PARALLEL BLOCK-DIAGONAL-BORDERED SPARSE LINEAR SOLVERS FOR POWER SYSTEMS APPLICATIONS

by

DAVID P. KOESTER

ABSTRACT OF DISSERTATION

October, 1995

This thesis presents research into parallel linear solvers for block-diagonal-bordered sparse matrices. The block-diagonal-bordered form identifies parallelism that can be exploited for both direct and iterative linear solvers. We have developed efficient parallel block-diagonal-bordered sparse direct methods based on both LU factorization and Choleski factorization algorithms, and we have also developed a parallel block-diagonal-bordered sparse iterative method based on the Gauss-Seidel method. Parallel factorization algorithms for block-diagonal-bordered form matrices require a specialized ordering step coupled to an explicit load balancing step in order to generate this matrix form and to distribute the computational workload uniformly for an irregular matrix throughout a distributed-memory multi-processor. Matrix orderings are performed using a diakoptic technique based on node-tearing-nodal analysis. Parallel Gauss-Seidel algorithms for block-diagonal-bordered form matrices require a two-part matrix ordering technique — first to partition the matrix into block-diagonal-bordered form, again, using the node-tearing diakoptic techniques and then to multi-color the data in the last diagonal block using graph coloring techniques. The ordered matrices have extensive parallelism, while maintaining the strict precedence relationships in the Gauss-Seidel algorithm.

Empirical performance measurements for real power system networks are presented for implementations of a parallel block-diagonal-bordered LU algorithm, a similar Choleski algorithm, and a parallel block-diagonal-bordered Gauss-Seidel algorithm run on a distributed memory Thinking Machines CM-5 multi-processor. We have compared the performance of the direct and iterative parallel implementations on the CM-5, and show that significant algorithmic speedup may be possible for the Gauss-Seidel algorithm versus Choleski factorization for positive definite matrices. We have developed a simple technique that uses empirical data to predict the performance of these algorithms on future architectures. We apply these techniques to develop algorithm performance predictions for future Scalable Parallel Processing (SPP) architectures.

Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Power Systems Applications

by

David P. Koester

M.A.S., The Ohio State University, 1978

B.A., Augustana College, 1977

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

October, 1995

Approved ______________________________
Professor Geoffrey C. Fox

Approved ______________________________
Professor Sanjay Ranka

Date ________________________________

© Copyright 1995
David P. Koester

To my loving wife

Ann

and my super children

Yuri, Zachary, Naomi, and Abram

Acknowledgments

I am deeply indebted to the late Professor Yehuda Wallach, my first advisor, for both inspiring me to pursue my degree and providing me with my initial insights into the use of diakoptics to provide parallelism in power systems network applications. To say that Professor Wallach was passionate about diakoptics would have been an understatement!

I am grateful that Professor Wallach accommodated my part-time student status with weekend meetings at his home. These meetings invariably included the gracious hospitality of tea and snacks provided by his wife, Rachel. I worked with Professor Wallach during the Gulf War (Spring semester 1991), and several meetings were interrupted as Yehuda and Rachel anxiously awaited telephone calls from their daughter in Israel to indicate that she was safe after Iraqi Scud missile attacks. After a telephone call confirmed his daughter's safety, Professor Wallach would resume our meeting.

My research confirmed that Professor Wallach's intuition was correct concerning the applicability of diakoptic techniques as the basis for efficient parallel linear solver algorithms for power systems network applications. May this research stand as a testament to his memory.

* * *

There are many individuals in the academic community to whom I am deeply indebted. First, I would like to thank my co-advisors, Professors Geoffrey C. Fox and Sanjay Ranka, for their assistance and guidance during my years of work on my research. As co-advisors, they provided a good balance of interest in my research. Professor Nancy McCracken, the third member of my research committee, provided guidance and much needed support as I attempted to balance a (nearly) full-time job, my academic research, and my family. I would like to thank Professor Tony Skjellum of Mississippi State University and Professor Chis Pottle of Cornell University for their contributions to this research: their comments helped me make notable improvements to this work. Special thanks go to Alvin Leung, a Graduate Research Assistant at the Northeast Parallel Architectures Center (NPAC), for porting software developed for the Thinking Machines CM-5 to the IBM SP1 and SP2 and for running benchmarks of the software on that architecture. Lastly, I would like to express my gratitude to Dr. Paul Coddington, Senior Researcher Scientist at NPAC, for his contributions to this research and for his friendship.

I owe additional thanks to Professor Fox, who as director of NPAC provided me with the infrastructure, an office, a workstation, travel funds, and access to parallel computer resources, to pursue my research. This support was funded 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. I am also grateful to have had the opportunity to professionally associate with the many fine intellects that Geoffrey has assembled at Syracuse University.

* * *

I gratefully acknowledge the financial assistance and the time provided me to work on my research by my employer, The MITRE Corporation, through their Advanced Graduate Degree program.

* * *

Finally, I wish to express my special gratitude for the many years of support from my loving wife, Ann, and my super children: Yuri, Zachary, Naomi, and Abram. I hope that my children are as proud of me as I, at the age of thirteen, was of my father when he received his high school equivalency diploma.

Ann, I love you so very much.

Contents

Acknowledgments **iv**

List of Tables **x**

List of Figures **xi**

1 Introduction **1**

1.1 The State of Parallel Power Systems Linear Solver Research 2

1.2 Block-Diagonal-Bordered Power System Matrices 3

1.2.1 Block-Diagonal-Bordered Direct Linear Solvers 5

1.2.2 Block-Diagonal-Bordered Iterative Linear Solvers 6

1.3 Low-Latency Communications 6

1.4 Embedded Software Applications 7

1.5 Organization of this Thesis 8

2 Power System Applications **10**

2.1 Load-Flow Analysis 10

2.2 Transient Stability Analysis 11

2.3 Power System Network Matrices 12

2.4 Pseudo-Images Representing Sparse Power Systems Network Matrices 14

3 Linear Solvers **17**

3.1 Direct Methods 17

3.1.1 LU Factorization 18

3.1.2 Choleski Factorization 18

3.1.3 Ordering Sparse Matrices for Direct Methods 21

3.1.4 A Survey of the Literature for Parallel Direct Methods 23

3.2 Iterative Methods 25

3.2.1 Gauss-Seidel . . . 25
3.2.2 Ordering Sparse Matrices for Iterative Methods . . . 27
3.2.3 A Survey of the Literature for Parallel Iterative Methods . . . 28

4 Available Parallelism 29
4.1 Available Parallelism in Block-Diagonal-Bordered Form Matrices for Direct Methods 30
4.2 Available Parallelism in Block-Diagonal-Bordered Form Matrices for Iterative Methods 35
4.2.1 Parallelism in Block-Diagonal-Bordered Matrices . . . 36
4.2.2 Parallelism in Multi-Colored Matrices . . . 38

5 The Preprocessing Phase 41
5.1 The Preprocessing Phase for Direct Methods . . . 42
5.1.1 Ordering . . . 42
5.1.2 Pseudo-Factorization . . . 43
5.1.3 Load Balancing . . . 44
5.2 The Preprocessing Phase for Iterative Methods . . . 45
5.2.1 Ordering . . . 46
5.2.2 Pseudo-Solution . . . 47
5.2.3 Load Balancing . . . 47

6 Parallel Sparse Matrix Solver Implementations 49
6.1 Sequential Code Optimization . . . 49
6.2 Parallel Sparse Direct Solver Implementations . . . 51
6.2.1 The Hierarchical Data Structure . . . 53
6.2.2 Parallel Blocked-Diagonal-Bordered LU Factorization . . . 56
6.2.3 Forward Reduction and Backward Substitution . . . 59
6.3 Parallel Sparse Iterative Solver Implementations . . . 61
6.3.1 The Hierarchical Data Structure . . . 62
6.3.2 Parallel Blocked-Diagonal-Bordered Gauss-Seidel . . . 63

7 Empirical Results for Parallel Linear Solvers 67
7.1 Empirical Results for Parallel Direct Linear Solvers . . . 67
7.1.1 Ordering Power Systems Network Matrices into Block-Diagonal-Bordered Form 68
7.1.2 Parallel Direct Sparse Solver Performance . . . 72
7.1.2.1 Selecting Partitioned Matrices with *Best* Parallel Solver Performance 75
7.1.2.2 Timing Performance Comparisons . . . 76
7.1.2.3 Examining Speedup . . . 78

7.1.2.4 Analyzing Algorithm Component Performance 81
7.1.3 Comparing Communications Paradigms 85
7.1.4 Algorithm Performance for the IBM SP1 and SP2 87
7.1.5 Conclusions 89
7.2 Empirical Results for the Parallel Iterative Linear Solver 89
7.2.1 Ordering Power Systems Network Matrices into Block-Diagonal-Bordered Form 90
7.2.2 Parallel Sparse Gauss-Seidel Performance 93
7.2.2.1 Selecting Partitioned Matrices with *Best* Parallel Solver Performance 95
7.2.2.2 Timing Performance Comparisons 97
7.2.2.3 Examining Speedup 98
7.2.2.4 Analyzing Algorithm Component Performance 99
7.2.3 Convergence Rate 102
7.2.4 Comparing Communications Paradigms 103
7.2.5 Conclusions 110
7.3 Comparing Parallel Direct and Iterative Algorithms 110
7.3.1 Parallel Choleski versus Parallel Gauss-Seidel 111
7.3.2 Parallel LU Solvers versus Parallel Gauss-Seidel 112
7.3.3 Conclusions 114

8 Algorithm Performance on Future SPP Architectures 115
8.1 Parallel Computing for the Power Utility Industry 116
8.2 Overhead-Based Performance Estimates 117
8.3 Performance Predictions for Direct Solvers 119
8.3.1 Model Validation 119
8.3.2 Performance Predictions 120
8.4 Performance Predictions for Iterative Solvers 123
8.5 Low-Latency Communications Considerations 125
8.6 Conclusions 126

