Parallel LU Factorization of Block-Diagonal-Bordered Sparse Matrices

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

NPAC Technical Report — SCCS 550

22 August 1993

Abstract

Research is being performed to examine the applicability of parallel direct block-diagonal-bordered sparse matrix solvers for irregular sparse matrix problems derived from the electrical power systems community. Moreover, we believe that this research also has utility for irregular sparse matrix factorization for applications where the data is hierarchical. Direct block-diagonal-bordered sparse matrix algorithms exhibit distinct advantages when compared to current general parallel direct sparse matrix solvers. Task assignments for numerical factorization on distributed-memory multi-processors depend only on the assignment of independent blocks to processors and the processor assignments of data in the last diagonal block. In addition, data communications are significantly reduced and those remaining communications are generally uniform and structured. Parallel block-diagonal-bordered sparse matrix algorithms require modifications to the traditional sparse matrix preprocessing phase that include an explicit *load balancing* step coupled to a specialized *ordering* step to uniformly distribute the workload throughout a distributed-memory multi-processor. In this paper, we propose a new preprocessing phase that includes specialized *ordering* and *load balancing* techniques, we describe in detail the mathematics of block-diagonal-bordered sparse matrix solvers, and we present implementation details and empirical parallel performance data for a prototype direct block-diagonal-bordered sparse matrix solver running on a Thinking Machines CM-5 using message passing.

1 Introduction

Solving sparse linear systems practically dominates scientific computing [15], but the performance of sparse matrix solvers have tended to trail behind their dense matrix counterparts [12]. 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. The limited success with efficient sparse matrix solvers is not surprising, because general sparse matrix solvers require more complicated algorithms and significantly more complicated data structures that require irregular memory reference patterns. The irregular nature of these problems has aggravated the problems of implementing sparse matrix solvers on vector or parallel architectures: efficient algorithms for these classes of machines require regularity in available data vector lengths and in interprocessor communications patterns. The greater complexity and irregularity of sparse matrix computations make this field an area with active research as parallel sparse matrix solvers are developed and optimized for particular applications.

Our research has focused on applications from the electrical power systems community where portions of the sparse matrix are naturally in block-diagonal-bordered form and the remaining portions of the sparse matrix can be ordered to yield additional independent diagonal blocks and reduce the number of equations in the borders. These naturally block-bordered-diagonal sparse matrices occur when simulating power systems in order to examine the stability of electrical power system generators when a transient-type event occurs [1, 2, 6]. Given adequate computing power, electrical power system transient stability analysis could be incorporated into real-time control software for electrical power utilities, and a detailed view of an unfolding power grid failure could be provided to operators so that they could rapidly make informed, proactive decisions to minimize the extend of cascading power system network and generator failures. As a real-time grand challenge computing application, transient stability analysis would require a fast, efficient parallel sparse matrix solver.

These matrix forms remain static for extended periods of time because the sparse matrices represent actual electrical power distribution networks and electrical generators. Modifications to the electrical distribution networks are costly and represent the addition or removal of actual high voltage electrical distribution lines. Meanwhile, generators usually have high startup and shutdown costs, so they are brought on-line for periods usually lasting a minimum of several hours. Consequently, sparse matrices representing electrical power systems remain static for a sufficiently long period of time to justify the additional effort for special matrix ordering to minimize the number of fillin and the number of calculations and also justify the additional effort for balancing the number of calculations on individual

processors in order to ensure efficient use of parallel computing resources.

The work presented in this paper could significantly improve performance of electrical power systems transient stability analysis and also electrical power systems load flow analysis. Moreover, we believe that this research also has utility beyond this application and the techniques developed in this work can readily be extended to develop efficient parallel direct solvers for other irregular, hierarchical sparse matrices encountered in other scientific and engineering applications.

1.1 Parallel Direct Block-Diagonal-Bordered Sparse Matrix Solvers

Research is in progress to examine efficient parallel algorithms for the direct solution of the sparse systems of equations

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

when the sparse block-diagonal-bordered A matrix is factored into upper and lower triangular matrices, e.g., $A = LU$. Forward reduction and backward substitution steps are then required to solve for the vector x in the initial system of linear equations. The parallel direct solution of sparse matrices in block-diagonal-bordered form is performed in three distinct steps:

1. factorization,
    - factorization of the independent blocks,
    - updates to the last block using data from the borders,
    - factorization of the last block,
2. forward reduction,
    - reduction of the independent blocks,
    - updates using data from the borders,
    - reduction of the last block,
3. backward substitution,
    - substitution of the last block,
    - broadcast of data to the processors,
    - substitution of the independent blocks.

Each step involves highly parallel operations on data in the independent blocks, operations on data in the last block, and operations that couple the data in the independent blocks to data in the last block. This step involves data communications in algorithms developed

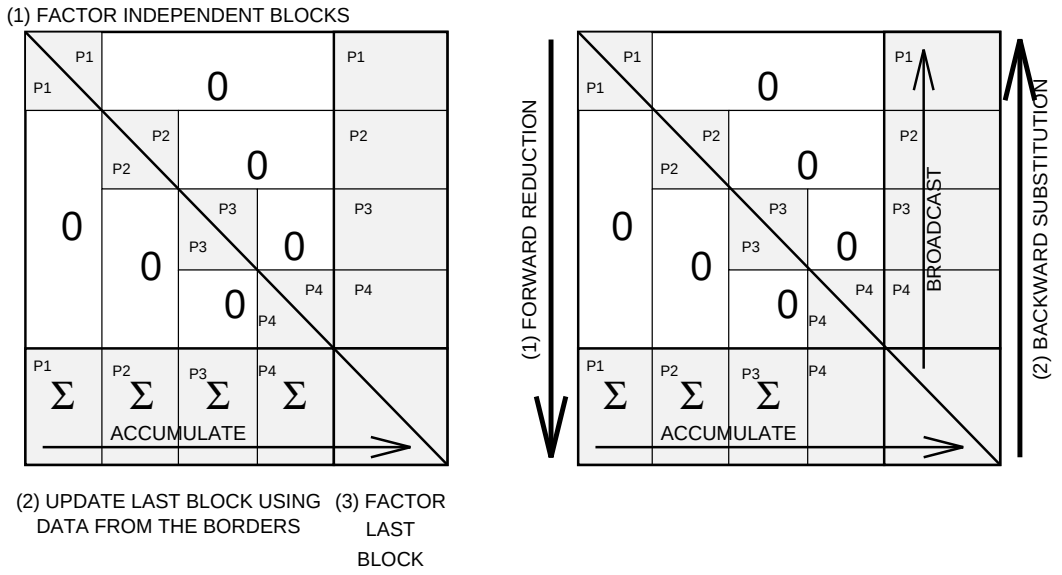

Figure 1: Block Bordered Diagonal Form Sparse Matrix Solution Steps

for distributed-memory multi-processor architectures. Figure 1 illustrates both the factorization steps and the reduction/substitution steps for a block-diagonal-bordered sparse matrix.

LU factorization of block-diagonal-bordered sparse matrices has significant advantages over general sparse matrix solvers. First, for all but the last block, every processor is assigned all the data required for calculations performed by that processor. Second, all calculations in the independent blocks and borders can be performed in parallel, without requiring communications to fetch data to perform updates. The LU factorization of the diagonal blocks can be performed in parallel without requiring interprocessor communications, and the numerical updates of the last block utilizing data in the borders, can also be performed in parallel. For this step, communications are only required to send partial sums of updates to the appropriate processors that possess data values in the last block of the matrix. Interprocessor communications are significantly reduced and are now also uniform and structured. Finally, the last block is factored in the most efficient manner depending on the density of this sub-matrix. The factorization steps are illustrated in part (a) of figure 1.

After the numerical factorization of the block-diagonal-bordered matrix, most of the calculations in the remaining forward reduction and backward substitution phases can also be performed in parallel without requiring communications. The reduction/substitution steps are illustrated in part (b) of figure 1. The forward reduction stage can be performed in

parallel until the last block is reduced. Communications are required to accumulate partial sums of products as the border equations are reduced. Likewise, most of the calculations in the backward substitution phase can be calculated in a highly parallel manner after the substitution is performed for the last block, and the results are broadcast to all processors.

Block-diagonal-bordered matrix forms offer significant advantages when solving sparse linear systems of equations. In a manner dissimilar to sparse linear solvers described in [7, 8, 9, 12, 23, 24, 25, 26, 27, 31], the task assignments for parallel numerical factorization of a block-diagonal-bordered matrix depend only on the assignment of independent blocks to processors and the processor assignments of data in the last diagonal block. However, additional computational work must be performed in the initial stages that determine which independent blocks are assigned to particular processors.

Block-diagonal-bordered sparse matrix algorithms require modifications to the normal preprocessing phase. 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 propose modifying 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.

To transform a sparse matrix into block-diagonal-bordered form requires relatively sophisticated ordering techniques, and also requires that load balancing be performed based on the number of calculations in each independent block. Due to the poor correlation of the number of rows or columns in a sparse matrix independent block with the workload, the actual number of calculations in each independent block must be determined in a *pseudo factorization* step during the preprocessing phase. *Pseudo factorization* also determines the location of all fillin so that efficient static data structures can be utilized when pivoting for numerical stability is not required. The three-step preprocessing phase required for efficient factorization of this matrix form will be more computationally intensive than the simple algorithms required to prepare a matrix for numerical factorization presented in the recent literature. The additional processing requirements for these steps limit the applicability of this technique to repetitive solutions of static network structures, where the additional effort can be amortized over multiple solutions of similar linear systems.

1.2 Overview

In this paper, we describe both sequential and parallel variants of the LU factorization algorithm used to solve the system of linear equations $Ax = b$, where A is a sparse matrix in block-diagonal-bordered form. We describe methods to transform a general sparse matrix to block-diagonal-bordered form and we also describe an implementation of the parallel block-diagonal-bordered sparse matrix solver on the Thinking Machines CM-5 using explicit message passing. In section 2 of this paper, we describe in detail the mathematics of sparse LU factorization including the precedence relationships in the algorithms, the two step process presented in recent literature to preprocess a sparse matrix before performing sparse Choleski factorization, and a version of Crout's LU factorization algorithm for sparse matrix factorization. This section includes a general discussion of forward reduction and backward substitution, and also includes a survey of the sparse matrix literature in order to place this work in context with other research.

In section 3, we propose a new three-step preprocessing phase to replace the present two-step preprocessing phase. We describe the specialized *ordering*, *pseudo factorization*, and *load balancing* steps required for efficient parallel algorithm implementations on distributed-memory multi-processors. This preprocessing step is the key to the efficient use of block-diagonal-bordered sparse matrix algorithms for highly irregular sparse matrices. In section 4, we describe the options available for the LU factorization of block-diagonal-bordered matrices that are dependent on the precedence relationships inherent in the algorithm. In section 5, we describe both a sequential block-diagonal-bordered sparse matrix algorithm and a parallel version implemented on the CM-5. In section 6, we present performance results when testing the software implementations on sample matrices. These data include measured speedup and efficiency as a function of the order of the matrix for both the LU factorization step and the forward reduction/backward substitution steps. Lastly, in section 7, we present our conclusions and briefly describe future research.

2 Background

Consider the direct solution of the linear system

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

where A is an $N \times N$ sparse matrix. For this research, it has been assumed that this matrix is neither symmetric nor position symmetric, however, the algorithms can be extended to Choleski factorization of symmetric positive definite matrices with minimal modifications to the mathematics or the software implementation described in section 5. Additional discussions on the state of the literature for Choleski factorization are presented below.

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:

$$Ax = LUx = b, \tag{3}$$

A lower triangular matrix, L, has all zeros above the diagonal and an upper triangular matrix, U, has all zeros below the diagonal. Triangular linear systems can be readily solved numerically by solving for the first value in the triangular linear system and substituting that value into subsequent equations. This procedure is repeated for all equations in the linear system.

2.1 LU Factorization

LU factorization is a variant of Gaussian elimination, and has numerous variants that depend on the order of calculations in addition to other implementation factors. There any numerous algorithms for LU factorization of dense matrices, that have three nested **for** loops around the statement:

$$a_{i,j} = a_{i,j} - \frac{(a_{i,k} \times a_{k,j})}{a_{k,k}}. \tag{4}$$

In this statement, the indices run:

- k - along the diagonal,
- i - down the rows,
- j - across the columns.

Sparse matrix LU factorization can mirror any dense LU 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 LU factorization of a matrix by ordering the matrix before factorization [5]. Fillin is discussed in greater detail below.

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 in equation 4 can be performed in parallel. Due to precedence relationships in the algorithm, this is generally the inner most **for** loop. Sparse factorization algorithms have inadequate calculations using the inner most index for efficient parallelism; however, sparse matrices have additional parallelism as a result of the

nature of the data and the precedence rules governing the order of calculations. Instead of just parallelizing the inner most **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.

