A Parallel Gauss-Seidel Algorithm for Sparse Power Systems 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

A Comdensed Version of this Paper was presented at SuperComputing '94

NPAC Technical Report — SCCS 630

4 April 1994

Abstract

We describe the implementation and performance of an efficient parallel Gauss-Seidel algorithm that has been developed for irregular, sparse matrices from electrical power systems applications. Although, Gauss-Seidel algorithms are inherently sequential, by performing specialized orderings on sparse matrices, it is possible to eliminate much of the data dependencies caused by precedence in the calculations. A two-part matrix ordering technique has been developed — first to partition the matrix into block-diagonal-bordered form using diakoptic techniques and then to multi-color the data in the last diagonal block using graph coloring techniques. The ordered matrices often have extensive parallelism, while maintaining the strict precedence relationships in the Gauss-Seidel algorithm. We present timing results for a parallel Gauss-Seidel solver implemented on the Thinking Machines CM-5 distributed memory multi-processor. The algorithm presented here requires active message remote procedure calls in order to minimize communications overhead and obtain good relative speedup. The paradigm used with active messages greatly simplified the implementation of this sparse matrix algorithm.

1 Introduction

We have developed an efficient parallel Gauss-Seidel algorithm for irregular, sparse matrices from electrical power systems applications. Even though Gauss-Seidel algorithms for dense matrices are inherently sequential, it is possible to identify sparse matrix partitions without data dependencies so calculations can proceed in parallel while maintaining the strict precedence rules in the Gauss-Seidel technique. All data parallelism in our Gauss-Seidel algorithm is derived from within the actual interconnection relationships between elements in the matrix. We employed two distinct ordering techniques in a preprocessing phase to identify the available parallelism within the matrix structure:

1. partitioning the matrix into block-diagonal-bordered form,
2. multi-coloring the last diagonal matrix block.

Our challenge has been to identify available parallelism in the irregular sparse power systems matrices and develop an efficient parallel Gauss-Seidel algorithm to exploit that parallelism.

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

We show that it is possible to identify the hierarchical structure within a power system matrix using only the knowledge of the interconnection pattern by *tearing* the matrix into partitions and coupling equations that yield a block-diagonal-bordered matrix. Node-tearing-based partitioning identifies the basic network structure that provides parallelism for the majority of calculations within a Gauss-Seidel iteration. Meanwhile, without additional ordering, the last diagonal block would be purely sequential, limiting the potential speedup of the algorithm in accordance with Amdahl's law. The last diagonal block represents the interconnection structure within the equations that couple the partitions found in the previous step. Graph multi-coloring has been used to order this matrix partition and subsequently identify those rows that can be solved in parallel.

We implemented explicit load balancing as part of each of the aforementioned ordering steps to maximize efficiency as the parallel algorithm is applied to real power system load-flow matrices. An attempt was made to place equal amounts of processing in each partition, and in each matrix color. The metric employed when load-balancing the partitions is the number of floating point multiply/add operations, not simply the number of rows per partition. Empirical performance data collected on the parallel Gauss-Seidel algorithm illustrate the ability to balance the workload for as many as 32 processors.

We implemented the parallel Gauss-Seidel algorithm on the Thinking Machines CM-5 distributed memory multi-processor using the Connection Machine active message layer (CMAML). Using this communications paradigm, significant improvements in the performance of the algorithm were observed compared to more traditional communications paradigms that use the standard blocking send and receive functions in conjunction with packing data into communications buffers. To signif-

icantly reduce communications overhead and attempt to hide communications behind calculations, we implemented each portion of the algorithm using CMAML remote procedure calls. The communications paradigm we use throughout this algorithm is to send a double precision data value to the destination processor as soon as the value is calculated. The use of active messages greatly simplified the development and implementation of this parallel sparse Gauss-Seidel algorithm.

Parallel implementations of Gauss-Seidel have have generally been developed for regular problems such as the solution of Laplace's equations by finite differences [3, 4], where *red-black* coloring schemes are used to provide independence in the calculations and some parallelism. This scheme has been extended to multi-coloring for additional parallelism in more complicated regular problems [4], however, we are interested in the solution of irregular linear systems. There has been some research into applying parallel Gauss-Seidel to circuit simulation problems [12], although this work showed poor parallel speedup potential in a theoretical study. Reference [12] also extended traditional Gauss-Seidel and Gauss-Jacobi methods to waveform relaxation methods that trade overhead and convergence rate for parallelism. A theoretical discussion of parallel Gauss-Seidel methods for power system load-flow problems on an alternating sequential/parallel (ASP) multi-processor is presented in [15]. Other research with the parallel Gauss-Seidel methods for power systems applications is presented in [7], although our research differs substantially from that work. The research we present here utilizes a different matrix ordering paradigm, a different load balancing paradigm, and a different parallel implementation paradigm than that presented in [7]. Our work utilizes diakoptic-based matrix partitioning techniques developed initially for a parallel block-diagonal-bordered direct sparse linear solver [9, 10]. In reference [9] we examined load balancing issues associated with partitioning power systems matrices for parallel Choleski factorization.

The paper is organized as follows. In section 2, we introduce the electrical power system applications that are the basis for this work. In section 3, we briefly review the Gauss-Seidel iterative method, then present a theoretical derivation of the available parallelism with Gauss-Seidel for a block-diagonal-bordered form sparse matrix. Paramount to exploiting the advantages of this parallel linear solver is the preprocessing phase that orders the irregular sparse power system matrices and performs load-balancing. We discuss the overall preprocessing phase in section 5, and describe node-tearing-based ordering and graph multi-coloring-based ordering in sections 6 and 7 respectively. We describe our parallel Gauss-Seidel algorithm in section 8, and include a discussion of the hierarchical data structures to store the sparse matrices. Analysis of the performance of these ordering techniques for actual power system load flow matrices from the Boeing-Harwell series and for a matrix distributed with the Electrical Power Research Institute (EPRI) ETMSP software are presented in section 9. Examinations of the convergence of the algorithm are presented along with parallel algorithm performance. We state our conclusions in section 10.

2 Power System Applications

The underlying motivation for our research is to improve the performance of electrical power system applications to provide real-time power system control and real-time support for proactive decision making. Our research has focused on matrices from load-flow applications [15]. Load-flow analysis examines steady-state equations based on the positive definite network admittance matrix

that represents the power system distribution network, and is used for identifying potential network problems in contingency analyses, for examining steady-state operations in network planning and optimization, and for determining initial system state in transient stability calculations [15]. Load flow analysis entails the solution of non-linear systems of simultaneous equations, which are performed by repeatedly solving sparse linear equations. Sparse linear solvers account for the majority of floating point operations encountered in load-flow analysis. Load flow is calculated using network admittance matrices, which are symmetric positive definite and have sparsity defined by the power system distribution network. Individual power utility companies often examine networks in their operations centers that are represented by less than 2,000 sparse complex equations, while regional power authority operations centers would examine load-flow with matrices that have as many as 10,000 sparse complex equations. This paper presents data for power system networks of 1,723, 4,180, and 5,300 nodes.

3 The Gauss-Seidel Method

We are considering an iterative solution to the linear system

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

where $\mathbf{A}$ is an $(n \times n)$ sparse matrix, $\mathbf{x}$ and $\mathbf{b}$ are vectors of length n, and we are solving for $\mathbf{x}$. Iterative solvers are an alternative to direct methods that attempt to calculate an exact solution to the system of equations. Iterative methods attempt to find a solution to the system of linear equations by repeatedly solving the linear system using approximations to the $\mathbf{x}$ vector. Iterations continue until the solution is within a predetermined acceptable bound on the error.

Common iterative methods for general matrices include the Gauss-Jacobi and Gauss-Seidel, while conjugate gradient methods exist for positive definite matrices. Critical in the choice and use of iterative methods is the convergence of the technique. Gauss-Jacobi uses all values from the previous iteration, while Gauss-Seidel requires that the most recent values be used in calculations. The Gauss-Seidel method generally has better convergence than the Gauss-Jacobi method, although for dense matrices, the Gauss-Seidel method is inherently sequential. Better convergence means fewer iterations, and a faster overall algorithm, as long as the strict precedence rules can be observed. The convergence of the iterative method must be examined for the application along with algorithm performance to ensure that a useful solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be found.