9 Conclusions 127
9.1 Direct Methods 128
9.2 Iterative Methods 128
9.3 Comparisons of Direct and Iterative Methods 129
9.4 Future Architectures 130
9.5 Future Research Opportunities 130

A Nomenclature **138**
A.1 Parallel Processor Architectures . . . 138
A.2 Parallel Computing Analysis . . . 138

B Node-Tearing Nodal Analysis **142**
B.1 The Node-Tearing Algorithm . . . 142
B.2 An Example of Node-Tearing . . . 145
B.3 The Node-Tearing Implementation . . . 148

C Minimum-Degree Ordering **150**

D Graph Coloring **152**
D.1 The Basic Greedy Graph Coloring Heuristic . . . 152
D.2 The Saturation Degree Ordering Heuristic . . . 153

E Implementation Pseudo-Code **155**
E.1 Parallel Blocked-Diagonal-Bordered LU Factorization . . . 155
E.2 Parallel Blocked-Diagonal-Bordered Gauss-Seidel . . . 156

F Network Ordering Performance Statistics **174**
F.1 Parallel Direct Solver Statistics . . . 174
F.2 Parallel Iterative Solver Statistics . . . 174

List of Tables

2.1 Comparison of Power System Matrices and Boeing-Harwell Structural Matrices . . 16
7.1 EPRI6K — IBM SP1 and SP2 Performance Data — Complex Variate LU Solver . . 88
7.2 Convergence for the BCSPWR09 Power Systems Network 102
7.3 Convergence for the BCSPWR10 Power Systems Network 103
F.1 BCSPWR09 — LU and Choleski Factorization Ordering Statistics 175
F.2 BCSPWR10 — LU and Choleski Factorization Ordering Statistics 175
F.3 EPRI6K — LU and Choleski Factorization Ordering Statistics 176
F.4 NiMo-OPS — LU and Choleski Factorization Ordering Statistics 176
F.5 NiMo-PLANS — LU and Choleski Factorization Ordering Statistics 177
F.6 BCSPWR09 — Gauss-Seidel Ordering Statistics 177
F.7 BCSPWR10 — Gauss-Seidel Ordering Statistics 178
F.8 EPRI6K — Gauss-Seidel Ordering Statistics 179
F.9 NiMo-OPS — Gauss-Seidel Ordering Statistics 180
F.10 NiMo-PLANS — Gauss-Seidel Ordering Statistics 181

List of Figures

1.1 Summary of Completed and Future Research 4
2.1 Ordering the Admittance Sub-Matrix in Transient Stability Differential-Algebraic Equations 11
2.2 Relative Frequency Histogram of Edges per Graph Node 14
2.3 Pseudo-Images of Original Sparse Power Systems Matrices 15
3.1 Sparse LU Factorization — Doolittle Algorithm 19
3.2 Sparse Forward Reduction for Doolittle Factorization 19
3.3 Sparse Backward Substitution for Doolittle Factorization 20
3.4 Sparse Column Choleski Factorization 20
3.5 Sparse Forward Reduction for Choleski Factorization 21
3.6 Sparse Backward Substitution for Choleski Factorization 21
3.7 Graph Theoretical Explanation of Fillin 23
3.8 Sparse Gauss-Seidel Algorithm 27
4.1 Graph with Four Independent Sub-Matrices 31
4.2 Elimination Tree with Four Supernode Leaves 31
4.3 Block Bordered Diagonal Form Sparse Matrix Solution Steps 36
4.4 Block-Bordered-Diagonal Form Gauss-Seidel Method 39
4.5 Multi-Colored Gauss-Seidel Method for the Last Diagonal Block 40
6.1 Optimal Row-Major Data Storage for Gauss-Seidel Algorithms 50
6.2 Optimal Data Storage for Dense Factorization 51
6.3 Optimal Data Storage for Sparse Factorization — Doolittle's Algorithm 52
6.4 The Hierarchical Data Structure for Parallel LU Factorization 54
6.5 The Hierarchical Data Structure for Parallel Gauss-Seidel Methods 63
7.1 BCSPWR09 — Block-Diagonal-Bordered Form — Load Balanced for 8 Processors . 70
7.2 Last Diagonal Block Size after Partitioning 71
7.3 Block-Diagonal-Bordered Form Matrices — Load Balanced for 8 Processors 73
7.4 Parallel LU Factorization Timing Data — Double Precision 75

7.5 Parallel Choleski and LU Timing Comparisons 77
7.6 Relative Speedup — Parallel Direct Solvers 78
7.7 BCSPWR09 — Algorithm Component Timing Data 82
7.8 BCSPWR10 — Algorithm Component Timing Data 83
7.9 Speedup — Low-Latency versus Buffered Communications 85
7.10 Speedup — Low-Latency versus Buffered Communications — Update Last Block . 86
7.11 BCSPWR09 — Block-Diagonal-Bordered Form for Parallel Gauss-Seidel 92
7.12 Block-Diagonal-Bordered Form for Parallel Gauss-Seidel — Load Balanced for 8 Processors 94
7.13 Parallel Gauss-Seidel Timing Data — Double Precision 96
7.14 Best Empirical Performance for Parallel Gauss-Seidel Implementations 98
7.15 Relative Speedup — Parallel Gauss-Seidel 99
7.16 Algorithm Component Timing Data — Double Precision Gauss-Seidel 101
7.17 Relative Speedup — Double Precision Parallel Gauss-Seidel 104
7.18 Number of Low-Latency Messages Required to Distribute $\hat{\mathbf{x}}^{(k+1)}$ 105
7.19 Percentage of Broadcast Values Required to Distribute $\hat{\mathbf{x}}^{(k+1)}$ 106
7.20 BCSPWR09 — Histograms of the Number of Messages to Distribute $\hat{\mathbf{x}}^{(k+1)}$ 107
7.21 BCSPWR10 — Histograms of the Number of Messages to Distribute $\hat{\mathbf{x}}^{(k+1)}$ 108
7.22 Low-Latency Communications Speedup — Parallel Gauss-Seidel 109
7.23 Gauss-Seidel Iterations as a Function of Dishonest Reuses of the Choleski $\mathbf{LL}^T$ Matrix 112
7.24 Gauss-Seidel Iterations as a Function of Dishonest Reuses of the Double Precision **LU** Matrix 113
7.25 Gauss-Seidel Iterations as a Function of Dishonest Reuses of the Complex **LU** Matrix 114
8.1 Performance Validation for Parallel Complex LU Factorization 120
8.2 Performance Predictions for Parallel Complex LU Factorization 121
8.3 Performance Predictions for Parallel Complex Gauss-Seidel — Low-Latency Communications Paradigm 124
8.4 Performance Predictions for Parallel Complex Gauss-Seidel — Buffered Communications Paradigm 125
B.1 Sample Contour Tableau for the k^{th} Diagonal Block 144
B.2 Graph for a Node-Tearing Example 146
B.3 Example Contour Tableau 146
B.4 Relabeled Example Graph 147
B.5 Matrix Representation of the Example Graphs 147
B.6 Elimination Trees for the Example Graphs 147
B.7 The Node-Tearing Algorithm 149

C.1 The Minimum-Degree Algorithm 151

D.1 The Graph Multi-Coloring Algorithm 153

D.2 The Graph Multi-Coloring Algorithm for the Last Diagonal Block of a Block-Diagonal-Bordered Matrix 154

E.1 Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Diagonal Blocks and Border 157

E.2 Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Update the Last Diagonal Block — Low-Latency Communications Paradigm 158

E.3 Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Update the Last Diagonal Block — Buffered Communications Paradigm 159

E.4 Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Last Diagonal Block 160

E.5 Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Diagonal Blocks and Border 161

E.6 Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Update the Last Diagonal Block — Low-Latency Communications Paradigm 162

E.7 Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Update the Last Diagonal Block — Buffered Communications Paradigm 163

E.8 Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Last Diagonal Block 164

E.9 Parallel Block-Diagonal-Bordered Sparse Backward Substitution Algorithm — LU Factorization — Last Diagonal Block 165

E.10 Parallel Block-Diagonal-Bordered Sparse Backward Substitution Algorithm — LU Factorization — Diagonal Blocks and Border 166

E.11 The Iterative Framework for the Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm 167

E.12 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Diagonal Blocks 167

E.13 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Update $\hat{\mathbf{b}}$ — Low-Latency Communications Paradigm 168

E.14 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Update $\hat{\mathbf{b}}$ — Buffered Communications Paradigm 169

E.15 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Last Diagonal Block — Low-Latency Communications Paradigm 170

E.16 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Last Diagonal Block — Buffered Communications Paradigm 171

E.17 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Convergence Check — Low-Latency Communications Paradigm 172

E.18 Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Convergence Check — Buffered Communications Paradigm 173

Chapter 1

Introduction

This thesis presents research into parallel linear solvers for block-diagonal-bordered sparse matrices. The block-diagonal-bordered form identifies parallelism that can be exploited for both direct and iterative linear solvers. *Direct methods* obtain the exact solution for $\mathbf{Ax} = \mathbf{b}$ in a finite number of operations, whereas *iterative methods* calculate sequences of approximations that may or may not converge to the solution. In order to compare performance for parallel sparse direct and iterative linear solvers for power systems network applications, we have developed efficient parallel block-diagonal-bordered sparse direct methods based on LU factorization and Choleski factorization algorithms, and we have developed an efficient parallel block-diagonal-bordered sparse iterative method based on the Gauss-Seidel method. We are examining parallel sparse linear solvers for embedded power systems applications, so the direct solvers we implement also require parallel forward reduction and backward substitution algorithms.