2.2 Precedence in LU Factorization

Parallel implementations of direct linear solvers must cope with the precedence relationships that exist in the order of calculations. Before calculations can be completed on a row or column, non-zero values from previous rows and columns must be available and all calculations in those rows and columns must be completed. Precedence relationships in the calculations cause frequent synchronization in parallel algorithms, which reduce the granularity in the amount of available calculations making it more difficult to uniformly distribute processing load.

Varying the order of the **for** loops in a dense LU factorization causes different algorithms: some LU factorization algorithms offer more available parallelism than others. One class of LU factorization algorithms is referred to as fan-in algorithms, because data from previous rows or columns are sent inward to the column and row being modified. This algorithm type is illustrated in figure 2, in addition to the fan-out factorization technique. For fan-out factorization, data from one row or column can be used in the modification step for all subsequent rows and columns. However, a row or column is not completely factored until all previous rows and columns have been modified. In order to develop an efficient parallel block-diagonal-bordered LU factorization algorithm, a hybrid combination of both fan-in and fan-out precedence relations have been included in the same algorithm.

For dense matrix fan-in algorithms, only a single column or row can be modified at any time; however, multiple rows or columns can be modified concurrently in sparse matrices due to the fact that multiple rows and columns may be independent of other rows and columns. For a fan-in algorithm, the block-diagonal-bordered sparse matrix form presented in figure 1 clearly illustrates column and row independence for the independent diagonal blocks. There simply is no data in previous rows/columns to be included in the calculations. This phenomenon will be described in greater detail in section 3.

The goals of this research are to develop highly efficient, scalable parallel sparse LU factorization algorithms. To accomplish this goal, we propose redefining the matrix ordering phase to more efficiently exploit mutually independent calculations while significantly reducing concerns with precedence throughout as many of the calculations in a parallel implementation as possible. In this work, we plan to minimize the effects of precedence in block-diagonal-bordered sparse matrix algorithms by localizing calculations in independent

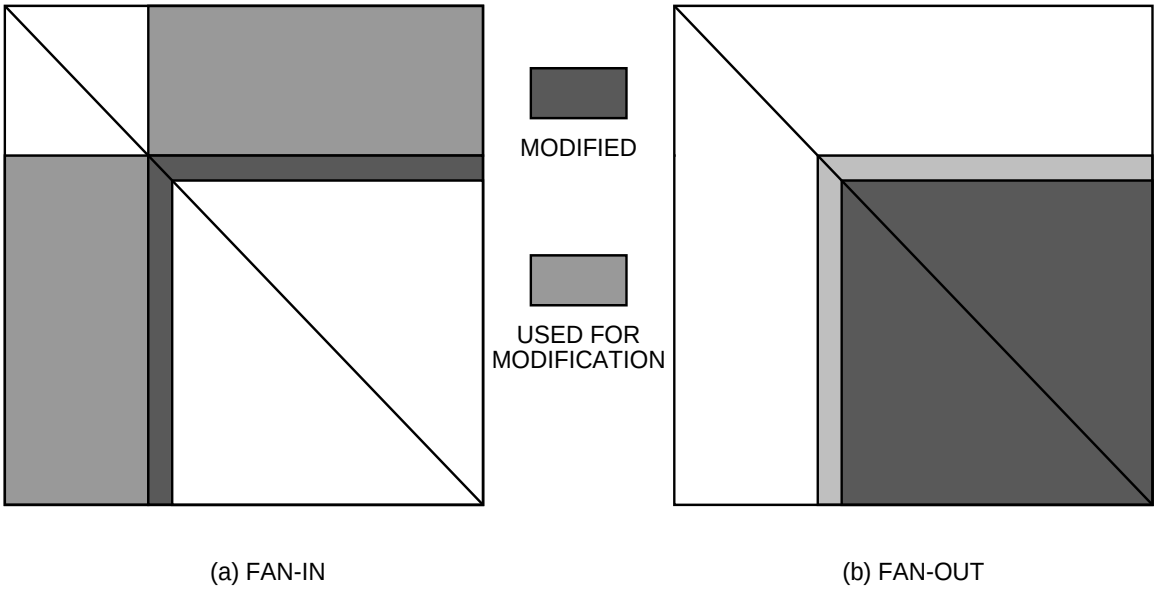

Figure 2: Fan-in and Fan-out LU Factorization Algorithms for Dense Matrices

sub-matrices that are spread throughout a multi-processor.

2.3 Crout's LU Factorization Algorithm

This research is based upon a factorization algorithm commonly attributed to Crout [5, 13]. We modify it to yield the general sequential sparse factorization algorithm presented in figure 3.

Element level dependencies for each of the two steps of this algorithm are illustrated in figure 4 for a general sparse matrix. To update a non-zero value $A_{i,k}$ in the lower triangular portion of the matrix (L), non-zero column elements $A_{j,k}$ in the upper triangular portion of the matrix are multiplied with corresponding non-zero row elements $A_{i,j}$ in the lower triangular portion of the matrix. The second update step modifies values $A_{k,j}$ in the upper triangular portion of the matrix (U) by multiplying non-zero lower triangular row elements $A_{k,i}$ by corresponding upper triangular column elements $A_{i,j}$. The LU matrix is over-specified so the values on the diagonal of either the upper or lower triangular matrix portions can be set equal to one [5, 13]. In Crout's implementation, the diagonal values of L are stored in the original diagonal matrix positions, while the diagonal values of U are all equal to 1 but never explicitly stored.

for $k = 1$ **to** N /* for all elements along the diagonal */
    /* lower triangular */
    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
    /* upper triangular */
    for each $j \in [k+1, N]$
        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
        $A_{k,j} \leftarrow (A_{k,j} / A_{k,k})$
    endfor
endfor

Figure 3: Sparse Crout LU Factorization

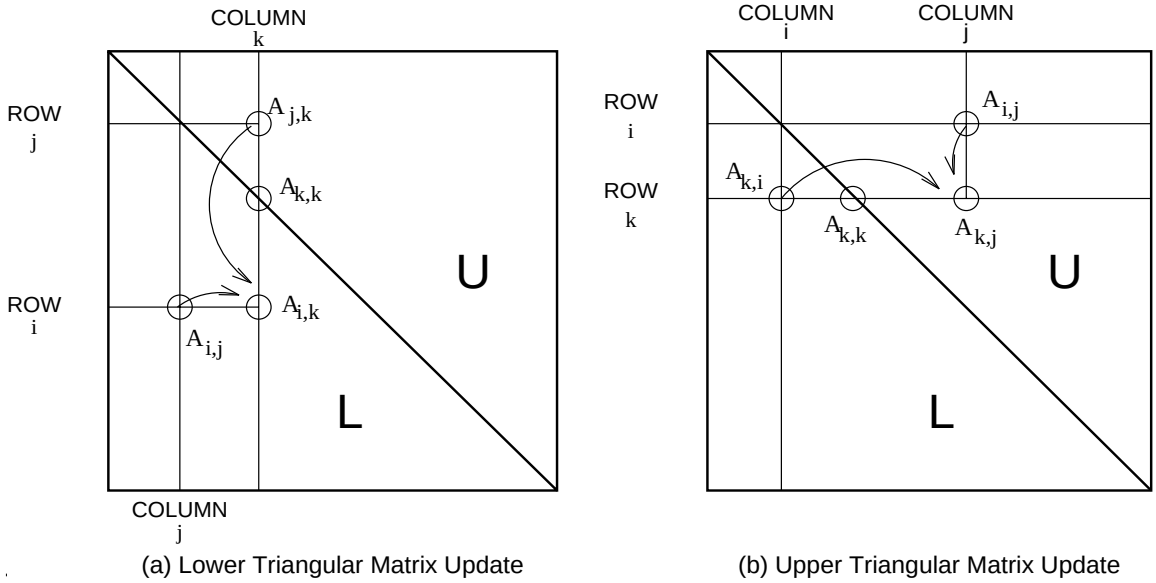

Figure 4: Element Level Dependencies for General Sparse LU Factorization

2.4 Fillin

It is important to note that in this implementation, L and U can reuse the memory from the initial sparse matrix A, as long as provisions for storage of fillin values are provided. This algorithm assumes no pivoting is required to ensure numerical stability. Fillin results when either $A_{i,j}$ and $A_{j,k}$ or $A_{k,i}$ and $A_{i,j}$ are both non-zero and $A_{i,k}$ or $A_{k,j}$ are initially equal to zero. The amount of fillin in the factorization of a sparse matrix is highly dependent on the order in which calculations are performed. There are many ways to *order* a sparse matrix to reduce the amount of fillin [5, 9, 12]. Two noteworthy ordering techniques are *minimum degree ordering* and *nested dissection*. The foundation for minimum degree ordering is based in graph theory, which shows that fillin can be minimized by first eliminating rows and columns that have the fewest number of non-zero values, thus minimizing early fillin, which will hopefully minimize fillin later in the factorization. Minimum degree ordering is a greedy algorithm that is far from optimal because no good tie breaking rules exist and numerous rows and columns have equal numbers of elements. Nested dissection recursively breaks a matrix into a block-diagonal-bordered form where fillin is limited by the resulting structure. This technique is marginally effective for irregular networks or sparsity patterns, although it is considered the theoretical benchmark to which the quality of orderings are compared [12]. We have developed other ordering techniques for highly irregular matrices [14] that

- partition the matrix into mutually independent blocks,
- balance the load for multi-processors.

This research is introduced later in this paper.

2.5 Symbolic Factorization and Static Data Structures

Sparse matrices have only a small percentage of non-zero values, consequently, they are stored in an implicit form, where only non-zero values and corresponding row and column location identifiers are stored. The most efficient sparse matrix algorithms, that do not require pivoting to maintain numerical stability, use implicit static data structures to store the matrix, in order to reduce the overhead of adding data to dynamic data structures. Static data structures require that all locations of all non-zero values be known when allocating memory for the data structures. This includes the location of all fillin. Example static data structures are the CSC or *compressed storage of columns* format [10, 24], or C language data structures that store the non-zero values, row and column location identifiers for a specific value, (i, j), and possibly additional pointers to subsequent values in columns or rows.

A naive approach to determine fillin in a sparse matrix when performing LU factorization or Choleski factorization is to simply symbolically duplicate the operations of the factorization [12]. Such a naive algorithm would require the same time complexity as numerical factorization, $O(N^m)$, where $1.2 < m < 1.5$ for many sparse matrix applications [18]. However, it is possible to reduce the time complexity of symbolic factorization for set-based algorithms to less than $O(\eta(L))$, where $\eta(L)$ represents the number of non-zero values in the factored matrix. References [9, 12] each contain excellent discussions of symbolic factorization for Choleski factorization. For asymmetric matrices, if the data is diagonally dominant and there are no requirements to use pivoting or partial-pivoting to maintain numerical stability of the calculations, then an algorithm based on symmetric symbolic factorization could be implemented to determine the location of all fillin in both the upper and lower triangular portions of the matrix. If pivoting is required to maintain numerical stability when factoring a sparse matrix, non-static data structures will be required because pivoting dynamically modifies the sparsity structure of a matrix.

2.6 Forward Reduction/Backward Substitution

After a matrix A is factored into the two matrices L and U in

$$Ax = LUx = b, \tag{5}$$

the solution of the vector x for a given b vector follows a two step process. First, there is the forward reduction step where

$$Ly = b \tag{6}$$

is solved for a temporary working vector y, and then the solution for x is obtained by the backward substitution step

$$Ux = y. \tag{7}$$

LU factored matrices can be reused many times when solving for multiple right-hand side $b_{\mathbf{m}}$ vectors in the linear equation $Ax_{\mathbf{m}} = b_{\mathbf{m}}$. It is common practice in iterative solutions of non-linear equations using Newton's method to evaluate the linear system in the Jacobian only every k ($k \geq 1$) iterations, because of the computational complexity to solve the linear system. Consequently, efficient parallel implementations of forward reduction and backward substitution are required even though the complexity of these steps are significantly less than the workload to factor the matrix.

The sequential version of forward reduction and backward substitution are traditionally viewed as straight forward implementations of loops to directly solve for y_i and x_i in the equations

$$y_i = \frac{(b_i - (y_1 \times L_{i,1} + \cdots + y_{(i-1)} \times L_{i,(i-1)}))}{L_{i,i}}, \tag{8}$$

and

$$x_i = (y_i - (x_N \times U_{i,N} + \cdots + x_{(i+1)} \times U_{i,(i+1)})). \tag{9}$$

In these equations, the values $L_{i,j}$ and $U_{i,j}$ are the lower and upper triangular matrices respectively, that are generated by the LU factorization step. For L and U matrices generated by Crout's factorization algorithm, equation 9 does not require a final division operation because the all values $U_{i,i}$ are equal to 1. However, like factorization, there are multiple distinct ways to manipulate the data in the nested **for** loops. Instead of performing reduction and substitution by rows using equation 8 and equation 9, the values y_j and x_j can be calculated using b_j and y_j respectively in the formulas

$$y_j = (b_j / L_{i,i}), \tag{10}$$

and

$$x_j = y_j. \tag{11}$$

Then the entire j^{th} column in forward reduction or backward substitution respectively can be updated using the formulas

$$b_i = (b_i - (y_j \times L_{i,j})), \forall i > j, \tag{12}$$

and

$$y_i = (y_i - (x_j \times U_{i,j})), \forall i < j. \tag{13}$$

After the entire j^{th} column has been updated, the values of $y_{(j+1)}$ and $x_{(j+1)}$ can be calculated and this procedure iteratively repeated for all columns.

