available, 54 Gauss-Seidel iterations could be performed in the same time as a single direct solution, and 24 iterations per solution for 10 *dishonest (re)uses*. Figure 62 illustrates similar performance for the BCSPWR10 power systems matrix — nearly 120 Gauss-Seidel iterations could be performed in the same time as a single direct solution for 32 processors, and 55 iterations per solution for 10 *dishonest (re)uses*. Given that there are good starting points for each successive iterative solution, there is a strong possibility that the use of parallel Gauss-Seidel should yield significant algorithmic speedups for diagonally dominant or positive definite sparse matrices.

The parallel block-diagonal-bordered direct solvers, presented in this paper, address the most difficult power systems applications to implement on a multi-processor — solutions relating only to power system networks. Load-flow has the smallest matrices and the fewest calculations due

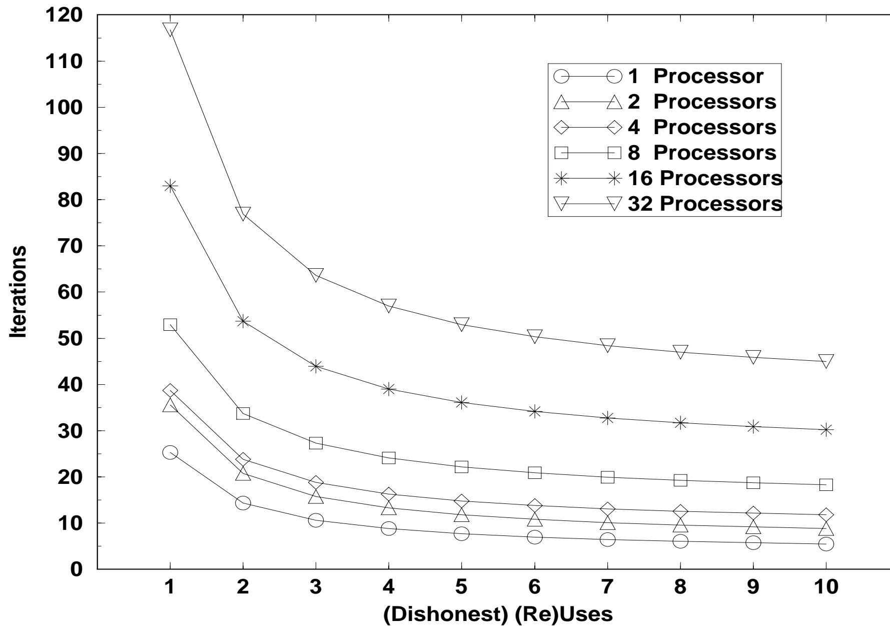

Figure 62: Gauss-Seidel Iterations as a Function of Dishonest Reuses of LU Matrix — BCSPWR10

to symmetry and lack of requirements for pivoting to ensure numerical stability. LU factorization of network equations for decoupled solutions of differential-algebraic equations has additional calculations, but often is solved without numerical pivoting. We have shown in this paper that by simply increasing the number of floating point operations by a factor of six, parallel speedup of the algorithm improves significantly.

Parallel block-diagonal-bordered sparse linear solver algorithms can readily be extended to applications that have power systems networks as a small portion of a larger matrix, for example, the entire system of linearized differential-algebraic equations encountered in transient stability analysis or small-signals analysis applications. These applications add many natural blocks of linearized differential equations that significantly increase the size of the matrix. The linearized differential equations are less-sparse than the network equations and may require pivoting to ensure numerical stability. Pivoting for this matrix would be limited to within diagonal-blocks to place limits on fillin, but the efficient static data structures would need to be replaced by less-efficient dynamic linked-list-based data structures. Any of these modifications would increase computational workload — work that does not require interprocessor interactions. As a result, any modifications to algorithms to include these additional features would improve parallel speedup, for the Thinking Machines CM-5 and for future machines that will be significantly faster than the MPPs and SPPs of today.

Acknowledgments

We thank Alvin Leung, Nancy McCracken, Paul Coddington, Chris Pottle, 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] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. *Proceedings of the 1994 International Parallel Processing Symposium*, 1994.

[4] J. S. Chai and A. Bose. Bottlenecks in Parallel Algorithms for Power System Stability Analysis. *IEEE Transactions on Power Systems*, 8(1):9–15, February 1993.

[5] T. A. David. Performance of an Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization. Technical Report TR-92–14, University of Florida, Computer and Information Sciences Department, May 1992.

[6] T. A. David and I. S. Duff. Unsymmetric-Pattern Multifrontal Methods for Parallel Sparse LU Factorization. Technical Report TR-91-23, University of Florida, Computer and Information Sciences Department, September 1991.

[7] T. A. David and I. S. Duff. An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization. Technical Report TR-93-018, University of Florida, Computer and Information Sciences Department, March 1993.

[8] 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.

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

[10] 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).

[11] Electrical Power Research Institute, Palo Alto, California. *Extended Transient-Midterm Stability Program: Version 3.0 - Volume 4: Programmers Manual , Part 1*, April 1993.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] A. Gupta and V. Kumar. A Scalable Parallel Algorithm for Sparse Cholesky Factorization. In *SuperComputing '94*, pages 793–802. IEEE Computer Society and ACM, November 1994.