Solving sparse linear systems practically dominates scientific computing, but the performance of direct sparse matrix solvers has tended to trail behind its dense matrix counterparts [29]. Parallel sparse matrix solver performance generally is less than similar dense matrix solvers even though there is more inherent parallelism in sparse matrix algorithms than dense matrix algorithms. This additional parallelism is often described by *elimination trees*, graphs that illustrate the dependencies in the calculations [19, 20, 21, 29, 55, 56, 57, 58, 64]. Parallel sparse linear solvers can simultaneously factor entire groups of mutually independent contiguous blocks of columns or rows without communications; meanwhile, dense linear solvers can only update blocks of contiguous columns or rows during each pipelined communication cycle. The limited success with efficient sparse matrix solvers is not unexpected, because general sparse linear solvers require more complicated data structures and algorithms that must contend with irregular memory reference patterns. The irregular nature of many real-world sparse matrices has aggravated the task of implementing 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 [11, 25, 47].

We have focused on developing parallel linear solvers optimized for sparse matrices from the power systems community — in particular, we have examined linear solvers for matrices resulting from power distribution system networks. These matrices are some of the most sparse matrices encountered in real-life applications, while also being irregular. Recently, references [25, 33] have reported scalable Choleski solvers with extremely good performance for large numbers of processors, but they are for matrices that have more rows/columns, that have more nonzero elements per row/column, and that are more regular than power systems matrices. When empirical performance of sparse linear solvers is examined using real, irregular sparse matrices, available parallelism in the sparse matrix or load-imbalance overhead can be as much the reason for poor parallel efficiency as the parallel algorithm or implementation [37, 47].

1.1 The State of Parallel Power Systems Linear Solver Research

Power systems matrices are both irregular and the sparsest matrices available to the academic and industrial communities. As a result, research into efficient sparse linear solvers for power systems applications has not met with the same success as research into efficient general parallel sparse linear solvers [7, 54]. The state of parallel direct linear solver development in the power systems community has yielded solvers sufficiently inefficient that sequential algorithms are used to factor and triangular solve equations that may be formulated in parallel [7]. While efficient parallel linear solvers have not been reported in the power systems community journals to solve the special very sparse irregular power systems network matrices, there has been significant research into efficient general sparse linear solvers for general matrices, always larger and less sparse than power systems network matrices [8, 9, 10, 19, 20, 21, 25, 29, 46, 47, 55, 56, 57, 58, 64].

In the research presented in this thesis, we have developed specialized, efficient parallel sparse linear solvers for linear systems derived from power systems networks. The performance of our parallel linear solvers is significantly better than the performance of linear solvers reported in the power systems literature [7, 54]. In order to develop efficient parallel linear solvers, we have utilized state-of-the-art research into:

- efficient general parallel sparse linear solvers,
- power systems analysis,
- computational science,

and applied these concepts to the special nature of power systems applications with deliberate attention to maximizing performance for power systems network matrices. We have combined:

- load-balanced elimination trees from the general parallel sparse linear solver community [21, 25, 29],
- diakoptics from the power system community [26, 62],
- early work with block-diagonal-bordered form matrices [15],
- efficient parallel algorithm design guidelines [17, 23, 24, 41, 52],

to develop our efficient parallel block-diagonal-bordered linear solvers.

Research by others into general parallel linear solvers has answered many of the outstanding questions concerning the development of techniques for efficient parallel algorithms to solve general sparse matrices from the structural analysis community. The research presented in this thesis presents efficient parallel direct and iterative linear solvers that yield efficient performance on power systems network linear equations for either symmetric positive definite matrices or position symmetric matrices that do not require pivoting to ensure numerical stability. Developing linear solvers for power systems applications that require pivoting remains as future research, as has the optimization of the use of preconditioning techniques with iterative solvers. Another power systems application problem that remains for future research, is the development of new, more efficient differential-algebraic equation (DAE) solvers that would utilize the parallel block-diagonal-bordered linear solvers and address the entire linearized block-diagonal-bordered DAE matrix [34]. A summary of the research to date and future research is presented in figure 1.1

1.2 Block-Diagonal-Bordered Power System Matrices

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

Diakoptics, or the *tearing* of systems into smaller subsystems then solving the subsystems in a piecewise manner before reconstructing the system, has offered promise to be used as the basis

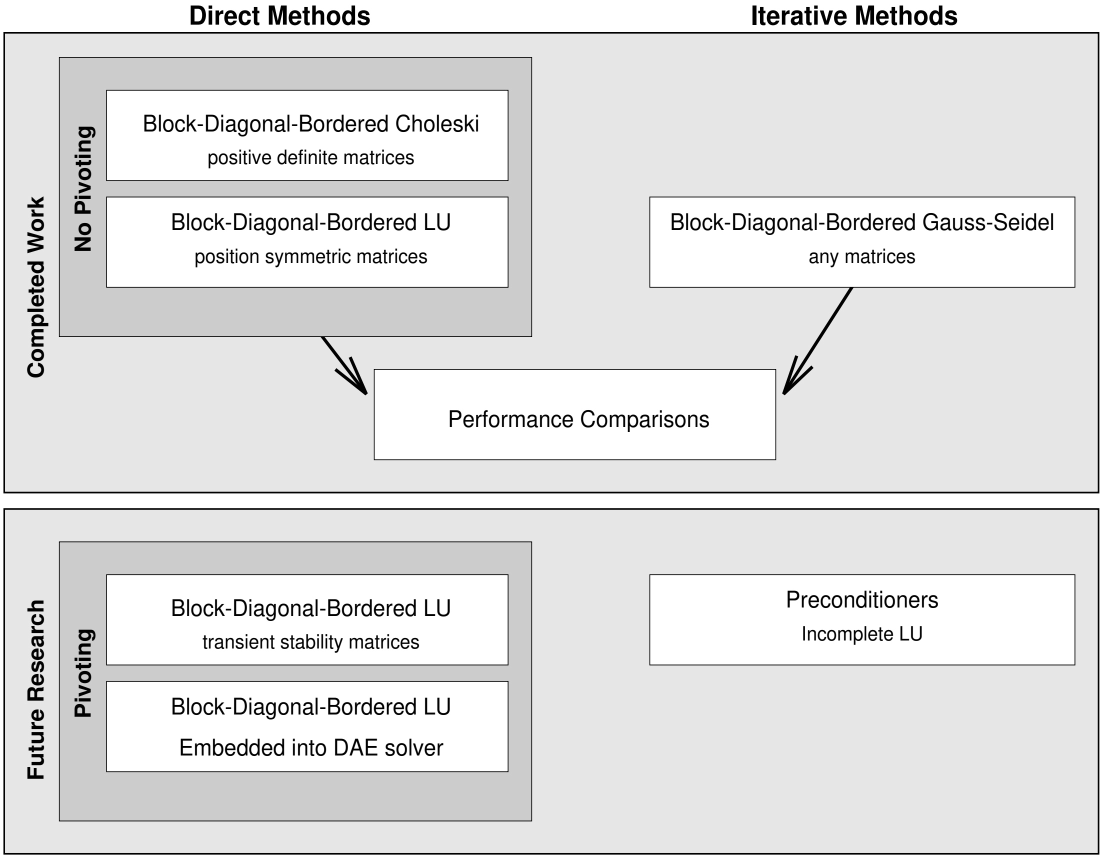

Direct Methods
Iterative Methods
Completed Work
No Pivoting
Block-Diagonal-Bordered Choleski
positive definite matrices
Block-Diagonal-Bordered LU
position symmetric matrices
Block-Diagonal-Bordered Gauss-Seidel
any matrices
Performance Comparisons
Future Research
Pivoting
Block-Diagonal-Bordered LU
transient stability matrices
Block-Diagonal-Bordered LU
Embedded into DAE solver
Preconditioners
Incomplete LU

Figure 1.1: Summary of Completed and Future Research

for parallel sparse linear solvers for power systems applications [12, 26]. A specialized form of diakoptics, node-tearing nodal analysis, has been used to partition power systems network matrices into block-diagonal-bordered form [49]. The application of diakoptic techniques identifies inherent power systems network structure that in turn can be exploited to provide parallelism for sparse linear solvers embedded within power systems applications. Node-tearing diakoptic techniques are readily identified with methods to identify parallelism in general sparse linear solvers. In particular, node-tearing-based partitioning of power systems networks identifies block-diagonal-bordered form matrices, that are related to *elimination trees*, and *supernodes* within the network where there is inherent parallelism. Diakoptic node-tearing-based partitioning identifies the basic network structure that provides parallelism for the majority of calculations within both direct and iterative solutions of power systems network-based linear systems.

In this thesis, we examine the applicability of parallel block-diagonal-bordered sparse solvers for real power system applications that require either the solution of symmetric positive definite sparse matrices or location symmetric sparse matrices that result from solving problems relating to power systems networks. Variations of this technique could be used to solve other power system sparse linear systems such as those that result from solving linearized differential-algebraic equations from transient stability analysis applications or small-signal stability assessments. The implementations we describe in this paper work directly with the equations resulting from the power systems network, the smallest class of power system matrix.

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

1.2.1 Block-Diagonal-Bordered Direct Linear Solvers

Block-diagonal-bordered sparse matrix algorithms require modifications to the normal preprocessing phase described in numerous papers on parallel Choleski factorization [19, 20, 21, 29, 55, 56, 57, 58, 64]. Each of the numerous papers referenced above use the paradigm to *order* the sparse matrix and then perform *symbolic factorization* in order to determine the locations of all fillin values so that static data structures can be utilized for maximum efficiency when performing numerical factorization. We modify this commonly used sparse matrix preprocessing phase to include an explicit *load balancing* step coupled to the *ordering* step so that the workload is uniformly distributed throughout a distributed-memory multi-processor and parallel algorithms make efficient use of the computational resources.