Figure 5 presents a comparison of row and sub-matrix reduction/substitution. This figure illustrates the loop order for the two algorithm types by the large arrows, while smaller arrows depict the calculation order. For row reduction/substitution, the calculations are constrained to within rows, after which a value of y_i or x_i is calculated. However, for sub-matrix reduction/substitution, the value of y_j or x_j is calculated first, then all values remaining in the sub-matrix are used to update b_i and y_i values.

2.7 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 [4, 29, 30], 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 [29, 30], nevertheless, performance is deterministic for:

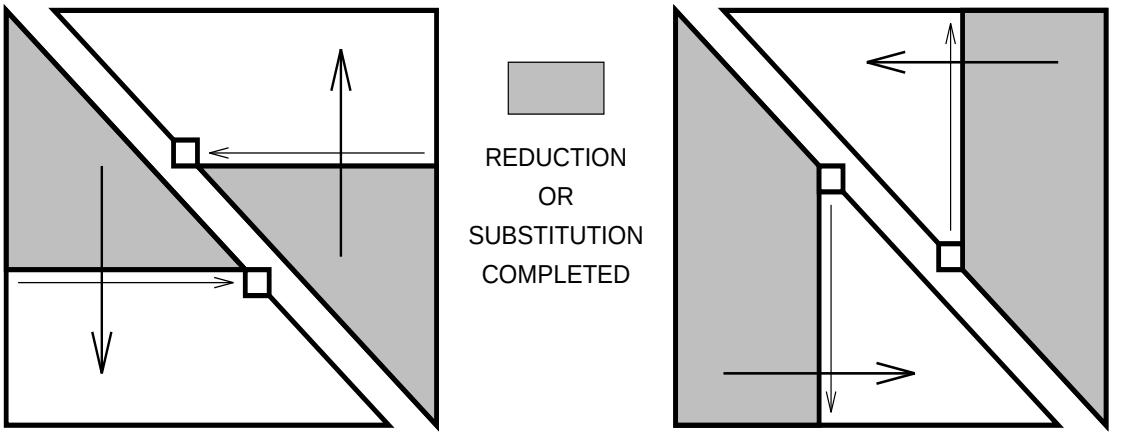

Figure 5: Alternative Forward Reduction/Backward Substitution Schemes

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

Direct sparse matrix solvers, on the other hand, have computational complexity significantly less than $O(N^2)$, and some papers refer to the complexity of particular applications ranging from $O(N^{1.2})$ to $O(N^{1.5})$ [18]. With significantly less calculations than dense direct solvers, and lacking uniform, organized communications patterns, direct parallel sparse matrix solvers often require detailed knowledge of the application to permit efficient implementations. The greater complexity and irregularity of sparse matrix computations make this field an area with active research as parallel sparse matrix solvers are developed and optimized for particular applications.

The basics for general sequential direct sparse matrix solvers, including numerical stability and ordering techniques to limit fillin are presented in [5]. There are many papers that examine sparse matrix algorithms for multi-processors and vector computers, however, there are limited numbers of papers describing research into parallel non-symmetric direct sparse matrix solvers. A concurrent non-symmetric sparse solver was developed as part of a petrochemical processing simulation [21]. Frontal or multi-frontal techniques offer some promise for algorithms designed for vector and vector computer-based multi-processors [4]. General sparse matrix solvers store data in compressed implicit data structures. In order

to use vector processors in an effective manner, the hardware must have indirect addressing capabilities or the data must be processed with vector *scatter and gather* operations.

Meanwhile, 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 [7, 8, 9, 12]. 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 [24, 25, 26, 27]. 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 [15]. 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 [7, 8, 9, 12, 24, 25, 26, 27] have kept with a general four step paradigm for parallel sparse Choleski factorization:

- order the matrix to minimize fillin,
- symbolic factorization to identify fillin and set up static data structures,
- numerical factorization,
- forward reduction/backward substitution.

In this paper, we break from this four step process and introduce a process that is applicable to highly irregular sparse matrices. We propose a new three-step preprocessing phase to replace the two-step preprocessing phase of *ordering* and *symbolic factorization*. While the original development of our method was directed toward applications from the electrical power systems community, it has wider applicability to many irregular sparse matrix applications. Because of the irregular nature of the sparse matrices, explicit load balancing of the workload among the processors is required.

The direct matrix technique presented in this paper may even compete with other parallel sparse matrix solvers for regular problems, because task assignments for numerical factorization of a block-diagonal-bordered matrix depend only on the assignment of independent blocks to processors and the processor assignments of data in the last diagonal block. Techniques based on block-diagonal-bordered sparse matrices have absolutely no task organization graph, because the entire problem of task assignments are performed in an initial stage that performs load-balancing, and all data is available on the processor for

calculations until the solution of the last diagonal block in the matrix. Communications are also more uniform and structured.

3 A New Three-step Preprocess Phase

The primary goal of this paper is to describe the significant benefits of using block-diagonal-bordered sparse matrices when performing sparse LU factorization in parallel on distributed-memory multi-processors. 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* into block-diagonal-bordered form,
- *pseudo factor* to identify both fillin and the number of calculations for independent blocks, and
- *load balance* to uniformly distribute the calculations among processors.

The first step determines the block-diagonal-bordered form; 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's 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. Future research may examine the applicability of producing an automated tool that has the capabilities of finding both an optimal matrix ordering and an optimal version of the multi-processor algorithm that is driven by the characteristics of the sparse matrix. Different versions of the block-diagonal-bordered LU factorization algorithm are possible depending on the specific architecture and characteristics of the matrix. These algorithms will be discussed in more detail in section 4.

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. Determining the actual workload may require a detailed simulation of all processing and interprocessor data communications.

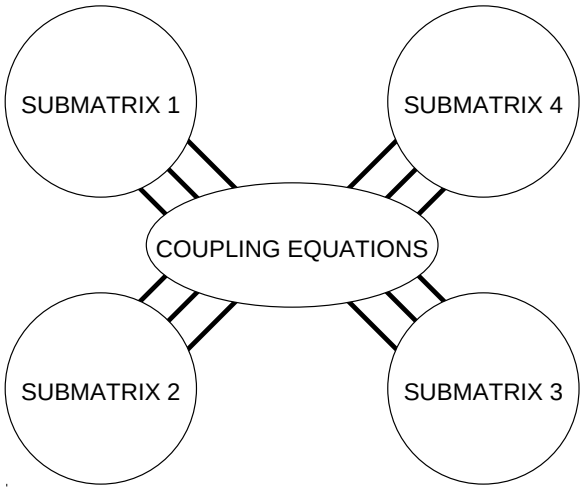

Figure 6: Sample Graph with Four Independent Subgraphs/Sub-Matrices

3.1 Ordering

The *ordering* portion in the preprocessing phase must identify subsections of the matrix that are mutually independent. For symmetric matrices, there is a graph-theoretical interpretation for independent sub-matrices. Independent sub-matrices simply have no shared edges in their undirected graph. A simple example to illustrate the concept of independent subgraphs is illustrated in figure 6. This figure has four independent portions of the graph connected by nodes that form the coupling equations. No subgraph element has edges to any portion of the graph other than within the local subgraph or extending to the coupling equations.

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 technique can produce balanced block-diagonal-bordered matrices on some regular matrices [12]. *Recursive spectral bisection* can be used to partition irregular matrices [3, 17, 20], and subsequently, the coupling equations can be extracted. Unfortunately, this technique, as well as nested dissection, relies on dividing the matrix into m equal sized partitions, without considering the coupling equations or considering the number of calculations in each independent block. A third method to order a sparse matrix into block-diagonal-bordered form is referred to as *node tearing* [5, 19], which is a specialized form of diakoptics [11]. 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. Load balancing techniques must be used after the node tearing matrix ordering step to uniformly distribute

the processing load onto a multi-processor. It is important to note that independent blocks can be assigned to any processor without requirements for interprocessor communications to factor the sub-block.

3.2 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 [9, 12, 24].

To accomplish the two-fold requirement for both identifying the location of fillin and determining the amount of calculations in each independent block, we propose that a *pseudo factorization* step be included in 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.

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. The workload during the numerical factorization step may differ substantially from the number of equations assigned to processors. Efficient 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.

3.3 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 may emphasize uniformly distributing the processing workload in any one of the steps, or it may attempt to optimize the distribution of processing workload for a combination of the three phases. A preliminary investigation 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 [14]. 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, we are assuming that communications can be made regular for all processors — a research goal — so the independent nature of calculations in the independent blocks will 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 independent blocks can be used for *load balancing*.

A *pigeon-hole* 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 many more independent blocks than processors. In this instance, the workloads in multiple small blocks can sum to equal the workload in a single larger block.

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 used extensively in irregular sparse matrix problems, 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.

4 LU Factorization of Block Bordered Diagonal Matrices

LU factorization of block-diagonal-bordered sparse matrices has significant advantages over general sparse matrix solvers. For all but the last block, every processor has all data required for calculations performed by that processor, so all calculations for the independent blocks and borders can be performed in parallel. Moreover, LU factorization of the blocks can be performed in a highly parallel manner that doesn't require interprocessor communications. For this step, communications are only required to send the partial updates to the appropriate processors that possess data values in the last block of the matrix. Finally, the last block is factored in the most efficient manner depending on the density of that sub-matrix. Research with ordering matrices representing real power system networks has shown that it is possible to reduce the number of coupling equations and, consequently, the size of the last block to a point where this sub-matrix becomes nearly dense after fillin. Moreover, efficient dense matrix solvers have been described in the literature [4, 29, 30].

After the numerical factorization of the block-diagonal-bordered matrix, most of the calculations in the remaining forward reduction and backward substitution phases can also be performed in a highly parallel manner. The forward reduction stage can be performed in parallel until the last block is reduced. Communications are required only to accumulate partial sums of products from the solved variables as the border equations are reduced. Likewise, most of the calculations in the backward substitution phase can be calculated in parallel without the need for additional communications after the substitution is performed for the last block, and the results are broadcast to all processors.

This section of the paper describes the benefits of LU factorization of block-diagonal-bordered sparse matrices and also describes variants of Crout's algorithm that are applicable to parallel block-diagonal-bordered form LU factorization. Variants of the algorithm are possible depending on implementation details dictated by the nature of the sparse matrix and the multi-processor architecture.

4.1 Numerical Factorization

The general nature of block-diagonal-bordered LU factorization algorithms described in this paper follows Crout's algorithm presented in section 2.3. Crout's algorithm is a fan-in algorithm that updates a row/column from previous values. During the preprocessing phase, the matrix is ordered into block-diagonal-bordered form, the location of all fillin is determined to permit static data structures, and load balancing is performed on the independent blocks in order to distribute the workload uniformly to all processors when

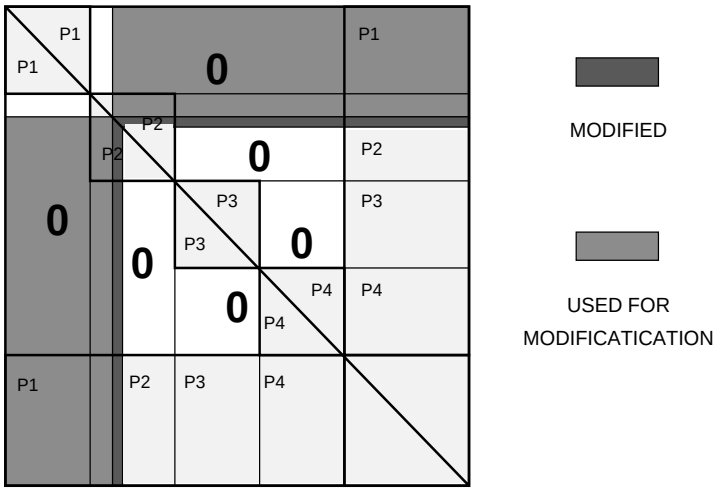

Figure 7: Precedence Relationship for Independent Blocks

factoring the independent blocks. The data is distributed to processors in such a manner that the data for an independent block and the corresponding portions of the borders are a assigned to the same processor. This is illustrated in figure 1(a).