The Gauss-Seidel method can be written as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \tag{2}$$

where:

$x_i^{(k)}$ is the i^{th} unknown in $\mathbf{x}$ during the k^{th} iteration, $i = 1, \cdots, n$ and $k = 0, 1, \ldots$,

$x_i^{(0)}$ is the initial guess for the i^{th} unknown in $\mathbf{x}$,

a_{ij} is the coefficient of $\mathbf{A}$ in the i^{th} row and j^{th} column,

```
ε ← ∞
while ε > ε_converge
    for k = 1 to n_iter
        for i = 1 to n
            x̃_i ← x_i
            x_i ← b_i
            for each j ∈ [1, n] such that a_ij ≠ 0
                x_i ← x_i − (a_ij * x_j)
            endfor
            x_i ← x_i / a_ii
        endfor
    endfor
    ε ← 0
    for i = 1 to n
        ε ← ε + abs(x̃_i − x_i)
    endfor
endwhile
```

Figure 1: Sparse Gauss-Seidel Algorithm

b_i is the i^{th} value in $\mathbf{b}$.

or

$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \mathbf{L})^{-1}[\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)}], \tag{3}$$

where:

$\mathbf{x}^{(k)}$ is the k^{th} iterative solution to $\mathbf{x}$, $k = 0, 1, \ldots$,

$\mathbf{x}^{(0)}$ is the initial guess at $\mathbf{x}$,

$\mathbf{D}$ is the diagonal of $\mathbf{A}$,

$\mathbf{L}$ is the of strictly lower triangular portion of $\mathbf{A}$,

$\mathbf{U}$ is the of strictly upper triangular portion of $\mathbf{A}$,

$\mathbf{b}$ is right-hand-side vector.

The representation in equation 2 is used in the development of the parallel algorithm, while the equivalent matrix-based representation in equation 3 is used below in discussions of available parallelism.

We present a general sequential sparse Gauss-Seidel algorithm in figure 1. This algorithm calculates a constant number of iterations before checking for convergence. For very sparse matrices, such as power systems matrices, the computational complexity of the section of the algorithm which checks convergence is $O(n)$, nearly the same as that of a new iteration of $\mathbf{x}^{(k+1)}$. Consequently, we perform multiple iterations between convergence checks. Only non-zero values in $\mathbf{A}$ are used when calculating $\mathbf{x}^{(k+1)}$.

It is very difficult to determine if one-step iterative methods, like the Gauss-Seidel method, converge for general matrices. Nevertheless, for some classes of matrices, it is possible to prove Gauss-Seidel methods do converge and yield the unique solution $\mathbf{x}$ for $\mathbf{A}\mathbf{x} = \mathbf{b}$ with any initial starting vector $\mathbf{x}^{(\mathbf{0})}$. Reference [4] proves theorems to show that this holds for both diagonally dominant and symmetric positive definite matrices. The proofs of these theorems state that the

Gauss-Seidel method will converge for these matrix types, however, there is no evidence as to the rate of convergence.

Symmetric sparse matrices can be represented by graphs with elements in equations corresponding to undirected edges in the graph [6]. Ordering a symmetric sparse matrix is actually little more than changing the labels associated with nodes in an undirected graph. Modifying the ordering of a sparse matrix is simple to perform using a permutation matrix $\mathbf{P}$ of either zeros or ones that simply generates elementary row and column exchanges. Applying the permutation matrix $\mathbf{P}$ to the original linear system in equation 1 yields the linear system

$$(\mathbf{PAP}^T)(\mathbf{Px}) = (\mathbf{Pb}), \tag{4}$$

that is solved using the parallel Gauss-Seidel algorithm. While ordering the matrix greatly simplifies accessing parallelism inherent within the matrix structure, ordering can have an effect on convergence [4]. In section 9, we present empirical data to show that in spite of the ordering to yield parallelism, convergence appears to be rapid for positive definite power systems load-flow matrices.

4 Available Parallelism

While Gauss-Seidel algorithms for dense matrices are inherently sequential, it is possible to identify portions of sparse matrices that do not have mutual data dependencies, so calculations can proceed in parallel on mutually independent matrix partitions while maintaining the strict precedence rules in the Gauss-Seidel technique. All parallelism in the Gauss-Seidel algorithm is derived from within the actual interconnection relationships between elements in the matrix. Ordering sparse matrices into block-diagonal-bordered form can offer substantial opportunity for parallelism, because the values of $\mathbf{x}^{(k+1)}$ in entire sparse matrix partitions can be calculated in parallel without requiring communications. Because the sparse matrix is a single system of equations, all equations (with off-diagonal variables) are dependent. Dependencies within the linear system requires data movement from mutually independent partitions to those equations that couple the linear system. After we develop the Gauss-Seidel algorithm for a block-diagonal-bordered matrix, the optimum data/processor assignments for an efficient parallel implementation are straightforward.

While much of the parallelism in this algorithm comes from the block-diagonal-bordered ordering of the sparse matrix, further ordering of the last diagonal block is required to provide parallelism in what would otherwise be a purely sequential portion of the algorithm. The last diagonal block represents the interconnection structure within the equations that couple the partitions in the block-diagonal portion of the matrix. These equations are rather sparse, often with substantially fewer off-diagonal matrix elements (graph edges) than diagonal matrix elements (graph nodes). Consequently, it is rather simple to color the graph representing this portion of the matrix. Separate graph colors represent rows where $\mathbf{x}^{(k+1)}$ can be calculated in parallel, because within a color, no two nodes have any adjacent edges. For the parallel Gauss-Seidel algorithm, a synchronization barrier is required between colors to ensure that all new $\mathbf{x}^{(k+1)}$ values are distributed to the processors so that the strict precedence relation in the calculations are maintained.

4.1 Parallelism in Block-Diagonal-Bordered Matrices

To clearly identify the available parallelism in the block-diagonal-bordered Gauss-Seidel method, we define a block diagonal partition on the matrix, apply that partition to formula 3, and equate terms to identify available parallelism. We must also define a sub-partitioning of the last diagonal block to identify parallelism after multi-coloring.

First, we define a partitioning of the system of linear equations $(\mathbf{PAP}^T)(\mathbf{Px}) = (\mathbf{Pb})$, where the permutation matrix $\mathbf{P}$ orders the matrix into block-diagonal-bordered form.

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{0} & & \mathbf{A}_{1,m+1} \\ \mathbf{0} & \ddots & & \vdots \\ & & \mathbf{A}_{m,m} & \mathbf{A}_{m,m+1} \\ \mathbf{A}_{m+1,1} & \cdots & \mathbf{A}_{m+1,m} & \mathbf{A}_{m+1,m+1} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1^{(k)} \\ \vdots \\ \mathbf{x}_m^{(k)} \\ \mathbf{x}_{m+1}^{(k)} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_m \\ \mathbf{b}_{m+1} \end{pmatrix}. \tag{5}$$