Parallel block-diagonal-bordered sparse direct linear solvers offer the potential for regularity often absent from other parallel sparse solvers [34, 35, 36, 38]. Our research into specialized matrix ordering techniques has shown that it is possible to order actual power system matrices readily into block-diagonal-bordered form, and to load-balance sufficiently effectively that relative speedups greater than ten have been observed in empirical performance measurements for direct solvers on a 32 processor Thinking Machines CM-5.

1.2.2 Block-Diagonal-Bordered Iterative Linear Solvers

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 [35, 36]. 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.

The same diakoptic node-tearing-based network partitioning used to order matrices into block-diagonal-bordered form for direct linear methods has been used to identify available parallelism within the irregular sparse power systems matrices for our parallel Gauss-Seidel implementation,

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 block-diagonal-bordered matrix. 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 Gauss-Seidel 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. Load-balancing for the parallel Gauss-Seidel algorithm is sufficiently effective that relative speedups greater than 20 have been observed in empirical performance measurements for iterative solvers on a 32 processor Thinking Machines CM-5.

1.3 Low-Latency Communications

For this work, active message-based communications on the Thinking Machines CM-5 provided implementations with extremely low-latency interprocessor communications. Separate algorithms have been developed for conventional non-blocking, buffered communications and low-latency, active message-based communications. Active message remote procedure calls (RPCs) provide protocolless access to the transport layer of the CM-5 communication network. The user must assume responsibilities for all aspects of communications — but is rewarded with very low-latency communications for short messages. The extremely low-latency communications available with active messages, permit a parallel-code development paradigm that greatly simplifies the implementation of these sparse matrix algorithms. Empirical data have been collected on both active message-based implementations and more traditional non-blocking, buffered message passing commands in order to illustrate the need for low-latency communications when solving matrices as sparse and irregular as power systems matrices.

1.4 Embedded Software Applications

Our research has examined the performance of block-diagonal-bordered direct solvers, with implementations for both Choleski and LU factorization, to be incorporated within electrical power system applications. Because we are considering software to be embedded within a more extensive application, we examine efficient parallel forward reduction and backward substitution algorithms in addition to parallel factorization algorithms. Due to the reduced amount of calculations in the triangular solution phases of solving a system of factored linear equations, these algorithms are often ignored when parallel Choleski or LU factorization algorithms are presented in the literature. However, we will show that for large power systems network matrices, there is less than an order of magnitude difference in parallel factorization and parallel triangular solution empirical run times.

In our research, we have found that the development of parallel factorization algorithms must consider forward reduction and backward substitution, because the choice of the order of calculations in factorization can greatly influence the performance of the parallel triangular solutions. In order to ensure cache hits, data structures are dependent on the order of calculations, and data structures affect the amount of communications in parallel forward reduction and backward substitution. We have found that the results of additional communications overhead can eliminate any potential speedup for parallel forward reduction with column oriented data storage. This communications overhead cannot be eliminated for Choleski factorization, where either forward reduction or backward substitution must be performed with an implicit transpose of the factored matrix [29, 40]. Fortunately, the LU factorization algorithm can be implemented in a manner to eliminate this communications overhead problem.

One goal of our research has been to compare performance for parallel block-diagonal-bordered direct and iterative solvers to determine which algorithm could provide the best performance when embedded within a parallel power systems application. Direct methods obtain the exact solution for $\mathbf{Ax} = \mathbf{b}$ in a finite number of operations, whereas iterative methods calculate sequences of approximations that may or may not converge to the solution. Nevertheless, if the parallel sparse Gauss-Seidel method does converge and does so rapidly for an application, the iterative technique can provide algorithmic speedup when compared to parallel direct sparse methods. For sequential dense Gauss-Seidel and LU factorization, both a Gauss-Seidel iteration and forward-reduction and backward substitution have the same computational complexity, $O(N^2)$; however, a single iteration of the sparse Gauss-Seidel algorithm has less computations than the combination of sparse forward reduction and backward substitution. Both algorithms perform a sparse matrix × vector product; however, the a Gauss-Seidel iteration has only the original non-zeros in the sparse matrix, while the forward reduction and backward substitution for a direct method also includes fillin — or values that become non-zero during the process of factorization. The computational complexity for dense factorization, $O(N^3)$, is even greater than for forward reduction and backward substitution, adding computational costs to the parallel sparse direct methods.

To illustrate the potential algorithmic speedup available for the parallel iterative solver, we provide parametric comparisons of the parallel sparse Gauss-Seidel algorithm and the parallel sparse direct methods using empirical data collected on the Thinking Machines CM-5. Comparisons of algorithm performance can be made as a function of the number of iterations that can be performed in the time to perform the factorization and any number of forward reductions and backward substitutions. Performance improvements can only be assured for the solution of diagonally dominate or positive definite matrices, where convergence with the Gauss-Seidel method is ensured [23].

1.5 Organization of this Thesis

This thesis is organized as follows. In chapter 2, we describe the electrical power system applications that are the basis for this work. We compare the size and sparsity of five sample power systems matrices to other sample matrices used frequently to compare the performance of parallel sparse linear solvers. Pseudo images representing the original sparse power matrices are provided for comparison to other pseudo images that represent the matrices after ordering into block-diagonal-bordered form.

In chapter 3, we briefly review techniques for both direct and iterative linear solvers. We discuss the direct techniques, factorization and forward reduction/backward substitution, and the Gauss-Seidel iterative method. We review the extensive literature describing research into general parallel LU and Choleski factorization algorithms, and the literature describing research into parallel iterative methods. This is followed in chapter 4 by a theoretical derivation of the available parallelism in both direct methods and the Gauss-Seidel iterative method when solving block-diagonal-bordered form sparse matrices.

Paramount to exploiting the advantages of either the parallel sparse block-diagonal-bordered direct methods or the parallel sparse block-diagonal-bordered iterative Gauss-Seidel method is the process of ordering the irregular sparse power system matrices into this form in a manner that balances the workload among multi-processors. In chapter 5, we describe the preprocessing phase used to generate matrix ordering for block-diagonal-bordered matrices with uniformly distributed processing load. Both solver types require a three-step process that includes network partitioning, determining the workload in each partition, and an explicit load-balancing step. In this chapter, we introduce pseudo-factorization and we review minimum degree ordering and pigeon-hole load balancing algorithms.

In chapter 6, we describe the Thinking Machines CM-5 implementations of our parallel block-diagonal-bordered sparse LU, Choleski, and Gauss-Seidel algorithms. Analysis of the performance of the ordering techniques and the parallel implementations is presented in chapter 7 for actual power system network matrices from the Boeing-Harwell series, the Electrical Power Research Institute (EPRI), and an electrical utility, the Niagara Mohawk Power Corporation. Comparisons of the direct and iterative methods also are presented in this chapter. Predictions of performance on future parallel architectures are presented in chapter 8. We present our conclusions concerning parallel block-diagonal-bordered linear solvers for electrical power system applications in chapter 9.

In appendix A, we provide a description of selected terminology used throughout this work. In appendices B through D, we present detailed discussions of algorithms used in the preprocessing phase: the node-tearing algorithm developed to order matrices into block-diagonal-bordered form, the minimum degree ordering algorithm, and the graph multi-coloring algorithm developed specifically to order the last diagonal block in the parallel Gauss-Seidel algorithm. In appendix E, we present pseudo-code descriptions of the parallel algorithms described in chapter 6, and in appendix F, we present tables of statistics to illustrate the performance of our diakoptics-based power systems network matrix ordering technique.

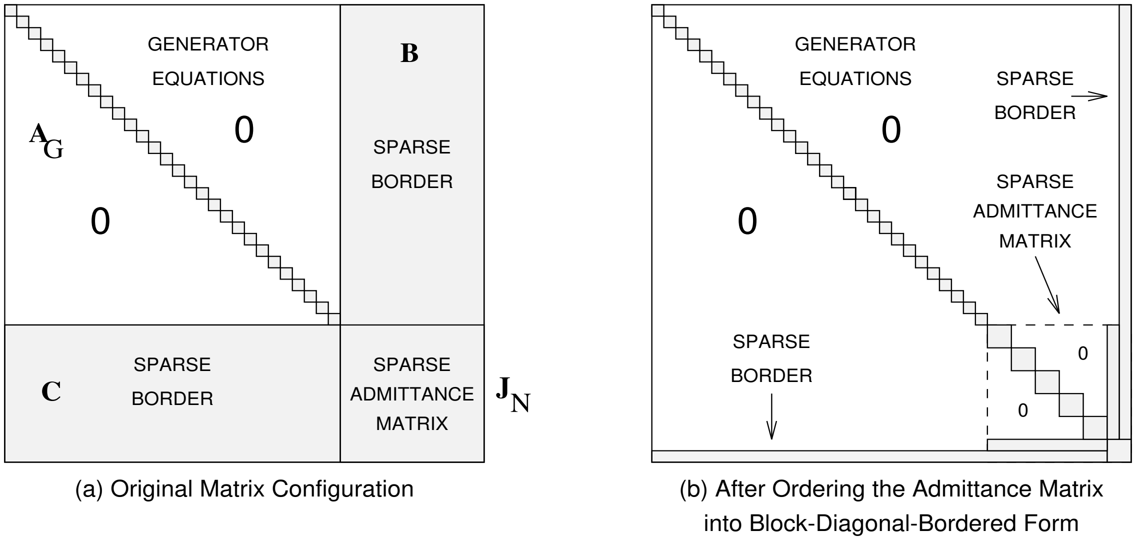
Chapter 2

Power System Applications

The underlying impetuous for our research is to improve the performance of electrical power system applications that provide real-time power system control and real-time support for proactive decision making with efficient parallel linear solver algorithms. In particular, our research has focused on load-flow and transient stability applications [1, 2, 4, 62], where sparse linear solvers account for a substantial percentage of floating point operations encountered [7, 54]. Efficient parallel linear solvers could significantly affect real-time performance of these applications.