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

[20] 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.

[21] W. Hoffmann. Solving Linear Systems by Direct Methods Related to Gaussian Elimination. In *Algorithms and Applications on Vector and Parallel Computers*. Elsevier Science Publishers B. V., 1987.

[22] G. Karypis, A. Gupta, and V. Kumar. A Parallel Formulation of Interior Point Algorithms. In *SuperComputing '94*, pages 204–213. IEEE Computer Society and ACM, November 1994.

[23] D. P. Koester, S. Ranka, and G. C. Fox. Parallel LU Factorization of Block-Diagonal-Bordered Sparse Matrices. NPAC Technical Report SCCS-550, Northeast Parallel Architectures Center (NPAC), Syracuse Univeristy, August 1993.

[24] D. P. Koester, S. Ranka, and G. C. Fox. A Parallel Gauss-Seidel Algorithm for Sparse Power System Matrices. In *SuperComputing '94*, pages 184–193. IEEE Computer Society and ACM, November 1994.

[25] D. P. Koester, S. Ranka, and G. C. Fox. A Parallel Gauss-Seidel Algorithm for Sparse Power System Matrices. NPAC Technical Report SCCS 630, Northeast Parallel Architectures Center (NPAC), Syracuse University, April 1994.

[26] 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.

[27] D. P. Koester, S. Ranka, and G. C. Fox. Parallel Choleski Factorization of Block-Diagonal-Bordered Sparse Matrices. NPAC Technical Report SCCS 604, Northeast Parallel Architectures Center (NPAC), Syracuse University, January 1994.

[28] W. Oed. The Cray Research Massively Parallel Processor System — Cray T3D. Technical report, Cray Research GmbH, November 1993.

[29] 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.

[30] 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.

[31] E. E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization*. PhD thesis, Stanford University, December 1992.

[32] E. Rothenberg and R Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *SuperComputing '94*, pages 783–792. IEEE Computer Society and ACM, November 1994.

[33] 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.

[34] 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.

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

[36] D. J. Tylavsjy, A. Bose, and et. al. Parallel Processing in Power Systems Computation. *IEEE Transactions on Power Systems*, 7(2):629–638, May 1992.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

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

[45] 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:

$$r_i^{(k)} = \min_t r_t^{(k)}, \tag{33}$$

where:

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

When factoring the kth row, the row with the minimum number of variables is selected, moved by elementary row and column exchange rules to the kth row, and then factored. Algorithms to implement this iterative formula are best described using the graph theoretical explanation of fillin presented in figure 9. 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* [17]. 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 63. 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 9, 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

$$\Lambda_{G_\nu}(\upsilon) = (\Lambda_G(\upsilon) \cup \Lambda_G(\nu)) - \{\nu, \upsilon\}. \tag{34}$$

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 [17]. 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

$G \leftarrow$ *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* $\upsilon \in \Lambda_G(\nu)$
        $\Lambda_{G_\nu}(\upsilon) \leftarrow (\Lambda_G(\upsilon) \cup \Lambda_G(\nu)) - \{\nu, \upsilon\}$
    end for
    $G \leftarrow G_\nu$
end while

Figure 63: The Minimum-Degree Algorithm

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 64 through 67. The example graph, presented in figure 64, 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 65. 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 66 illustrates the ordered graph, note that only the labels on the modes have changed from figure 64. To illustrate the effect of ordering the matrix, the matrix sparsity structure for the original and ordered graphs are presented in figure 67. 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.

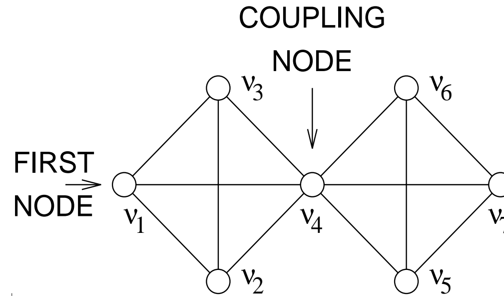

Figure 64: Graph for a Node-Tearing Example

	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 65: Example Contour Tableau

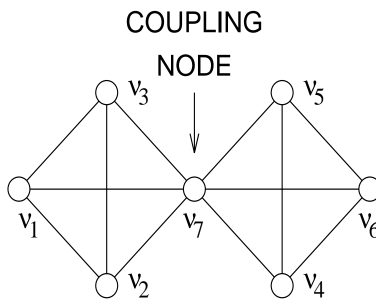

Figure 66: Relabeled Example Graph

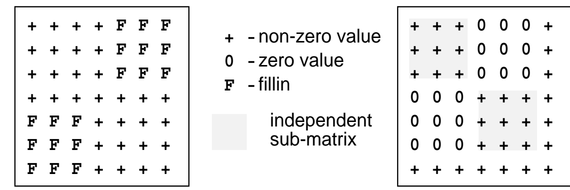

Figure 67: Matrix Representation of the Example Graphs