Equation 3 divides the $\mathbf{PAP}^T$ matrix into a diagonal component $\mathbf{D}$, a strictly lower diagonal portion of the matrix $\mathbf{L}$, and a strictly upper diagonal portion of the matrix $\mathbf{U}$ such that:

$$\mathbf{PAP}^T = \mathbf{D} + \mathbf{L} + \mathbf{U} \tag{6}$$

Derivation of the block-diagonal-bordered form of the$\mathbf{D}$, $\mathbf{L}$, and $\mathbf{U}$ matrices is straightforward. Equation 3 requires the calculation of $(\mathbf{D} + \mathbf{L})^{-1}$, which also is simple to determine explicitly, because this matrix has block-diagonal-lower-bordered form. Given these partitioned matrices, it is relatively straightforward to identify available parallelism by substituting the partitioned matrices and partitioned $\mathbf{x}^{(k)}$ and $\mathbf{b}$ vectors into the definition of the Gauss-Seidel method and then performing the matrix multiplications. As a result we obtain:

$$\mathbf{x}^{(k+1)} = \begin{pmatrix} (\mathbf{D}_{1,1} + \mathbf{L}_{1,1})^{-1} \left[\mathbf{b}_1 - \mathbf{U}_{1,1}\mathbf{x}_1^{(k)} - \mathbf{U}_{1,m+1}\mathbf{x}_{m+1}^{(k)}\right] \\ \vdots \\ (\mathbf{D}_{m,m} + \mathbf{L}_{m,m})^{-1} \left[\mathbf{b}_m - \mathbf{U}_{m,m}\mathbf{x}_m^{(k)} - \mathbf{U}_{m,m+1}\mathbf{x}_{m+1}^{(k)}\right] \\ (\mathbf{D}_{m+1,m+1} + \mathbf{L}_{m+1,m+1})^{-1} \left[\mathbf{b}_{m+1} - \sum_{i=1}^{m}(\mathbf{L}_{m+1,i}^{-1}\mathbf{x}_i^{(k+1)}) - \mathbf{U}_{m+1,m+1}\mathbf{x}_{m+1}^{(k)}\right] \end{pmatrix}. \tag{7}$$

We can identify the parallelism in the block-diagonal-bordered portion of the matrix by examining equation 7. If we assign each partition i, $(i = 1, \cdots, m)$, to a separate processor the calculations of $\mathbf{x}_i^{(k+1)}$ are independent and require no communications. Note that the vector $\mathbf{x}_{m+1}^{(k)}$ is required for the calculations in each partition, and there is no violation of the strict precedence rules in the Gauss-Seidel because it is calculated in the last step. After calculating $\mathbf{x}_i^{(k+1)}$ in the first m partitions, the values of $\mathbf{x}_{m+1}^{(k+1)}$ must be calculated using the lower border and last block. From the previous step, the values of $\mathbf{x}_i^{(k+1)}$ would be available on the processors where they were calculated, so the values of $(\mathbf{L}_{m+1,i}^{-1}\mathbf{x}_i^{(k+1)})$ can be readily calculated in parallel. Only (matrix $\times$ vector) products, calculated in parallel, are involved in the communications phase. Furthermore, if we assign

$$\hat{\mathbf{b}} = \mathbf{b}_{m+1} - \sum_{i=1}^{m} \left(\mathbf{L}_{m+1,i}^{-1}\mathbf{x}_i^{(k+1)}\right), \tag{8}$$

then the formulation of $\mathbf{x}_{m+1}^{(k+1)}$ looks similar to equation 3:

$$\hat{\mathbf{x}}^{(k+1)} = \mathbf{x}_{m+1}^{(k+1)} = \left(\mathbf{D}_{m+1,m+1} + \mathbf{L}_{m+1,m+1}\right)^{-1} \left[\hat{\mathbf{b}} - \mathbf{U}_{m+1,m+1}\mathbf{x}^{(k)}\right]. \tag{9}$$

4.2 Parallelism in Multi-Colored Matrices

The ordering imposed by the permutation matrix $\mathbf{P}$, includes multi-coloring-based ordering of the last diagonal block that produces sub-partitions with parallelism, We define the sub-partitioning as:

$$\mathbf{A}_{m+1,m+1} = \begin{pmatrix} \hat{\mathbf{D}}_{1,1} & \hat{\mathbf{A}}_{1,2} & \cdots & \hat{\mathbf{A}}_{1,c} \\ \hat{\mathbf{A}}_{2,1} & \hat{\mathbf{D}}_{2,2} & \cdots & \hat{\mathbf{A}}_{2,c} \\ \vdots & & \ddots & \vdots \\ \hat{\mathbf{A}}_{c,1} & \hat{\mathbf{A}}_{c,2} & \cdots & \hat{\mathbf{D}}_{c,c} \end{pmatrix}. \tag{10}$$

where $\hat{\mathbf{D}}_{i,i}$ are diagonal blocks and c is the number of colors. After forming $\mathbf{L}_{m+1,m+1}$ and $\mathbf{U}_{m+1,m+1}$, it is straight forward to prove that:

$$\hat{\mathbf{x}}^{(k+1)} = \begin{pmatrix} \hat{\mathbf{x}}_1^{(k+1)} \\ \hat{\mathbf{x}}_2^{(k+1)} \\ \vdots \\ \hat{\mathbf{x}}_c^{(k+1)} \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{D}}_{1,1}^{-1}\left[\hat{\mathbf{b}}_1 - \sum_{j>1}\hat{\mathbf{A}}_{1,j}\hat{\mathbf{x}}_j^{(k)}\right] \\ \hat{\mathbf{D}}_{2,2}^{-1}\left[\hat{\mathbf{b}}_2 - \sum_{j<2}\hat{\mathbf{A}}_{2,j}\hat{\mathbf{x}}_j^{(k+1)} - \sum_{j>2}\hat{\mathbf{A}}_{2,j}\hat{\mathbf{x}}_j^{(k)}\right] \\ \vdots \\ \hat{\mathbf{D}}_{c,c}^{-1}\left[\hat{\mathbf{b}}_c - \sum_{j<c}\hat{\mathbf{A}}_{c,j}\hat{\mathbf{x}}_j^{(k+1)}\right] \end{pmatrix}. \tag{11}$$

Calculating $\hat{\mathbf{x}}_i^{(k+1)}$ in each sub-partition of $\hat{\mathbf{x}}^{(k+1)}$ does not require values of $\hat{\mathbf{x}}_i^{(k+1)}$ within the sub-partition, so we can calculate the individual values within $\hat{\mathbf{x}}_i^{(k+1)}$ in any order and distribute these calculations to separate processors without concern for precedence. In order to maintain the strict precedence in the Gauss-Seidel algorithm, the values of $\hat{\mathbf{x}}_i^{k+1}$ calculated in each step must be broadcast to all processors, and processing cannot proceed for any processor until it receives the new values of $\hat{\mathbf{x}}_i^{(k+1)}$ from all other processors.

If the block-diagonal-bordered matrix partitions $\mathbf{A}_{i,i}$, $\mathbf{A}_{m+1,i}$, and $\mathbf{A}_{i,m+1}$ $(1 \leq i \leq m)$ are assigned to the same processor, then there are *no* communications until $\mathbf{x}_{m+1}^{(k+1)}$ is calculated. At that time, only (matrix $\times$ vector) products are sent to the processors that hold the appropriate data in the last diagonal block. This processor/data assignment to processors is defined by multi-coloring only the last diagonal block.

Figure 2 describes the calculation steps in the parallel Gauss-Seidel for a block-diagonal-bordered sparse matrix. This figure depicts four diagonal blocks, and data/processor assignments (P1, P2, P3, and P4) are listed for the data block. Figure 3 illustrates the data/processor assignments in the last diagonal block.

5 The Preprocessing Phase

In the previous section, we developed the theoretical foundations of parallel Gauss-Seidel methods with block-diagonal-bordered sparse matrices, and now we will discuss the procedures required

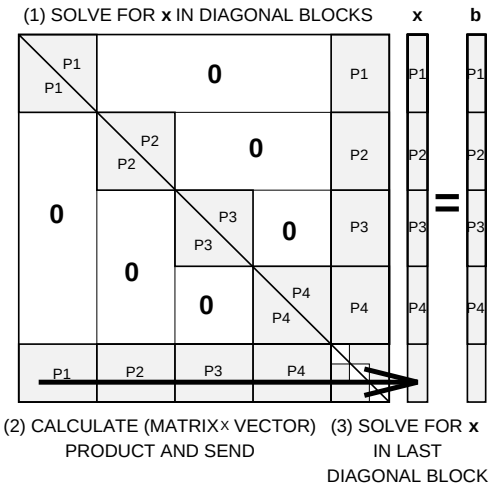

Figure 2: Block-Bordered-Diagonal Form Gauss-Seidel Method

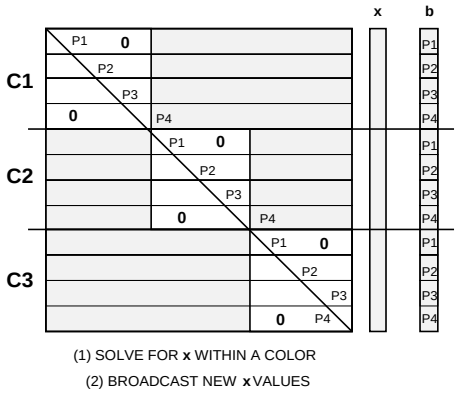

Figure 3: Multi-Colored Gauss-Seidel Method for the Last Diagonal Block

to generate the permutation matrices, $\mathbf{P}$, to produce block-diagonal-bordered/multi-colored sparse matrices so that our parallel Gauss-Seidel algorithm is efficient. We must reiterate that all parallelism for our Gauss-Seidel algorithm is identified from the interconnection structure of elements in the sparse matrix during this preprocessing phase. We must order the sparse matrix in such a manner that processor loads are balanced. The technique we have chosen for this preprocessing phase is to:

1. *order* the matrix into block-diagonal-bordered form while minimizing the size of the last diagonal block,

2. *order* the last diagonal block using multi-coloring techniques.