2.1 Load-Flow Analysis

Load-flow analysis examines steady-state equations based on the positive definite network admittance matrix that represents the power system distribution network. Load-flow analysis is used for identifying potential network problems in contingency analyses, for examining steady-state operations in network planning and optimization, and also for determining initial system state in transient stability calculations [62]. Load flow analysis entails the solution of non-linear systems of simultaneous equations, which are performed by repeatedly solving sparse linear equations. Load flow is calculated using the network admittance matrices, which are symmetric positive definite and have sparsity defined by the power system distribution network. The size of these matrices is limited because individual power systems generally use networks with less than 2,000 sparse complex equations in their operations centers, while regional power authority operations centers would also be limited to sparse load-flow matrices with less than 10,000 sparse complex equations. Power systems planning studies often incorporate larger networks as lower voltage distribution lines are included in these studies. Sparse matrices employed in planning studies can have from 5,000 to 50,000 sparse equations. This paper presents data for power system networks with 1,723, 1,766, 5,300, 6,692, and



(a) Original Matrix Configuration

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

Figure 2.1: Ordering the Admittance Sub-Matrix in Transient Stability Differential-Algebraic Equations

9,430 nodes.

2.2 Transient Stability Analysis

Transient stability analysis is a detailed simulation of the power system that models the dynamic behavior of the electrical distribution networks, electrical loads, and the electro-mechanical equations of motion of the interconnected generators [4]. Transient stability analysis can be used to perform selective detailed analyses of generator commitment stability, and to support crisis decision-making during network recovery. The transient stability problem is modeled by differential algebraic equations (DAEs) with differential equations representing the generators and non-linear algebraic equations representing the power system network that interconnects the generators. The DAEs are in natural non-symmetric block-diagonal-bordered form, with diagonal blocks of generator equations coupled by the power system distribution network. In this representation, there are as many coupling equations as the entire sparse admittance matrix. It is also possible to order the admittance matrix to block-diagonal-bordered form in order to increase available parallelism. This is illustrated in figure 2.1. The size of the sparse matrices representing the DAEs have as many as 10,000 complex equations for an individual power system, while regional power authorities could have as many as 50,000 sparse complex equations in the matrix formed from the DAEs.

It is also possible to solve the above equations by decoupling the generator equations from the network equations. For decoupled transient stability analysis, the transient stability differential-algebraic equation matrix is partitioned into four sub-matrices. In figure 2.1, the generator equations are in the block-diagonal matrix labeled $\mathbf{A}_G$. The generator equations are solved independently of the network equations, then the sparse admittance matrix is modified by the matrix coefficients in the sparse borders, labeled **B** and **C**. The admittance matrix is labeled $\mathbf{J}_N$ in this figure. Instead of the common practice of decoupling the generator and network calculations in a transient stability simulation, we hope to continue this research and eventually examine the use of more powerful differential-algebraic equation solvers for transient stability analysis that do not decouple the generator and network equations. The fully-coupled differential-algebraic equations will offer more potential for good parallel performance because

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

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

2.3 Power System Network Matrices

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

Matrices representing power system networks are some of the most sparse matrices encountered throughout the academic or industrial community. Figure 2.2 illustrates the proportion of graph nodes with a particular number of graph edges or the number of non-zero values in a matrix row or column for five separate power system matrices:

- Boeing-Harwell matrix BCSPWR09 — 1,723 nodes and 2,394 graph edges [13],

- Boeing-Harwell matrix BCSPWR10 — 5,300 nodes and 8,271 graph edges [13],
- EPRI matrix EPRI6K matrix — 6,692 nodes and 10,535 graph edges [14],
- Niagara Mohawk Power Corporation operations matrix NiMo-OPS — 1,766 nodes and 2,506 graph edges,
- Niagara Mohawk Power Corporation planning matrix NiMo-PLANS — 9,430 nodes and 14,001 graph edges.

Matrices BCSPWR09 and BCSPWR10 are from the Boeing Harwell series and represent electrical power system networks from the Western and Eastern US respectively. The EPRI6K matrix is distributed with the Extended Transient-Midterm Stability Program (ETMSP) from the Electrical Power Research Institute (EPRI). Matrices NiMo-OPS and NiMo-PLANS have been made available by the Niagara Mohawk Power Corporation, Syracuse, NY.

In this relative frequency histogram, the most frequently occurring number of edges per node is *only* 2! Table 2.1 provides additional data to illustrate that power system matrices are both relatively small in size and also have the fewest average edges per node of available matrices. In this table, all data except that from EPRI and Niagara Mohawk are from the Boeing-Harwell series [13]. The structural matrices, BCSSTK13 to BCSSTK32, are frequently used in papers to benchmark parallel sparse linear algorithms [19, 20, 21, 25, 29, 33, 46, 47, 55, 56, 57, 58, 64]. For power system matrices, the average number of edges per node is less than two while for many of the structural matrices, the average number of nodes per edge is greater than ten. Also the number of nodes in power system matrices are limited when compared to the Boeing-Harwell structural matrices.

While power systems matrices are extremely sparse, they are also irregular, with the larger matrices having some nodes with greater than twenty edges. The histogram presented in figure 2.2 has been truncated at ten edges per node to emphasize the high incidence of edges with less than three nodes. As a result of the degree of sparsity and irregularity in these matrices, developing parallel sparse linear solvers for power systems application has proven to be a challenge [7, 38, 54]. Nevertheless, by developing parallel algorithms that actively address the irregular nature of the graphs with explicit load-balancing and by making all necessary communications as balanced, regular, and as asynchronous as possible, we will show in sections 7.1 and 7.2 that our block-diagonal-bordered approach to addressing linear solvers for power system applications can yield respectable speedups even for as many as 32 processors.

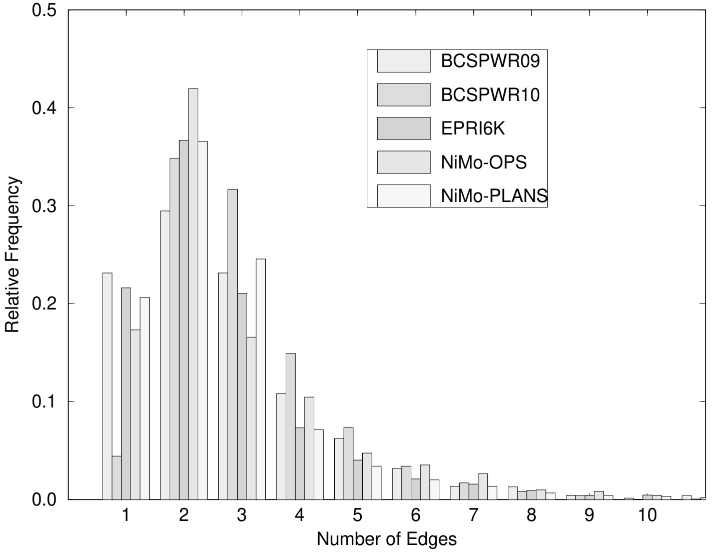


Figure 2.2: Relative Frequency Histogram of Edges per Graph Node

2.4 Pseudo-Images Representing Sparse Power Systems Network Matrices

In order to illustrate the sparsity and irregularity of the matrices that represent the power systems networks and to provide a baseline with which to illustrate the performance of the node-tearing algorithm in sections 7.1 and 7.2, we provide representations of the original matrices for the five power systems networks in figure 2.3. These pseudo-images illustrate the locations of the non-zero values in the matrices as black pixels, and the matrices are symmetrical around the diagonal. A bounding box has been placed around each sparse matrix, with the matrix identifier located in the upper triangular portion of each symmetric matrix.

These matrices have no fillin and are presented with the graph node identifiers as supplied in the distribution of the data — without any additional ordering. When examining these unordered matrices, there appears to be significant differences between the power systems networks from the Boeing-Harwell series and power systems networks from the Niagara Mohawk Power Corporation. The Niagara Mohawk Power Corporation matrices have distinct block structure, while the Boeing-Harwell matrices and the EPRI matrix appear that they have been previously ordered with a minimum degree ordering. The upper left portion of these matrices appears to have fewer values in rows/columns, while the lower right hand portion of the matrices appears denser.