The first step of the parallel algorithm performs LU factorization of the independent blocks is straightforward. Precedence permits the parallel factorization of each independent block and its associated section of border. Figure 7 depicts the precedence in the calculations for a block in an idealized block-diagonal-bordered sparse matrix. The data in each block and border portion may also be sparse. Figure 8 illustrates the element level dependencies for non-zero elements in both the diagonal block and the associated portion of the border. Note that normal fan-in precedence relationships illustrated in figure 7 at first glance suggest that data may be required from previous border sections possibly stored on other processors; however, due to the block-diagonal-bordered form of the matrix, any value $A_{j,k}$ that would be used for an update in a previous block is equal to zero by definition. Consequently, no interprocessor communications is required in this parallel LU factorization step, because all data is assigned to the processor that uses it to perform updates.

The second step in the numerical factorization of a block-diagonal-bordered matrix is to update the last block using data in the borders. For a fan-in algorithm, figure 9 illustrates the required calculations for the factorization of the first column and row of the last block. However, precedence permits the updates from the borders to occur in any order as long as they occur before the column and row updates are completed. For parallel algorithms, there are definite advantages in calculating all updates from the borders at the same time. For systems where the communications latency when generating a message

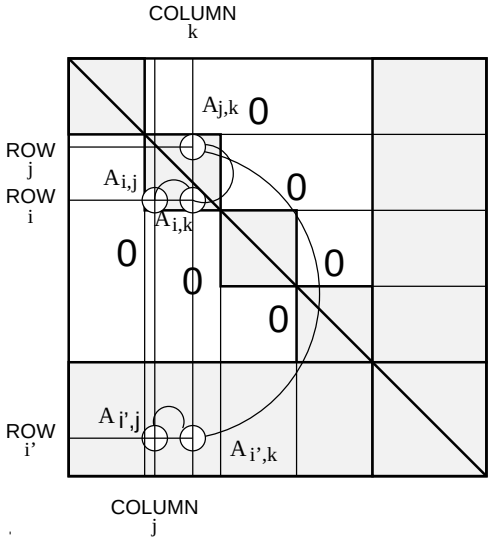

Figure 8: Element Level Dependencies for Independent Matrix Blocks

dominates the communications costs, limited numbers of long messages have significant advantages over numerous short messages. Performing all updates at the same time in order to generate long messages can save significant time that would otherwise be part of the communication overhead in the parallel algorithm. When implemented on processors with low latency communications support for short messages, it may be advantageous to perform these calculations in a manner that more closely resembles a general fan-in algorithm. It may even be advantageous to calculate the partial sums and store the values until the processors holding a data value that is part of the last block initiates the retrieval of a value to complete the factorization of a column or row.

Figure 10 illustrates the element level data dependencies for updates to elements in the last block by data in the borders. To update value $A_{i,k}$ in the last block, the value $A_{j,k}$ in the upper border and the value $A_{i,j}$ in the lower border must be multiplied together. All other values in the same column as $A_{j,k}$ (within the independent block) and the same row as $A_{i,k}$ (within the independent block) can be multiplied together and summed to minimize communications further. Partial sums are calculated for all independent blocks and corresponding borders, so partial sums now become a function of the number of processors, rather than the number of independent blocks. The partial sums can be further reduced in parallel if the partial sums are generally non-zero. Parallel reduction of partial sums is illustrated in figure 11. If the partial sums are sparse relative to the various processors, then the partial sums should be sent directly to the corresponding processor to minimize communications. Eventually the partial sums are subtracted from the elements in the last

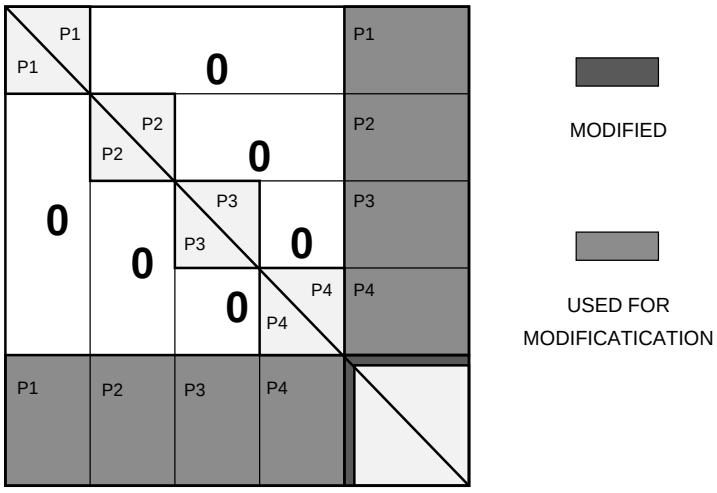

Figure 9: Fan-in Data Pattern for the Factorization of the Last Matrix Block

block. Similar calculations are required for the upper triangular portion of the last block and the borders.

The third step in the numerical factorization of a block-diagonal-bordered matrix is the factorization of the last block in the matrix. Due to the fillin in the matrix, it is common for the lower right portion of the last block to be nearly dense. By not explicitly storing zero values, implicit sparse matrix storage overcomes the overhead of storing row and column location indicators and pointers to the next value in a row or column. However, when a portion of a matrix becomes dense due to fillin, that portion of the matrix should be stored and factored in dense form. The most desirable situation would be to have a small last block that is significantly dense to permit explicit storage of the matrix so that this sub-matrix can be updated efficiently as a dense matrix. In this situation, the best parallel dense matrix factorization algorithm for the particular target architecture could be used to factor the last block. Otherwise, the algorithm would be more complicated and include a partitioning of the last block sub-matrix into a dense block in the lower right hand corner with the remainder of this sub-matrix being sparse. The size of the last block is dependent on the structure in the data and the ordering algorithm in the preprocessing phase.

For sparse matrices that have nearly dense last blocks, one method of enhancing LU factorization algorithms on either sequential or parallel architectures is to control the size of the last block. It is better to have a small, very dense last block rather than a larger block that is less dense, but still sufficiently dense as to require explicit storage of the matrix for efficient factorization. It can be shown that the number of multiplication and subtraction operations to factor either the lower triangular or the upper triangular portions of the last

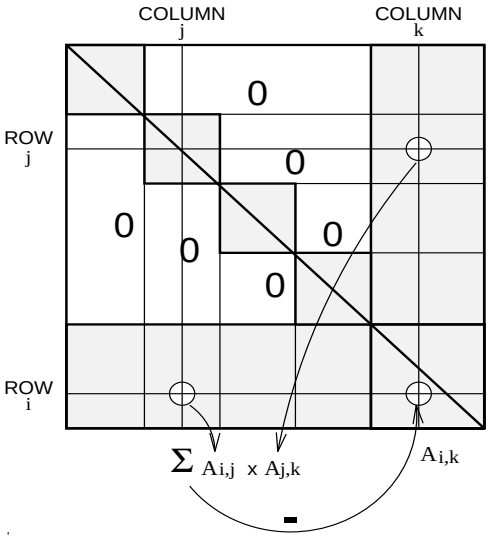

Figure 10: Element Level Dependencies for Independent Matrix Borders

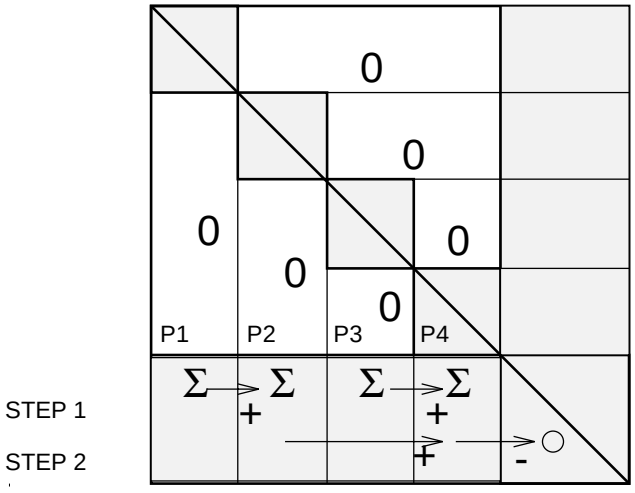

Figure 11: Parallel Reduction of Partial Sums

block, assuming that the data will be stored explicitly as a dense matrix, are

$$N_{ops} = \frac{(N_{lb}^3 - N_{lb})}{6} \tag{14}$$

Where N_{lb} is the size of the last block. Thus, the number of calculations are $O(N_{lb}^3)$, and even small increases in the size of the last block cause large increases in the number of calculations. A 25% increase in the size of the last block doubles the number of calculations, and a 60% increase in the number of equations in the last block increases the number of calculations by a factor of four. Consequently, controlling the size of the last block during the ordering step of the preprocessing phase can have significant ramifications on the run time of block-diagonal-bordered sparse matrix factorization algorithms.

4.2 Forward Reduction/Backward Substitution

Block-diagonal-bordered matrices have the potential for efficient reduction and substitution steps because of limited, regular communications, as seen in figure 1(b). The solution of values in the y vector corresponding to the independent diagonal blocks can be solved in a highly parallel manner. For the forward reduction step, the only communications required are to send the partial sums of products from rows in the borders to those processors that hold the data for the last block and these partial sums can be reduced in parallel in the same manner as the reduction of partial sums in the factorization step, depicted in figure 11. In essence, the partial sums of products of $(y_j * A_{i,j})$, $\forall j$ such that $A_{i,j} \neq 0$ are used to modify the value of b_i. After all updates have been made using the values of y_j in the independent blocks, the forward reduction of the last block can proceed using the best available techniques for the nature of this sub-matrix.

For the backward substitution step that calculates the values in the vector x, the values must be broadcast to all processors. If the data in the last block is distributed to multiple processors, each value must be broadcast to all other processors immediately after they are calculated. In this instance, special techniques that support lightweight communications processes, such as *active messages* [22] on the Thinking Machines CM-5 can be used to minimize communications latency. Regardless of the communications capabilities of the distributed-memory multi-processor, as soon as values in the x vector from the last block have been broadcast to all processors, calculations to update the y vector can be made with $U_{i,j}$ values in the borders because of the sub-matrix precedence relationship in backward substitution. After all values in the x vector from the last block have been solved, the remaining calculations to solve for values of x in the independent blocks can be performed in parallel without requiring additional interprocessor communications, because all data for rows are allocated to a processor and because blocks are independent with no requirement to send x vector values to other processors.

There are additional algorithm implementation features that can improve the efficiency of forward reduction and backward substitution. By including information in the data structures on the row and column locations of a value in the implicit storage of the matrix, indexing overhead can be minimized by permitting modifications in both the forward reduction and backward substitution steps that do not require the calculation of the row or column locations of a value.

5 Sparse Matrix Solver Implementations

Implementations of a block-diagonal-bordered sparse matrix solver have been developed in the C programming language for both sequential computers and for the Thinking Machines CM-5 multi-computer using message passing and a *host-node* paradigm. Performance data has been gathered for each of the two software implementations and performance comparisons are presented in the next section. Both the sequential and parallel block-diagonal-bordered sparse matrix solvers use implicit hierarchical data structures based on vectors of C programming language structures. These data structures are optimal for either sequential or parallel implementations and provide good cache coherence, because non-zero data values and row and column location indicators are stored in adjacent physical memory locations. The parallel implementation presented in this section has been developed as an instrumented proof-of-concept to examine the efficiency of the data structures and the efficiency of the basic message passing when partial sums are sent to the last block to update values. The last block in this initial parallel implementation has been factored sequentially in the host processor. In order to minimize communications times in the second step of factorization when partial sums of updates from the borders are sent to the host processor, vectors of all non-zero partial sums from a processor are constructed. For this implementation, there has been no attempt at parallel reduction of the partial sums of updates in the borders.

Section 5.2 and section 5.3 includes pseudo-code outlines of the sequential and parallel algorithms respectively. Appendix B contains more detailed versions of these algorithms that include all **for** and **for each** loops.

5.1 The Hierarchical Data Structure

An implicit hierarchical data structure has been developed to efficiently store and retrieve data for a non-symmetric block-diagonal-bordered sparse matrix. This data structure is static, consequently, the locations of all fillin must be determined before memory is allocated for the data structures. In addition, due to the static nature of the data structure, explicit pointers to subsequent data locations can be used in order to reduce indexing overhead. Row

and column location indicators are explicitly stored as are pointers to subsequent values in columns and rows that are required when updating values in the matrix. This sparse matrix research has considered real-time applications, so the use of additional memory in the data structures is traded for reduced indexing overhead. Modern distributed memory multiprocessors can be purchased 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 or columns 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 12 graphically illustrates the hierarchical nature of the data structure, where the eight separate C structure elements presented in that figure are:

1. block identifier,
2. matrix diagonal element,
3. non-zero value in a lower triangular matrix diagonal block (arranged by rows),
4. non-zero value in a upper triangular matrix diagonal block (arranged by columns),
5. non-zero row in the lower border,
6. non-zero column in the upper border,
7. non-zero value in the lower border (arranged by rows),
8. non-zero value in the upper border (arranged by columns).