Inherent in both preprocessing steps is explicit load-balancing to determine processor/data mappings for efficient implementation of the Gauss-Seidel algorithm.

This preprocessing phase incurs significantly more overhead than solving a single instance of the sparse matrix; 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 solutions.

5.1 Ordering the Matrix into Block-Bordered-Diagonal Form

We require a technique that orders irregular matrices into block-diagonal-bordered form while limiting the number of coupling equations. Minimizing the number of coupling equations minimizes the size of the last diagonal block in a block-diagonal-bordered sparse matrix, and minimizes the amount of broadcast communications required when calculating values of $\mathbf{x}^{(k+1)}$ in the last diagonal block. The effects of minimizing the size of the last diagonal block are not all positive. We have found that minimizing the size of the last block can affect potential parallelism if the resulting workload for calculating $\mathbf{x}^{(k+1)}$ in the diagonal blocks cannot be distributed uniformly throughout a multi-processor — in which case there is load imbalance between multi-processors [9]. When determining the optimal ordering for a sparse matrix, the size of the last diagonal block and the subsequent additional communications may be traded for an ordering that yields good load balance in the highly parallel portion of the calculations, especially when using larger numbers of processors.

The method we have chosen to order a sparse matrix into block-diagonal-bordered form is referred to as node-tearing [13], which is a specialized form of diakoptics [5]. We have selected node-tearing nodal analysis because this algorithm determines the natural structure in the matrix while providing the means to minimize the number of coupling equations. With the node-tearing algorithm, we can determine the hierarchical structure in a power system distribution grid solely from the interconnection relationships in the sparse matrices. Tearing here refers to breaking the original problem into smaller sub-problems whose partial solutions can be combined to give the solution of the original problem. Load balancing techniques must be used after the node tearing matrix ordering step to uniformly distribute the processing load onto a multi-processor.

The node-tearing-based ordering algorithm has the ability to adjust the characteristics of the ordering by varying an input parameter. Empirical data is presented later in section 9 for multiple orderings to illustrate the parallel linear solver algorithm performance as a function of input parameters to the node-tearing algorithm.

Load balancing for node-tearing-based ordering can be performed with a simple pigeon-hole type algorithm that uses a metric based on the number of floating point multiply/add operations in a partition, instead of simply using the number of rows per partition. Load balancing examines the number of operations when calculating $\mathbf{x}^{(k+1)}$ in the matrix partitions and the number of operations when calculating the sparse matrix vector products in preparation to solve for $\mathbf{x}^{(k+1)}$ in the last diagonal block. These metrics do not consider indexing overhead, which can be rather extensive when working with very sparse matrices stored in an implicit form. This algorithm finds an optimal distribution for workload to processors, however, actual disparity in processor workload is dependent on the irregular sparse matrix structure. This algorithm works best when there are minimal disparities in the workloads for independent blocks or when there are significantly more independent blocks than processors. In this instance, the workloads in multiple small blocks can sum to equal the workload in a single block with more computational workload.

5.2 Ordering the Last Diagonal Block

The application of diakoptic techniques yields a block-diagonal-bordered matrix form that identifies the basic network structure and provides parallelism for the majority of calculations within a Gauss-Seidel iteration. However, without additional ordering, the last diagonal block would be purely sequential, limiting the potential speedup of the algorithm in accordance with Amdahl's law. The last diagonal block represents the interconnection structure within the equations that couple the partitions found in the previous step. In other words, the variables in the last-diagonal block are the interconnections within the equations that tie the entire matrix together. Graph multi-coloring has been used for ordering this portion of the matrix — all nodes of the same color share no interconnections, consequently, the values of $\mathbf{x}^{(k+1)}$ in these rows can be calculated in any order without violating the strict precedence rules in the Gauss-Seidel method. As a result, rows within a color can be solved in parallel.

The multi-coloring algorithm we selected for this work is based on the saturation degree ordering algorithm. We also require load balancing, a feature not commonly implemented within graph multi-coloring. As part of our implementation we added a feature that equalizes the number of rows per color to provide some basic load balancing. The graph multi-coloring technique is discussed in greater detail in section 7.

6 Node-tearing Nodal Analysis

A detailed theoretical derivation of node-tearing is too lengthy to describe here in rigorous mathematical terms. We refer interested readers to references [10, 13] for proofs of the mathematics, although a brief description of node-tearing follows.

Let the set $\mathcal{N}$ denote the nodes of a graph $\mathcal{G}$ and let $\mathcal{E}$ denote the edges in $\mathcal{G}$, or $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. In summary, node-tearing is a greedy algorithm that partitions the nodes $\mathcal{N}$ in $\mathcal{G}$ into

$$\begin{aligned} \mathcal{N}_1 &\equiv \cup_{i=1}^{m} \mathcal{N}_1^i \\ \mathcal{N}_2 &\equiv \mathcal{N} - \mathcal{N}_1 \end{aligned} \tag{12}$$

where:

$\mathcal{N}_1$ is the set of nodes in the mutually independent partitions

$\mathcal{N}_1^i$ is the set of nodes in a mutually independent partition

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

Mutual independence occurs when no edges in $\mathcal{E}_1^i$ are connected to edges in $\mathcal{E}_1^j$ $\forall\, i \neq j$ and $i, j = 1, 2, \ldots, m$. Consequently, $\mathcal{G}_1^i$ has no edges in common with $\mathcal{G}_1^j$, $\forall i \neq j$, and there are no edges directly interconnecting any nodes in $\mathcal{N}_1^i$ and $\mathcal{N}_1^j$, $\forall i \neq j$. Connectivity between $\mathcal{G}_1^i$ and $\mathcal{G}_1^j$, $\forall i \neq j$, is indirect and must go through nodes in $\mathcal{N}_2$.

In addition to ordering matrices into block-diagonal-bordered form using node-tearing, we require that the number of coupling equations, $\mid \mathcal{N}_2 \mid$, is minimized over all distinct partitions $\{\mathcal{N}_1, \mathcal{N}_2\}$ of $\mathcal{G}$ while also specifying that $|\mathcal{N}_1^k| \leq max_{DB}$, $k = 1, 2, \ldots, m$. This constraint permits some control of the maximum size of diagonal blocks, max_{DB}, which can prove quite useful when tearing a graph for solving on multi-processors. By modifying this parameter, control can be exercised over the shape of the ordered sparse matrix — yielding small blocks when max_{DB} is small and limiting the size of the borders in a block-diagonal-bordered matrix when max_{DB} is large. This optimization problem belongs to the family of NP-complete problems [13], so a simple, efficient heuristic algorithm has been developed based on examining the contour of the graph [13]. A contour-tableau contains three lists:

1. the iterating sets $\mathcal{I}_i^k$ or the potential elements of a set of nodes in the sub-graph $\mathcal{N}_i^k$,
2. the adjacency sets $\mathcal{A}_i^k$ or the set of nodes adjacent to, but not including any elements in the corresponding iterating set,
3. contour numbers c_i^k or the cardinality of the adjacency set.

As we perform node-tearing, we want to minimize the size of the adjacency set, $\left|\mathcal{A}_i^k\right|$, for each partition and subsequently this will minimize $|\mathcal{N}_2|$. A separate contour-tableau is developed for each diagonal block.