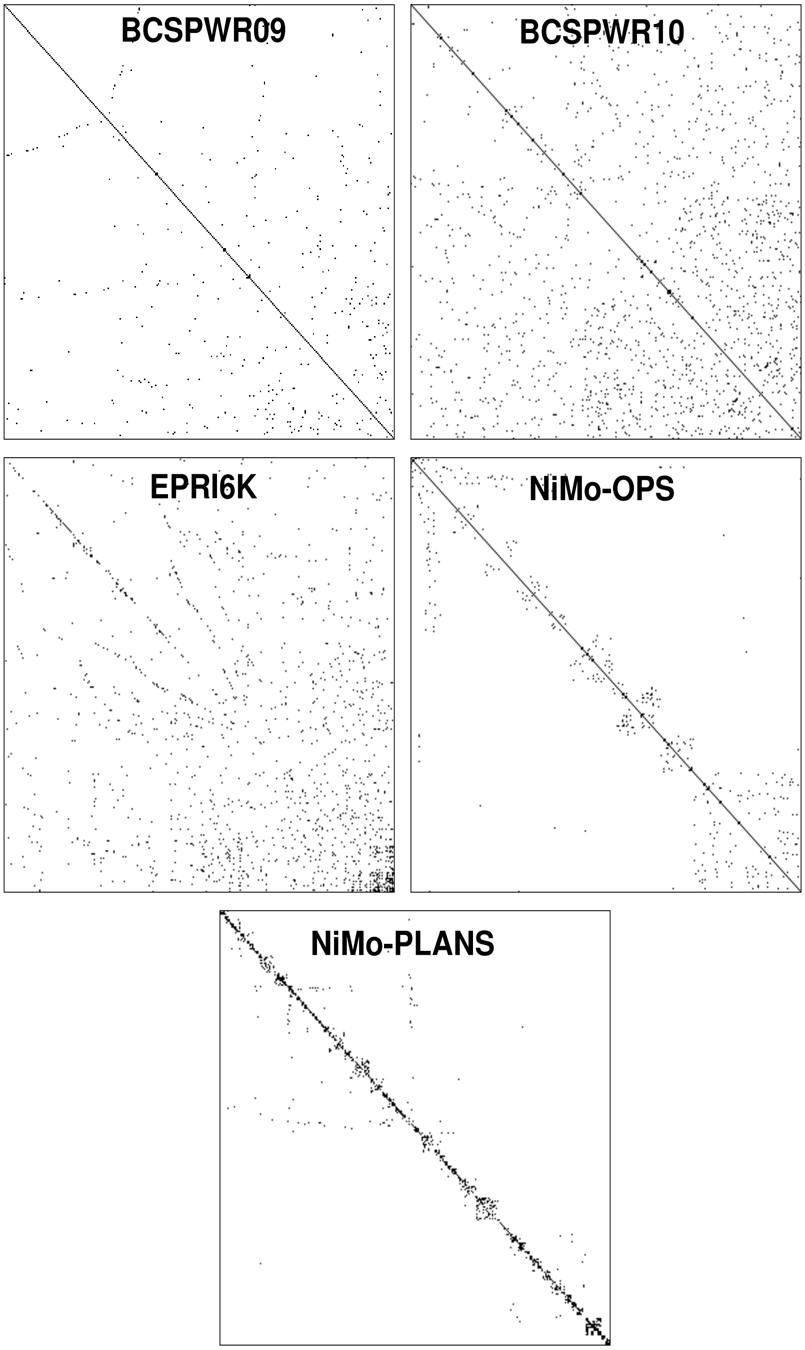


Figure 2.3: Pseudo-Images of Original Sparse Power Systems Matrices

Table 2.1: Comparison of Power System Matrices and Boeing-Harwell Structural Matrices

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

Chapter 3

Linear Solvers

This thesis presents research into parallel linear solvers for block-diagonal-bordered sparse matrices, but first we review algorithms for sequential sparse linear solvers. We address both direct and iterative solvers in this work, each having their own advantages and disadvantages. Direct methods obtain the exact solution for a series of simultaneous linear equations in a finite number of operations, whereas iterative methods calculate sequences of approximations that may or may not converge to the solution. While direct methods obtain an exact solution of the linear system, they may require significantly more computations than required for iterative methods to obtain a usable solution. The remainder of this chapter discusses two related direct methods, LU factorization and Choleski factorization, and one iterative solver, the Gauss-Seidel method.

3.1 Direct Methods

We are considering the solution of the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.1}$$

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

$$\mathbf{A}\mathbf{x} = \mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{b}, \tag{3.2}$$

A lower triangular matrix, $\mathbf{L}$, has all zeros above the diagonal and an upper triangular matrix, $\mathbf{U}$, has all zeros below the diagonal [12].

3.1.1 LU Factorization

The sparse matrix $\mathbf{A}$ can be numerically factored into a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ as in equation 3.2, where all values on the diagonal of either $\mathbf{L}$ or $\mathbf{U}$ must equal 1 — $l_{kk} = 1$ or $u_{kk} = 1$, where $k = 1, \cdots, N$. Equation 3.2 is solved by setting $\mathbf{Ux} = \mathbf{y}$, and substituting $\mathbf{y}$ for $\mathbf{Ux}$. The numerical solution for $\mathbf{Ly} = \mathbf{b}$ is found by forward reduction, and the numerical solution for $\mathbf{x}$ is calculated by backward substitution in the equation $\mathbf{Ux} = \mathbf{y}$. Triangular linear systems can be readily solved numerically by solving for the first value in the triangular linear system and substituting that value into subsequent equations.

Sparse LU factorization can mirror any similar dense factorization algorithm, although generally a sparse matrix algorithm has only one explicit **for** loop, which can be for any single index in the dense case. The remaining indices are examined only for non-zero values in the original matrix or for non-zero values that will occur from fillin in the matrix. Sparse matrix fillin occurs when a value that originally was zero becomes non-zero in the process of factoring the matrix. Fillin can be controlled in sparse factorization by *ordering* the matrix before performing the factorization if there is no requirement for pivoting to ensure numerical stability of the calculations [12]. There are many ordering techniques for position symmetric matrices, with one of the most common being *minimum degree ordering*. If pivoting is required to ensure numerical stability, a Markowitz ordering/pivoting strategy can be employed, and fillin determined during the solution of the matrix. The Markowitz ordering strategy selects pivots with the added constraint of minimizing fillin [12]. Additional discussions on the state of the literature for LU factorization are presented below.

We present a general sequential sparse factorization algorithm, in figure 3.1, based upon the factorization algorithms commonly attributed to Doolittle for matrices that do not require pivoting. In Doolittle factorization, all values on the diagonal of $\mathbf{L}$ equal 1 — $l_{kk} = 1$. We also present general sequential sparse forward reduction and backward substitution algorithms in figures 3.2 and 3.3 respectively that would be used in conjunction with the Doolittle-based algorithm to solve for $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$.

3.1.2 Choleski Factorization

If the matrix $\mathbf{A}$ is an $(N \times N)$ symmetric positive definite sparse matrix, then a special form of LU factorization can be used that exploits the symmetry and inherent numerical stability of this matrix form [12]. A symmetric positive definite sparse matrix $\mathbf{A}$ can be numerically factored into a single lower triangular matrix $\mathbf{L}$:

$$\mathbf{Ax} = \mathbf{LL}^T\mathbf{x} = \mathbf{b}, \tag{3.3}$$

Equation 3.3 is solved by setting $\mathbf{L}^T\mathbf{x} = \mathbf{y}$, and substituting $\mathbf{y}$ for $\mathbf{L}^T\mathbf{x}$. The numerical solution for $\mathbf{Ly} = \mathbf{b}$ is found by forward reduction, and the numerical solution for $\mathbf{x}$ is calculated

```
for $k = 1$ to $N$ /* for all elements along the diagonal */
    for each $i \in [k, N]$
        for each $j \in [1, k-1]$ such that $a_{ij} \neq 0$ and $a_{jk} \neq 0$
            $a_{ik} \leftarrow a_{ik} - (a_{ij} * a_{jk})$
        endfor
    endfor
    for each $i \in [k+1, N]$
        $a_{kj} \leftarrow (a_{kj} / a_{kk})$
    endfor
    for each $j \in [k+1, N]$
        for each $i \in [1, k-1]$ such that $a_{ki} \neq 0$ and $a_{ij} \neq 0$
            $a_{kj} \leftarrow a_{kj} - (a_{ki} * a_{ij})$
        endfor
    endfor
endfor
```

Figure 3.1: Sparse LU Factorization — Doolittle Algorithm

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

Figure 3.2: Sparse Forward Reduction for Doolittle Factorization

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

Figure 3.3: Sparse Backward Substitution for Doolittle Factorization

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

Figure 3.4: Sparse Column Choleski Factorization

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

We present a general sequential sparse factorization algorithm based upon the column Choleski factorization algorithm [29], which is similar to the factorization algorithms commonly attributed to Crout and Doolittle, and similar to the LU algorithm presented in figure 3.1. A sequential sparse factorization algorithm is presented in figure 3.4, and we present sequential sparse forward reduction and backward substitution algorithms for Choleski factorization in figures 3.5 and 3.6 respectively. In the backward substitution algorithm, the calculations are performed by implicitly transposing $\mathbf{L}$.

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

Figure 3.5: Sparse Forward Reduction for Choleski Factorization

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

Figure 3.6: Sparse Backward Substitution for Choleski Factorization

3.1.3 Ordering Sparse Matrices for Direct Methods

Position symmetric sparse matrices can be represented by graphs with elements in equations corresponding to undirected edges in the graph [12, 29]. The motivating applications for this research have position symmetric or position symmetric sub-matrices that are derived from power system networks that in turn, have graph representations. Ordering a symmetric sparse matrix modifies the order in which nodes are factored and is actually little more than changing the labels associated with nodes in an undirected graph; however, this simple task can drastically effect the amount of calculations involved when factoring a sparse matrix by affecting the amount of fillin. Fillin are those matrix locations with zeros as initial values that become non-zeros during factorization.

For symmetric positive definite matrices, there is much latitude in the order to perform the calculations, because there is no requirement for pivoting for numerical stability and the only effect of modifying the order of calculations might result from changes in round-off errors [29]. Diagonally dominant matrices also can be factored with little concern for pivoting, and there are many applications where time constraints are critical, so in order to speedup sparse LU factorization, pivoting is ignored. If there is no pivoting, ordering can be performed a priori and static data structures can be used for the most efficient sequential algorithm.

There is a graph-theoretical interpretation for fillin; factoring a node is equivalent to removing the node from the graph; however, any path through the factored node to adjacent edges must remain and must now be explicitly listed. This phenomenon is illustrated in figure 3.7 for a segment of a graph. In this example, the node with the least number of edges is selected for factoring, and two of three possible new edges are created. Only two new edges are created because there is an existing edge already connecting a pair of nodes. Fillin causes the number of edges in the remaining nodes to increase, often drastically increasing the number of calculations. The amount of fillin generated when any node is factored is bounded by the binomial coefficient of *the number of edges at a node choose 2* or

$$f_k \leq \begin{pmatrix} \nu_k \\ 2 \end{pmatrix} = \frac{\nu_k!}{2 \times (\nu_k - 2)!} = \frac{(\nu_k \times (\nu_k - 1))}{2}, \tag{3.4}$$

where:

f_k is the number of fillin when factoring node k.