At the top of the hierarchical data structure is the information on the storage locations of independent blocks, the last block, and the lower and upper borders. The next layer in the data structure hierarchy are 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.

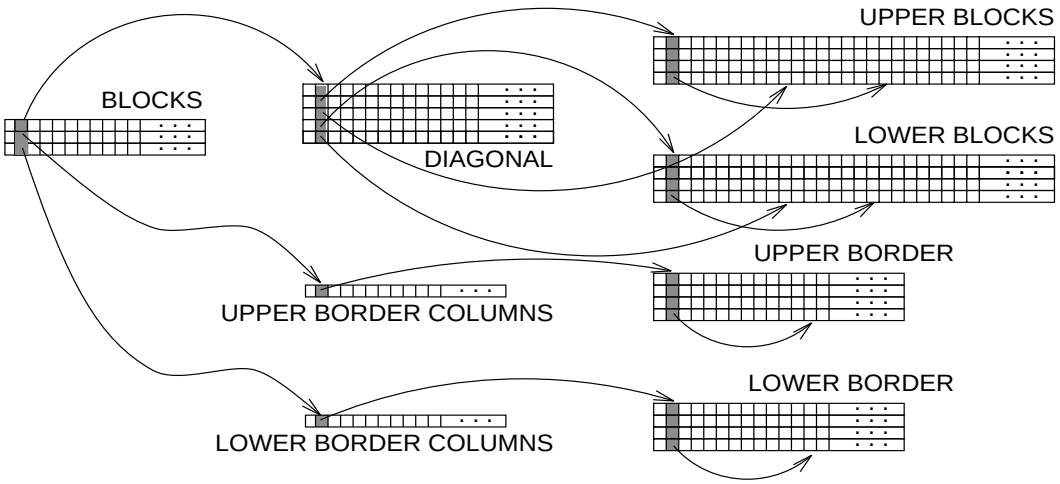

Figure 12: The Hierarchical Data Structure

Data in the strictly lower triangular portion of the matrix is stored by rows; likewise, data in the strictly upper triangular portion of the matrix is stored by columns. 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's algorithm requires 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 data pertaining to any diagonal element has pointers to the first non-zero element in the lower triangular row and upper triangular column. There are also pointers to the next non-zero value in the lower triangular column and the upper triangular row. This data structure trades memory utilization for speed by storing all row and column indicators with a non-zero value. In addition, the combination of adjacent storage of non-zero row and column values and the explicit storage of row and 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. A complete listing of the

elements in the data structure is presented in appendix A.

Conventional compressed data formats require less storage than this data structure; however, 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 support real-time applications. The compressed data format requires

$$S_c = (\lambda_{fp} + \lambda_{int}) \times \eta(A) + (\lambda_{int} \times n) \tag{15}$$

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

$$S_h = (\lambda_{fp} + (3 \times \lambda_{int})) \times \eta(A) + (\lambda_{int} \times n) + ((3 \times \lambda_{int}) \times N_{blocks}) + ((2 \times \lambda_{int}) \times N_{border}) \tag{16}$$

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 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, there is a significant decrease in indexing overhead, especially considering that to find a value in a row or column, the average 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.

Nevertheless, there is nothing in the nature of block-diagonal-bordered form matrices that limits the use of modified compressed data structures that simply use the hierarchical nature of this data structure to identify independent blocks with compressed data structures substituted throughout the remainder of the data structure. However, the data structure is closely tied to the implementation of the block-diagonal-bordered sparse matrix algorithms

and modifying the data structures would require changes in the code used to identify values to be used in updates of matrix values.

While the general Crout factorization algorithm permits partial pivoting [13], 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 [5]. 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 our applications will be diagonally dominate if not positive definite.

5.2 The Sequential Algorithm

The sequential algorithm we propose for LU factorization of block-diagonal-bordered sparse matrices follows the version of Crout's algorithm presented during the background discussion in section 2. The fan-in form of Crout's algorithm is followed for each independent block and the corresponding portion of the border. The last block is also factored in a fan-in manner, however, data in the borders must be used in the update of $A_{i,j}$ values in the last block. None of the independent blocks need be concerned with data from other blocks or corresponding border sections, so the algorithm to factor the last block differs from the algorithm for the independent blocks. In this implementation, the last block is also factored using sparse techniques. This factorization algorithm is presented in figure 13. Other implementations of this algorithm may consider the last block to be dense, or a combination of sparse and dense. Such a consideration would be dependent on the nature of the sparse matrix.

Our research into ordering power systems distribution network matrices into block-diagonal-bordered form has illustrated that there are substantial reductions in the number of fillin for this ordering technique when compared to conventional ordering techniques such as minimum degree ordering [14]. Consequently, the ordering techniques required for efficient parallel algorithms also benefit sequential algorithms.

The remaining steps in the sequential algorithm are forward reduction and backward substitution. The sequential version of these algorithms are straight forward implementations of the row reduction/substitution technique discussed in section 2.6. Pseudo-code versions of the algorithms for sequential forward reduction and sequential backward substi-

/* factor the independent blocks and corresponding borders */
for all independent blocks l
    for all elements k along the diagonal in block l
        Update the k^{th} *column in the* l^{th} *lower diagonal block*
        Update the k^{th} *column in the* l^{th} *lower border piece*
        Update the k^{th} *row in the* l^{th} *upper diagonal block*
        Update the k^{th} *row in the* l^{th} *upper border piece*
    endfor
endfor
/* factor the last block */
for all elements k **along the diagonal in the last block**
    Update the k^{th} *column using data from the last block*
    Update the k^{th} *column using data from the borders*
    Update the k^{th} *row using data from the last block*
    Update the k^{th} *row using data from the borders*
endfor

Figure 13: Sequential Sparse LU Factorization

```
/* reduce the independent blocks */
for all independent blocks l
    for all rows i in block l
        Update b_i using y_j and L_{i,j} values in the independent diagonal blocks
        y_i ← (b_i/L_{i,i})
    endfor
endfor
/* reduce the last block */
for all rows i in the border and last block
    for all independent blocks l
        Update b_i using y_j and L_{i,j} values in the borders
    endfor
    Update b_i using y_j and L_{i,j} values in the last block
    y_i ← (b_i/L_{i,i})
endfor
```

Figure 14: Sequential Sparse Forward Reduction

tution are presented in figures 14 and 15 respectively. The backward substitution step does not require a final division operation, because the all values $U_{i,i}$ are equal to 1. Detailed versions of the factorization, forward reduction, and backward substitution algorithms are presented in appendix B.

5.3 The Parallel Algorithm

Implementation of the parallel block-diagonal-bordered sparse matrix solver has been developed in the C programming language for the Thinking Machines CM-5 multi-processor using message passing and a host-node paradigm. The same implicit hierarchical data structure used in the sequential implementation is used in the parallel implementation, although the data structure is partitioned and the data are physically allocated to the distributed memory located with the various processors. This parallel implementation has been developed as an instrumented proof-of-concept to examine the efficiency of the data structures and the overhead of message passing when partial sums are sent to the last block to update values located there. This implementation is a simplified version of a block-diagonal-bordered sparse matrix solver, where the last block in this initial parallel implementation has been factored sequentially on the host processor. The actual code that implements the factoriza-

```
/* substitute in the last block */
for all rows i in the last block
    Update y_i using x_j and U_{i,j} values in the last block
    x_i ← y_i
endfor
/* substitute in the independent blocks and the border*/
for all independent blocks l
    for all rows i in block l
        Update y_i using x_j and U_{i,j} values in the border
        Update y_i using x_j and U_{i,j} values in the diagonal block
        x_i ← y_i
    endfor
endfor
```

Figure 15: Sequential Sparse Backward Substitution

tion algorithm presented in figure 16 contains extra node timing calls and communications to send this timing data back to the host processor for display. In spite of the extra communications overhead due to the instrumentation and the significant sequential portion of this proof-of-concept code, empirical performance data illustrates good speedup potential. Performance of this algorithm is discussed in the next section.

By limiting the factorization of the last block to a sequential process on the host processor, all partial sums of updates for values in the borders in the second phase of the factorization step can be sent with a single message to the host processor. A vector of structures identifying all non-zero elements in the last block is used to determine those partial sum values that need to be calculated. In order to minimize communications times in this factorization step, a single vector of all non-zero partial sums is constructed at each processor. For this implementation, there has been no attempt at parallel reduction of the partial sums of updates from the borders.

The remaining steps in the parallel algorithm are forward reduction and backward substitution. The parallel version of these algorithms combine row reduction/substitution and sub-matrix reduction/substitution. The combination of these techniques is utilized to minimize communications times, by generating long vectors. Later implementations of block-diagonal-bordered matrix algorithms will include the use of *active messages* for fast transfers of individual partial sums or for fast broadcast values of x_i to other processors.

Node Program

/* factor the independent blocks and corresponding borders */

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

  for all elements k along the diagonal in block l

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

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

    Update the k^{th} *row in the* l^{th} *upper diagonal block*

    Update the k^{th} *row in the* l^{th} *upper border piece*

  endfor

endfor

/* update the last block */

for all non-zero elements m **in the vector of partial sums**

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

    $\Sigma_m \leftarrow \Sigma_m + (L_{i,k} * U_{k,j})$ where $m \rightarrow (i,j)$

  endfor

endfor

Send the sparse vector of partial sums to the host processor

Host Program

/* Update values in the last block using partial sums calculated by nodes from the border */

for all processors p

  Receive the next available vector of partial sums

  for all elements m **in the vector of partial sums**

    $A_{i,j} \leftarrow A_{i,j} - \Sigma_m$ where $m \rightarrow (i,j)$

  endfor

endfor

/* factor the last block */

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

  Update the k^{th} *column using data from the last block*

  Update the k^{th} *row using data from the last block*

endfor

Figure 16: Parallel Sparse LU Factorization

Pseudo-code versions of the algorithms for parallel forward reduction and parallel backward substitution are presented in figures 17 and 18 respectively. As in the sequential algorithm, the parallel backward substitution step does not require a final division operation, because all values $U_{i,i}$ are equal to 1, Detailed versions of the factorization, forward reduction, and backward substitution algorithms are presented in appendix B.

6 Preliminary test results

6.1 Test Matrices

Preliminary empirical data collected using the block-diagonal-bordered sparse matrix solver has been collected on the Thinking Machines CM-5. In spite of the fact that the parallel implementation has instrumentation messages creating additional communications overhead, and the last block is factored sequentially on the host processor, the speedup for the parallel implementation performance has been substantial for large sparse matrices. The data used in the tests has been machine generated with a regular pattern, although the software implementation does not exploit the data regularities — the matrix was solved as if it were irregular. A regular data matrix was used because of the ease with which the location of all fillin values could be determined. In order to eliminate the need for load balancing, the number of independent blocks used in each test matrix is a multiple of the number of available processors on the Northeast Parallel Architectures Center (NPAC) CM-5.