The software implementation to perform node-tearing nodal analysis utilizes the basic concept of building a contour tableau to identify independent sub-matrices and the coupling equations in an undirected graph representing a sparse matrix. In our implementation, the search for the local minimum of the contour number is limited to within the range $(\alpha \times max_{DB}) \leq i \leq max_{DB}$, $0 < \alpha < 1$. When an independent sub-matrix is found, this iterating set is moved into a set $\mathcal{N}_1^k$, where $\mid \mathcal{N}_1^k \mid = i$. Figure 4 illustrates the major steps in the node-tearing ordering algorithm. The algorithm examines all nodes essentially once, where the size of the independent sub-blocks are limited to max_{DB}. The computational complexity of this algorithm is $O(\max_i |\mathcal{A}_i^k| \times n)$, due to the fact that all nodes in the graph must be examined, and for each element in the contour tableau — all elements of the adjacency set must be examined for the next node. The value of $\max_i |\mathcal{A}_i^k|$ must be less than n, and because the graphs will be sparse, the maximum number in the adjacency set will be substantially less than n.

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

Figure 4: The Node-Tearing Algorithm

$\hat{\mathcal{N}} \leftarrow \mathcal{N}_2$ *(the nodes in the sparse last diagonal block)*
while $\hat{\mathcal{N}} \neq \phi$ **do**
    select a node ν *from* $\hat{\mathcal{N}}$ *such that* ν
        has the largest number of neighbors with different colors
    $\sigma(\nu) \leftarrow$ *the consistent color with the fewest occurrences*
    $\hat{\mathcal{N}} \leftarrow \hat{\mathcal{N}} - \nu$
end while

Figure 5: The Graph Multi-Coloring Algorithm

7 Graph Coloring

Multi-coloring a graph G is an NP-complete problem that attempts to define a minimum number of colors for the nodes of a graph where no adjacent nodes are assigned the same color [8, 11]. A greedy heuristic can yield an optimal ordering if the vertices are visited in the correct order. We selected the saturation degree ordering algorithm [8], but modified it to include load-balancing. The saturation degree ordering algorithm selects a node in the graph that has the largest number of differently colored neighbors. We have added the capability to the saturation degree ordering algorithm to select the color for a node in a manner that equalizes the number of nodes with a particular color. We simply select the consistent color with the fewest number of nodes.

We present our version of the saturation degree ordering-based graph multi-coloring algorithm in figure 5. The computational complexity of this algorithm is $O(\max_{\nu \in \hat{\mathcal{N}}} |\Lambda_{\hat{\mathcal{G}}}(\nu)| \times \hat{n})$, where $\Lambda_{\hat{\mathcal{G}}}(\nu)$ defines the set of nodes in $\hat{\mathcal{G}}$ adjacent to ν. The graphs encountered for coloring in this work were very sparse, generally with no more than three nodes adjacent to any single node.

8 Parallel Gauss-Seidel Implementation

We have implemented a parallel version of a block-diagonal-bordered sparse Gauss-Seidel algorithm in the C programming language for the Thinking Machines CM-5 multi-computer using the Connection Machine active message layer (CMAML) remote procedure call as the basis for interprocessor communications [14]. Significant improvements in the performance of the algorithm were observed for active messages, when compared to more traditional communications paradigms that use the standard blocking `CMMD_send` and `CMMD_receive` functions in conjunction with packing data into communications buffers. A significant portion of the communications require each processor to send short data buffers to every other processor, imposing significant communications overhead due to latency. To significantly reduce communications overhead and attempt to hide communications behind calculations, we implemented each portion of the algorithm using CMAML remote procedure calls (`CMAML_rpc`). The communications paradigm we use throughout this algorithm is to send a double precision data value to the destination processor as soon as the value is calculated. Communications in the algorithm occur at distinct time phases, making polling for the active message handler function efficient. An active message on the CM-5 has a four word payload, which is more than adequate to send a double precision floating point value and an integer position indicator. The use of active messages greatly simplified the development and implementation of this parallel sparse Gauss-Seidel algorithm, because there was no requirement to maintain and pack communications buffers.

This implementation uses implicit data structures based on vectors of C programming language structures to store and retrieve data efficiently within the sparse matrix. These data structures provide good cache coherence, because non-zero data values and column location indicators are stored in adjacent physical memory locations. The data structure is composed of six separate parts that implicitly store the block-diagonal-bordered sparse matrix and the last block. Figure 6 graphically illustrates the relationships within the data structure. As illustrated in the figure, the block-diagonal structure, the border-row structure, and the last-block-diagonal structure contain pointers to the sparse row vectors. The second values in the two diagonal pointers are the values of a_{ii}, while the second value in the border-row structure is the destination processor for the (vector $\times$ vector) product from this border row used in calculating values in the last diagonal block.

Our parallel Gauss-Seidel algorithm has the following distinct sections where blocks are defined in section 4:

1. solve for $\mathbf{x}^{(k+1)}$ in the diagonal blocks
2. calculate $\hat{\mathbf{b}} = \mathbf{b}_{m+1} - \sum_{i=1}^{m} \left(\mathbf{L}_{m+1,i}^{-1} \mathbf{x}_i^{(k+1)} \right)$ by forming the (matrix $\times$ vector) products in parallel
3. solve for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block

A pseudo-code representation of the parallel Gauss-Seidel solver is presented in figure 7. A version of the software is available that runs on a single processor on the CM-5 to provide empirical speed-up data to quantify multi-processor performance. This sequential software includes the capability to gather convergence-rate data. The parallel implementation has been developed as an instrumented

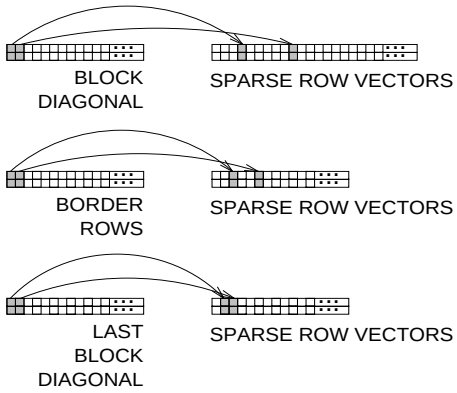

Figure 6: The Data Structure

proof-of-concept to examine the efficiency of each section of the code described above. The host processor is used to gather and tabulate statistics on the multi-processor calculations. Statistics are gathered at synchronization points, so there is no impact on total empirical measures of performance. Empirical performance data is presented in the next section for varied numbers of processors solving real power systems sparse load-flow matrices.

9 Empirical Results

Overall performance of our parallel Gauss-Seidel linear solver is dependent on both the performance of the matrix ordering in the preprocessing phase and the performance of the parallel Gauss-Seidel implementation. Because these two components of the parallel Gauss-Seidel implementation are inextricably related, the best way to assess the potential of this technique is to measure the speedup performance using real power system load-flow matrices. We first present speedup and efficiency data for three separate power systems matrices:

- Boeing-Harwell matrix BCSPWR09 — 1,723 nodes and 2,394 edges in the graph [1]
- Boeing-Harwell matrix BCSPWR10 — 5,300 nodes and 8,271 edges in the graph [1]
- EPRI matrix EPRI-6K matrix — 4,180 nodes and 5,226 edges in the graph [2]

Matrices BCSPWR09 and BCSPWR10 are from the Boeing Harwell series and represent electrical power system networks from the Western and Eastern US respectively. The EPRI-6K matrix is distributed with the Extended Transient-Midterm Stability Program (ETMSP) from EPRI. These matrices were preprocessed using a sequential program that ordered the matrix, load balanced each ordering step, and subsequently produced the implicit data structures required for the parallel block-diagonal-bordered Gauss-Seidel linear solver. Due to the static nature of the power system grid, such an ordering would be reused over many hours of calculations in real electrical power utility operations load-flow applications.