ν_k is the number of edges at node k.

There are several notable techniques to minimize fillin, with one of the commonly used techniques being minimum-degree ordering. This ordering technique is used for position symmetric matrices and attempts to minimize fillin by selecting the next node for elimination that has the lowest degree or least number of connected edges. Minimum-degree ordering is closely related to Markowitz ordering [12] — both heuristics select the next row/column to eliminate that has the least row/column elements. The node-tearing-based ordering technique is utilized to generate block-diagonal-bordered form sparse matrices, and minimum degree ordering is used to minimize fillin locally in the mutually independent blocks and the borders. This technique is commonly referred to as *multiple minimum degree ordering* [13, 22]. Additional detail on minimum degree-based sparse matrix ordering is presented in appendix C.

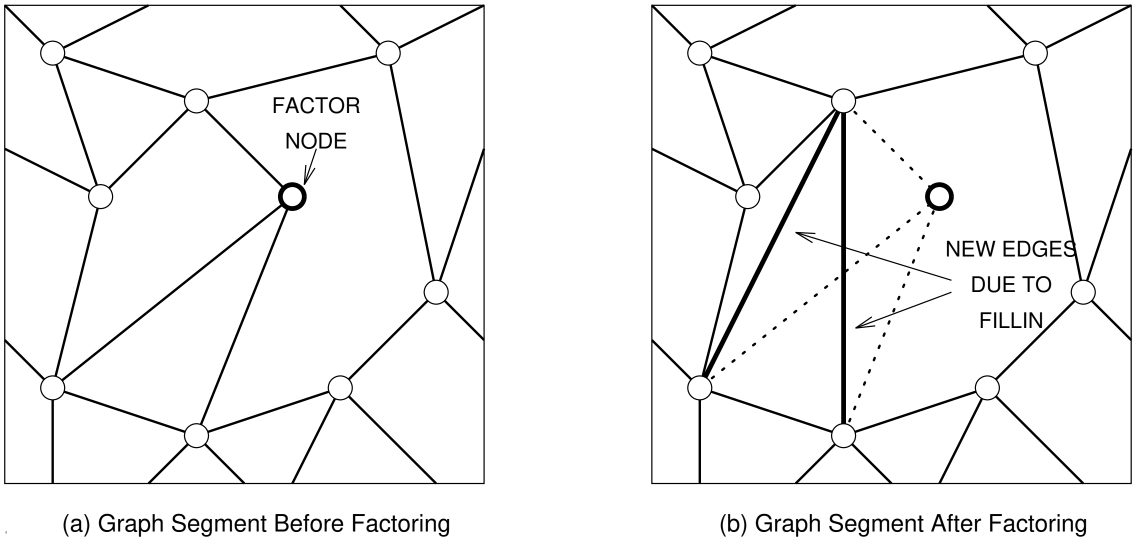
Modifying the ordering of a sparse matrix is simple to perform using a permutation matrix **P** of all zeros and ones that simply generates elementary row and column exchanges. Applying the permutation matrix **P** to the original linear system in equation 3.1 yields the linear system

$$(\mathbf{PAP}^T)(\mathbf{Px}) = (\mathbf{Pb}), \tag{3.5}$$

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

$$\bar{\mathbf{L}} = \mathbf{Pb}, \bar{\mathbf{U}}\mathbf{z} = \mathbf{y}, \mathbf{x} = \mathbf{P}^T\mathbf{z}. \tag{3.6}$$

For Choleski factorization, simply substitute $\bar{\mathbf{L}}^T$ for $\bar{\mathbf{U}}$ in equation 3.6. Also for Choleski factorization, as long as a symmetric positive definite matrix **A** is ordered with the permutation matrix **P**



(a) Graph Segment Before Factoring

(b) Graph Segment After Factoring

Figure 3.7: Graph Theoretical Explanation of Fillin

to $\mathbf{PAP}^T$, the resultant matrix after ordering remains symmetric positive definite.

3.1.4 A Survey of the Literature for Parallel Direct Methods

Significant research effort has been expended to examine parallel direct methods — for both dense and sparse matrices. Numerous papers have documented research on parallel dense matrix solvers [11, 60, 61], 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 [60, 61], nevertheless, performance is deterministic for:

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

Direct sparse matrix solvers, on the other hand, have computational complexity significantly less than $O(N^{2.0})$, and actual power system sparse matrices used in this work have orders of complexity less than $O(N^{1.5})$. These orders of complexity are consistent with matrices from circuit analysis applications that have complexities ranging from $O(N^{1.2})$ to $O(N^{1.5})$ [48]. With significantly less calculations than dense direct solvers, and lacking uniform, organized communications patterns, direct parallel sparse matrix solvers often require detailed knowledge of the application to permit efficient implementations.

The bulk of recent research into parallel direct sparse matrix techniques has centered around symmetric positive definite matrices, and implementations of Choleski factorization. A significant number of papers concerning parallel Choleski factorization for symmetric positive definite matrices have been published recently [19, 20, 21, 29]. 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 [46, 56, 57, 58].

Some of the most celebrated recent work has revived research into parallel sparse multifrontal Choleski techniques [25, 33]. Multifrontal techniques identify parallelism within the matrix structure in a manner similar to references [19, 20, 21, 29], but then create multiple small, dense matrices from independent rows/columns of data, and update each frontal matrix with dense techniques. Parallel sparse multifrontal algorithms have shown scalable performance for very-large, extremely regular sparse structural matrices. There has been some work on solving less-regular problems. Research has recently been published in [47] that describes load balancing techniques to support the work in [46]. Also, research has been ongoing to examine techniques that can efficiently factor irregular matrices using multifrontal techniques [8, 9, 10].

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 [44]. These techniques rely on regularity in the data to avoid processor load-imbalance.

Developing efficient parallel sparse matrix factorization algorithms requires more than just implementing parallel versions of sparse direct algorithms. All parallelism is identified in the structure of the sparse matrix, so before parallel factorization of the matrix can proceed, preprocessing of the matrix must occur. References [19, 20, 21, 29, 56, 57, 58] have utilized a general two step preprocessing paradigm for parallel sparse Choleski factorization:

1. order the matrix to minimize fillin,
2. symbolically factor the matrix to identify fillin and to identify static data structures.

In this paper, we break from this two step preprocessing paradigm and introduce a new three-step preprocessing phase that includes:

1. order the matrix,
2. pseudo-factor the matrix,

3. explicit load balance the matrix.

Pseudo-factorization is similar to the symbolic factorization step, although we require that the number of calculations in matrix partitions be calculated so that we can perform explicit load-balancing on the majority of the sparse matrix. Our three-step preprocessing phase is described in chapter 5.

This discussion is by no means an exhaustive literature survey, although it does represent a significant portion of the general direct sparse matrix research performed for vector and multi-processor computers.

3.2 Iterative Methods

We are considering an iterative solution to the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.7}$$

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.

3.2.1 Gauss-Seidel

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 calculated in the previous iteration, while Gauss-Seidel requires that the most recent values calculated be used in the present iteration. 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:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \tag{3.8}$$

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

or

$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \mathbf{L})^{-1}[\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)}], \tag{3.9}$$

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 strictly lower triangular portion of $\mathbf{A}$,
$\mathbf{U}$ is the strictly upper triangular portion of $\mathbf{A}$,
$\mathbf{b}$ is right-hand-side vector.

The representation in equation 3.8 is used in the development of the parallel algorithm, while the equivalent matrix-based representation in equation 3.9 is used below in discussions of available parallelism.

We present a general sequential sparse Gauss-Seidel algorithm in figure 3.8. Only non-zero values in $\mathbf{A}$ are used when calculating $\mathbf{x}^{(k+1)}$. 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.

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 [23] 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 — the rate of convergence is data dependent.

It may be possible to improve the convergence rate of iterative methods such as Gauss-Seidel by using preconditioning techniques such as incomplete LU factorization. This preconditioning technique performs operations similar to an LU factorization, but no calculations are performed that would generate fillin [23]. The use of preconditioners for parallel Gauss-Seidel algorithms raises many questions over and above the convergence performance improvement that may be possible for power

```
$\epsilon \leftarrow \infty$
while $\epsilon > \epsilon_{converge}$
    for $k = 1$ to $n_{iter}$
        for $i = 1$ to $N$
            $\tilde{x}_i \leftarrow x_i$
            $x_i \leftarrow b_i$
            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
    endfor
    $\epsilon \leftarrow 0$
    for $i = 1$ to $N$
        $\epsilon \leftarrow \epsilon + abs(\tilde{x}_i - x_i)$
    endfor
endwhile
```

Figure 3.8: Sparse Gauss-Seidel Algorithm

systems network matrices. These questions deal with compromises in available parallelism, effective load-balancing, and matrix ordering priorities between the two distinct algorithms. Implementation and testing of parallel preconditioners for parallel Gauss-Seidel linear solvers has not been examined in this work, although due to its potential impact on parallel iterative solver performance, we note that such algorithms should be examined in future research.

3.2.2 Ordering Sparse Matrices for Iterative Methods

Symmetric sparse matrices can be represented by graphs with elements in equations corresponding to undirected edges in the graph [29]. Ordering a symmetric sparse matrix modifies the order in which rows are solved and 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 3.1 yields the linear system

$$(\mathbf{P}\mathbf{A}\mathbf{P}^T)(\mathbf{P}\mathbf{x}) = (\mathbf{P}\mathbf{b}), \tag{3.10}$$

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 [23]. In section 7.2, 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.