An example of the test matrices is presented in figure 19. This figure depicts the sparsity structure in a matrix, where non-zero matrix elements can be interpreted as either the edges in a graph or they can be interpreted as non-zero coefficients in sparse linear equations. Non-zero values are black, zero values are light gray, and fillin values are medium gray. This matrix is the smallest test matrix for which performance data is presented in this paper, and it has 32 independent blocks, with four sub-blocks and four separate sets of coupling equations in the sparse borders in each independent block. Matrices with larger numbers of equations are generated by increasing the number of independent blocks and the corresponding coupling equations in the borders. The choice of this test matrix form has been motivated partially by the simplicity of the form in order to generate rapidly regular matrices for code verification and preliminary benchmarking purposes, but also motivated by the desire to expose the algorithms described in the previous section to controlled data in order to investigate the performance of each portion of the sequential and parallel block-diagonal-bordered sparse matrix algorithms. The use of these controlled matrices limits all load imbalance overhead in subsequent benchmarking. As a result, processing bottlenecks are not masked by artifacts in the data. When working with irregular sparse matrices,

Node Program

/* reduce the independent blocks */

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

  for all rows i **in block** l

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

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

  endfor

endfor

/* update the last block */

for all rows i **in the last block**

  if row i has any non-zero $L_{i,j}$

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

      $\Sigma_m \leftarrow \Sigma_m + (y_i * L_{i,j})$ where $m \leftarrow i$

    endif

  endif

endfor

Send the sparse vector of partial sums to the host processor

Host Program

/* Update b_i values using partial sums from the borders */

for all processors p

  Receive the next available vector of partial sums

  for all elements m **in the vector of partial sums**

    $b_i \leftarrow b_i - \Sigma_m$ where $m \rightarrow i$

  endfor

endfor

/* reduce the last block */

for all rows i **in the last block**

  Update b_i *using* y_j *and* $L_{i,j}$ *values in the last block*

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

endfor

Figure 17: Parallel Sparse Forward Reduction

Host Program

/* substitute in the last block */

for all rows i **in the last block**

Update y_i *using* x_j *and* $U_{i,j}$ *values in the last block*

$x_i \leftarrow y_i$

endfor

Broadcast the x-values to the node processors

Node Program

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

Receive the x values from the host

for all independent blocks l

for all rows i **in block** l

Update y_i *using* x_j *and* $U_{i,j}$ *values in the border*

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

$x_i \leftarrow y_i$

endfor

endfor

Figure 18: Parallel Sparse Backward Substitution

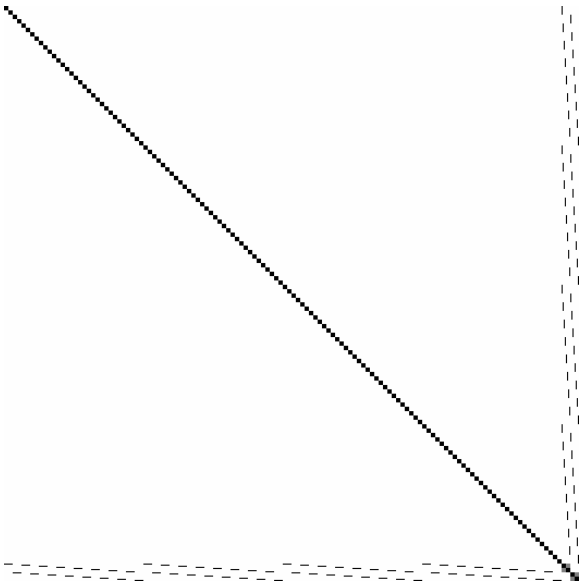

Figure 19: Sample Test Matrix With 32 Independent Blocks

it will be the task of the load balancing step to minimize any load imbalance overhead. Eventually, performance of the algorithms will be dependent on the inherent structure of real, irregular sparse matrices. Moreover, we are examining matrix partitioning and load balancing algorithms that will yield better performance than nested dissection algorithms on irregular data.

6.2 Parallel Algorithm Performance

Parallel algorithm performance was measured using 32 processors, the default number of processors on NPAC's Thinking Machines CM-5. The number of independent blocks, number of equations per matrix, the number of non-zero values, and the percentage of non-zero values are presented in table 1. The data used to test the algorithms should exhibit linear timing performance for factorization as the number of equations is increased, because the performance for sequential and parallel factorization can be described as the sum of multiple terms where the only variables in the equations are N_{blocks}, the number of independent blocks, and $N_{blocks/proc}$, the number of independent blocks assigned to each processor.

$$T_{seq} \approx (N_{blocks} \times T_{ind}) + (N_{blocks} \times T_{\Sigma}) + T_{last} \tag{17}$$

$$T_{par} \approx (N_{\frac{blocks}{proc}} \times T_{ind}) + ((N_{\frac{blocks}{proc}} \times T_{\Sigma}) + (N_{procs} \times T_{comm})) + (N_{procs} \times T_{vps}) + T_{last} \tag{18}$$

where:

T_{seq} is the total factorization time for the sequential algorithm.
T_{par} is the total factorization time for the parallel algorithm.
N_{blocks} is the number of independent blocks.
$N_{blocks/proc}$ is the number of independent blocks per processor.
T_{ind} is the time to factor an independent block.
T_{Σ} is the time to calculate the partial sums of updates in the borders.
T_{comm} is the time to communicate the partial sums to the host node.
N_{procs} is the number of processors.
T_{vps} is the time to update $A_{i,j}$ values using the partial sums.
T_{last} is the time to factor the last block.

All values in these equations are essentially constants, except for N_{blocks} and $N_{blocks/proc}$. Given the equations to predict processing time for both the sequential and parallel factorization with this data, run-time algorithm performance is expected to be proportional to N_{blocks}, the number of blocks. There are a constant number of equations per block, so the time to factor these test matrices should be proportional to the total number of equations or the order of the matrix. The computational complexity to factor an independent block, T_{ind}, in equations 17 and 18 is $O(n_{ind}^3)$ where n_{ind} is the number of equations in the independent sub-matrix. As long as n_{ind} remains constant, the time to factor the entire matrix should grow at a constant rate.

Figure 20 presents the wall-clock time required to factor test matrices with the sequential algorithm running only on the CM-5 host processor and the parallel algorithm running on the CM-5 host processor and the 32 node processors. The time required to factor these test matrices is linear as predicted for both the sequential and parallel algorithms. Classes of test matrices with linear growth in the number of calculations provide conservative estimates of speedup and efficiency for parallel algorithms when compared to classes of test matrices that have numbers of calculations that increase at polynomial rates. These test matrices add independent blocks of constant size rather than enlarging the independent blocks. Consequently, this class of test matrices has linear growth in the number of calculations as a function of the matrix order, in contrast to classes of test matrices where the size of sub-blocks is increased and the numbers of calculations would increase at a polynomial rate. For the class of test matrices with enlarged sub-blocks, improved performance could be a function of the increasing number of calculations that are available to be performed in parallel. The test matrices used throughout this preliminary examination of block-diagonal-bordered sparse matrix factorization algorithms exhibit good correlations of these theoretical predictions with empirical data,

# *blocks/proc*	**n**	**# non-zero**	**% non-zero**
1	2112	38912	0.87210
2	4160	75776	0.43790
4	8256	149504	0.21930
8	16448	296960	0.10980
16	32832	591872	0.05490

Table 1: Sample Matrix Sizes

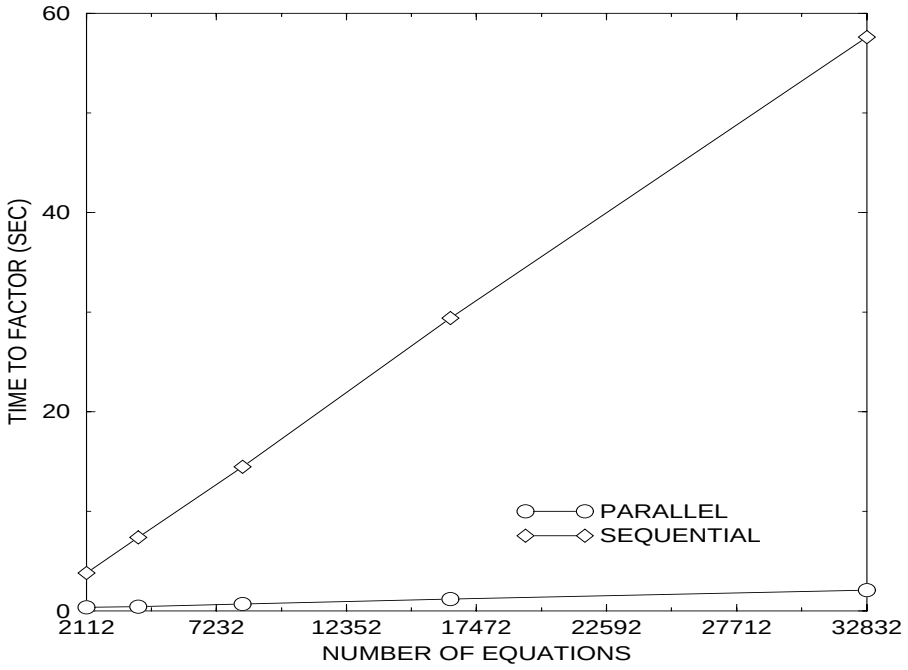

Figure 20: Linear Performance of Block-Bordered-Diagonal Sparse Matrix Factorization

The empirical data on the performance of the parallel block-diagonal-bordered sparse matrix solver described in section 5 has been collected using the 32 processor NPAC CM-5 for the block-diagonal-bordered sparse matrix factorization algorithms implemented in C using the CMMD version 2.0 library of message passing functions. Speedup has been calculated for the five sample matrices, and is graphically presented in figure 21. This graph plots speedup versus the number of equations in the matrix for

- total ($\circ$) — factorization, a single forward reduction, and a single backward substitution,
- a single factorization ($\diamond$),
- a single forward reduction ($\triangleright$),
- a single backward substitution ($\triangleleft$).

Likewise, efficiency is graphically represented in figure 22. For factorization, speedup ranges from 10.2 to 27.5 for the 32 processor NPAC CM-5 which corresponds to efficiency values ranging from 32% to 86%.

Parallel factorization performance is rather good; however, forward reduction and backward substitution performance is not as promising for this implementation. Speedup for forward reduction ranges from 0.5 to 2.0 and more significantly, the actual times for parallel factorization and parallel forward substitution are nearly equal. For a matrix with 32832 equations, the ratio of the time for factorization to the time for forward reduction is 3.5. For 2112 equations, the time ratio is only 2.3. Performance difficulties with the parallel forward reduction algorithm are caused by fewer calculations in the forward reduction step than the factorization step, and inadequate calculations exist to overcome the communications overhead. Additional research will address faster communications techniques to attempt to minimize this problem,

Meanwhile, speedup for parallel backward substitution is less than for parallel factorization, although somewhat better than for parallel forward reduction. The broadcast communications step in parallel backward substitution is more efficient than the accumulation step in parallel forward reduction, primarily because there are less sequential operations involved. Parallel backward substitution performance yielded speedups of 1.3 to 19.2 that correspond to efficiency values of 4% to 60%. Parallel performance tends to increase rather rapidly as larger matrices are processed. The empirical data collected for backward substitution is not monotonically increasing, as is all other speedup curves in figure 21. The probable explanation for this phenomenon is that measured performance for the parallel algorithm at this point was corrupted by processor contention. Given the small actual run time, a small increase in execution time would drastically effect this measurement. There

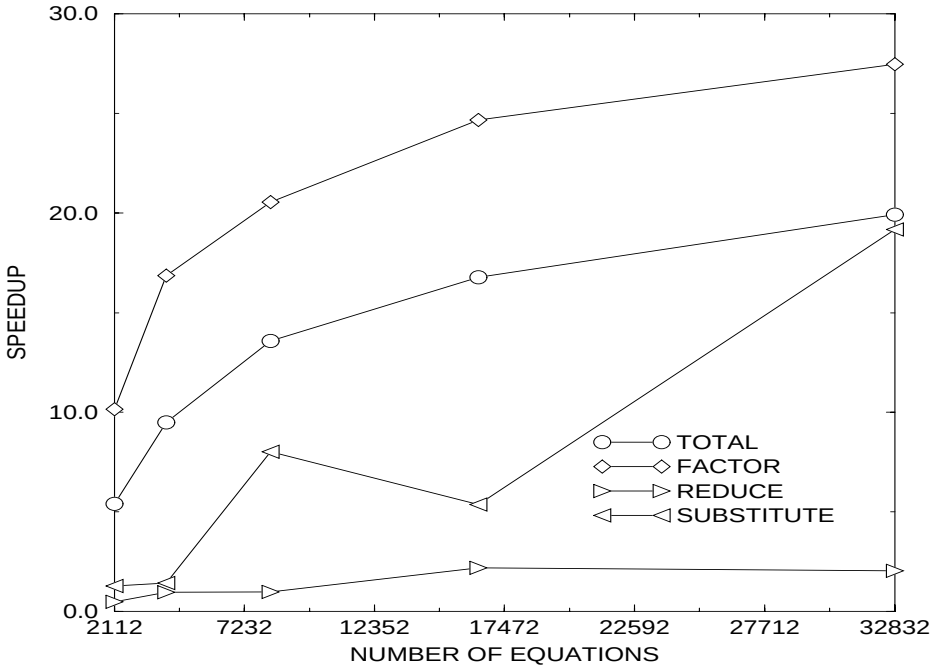

Figure 21: Speedup for the Block Bordered Diagonal Form Sparse Matrix Solver

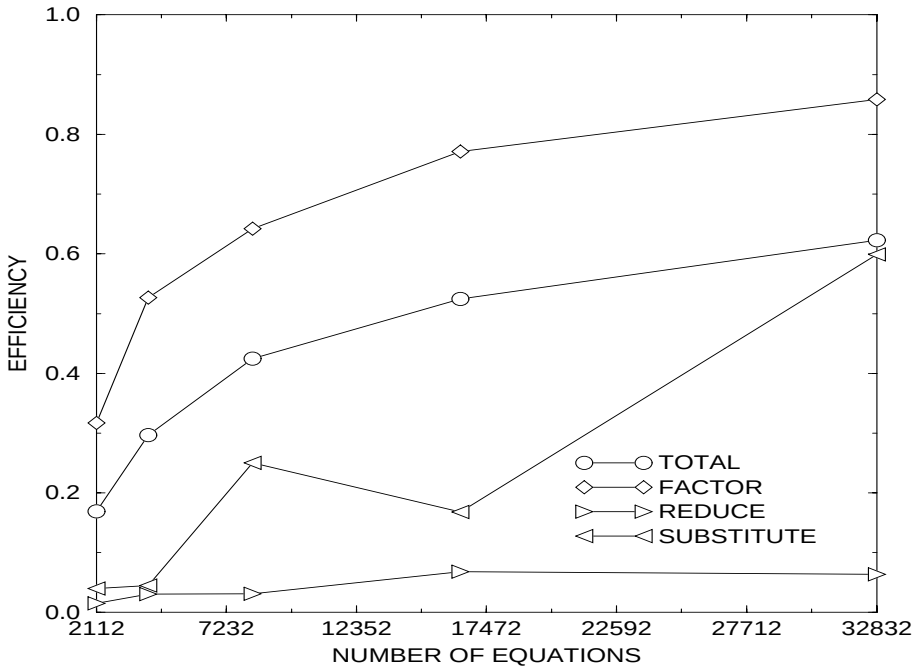

Figure 22: Efficiency for the Block Bordered Diagonal Form Sparse Matrix Solver

is no reason to believe that performance for backward substitution would not be follow a curve fitted to the other data points.