Matrix preprocessing was performed for multiple values of max_{DB}, the input value to the node-tearing algorithm. Empirical performance data was collected for each of the aforementioned power

Node Program
$\epsilon \leftarrow \infty$
while $\epsilon > \epsilon_{converge}$
    for $k = 1$ **to** n_{iter}
/* solve for $\mathbf{x}^{(k+1)}$ in the diagonal blocks */
        for all rows i **in blocks assigned to this processor**
            $\tilde{x}_i \leftarrow x_i$
            $x_i \leftarrow b_i$
            for each $j \in [1, n]$ such that $a_{ij} \neq 0$
                $x_i \leftarrow x_i - (a_{ij} * x_j)$
            endfor
            $x_i \leftarrow x_i / a_{ii}$
        endfor
/* calculate the (matrix $\times$ vector) products in the lower border */
        for all rows i **in the last block assigned to this processor**
            $\tilde{x}_i \leftarrow x_i$
            $x_i \leftarrow b_i$
        endfor
        for all non-zero rows i **in the lower border of this block**
            for each j such that $a_{ij} \neq 0$
                $\sigma \leftarrow \sigma - (a_{ij} * x_j)$
            endfor
            at processor $\rho_i \Longrightarrow x_i \leftarrow x_i - \sigma$ *using active message rpc*
        endfor
/* solve for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block */
        for all colors c
            for all rows i **in color** c **assigned to this processor**
                for each $j \in [1, n]$ such that $a_{ij} \neq 0$
                    $x_i \leftarrow x_i - (a_{ij} * x_j)$
                endfor
                $x_i \leftarrow x_i / a_{ii}$
                broadcast x_i *using active message rpc*
            endfor
        wait until all values of x_i **have arrived**
        endfor
    endfor
/* check convergence */
    $\epsilon_\rho \leftarrow 0$
    for all rows i **assigned to this processor**
        $\epsilon_\rho \leftarrow \epsilon_\rho + abs(\tilde{x}_i - x_i)$
    endfor
    $\epsilon \leftarrow \sum_{\forall \rho} \epsilon_\rho$ *using active message rpc*
endwhile

Figure 7: Parallel Gauss-Seidel

systems matrices using 1 through 32 processors on the Thinking Machines CM-5 at the Northeast Parallel Architectures Center at Syracuse University. The NPAC CM-5 is configured with all 32 nodes in a single partition, so user software was required to define the number of processors used to actually solve a linear system. Empirical data collected on the parallel Gauss-Seidel algorithm will be presented in two ways. We first present speedup and efficiency data from the three power systems matrices. Relative speedup and efficiency are presented using the times required to perform four iterations and a single convergence check. Next, we provide a detailed performance analysis using actual run times for the individual subsections of the parallel Gauss-Seidel linear solver. This detailed performance analysis illustrates the efficacy of the load balancing step in the preprocessing phase, and illustrates other performance bottlenecks.

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

$$S_p \equiv \frac{T_1}{T_p}, \tag{13}$$

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

Definition — Relative Efficiency *Relative efficiency is defined as*

$$E_p \equiv \frac{S_p}{p}, \tag{14}$$

where S_p is relative speedup and p is the number of processors.

9.1 Performance Analysis

As an introduction to the performance of the parallel Gauss-Seidel algorithm, we present a pair of graphs that plot relative speedup and relative efficiency versus the number of processors. Figure 8 plots the best speedup and efficiency measured for each of the power systems matrices for 2, 4, 8, 16, and 32 processors. These graphs show that performance for the EPRI-6K data set is the best of the three data sets examined. Speedup reaches a maximum of 11.6 for 32 processors and speedups of greater than 10.0 were measured for 16 processors. This yields a relative efficiency of 63% for 16 processors and 36% for 32 processors.

Relative speedups for the BCSPWR09 and BCSPWR10 matrices are less than for the EPRI-6K matrix, but each has speedup in excess of 7.0 for 16 processors. The reason for reduced performance with these matrices for larger numbers of processors is the size of the last block after ordering. For both the BCSPWR09 and BCSPWR10 matrices, the last diagonal block requires approximately 5% of the total calculations while the last block of the EPRI-6K matrix can be ordered so that only 1% of all calculations occur there. As the number of processors increases, communications overhead becomes a significant part of the overall processing time because $\mathbf{x}^{(k+1)}$ values in the last diagonal block must be broadcast to other processors before processing can proceed to the next color. Even though the second ordering phase is able to three-color the last diagonal blocks, communications overwhelms the processing time for larger numbers of processors and minimizes speedup in this portion of the calculations. There are insufficient parallel operations when solving for $\mathbf{x}^{(k+1)}$ in the

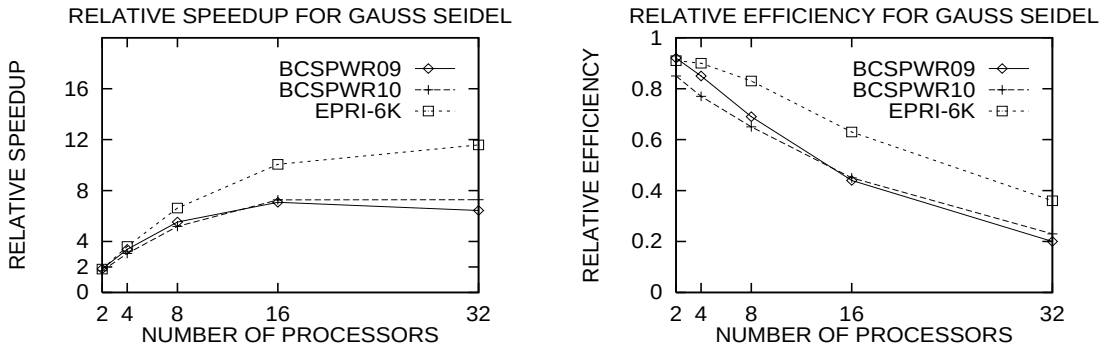

Figure 8: Relative Speedup and Efficiency — 2, 4, 8, 16, and 32 processors

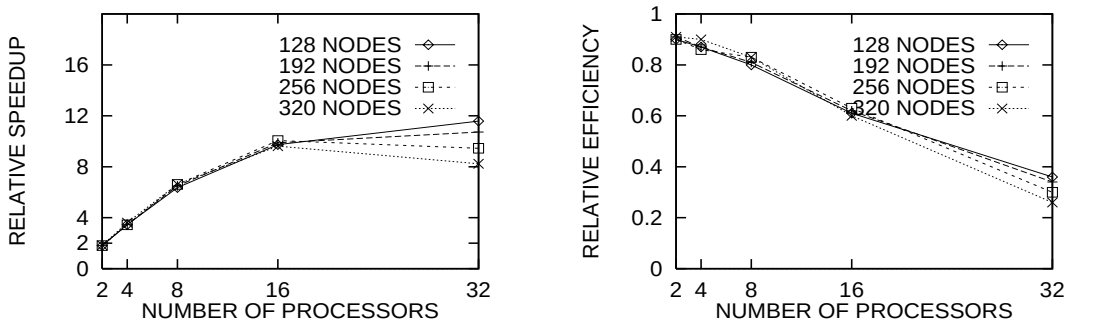

Figure 9: Relative Speedup and Efficiency for EPRI-6K Data — 2, 4, 8, 16, and 32 processors

diagonal blocks for these matrices to offset the effect of the nearly sequential last block. The effect of Amdahl's law is visible for larger numbers of processors due to the sequential nature of one portion of the algorithm.

The BCSPWR09 matrix encounters an additional problem in that the ordering phase was unable to effectively balance the workload in the portion of the software that processes all but the last block. This matrix is the smallest examined, and there is insufficient available parallelism in the matrix to support 16 or more processors.

A detailed examination of relative speedup and relative efficiency is presented in figure 9 for the EPRI-6K data. This figure contains two graphs that each have a family of four curves plotting relative speedup and relative efficiency for each of four maximum matrix partition sizes used in the node-tearing algorithm. The maximum partition sizes used when preprocessing this data are 128, 192, 256, and 320 nodes. The family of speedup curves for the various matrix orderings clearly illustrates the effects of load imbalance for some matrix orderings. For all four matrix orderings, speedup is nearly equal for 2 through 16 processors. However, the values for relative speedup diverge for 32 processors.

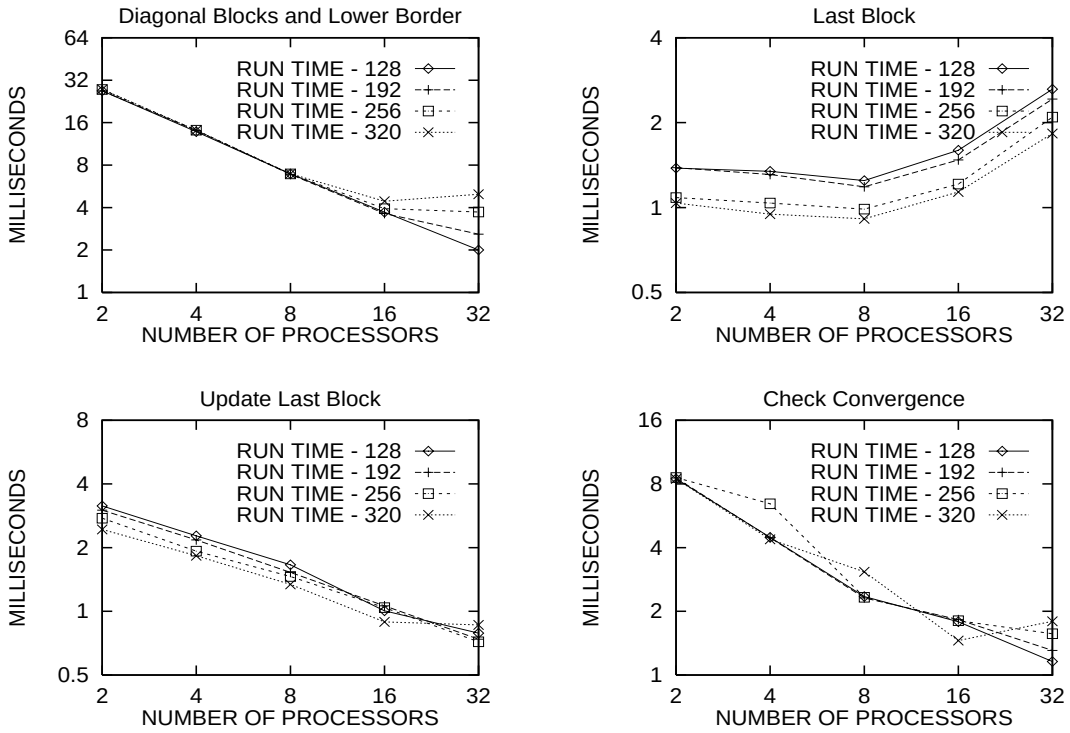

Figure 10: Timings for Algorithm Components — EPRI-6K Data — 2, 4, 8, 16, and 32 processors

We can look further into the cause of the disparity in the relative speedup values in the EPRI-6K data by examining the performance of each of the four distinct sections of the parallel algorithm. Figure 10 contains four graphs that each have a family of four curves that plot the processing time in milliseconds versus the number of processors for each of four values of max_{DB} from the node-tearing algorithm. The values of max_{DB} used when preprocessing this data are 128, 192, 256, and 320 nodes. These graphs are log-log scaled, so for perfect speedup, processing times should fall on a straight line with decreasing slope for repeated doubling of the number of processors. One or more curves on each of the performance graphs for the diagonal blocks and lower border, for updating the last diagonal block, and for convergence checks illustrate nearly perfect speedup with as many as 32 processors. Unfortunately the performance for calculating values of $\mathbf{x}^{(k+1)}$ in the last block does not also have stellar parallel performance.

The performance graph for the diagonal blocks and lower border clearly illustrates the causes for the load imbalance observed in the relative speedup graph in figure 9. For some matrix orderings, load balancing is not able to divide the work evenly for larger numbers of processors. This always occurs for larger values of max_{DB}, the maximum size of a block when ordering a matrix. Nevertheless, when ordering a matrix for sixteen or more processors, selecting small values of max_{DB} will provide good speedup for larger numbers of processors. The performance curves presented in figure 8, shows the best performance observed for the four matrix orderings.

Performance of updating the last block by performing sparse (matrix $\times$ vector) products and then

performing irregular communications yields good performance even for 32 processors. The times to perform updates is correlated to the size of the last diagonal block, which is inversely related to the magnitude of max_{DB}. The relationship between the magnitude of max_{DB} and the size of the last block is intuitive, because as the magnitude of max_{DB} increases, multiple smaller blocks can be incorporated into a single block. Not only can two smaller blocks be consolidated into the single block, but in addition, any elements in the coupling equations that are unique to those network partitions could also be moved into the larger block.

The performance graph for convergence checking illustrates that the load balancing step does not assign equal numbers of rows to all processors. The number of rows on a processor varies as a function of the load balancing. While the family of curves on this graph are more erratic than the curves representing performance in diagonal blocks and the lower border and the performance of updating the last diagonal block, performance generally is improving with near perfect parallelism even for 32 processors.

Information on the relative performance as a function of max_{DB} and the number of processors would be required when implementing this parallel algorithm in a load-flow analysis application. To minimize the effects of data movement, the application would require that the entire process to calculate the Jacobian when solving the systems of non-linear equations consider the processor/data assignments from the sparse linear solver. The time to solve each instance of the linear equations generated by the Jacobian is so small that all data redistribution must be eliminated, otherwise, the benefits observed from parallel processing speedup in an application will be lost.

Performance of this parallel Gauss-Seidel linear solver is dependent on the performance of the matrix preprocessing phase. We must reiterate that all available parallelism in this work is a result of ordering the matrix and identifying relationships in the connectivity pattern within the structure of the matrix. Power systems load flow matrices are some of the most sparse irregular matrices encountered. For the EPRI-6K data, the mode in a histogram of the number of edges per node is only two! In other words, the most frequent number of edges at a node is only two. 84.4% of the nodes in the EPRI-6K data have three or less edges. For the BCSPWR10 matrix, 71% of the nodes have three or less edges. Consequently, power systems matrices pose some of the greatest challenges to produce efficient parallel sparse matrix algorithms.

In figures 11 and 12, we present two orderings of the EPRI-6K data with max_{DB} equal to 128 and 256 nodes respectively. Non-zero entries in the matrix are represented as dots, and the matrices are delimited by a bounding box. Each of these figures contain three sub-figures: the ordered sparse matrix and two enlargements of the last block — before and after multi-coloring. Both matrices have been partitioned into block-diagonal-bordered form and load-balanced for eight processors. The numbers of nodes in the last diagonal blocks are 153 and 120 respectively, while the numbers of edges in this part of the matrix are 34 and 22 respectively. The graph multi-coloring algorithm is able to color these portions of the matrices with three and two colors respectively. These matrices represent the adjacency structure of the network graphs, and clearly illustrate the sparsity in these matrices.