While the speedups for parallel forward reduction and parallel backward substitution are less than the speedup for factorization, the performance for factoring the matrix and a single forward reduction and a single backward substitution step is still reasonably good. Speedup for the solution of a single b vector ranges from 5.4 to 19.9 with corresponding efficiency values of 17% to 62%.

6.3 Processing Bottlenecks

The implementation of the parallel block-diagonal-bordered sparse matrix solver included communications to gather timing information for the node processors. Timing information was sent from the nodes to the host processor for display. These timing values permitted a visual check of parallel algorithm performance during run time and also provided data to analyze processing bottlenecks. The sample data was by definition load balanced, so disparities in this timing data are unquestionably a result of processing bottlenecks. The only area where processing bottlenecks are apparent in the data occurs in the accumulation of partial sums in the factorization and forward reduction steps.

Table 2 presents the minimum and maximum times reported by the nodes when generating the sparse vector of partial sums. The minimum values represent the time required to calculate the partial sums of update values, to generate the sparse vector of partial sums, and to perform the communications with the host node. The maximum values also include any time the slowest node processor incurred while waiting for the host processor to complete the communications of the sparse vector. There are two distinct bottlenecks illustrated in this data. The first bottleneck is a result of the time required to develop the sparse vector of partial sums, and the second bottleneck is caused by the sequential processing of the partial sums at the host node. All partial sums are calculated in parallel during this step before any communications are performed: then all node processors send the partial sums to the host processor which must both receive the messages and finish sequentially accumulating the partial sums.

Until later versions of the parallel block-diagonal-bordered sparse matrix solver perform the factorization of the last block in parallel, there is nothing that could affect the second bottleneck, however, the use of a different communications model that permits the efficient use of short messages can minimize the first bottleneck. The initial startup time — represented by the values in the two columns of minimum processing times — can be reduced to nearly zero, with the use of *active messages*. With active messages, several words can be sent between any processors on the CM-5 in less than 2 microseconds [22, 28]. This is

	Factorization		Forward Reduction	
n	**minimum**	**maximum**	**minimum**	**maximum**
2112	60,987 μs	93,805 μs	34,572 μs	34,572 μs
4160	27,658 μs	82,843 μs	154,075 μs	154,079 μs
8256	13,9324 μs	245,315 μs	264,733 μs	264,756 μs
16448	117,640 μs	299,392 μs	252,238 μs	252,248 μs
32832	258,028 μs	806,859 μs	658,532 μs	658,547 μs

Table 2: Timings for Processing Bottlenecks

substantially less than the minimum times reported in table 2, so an active message construct in the algorithm should minimize the delay time that the host processor sits idle while waiting to begin processing.

Active messages will also be a significant factor in future implementations when the data for the last block is distributed onto multiple processors. For such implementations, multiple messages would be required to send the partial sums to the appropriate node processors. When data for the last block is distributed onto multiple processors, minimizing the idle time of a single processor waiting for data from the first block will no longer be of primary concern. Rather for these implementations, the focus will be to minimize the total communications latency incurred because more, although, shorter messages will be required than in the present implementation. Active messages are specifically designed to have reduced communications latency, so both processing bottlenecks observed above can be minimized in those implementations that perform the last calculations in parallel and use active messages for communications.

6.4 Comparison to Amdahl's Law

These parallel algorithm implementations each have a substantial sequential portion. Amdahl's law [16] describes theoretical limits on parallel performance due to the sequential portion of the algorithm. After reviewing the definition, Amdahl's law-based performance estimates are compared to measured performance for parallel block-diagonal-bordered sparse matrix factorization.

Definition (Amdahl's Law) *Given T_1, the time to solve a problem on a single processor, then T_{n_p}, the time to solve a problem on n_p processors, can be parameterized in $\alpha \in [0,1]$ by*

$$T_p \equiv \alpha T_1 + (1-\alpha)\frac{T_1}{p}, \tag{19}$$

where α is the inherently sequential fraction of computations. The aforementioned estimate of T_p can be used when estimating the relative speedup S_p [21] by

$$S_p = \frac{p}{1 + (p-1)\alpha} = \frac{1}{\alpha + (1-\alpha)/p} \leq S_\infty \equiv \alpha^{-1}, \tag{20}$$

where relative speedup is defined as

$$S_p \equiv \frac{T_1}{T_{n_p}}. \tag{21}$$

Amdahl's Law can be used to estimate the maximum potential relative speedup by taking the inverse of the sequential portion of the parallel problem.

It can be shown that the amount of processing required for the last diagonal block in the test matrices is approximately four times the processing required for each independent block. Independent diagonal blocks are divided into four sections, while the last block is divided into only two sections. The computational complexity of each independent matrix subsection is $O(n_{ind}^3)$. For the test matrix with 32 independent blocks, the sequential workload is approximately

$$\frac{(2 \times 8)}{((32 \times 4 \times 1) + (2 \times 8))} = \frac{1}{9}. \tag{22}$$

According to Amdahl's law, a task with $\frac{1}{9}$ of the operations being sequential limits speedup to no more than 9, regardless of the number of processors applied to the problem. The speedup obtained for parallel block-diagonal-bordered sparse matrix factorization for this matrix was 5.4 — which is consistent with Amdahl's law because the measured speedup does not exceed the Amdahl's law estimate of the bound on speedup. For the largest matrix reported in this paper, the sequential portion of the processing would be only $\frac{1}{129}$ of the total operations. Consequently, according to Amdahl's law, speedup would not be limited by sequential calculations for a 32 processor CM-5. Additional computations in block-diagonal-bordered sparse matrix algorithms must be performed in parallel for performance to be scalable for smaller sized matrices.

While the parallel block-diagonal-bordered matrix algorithms described in this report had significant amounts of sequential calculations in the last block, other versions of these algorithms are possible that significantly increase the amount of parallel operations when processing this section of the matrix. In spite of the large amount of sequential calculations in the last block, this simple test implementation illustrates that parallel factorization performance is very promising. Nevertheless, additional operations in future algorithms must be performed in parallel so that those algorithms are scalable. Depending on the characteristics of the data, it may be possible to factor the last block using proven parallel dense matrix techniques. Meanwhile, state-of-the-art communications techniques can also be employed to improve the communications performance of all three parallel algorithms. The

same active message communications techniques that may improve parallel factorization performance may also reduce the bottlenecks when performing parallel forward reduction and utilizing active messages may improve the broadcast of x vector values in the backward substitution phase.

7 Conclusions

This paper describes research into the applicability of parallel direct block-diagonal-bordered sparse matrix solvers. Direct sparse solvers for matrices in this form exhibit distinct advantages. In particular, data communications are significantly reduced when compared to general direct sparse matrix solvers described in the present literature and communications are also more uniform and structured. Communications in block-diagonal-bordered sparse matrix solvers is a function of the number of processors and less dependent on the data. Task assignments for numerical factorization of a block-diagonal-bordered sparse matrix depend only on the assignment of independent blocks to processors and the processor assignments of data in the last diagonal block. Block-diagonal-bordered sparse matrix solvers require special preprocessing to be efficient. Other papers in the literature have proposed a two step preprocessing phase that includes *ordering* the matrix and then performing *symbolic factorization* to determine the locations of all fillin values so that static data structures can be utilized for maximum efficiency when performing numerical factorization. In this paper, we describe the requirements for a three step preprocessing phase for efficient parallel block-diagonal-bordered sparse matrix solvers on distributed-memory multi-processors that includes:

- *ordering* the matrix into block-bordered-diagonal form,
- *pseudo factoring* the matrix to identify both fillin and number of calculations for independent blocks,
- *load balancing* the workload to distribute the calculations among processors uniformly.

This first implementation of block-diagonal-bordered sparse matrix algorithms demonstrates good speedup for factorization with test matrices, in spite of the sequential portions of the code that significantly limit parallel performance and algorithm scalability. There are areas for improvement in the algorithms presented herein. As part of our on-going research, we plan to parallelize all calculations, focusing on those calculations in the last matrix block. We plan to examine ordering techniques to determine whether or not efficient vectorization of the calculations are feasible for real power systems admittance matrices. We also plan to use new sophisticated communications techniques such as active

messages and virtual channels to minimize processing bottlenecks and permit hiding communications overhead behind calculations. Further research is planned to tune the parallel block-diagonal-bordered sparse matrix algorithms for optimum performance on the Thinking Machines CM-5 with power systems network matrices, and then develop additional versions of the parallel block-diagonal-bordered sparse matrix algorithms for efficient operation on other multi-processor architectures. Lastly, we intend to compare the performance and accuracy of these techniques with iterative methods, such as waveform relaxation [18].

Future research may even examine the applicability of producing an automated tool that has the capabilities of finding both an optimal matrix ordering and an optimal version of the multi-processor algorithm that is driven by the characteristics of the irregular sparse matrix and the target computer architecture. Numerous versions of the block-diagonal-bordered sparse matrix algorithms are possible depending on the specific computer architecture and characteristics of the matrix.

The research presented in this paper illustrates great promise for parallel block-diagonal-bordered sparse matrix solvers when the data is regular and there is no load imbalance overhead encountered in the parallel algorithm. In order to minimize load imbalance overhead with real sparse matrices, explicit load balancing will be required for distributed-memory multi-processor algorithms. To ensure the most uniform distribution of workload to processors, research to examine graph partitioning, node clustering, and node tearing techniques for irregular matrices will be required as part of the ordering phase. Techniques to order irregular matrices into recursive block-diagonal-bordered form have been examined for power systems distribution networks. This work is discussed in a companion paper [14].

References

[1] M. M. Adibi, P. M. Hirsch, and J. A. Jordan, Jr. Solution Methods for Transient and Dynamic Stability. *Proceedings of the IEEE*, 62(7):951–958, July 1974.

[2] P. M. Anderson and B. Demhart. Computational Aspects of Transient Stability Analysis. In A. M. Erisman, K. W. Neves, and M. H. Dwarakanath, editors, *Electrical Power Problems: The Mathematical Challenge*, pages 159–189. SIAM, Philadelphia, July 1980.

[3] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection. Technical Report RNR-92-?, NASA Ames Research Center, 1992.

[4] 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.

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

[6] J. Fong and C. Pottle. Parallel Processing of Power System Analysis Problems Via Simple Parallel Microcomputer Structures. *IEEE Transactions on Power Apparatus and Systems*, PAS-97(5):1834–1841, September/October 1978.

[7] 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.

[8] 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.

[9] 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.

[10] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

[12] 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.

[13] 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.

[14] D. P. Koester, S. Ranka, and G. C. Fox. Ordering and Load Balancing Irregular Matrices for Block-Diagonal-Bordered LU Factorization. Technical Report SCCS-xxx, Northeast Parallel Architectures Center, 1993. (work in progress).

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

[16] P. C. Patton. Performance Limits for Parallel Processors. In G. F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, chapter 1. John Wiley & Sons, New York, 1989.

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

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

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

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

[21] A. Skjellum. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic Process Systems in Chemical Engineering*. PhD thesis, California Institute of Technology, Division of Chemistry and Chemical Engineering, Pasadena, CA, 1990.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. Technical report, Computer Science Division — EECS, University of California, Berkeley, CA, March 1992. Report No. UCB/CSD 92/#675.

[29] 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.

[30] 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.

[31] 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 The Hierarchical Data Structure

The block-diagonal-bordered sparse matrix solver is written in the C programming language, and the implementation of the hierarchical data structure uses vectors of C structure data types. A detailed list of the eight parts of the hierarchical data structure, in addition to a description of each data structure field, are listed below.