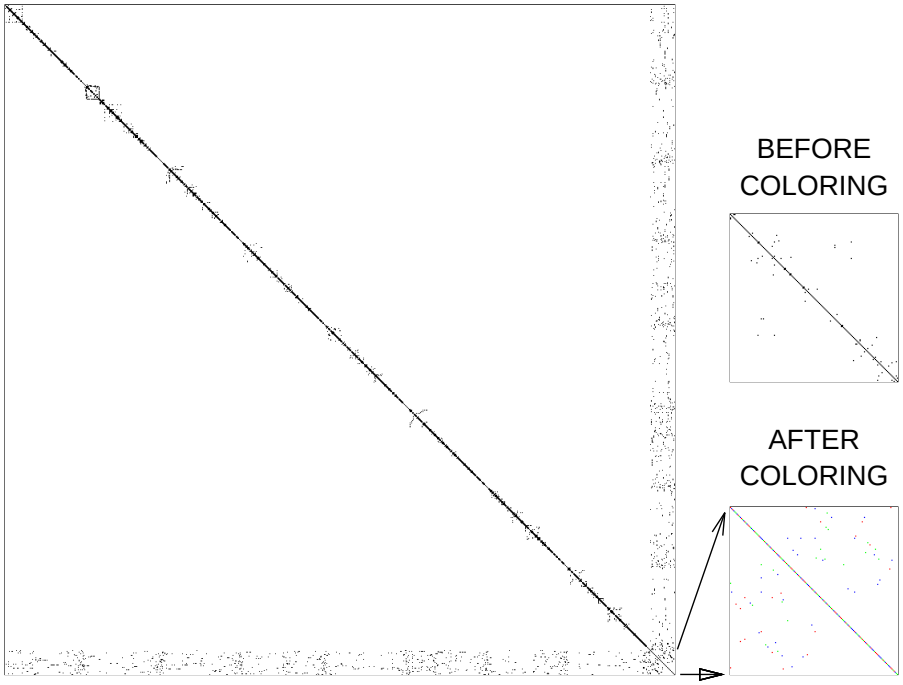

Figure 11: Ordered EPRI-6K Matrix — $max_{DB} = 128$

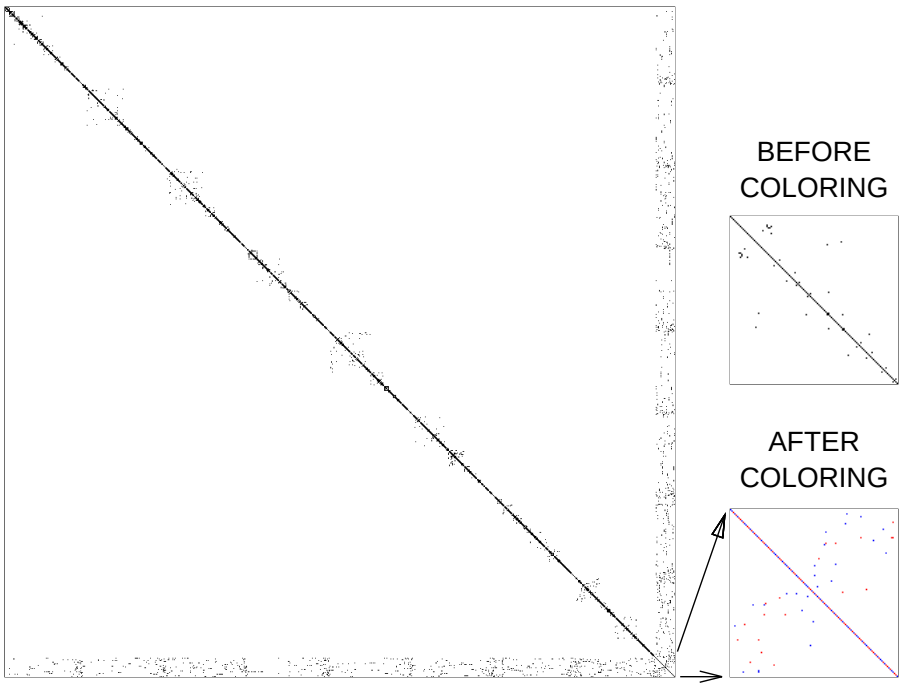

Figure 12: Ordered EPRI-6K Matrix — $max_{DB} = 256$

Iteration	**Total Error** $\sum_{\forall i} abs(x_i^{(k+1)} - x_i^{(k)})$	$\min_{\forall i} x_i^{(k+1)}$	$\max_{\forall i} x_i^{(k+1)}$
1	0.983654764216	0.000000119293	0.000478248320
2	0.000143661684	0.000000124521	0.000478321438
3	0.000000018556	0.000000124522	0.000478321442
4	0.000000000002	0.000000124522	0.000478321442
5	0.000000000000	0.000000124522	0.000478321442

Table 1: Convergence for EPRI-6K Data — $max_{DB} = 256$

9.2 Convergence Rate

Critical to the performance of an iterative linear solver is the convergence of the technique for a given data set. We have applied our solver to sample positive definite matrices that have actual power networks as the basis for the sparsity pattern, and random values for the entries. We have examined convergence for various matrices and various matrix orderings. A sample of the measured convergence data is presented in table 1. This table presents the total error for an iteration, and the minimum and maximum values encountered that iteration. All initial values, $\mathbf{x}^{(0)}$, have been defined to equal 0.0. Convergence is rather rapid, and after four iterations, total error equals 2×10^{-12}. Consequently, only a few iterations are required for reasonable convergence with this procedure on this data. We hypothesize that this good convergence rate is in part due to having good estimates of the initial starting vector. For actual solutions of power systems load flows, this solver would be used within an iterative non-linear solver, so good estimates of starting points for each solution also will be readily available.

9.3 Comparing Communications Paradigms

Underlying the whole concept of active messages is the paradigm that the user takes the responsibility for handling messages as they arrive at a destination. The user writes a handler function that takes the data from a register and uses it in a calculation or assigns the data to memory. By assigning message handling responsibilities to the user, communications overhead can be significantly reduced. The effect of reduced overhead can be clearly seen in this algorithm, when performance of an active message-based algorithm is compared to performance of an algorithm with more common blocking send and receive commands. The requirement in this algorithm to broadcast the values of $\mathbf{x}^{(k+1)}$ before the next color can proceed causes substantial amounts of communications. In the portion of the algorithm that solves for values of $\mathbf{x}^{(k+1)}$ in the last diagonal block, the amount of communications is $O(n_{procs}^2)$, and as the number of processors increases, the size of the messages for conventional message passing decreases. For traditional message passing paradigms, the cost for communications increases drastically as the number of processors increases, because each message incurs the same latency regardless of the amount of data sent. Meanwhile, with active messages, latency is greatly reduced because the user has the responsibility to process the message. This increase in the number of messages can be seen in figure 10, as the performance for solving for

values in the last block eventually increases slightly as the number of processors increases. For an algorithm based on a more traditional send and receive paradigm, performance quickly becomes unacceptable in this portion of the calculations as the number of processors increases.

10 Conclusions

We have developed a parallel sparse Gauss-Seidel solver with the potential for good relative speedup and relative efficiencies for the very sparse, irregular matrices encountered in electrical power system applications. Block-diagonal-bordered matrix structure offers promise for simplified implementation and also offers a simple decomposition of the problem into clearly identifiable subproblems. The node-tearing ordering heuristic has proven to be successful in identifying the hierarchical structure in the power systems matrices, and reducing the number of coupling equations so that the graph multi-coloring algorithm can usually color the last block with only two or three colors. All available parallelism in our Gauss-Seidel algorithm is derived from within the actual interconnection relationships between elements in the matrix, and identified in the sparse matrix orderings. Consequently, available parallelism is not unlimited. Relative speedup tends to increase nicely until either load-balance overhead or communications overhead cause speedup to level off.

We have shown that, depending on the matrix, relative efficiency declines rapidly after 8 or 16 processors, limiting the utility of applying large numbers of processors to a single parallel linear solver. Nevertheless, other dimensions exist in electrical power system applications that can be exploited to use large numbers of processors efficiently. While a moderate number of processors can be efficiently applied to a single power system simulation, multiple events can be simulated simultaneously.

Acknowledgments

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

References

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

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

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

[4] G. Golub and J. M. Ortega. *Scientific Computing with an Introduction to Parallel Computing.* Academic Press, Boston, MA., 1993.

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

[6] 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.

[7] G. Huang and W. Ongsakul. Managing the Bottlenecks in Parallel Gauss-Seidel Type Algorithms for Power Flow Analysis. *Proceedings of the 18th Power Industry Computer Applications (PICA) Conference*, pages 74–81, May 1993.

[8] M. T. Jones and P. E. Plassman. A Parallel Graph Coloring Heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–69, May 1993.

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

[10] D. P. Koester, S. Ranka, and G. C. Fox. Parallel Choleski Factorization of Block-Diagonal-Bordered Sparse Matrices. Technical Report SCCS-604, Northeast Parallel Architectures Center (NPAC), Syracuse University, Syracuse, NY 13244-4100, January 1994.

[11] D. W. Matula, G. Marble, and J. D. Isaacson. *Graph Coloring Algorithms.* Acedemic Press, Mew York, 1972.

[12] 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.

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

[14] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, 1993. Version 3.0.

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