1. block identifier
    - pointer to the last diagonal element in the block
    - pointer to the first non-zero lower border row pointer
    - pointer to the first non-zero upper border column pointer
2. matrix diagonal element
    - $A_{i,j}$ — non-zero value
    - pointer to the first non-zero value in row i in this lower diagonal block
    - pointer to the next non-zero value in column j in this lower diagonal block
    - pointer to the first non-zero value in column j in this upper diagonal block
    - pointer to the next non-zero value in row i in this upper diagonal block
3. non-zero value in a lower triangular matrix diagonal block (arranged by rows)
    - $A_{i,j}$ — non-zero value
    - i — row indicator
    - j — column indicator
    - pointer to the next non-zero value in column j in this lower diagonal block
4. non-zero value in a upper triangular matrix diagonal block (arranged by columns)
    - $A_{i,j}$ — non-zero value
    - i — row indicator
    - j — column indicator
    - pointer to the next non-zero value in row i in this upper diagonal block
5. non-zero row in the lower border
    - pointer to the first non-zero lower border row
6. non-zero column in the upper border

- pointer to the first non-zero upper border column

7. non-zero value in the lower border (arranged by rows)
   - $A_{i,j}$ — non-zero value
   - i — row indicator
   - j — column indicator
   - pointer to the next non-zero value in column j in the lower border
8. non-zero value in the upper border (arranged by columns)
   - $A_{i,j}$ — non-zero value
   - i — row indicator
   - j — column indicator
   - pointer to the next non-zero value in row i in the upper border

B Block Bordered Diagonal Form Sparse Matrix Algorithms

This appendix contains detailed descriptions of both the sequential and parallel algorithms implemented for the Thinking Machines CM-5. Figures 23 and 24 describe the sequential block-diagonal-bordered sparse LU factorization algorithm that was implemented using the hierarchical data structure described in section 5. Figure 23 describes the sequential algorithm to factor the independent blocks, and figure 24 describes the sequential algorithm to factor the last block. Figures 25 and 26 respectively describe the sequential forward reduction and backward substitution steps. Figures 28 and 27 describe the parallel block-diagonal-bordered sparse LU factorization that was implemented for the CM-5 using the CMMD message passing libraries for the node and host programs respectively. The parallel forward reduction algorithm is presented in figures 30 and 29, also for the node and host programs respectively. The parallel backward substitution algorithm is presented in figure 31. Additional implementation details are discussed in section 5.

```
/* factor the independent blocks and corresponding borders */
for l = 1 to (N_blocks − 1) /* for all independent blocks */
    for k = (n_(l−1) + 1) to n_l /* for all elements along the diagonal in block l */
        for each i ∈ [k, n_l] /* lower diagonal blocks */
            for each j ∈ [(n_(l−1) + 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 ∈ [(n_(N_blocks−1) + 1), n_{N_blocks}] /* lower borders */
            for each j ∈ [(n_(l−1) + 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 j ∈ [(k + 1), n_l] /* upper diagonal blocks */
            for each i ∈ [(n_(l−1)+1), k − 1] such that A_{i,j} ≠ 0 and  A_{k,i} ≠ 0
                A_{k,j} ← A_{k,j} − (A_{k,i} * A_{i,j})
            endfor
            A_{k,j} ← (A_{k,j}/A_{k,k})
        endfor
        for each j ∈ [(n_(N_blocks−1) + 1), n_{N_blocks}] /* upper borders */
            for each i ∈ [(n_(l−1)+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
            A_{k,j} ← (A_{k,j}/A_{k,k})
        endfor
    endfor
endfor
```

Figure 23: Sequential Sparse LU Factorization — Independent Blocks

```
/* factor the last block */
for k = (n_(N_blocks-1) + 1) to n_N_blocks /* for all elements along the diagonal in the last block */
    for each i ∈ [k, n_N_blocks] /* lower diagonal block */
        for each j ∈ [(n_(N_blocks-1) + 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
        for l = 1 to (N_blocks − 1) /* for all independent blocks */
            for each j ∈ [(n_(l-1) + 1), n_l] 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
    endfor
    for each j ∈ [k + 1, n_N_blocks] /* upper diagonal block */
        for each i ∈ [(n_(N_blocks-1) + 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
        for l = 1 to (N_blocks − 1) /* for all independent blocks */
            for each i ∈ [(n_(l-1) + 1), n_l] such that A_k,i ≠ 0 and A_i,j ≠ 0
                A_k,j ← A_k,j − (A_k,i * A_i,j)
            endfor
        A_k,j ← (A_k,j/A_k,k)
        endfor
    endfor
endfor
```

Figure 24: Sequential Sparse LU Factorization — Last Block

```
/* reduce the independent blocks */
for l = 1 to (N_blocks − 1) /* for all independent blocks */
    for i = (n_(l−1) + 1) to n_l /* for all rows in block l */
        for each j ∈ [(n_(l−1) + 1), (i − 1)] such that L_{i,j} ≠ 0
            b_i ← b_i − (y_j * L_{i,j})
        endfor
        y_i ← (b_i / L_{i,i})
    endfor
endfor
/* reduce the last block */
for i = (n_(N_blocks−1) + 1) to n_{N_blocks} /* for all rows in the border and last block */
    for l = 1 to (N_blocks − 1) /* for all independent blocks */
        for each j ∈ [(n_(l−1) + 1), n_l] such that L_{i,j} ≠ 0
            b_i ← b_i − (y_j * L_{i,j})
        endfor
        for each j ∈ [(n_(N_blocks−1) + 1), (i − 1)] such that L_{i,j} ≠ 0
            b_i ← b_i − (y_j * L_{i,j})
        endfor
        y_i ← (b_i / L_{i,i})
    endfor
endfor
```

Figure 25: Sequential Sparse Forward Substitution

/* substitute in the last block */
for $i = n_{N_{blocks}}$ **to** $(n_{(N_{blocks}-1)} + 1)$ /* for all rows in the last block */
    for each $j \in [(i+1), (n_{N_{blocks}}]$ such that $U_{i,j} \neq 0$
        $y_i \leftarrow y_i - (x_j * U_{i,j})$
    endfor
    $x_i \leftarrow y_i$
endfor
/* reduce the independent blocks and the border*/
for $l = 1$ **to** $(N_{blocks} - 1)$ /* for all independent blocks */
    for $i = n_l$ **to** $(n_{(l-1)} + 1)$ /* for all rows in block l */
        for each $j \in [(n_{(N_{blocks}-1)} + 1), n_{N_{blocks}}]$ such that $U_{i,j} \neq 0$
            $y_i \leftarrow y_i - (x_j * U_{i,j})$
        endfor
        for each $j \in [(i+1), n_l]$ such that $U_{i,j} \neq 0$
            $y_i \leftarrow y_i - (x_j * U_{i,j})$
        endfor
        $x_i \leftarrow y_i$
    endfor
endfor

Figure 26: Sequential Sparse Backward Substitution

/* factor the independent blocks and corresponding borders */
for those independent blocks assigned to this processor $l \in \{1, \ldots, (N_{blocks} - 1)\}$
  for $k = (n_{(l-1)} + 1)$ **to** n_l /* for all elements along the diagonal in block l */
    for each $i \in [k, n_l]$
      for each $j \in [(n_{(l-1)} + 1), k - 1]$ 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 $i \in [(n_{(N_{blocks}-1)} + 1), n_{N_{blocks}}]$
      for each $j \in [(n_{(l-1)} + 1), k - 1]$ 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 $j \in [(k + 1), n_l]$
      for each $i \in [(n_{(l-1)} + 1), k - 1]$ such that $A_{i,j} \neq 0$ **and** $A_{k,i} \neq 0$
        $A_{k,j} \leftarrow A_{k,j} - (A_{k,i} * A_{i,j})$
      endfor
      $A_{k,j} \leftarrow (A_{k,j}/A_{k,k})$
    endfor
    for each $j \in [(n_{(N_{blocks}-1)} + 1), n_{N_{blocks}}]$
      for each $i \in [(n_{(l-1)} + 1), k - 1]$ 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
      $A_{k,j} \leftarrow (A_{k,j}/A_{k,k})$
    endfor
  endfor
endfor
/* update the last block */
for all non-zero elements m **in the vector of partial sums** $m = 1, \ldots, n_{vps}$
  for each k such that $L_{i,k}$ and $U_{k,j} \neq 0$
    $\Sigma_m \leftarrow \Sigma_m + (L_{i,k} * U_{k,j})$ where $m \rightarrow (i, j)$
  endfor
endfor
Send the sparse vector of partial sums to the host processor

Figure 27: Parallel Sparse LU Factorization - Node Program

/* Update values in the last block using partial sums calculated by nodes from the border */
for all processors $p = 1, \ldots, N_{procs}$
    Receive the next available vector of partial sums
    for all elements m **in the vector of partial sums** $m = 1, \ldots, n_{vps}$
        $A_{i,j} \leftarrow A_{i,j} - \Sigma_m$ where $m \rightarrow (i,j)$
    endfor
endfor
/* factor the last block */
for $k = (n_{(N_{blocks}-1)} + 1)$ **to** $n_{N_{blocks}}$ /* for all elements along the diagonal in the last block */
    for each $i \in [k, n_{N_{blocks}}]$ /* lower diagonal block */
        for each $j \in [(n_{(N_{blocks}-1)} + 1), k-1]$ 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 $j \in [k+1, n_{N_{blocks}}]$ /* upper diagonal block */
        for each $i \in [(n_{(N_{blocks}-1)} + 1), k-1]$ 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
        $A_{k,j} \leftarrow (A_{k,j}/A_{k,k})$
    endfor
endfor

Figure 28: Parallel Sparse LU Factorization - Host Program

/* reduce the independent blocks */
for those independent blocks assigned to this processor $l \in \{1, \ldots, (N_{blocks} - 1)\}$
    for $i = (n_{(l-1)} + 1)$ **to** n_l /* for all rows in block l */
        for each $j \in [(n_{(l-1)} + 1), (i - 1)]$ such that $L_{i,j} \neq 0$
            $b_i \leftarrow b_i - (y_j * L_{i,j})$
        endfor
        $y_i \leftarrow (b_i / L_{i,i})$
    endfor
endfor
/* update the last block */
for all independent blocks $l = 1, \ldots, (N_{blocks} - 1)$
    for all rows i **in the last block** $i \in [(n_{(N_{blocks-1})} + 1), n_{N_{blocks}}]$
        if row i has any non-zero $L_{i,j}$
          $m \rightarrow i$
          for each j such that $L_{i,j} \neq 0$
              $\Sigma_m \leftarrow \Sigma_m + (y_i * L_{i,j})$
          endfor
        endif
    endfor
endfor
Send the sparse vector of partial sums to the host processor

Figure 29: Parallel Sparse Forward Reduction - Node Program

/* Update b_i values using partial sums from the borders */

for all processors $p = 1, \ldots, N_{procs}$

Receive the next available vector of partial sums

for all elements m **in the vector of partial sums** $m = 1, \ldots, n_{vps}$

$b_i \leftarrow b_i - \Sigma_m$ where $m \rightarrow i$

endfor

endfor

/* reduce the last block */

for $i = (n_{(N_{blocks}-1)} + 1)$ **to** $n_{N_{blocks}}$ /* for all rows in the last block */

for $l = 1$ **to** $(N_{blocks} - 1)$ /* for all independent blocks */

for each $j \in [(n_{(l-1)} + 1), n_l]$ such that $L_{i,j} \neq 0$

$b_i \leftarrow b_i - (y_j * L_{i,j})$

endfor

for each $j \in [(n_{(N_{blocks}-1)} + 1), (i - 1)]$ such that $L_{i,j} \neq 0$

$b_i \leftarrow b_i - (y_j * L_{i,j})$

endfor

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

endfor

endfor

Figure 30: Parallel Sparse Forward Reduction - Host Program

Host Program

/* substitute in the last block */

for $i = n_{N_{blocks}}$ **to** $(n_{(N_{blocks}-1)} + 1)$ /* for all rows in the last block */

  for each $j \in [(i+1), n_{N_{blocks}}]$ such that $U_{i,j} \neq 0$

    $y_i \leftarrow y_i - (x_j * U_{i,j})$

  endfor

  $x_i \leftarrow y_i$

endfor

Broadcast the x-values to the node processors

Node Program

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

Receive the x-values

for those independent blocks assigned to this processor $l \in \{1, \ldots, (N_{blocks} - 1)\}$

  for $i = (n_{(l-1)} + 1)$ **to** n_l /* for all rows in block b */

    for each $j \in [(n_{(N_{blocks-1})} + 1), n_{N_{blocks}}]$ such that $U_{i,j} \neq 0$

      $y_i \leftarrow y_i - (x_j * U_{i,j})$

    endfor

    for each $j \in [(i+1), n_l]$ such that $U_{i,j} \neq 0$

      $y_i \leftarrow y_i - (x_j * U_{i,j})$

    endfor

    $x_i \leftarrow y_i$

  endfor

endfor

Figure 31: Parallel Sparse Backward Substitution