3.2.3 A Survey of the Literature for Parallel Iterative Methods

Parallel implementations of Gauss-Seidel have generally been developed for regular problems such as the solution of Laplace's equations by finite differences [17, 23], 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 [23]; 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 [48], although this work showed poor parallel speedup potential in a theoretical study. Reference [48] 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 [63]. Other research with the parallel Gauss-Seidel methods for power systems applications is presented in [31], 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 [31]. Our work utilizes diakoptic-based matrix partitioning techniques developed initially for a parallel block-diagonal-bordered direct sparse linear solver [35, 36, 37, 39, 40]. In reference [37], we examined load balancing issues associated with partitioning power systems matrices for parallel Choleski factorization.

Chapter 4

Available Parallelism

The most significant aspect of parallel sparse LU factorization is that the sparsity structure can be exploited to offer more parallelism than is available with dense matrix solvers. Parallelism in dense matrix factorization is achieved by distributing the data in a manner that the calculations in one of the **for** loops can be performed in parallel. Sparse factorization algorithms have inadequate calculations in any row or column for efficient parallelism; however, sparse matrices offer additional parallelism as a result of the nature of the data and the precedence rules governing the order of calculations. Instead of just parallelizing a single **for** loop as in parallel dense matrix factorization, entire independent portions of a sparse matrix can be factored in parallel — especially when the sparse matrix has been ordered into block-diagonal-bordered form.

Provided that a matrix can be ordered into block-diagonal-bordered form, the parallel sparse LU algorithm can reap additional benefits, such as the elimination of task graphs for distributed-memory multi-processor implementations. Minimizing or eliminating task graphs is significant because the task graph can contain as much information as the representation of the sparse matrix for more conventional parallel sparse LU solvers [18].

The same independence between diagonal blocks in these sparse matrices can also be exploited in our parallel sparse Gauss-Seidel algorithm. 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. Furthermore, the extremely sparse last diagonal block also has inherent parallelism that can be identified by using graph multi-coloring techniques.

There are several distinct ways to illustrate available parallelism in block-diagonal-bordered form matrices. Available parallelism in a block-diagonal-bordered sparse matrix can be illustrated by the graph of the matrix. Figure 4.1 represents the form of a graph with four mutually independent submatrices (subgraphs) interconnected by shared coupling equations. No graph node in a subgraph has an interconnection to another subgraph except through the coupling equations. It should be intuitive that data in columns associated with nodes in subgraphs can be factored independently up to the point where the coupling equations are factored. The description of parallelism presented here for direct methods is also closely related to the concept of elimination graphs and super-nodes described in [29]. For the parallel Gauss-Seidel, a concept similar to elimination graphs could be utilized to depict the precedence in the calculations.

A block-diagonal-bordered form sparse matrix can be represented by an elimination or general precedence tree with supernodes at only two levels. Supernodes are collections of nodes in the elimination tree that are considered as a single entity, and form the elimination tree leaves, with another supernode as the root of the $N_{procs}ary$ tree. By simply restructuring the graph presented in figure 4.1, it is possible to represent the same concept as a tree. An elimination tree for a block-diagonal-bordered form matrix with four supernodes as leaves and a single supernode as the tree's root is presented in figure 4.2. Each leaf supernode will have an inherent hierarchy of calculations, that can be calculated independently of other leaf supernodes. The root supernode for factorization algorithms has little additional parallelism as a result of independent calculations, and would be represented generally as a vertical chain of nodes. This is due to fillin as a result of factorization. Conversely, the last block for the block-diagonal-bordered parallel Gauss-Seidel method represents only the interconnection structure within the equations that couple the partitions in the block-diagonal portion of the matrix.

For iterative methods, the reduced interconnectivity in the last diagonal block offers more available parallelism, but at the cost of requiring multiple iterations for change in the value of any $x_i^{(k)}$ to propagate throughout any interconnected values. Fillin in a factored matrix increases the amount of calculations, but any interconnection theoretically possible from the matrix graph representation has been considered. Unfortunately, there is little available parallelism inherent in factoring the last diagonal block, so pipelined techniques similar to those used for dense matrices are required for parallel factorization of the last diagonal block.

4.1 Available Parallelism in Block-Diagonal-Bordered Form Matrices for Direct Methods

While an elimination graph offers intuition into the available parallelism in block-diagonal-bordered sparse matrices for direct methods, it is possible to examine the theoretical mathematics of matrix

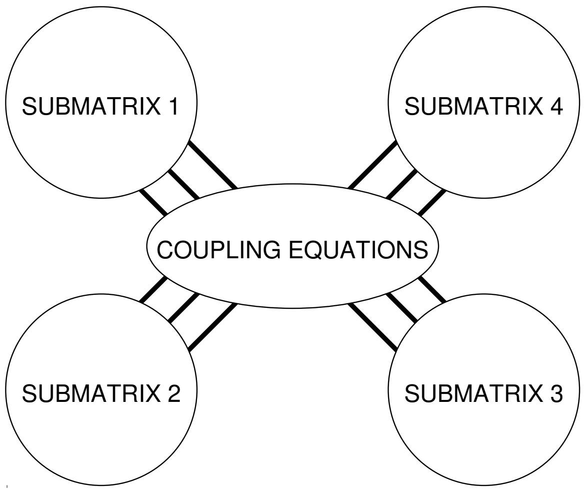


Figure 4.1: Graph with Four Independent Sub-Matrices

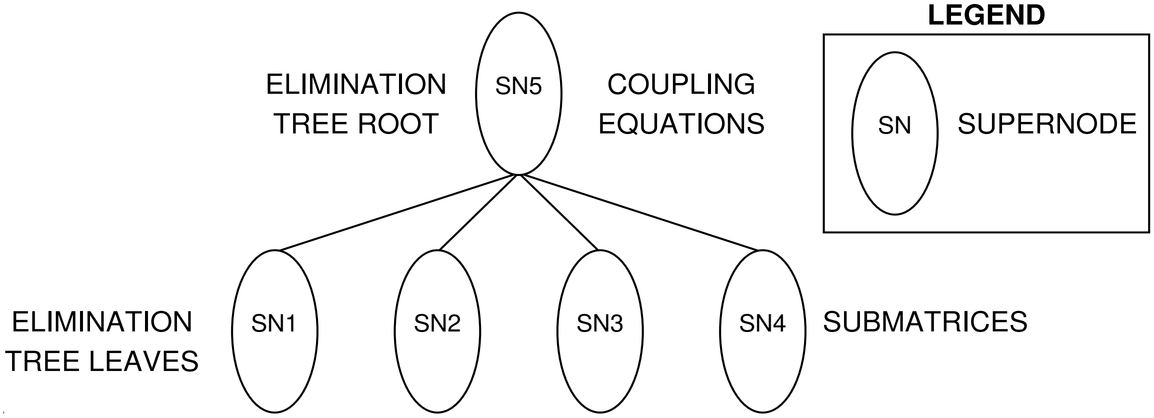


Figure 4.2: Elimination Tree with Four Supernode Leaves

partitioning to clearly identify available parallelism in this sparse matrix form. By partitioning the block-diagonal-bordered matrix into:

- a block-diagonal matrix,
- an upper border,
- a lower border,
- a last block,

and calculating the Shur complement [12], it is possible to identify available parallelism for direct methods by proving a theorem that states the LU factors of a block-diagonal-bordered matrix are also in block-diagonal-bordered form. A supporting lemma stating that the LU factors of a block-diagonal matrix are also block-diagonal form is required to complete the proof of the theorem. A similar version of this derivation can be used to identify the parallelism in Choleski factorization.

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

$$\mathbf{A} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{1,2} \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathcal{L}_{2,1} & \mathcal{L}_{2,2} \end{pmatrix} \begin{pmatrix} \mathcal{U}_{1,1} & \mathcal{U}_{1,2} \\ \mathbf{0} & \mathcal{U}_{2,2} \end{pmatrix} = \mathbf{LU} \tag{4.1}$$

where:

$\mathcal{A}_{1,1}$, $\mathcal{L}_{1,1}$, and $\mathcal{U}_{1,1}$ are of size $N_1 \times N_1$
$\mathcal{A}_{2,1}$ and $\mathcal{L}_{2,1}$ are of size $N_2 \times N_1$
$\mathcal{A}_{1,2}$ and $\mathcal{U}_{1,2}$ are of size $N_1 \times N_2$
$\mathcal{A}_{2,2}$, $\mathcal{L}_{2,2}$, and $\mathcal{U}_{2,2}$ are of size $N_2 \times N_2$.

The Shur complement of the partitioned matrices in equation 4.1 can be calculated by simply performing the matrix multiplication on the $\mathbf{LU}$ partitions which yields:

$$\mathbf{A} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{1,2} \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1}\mathcal{U}_{1,1} & \mathcal{L}_{1,1}\mathcal{U}_{1,2} \\ \mathcal{L}_{2,1}\mathcal{U}_{1,1} & \mathcal{L}_{2,1}\mathcal{U}_{1,2} + \mathcal{L}_{2,2}\mathcal{U}_{2,2} \end{pmatrix} \tag{4.2}$$

By equating blocks in equation 4.2, we can easily identify how to solve for the partitions:

$$\begin{aligned}
\mathcal{A}_{1,1} = \mathcal{L}_{1,1}\mathcal{U}_{1,1} \quad &\Rightarrow \quad \mathcal{L}_{1,1}\mathcal{U}_{1,1} = \mathcal{A}_{1,1} \\
\mathcal{A}_{1,2} = \mathcal{L}_{1,1}\mathcal{U}_{1,2} \quad &\Rightarrow \quad \mathcal{U}_{1,2} = \mathcal{L}_{1,1}^{-1}\mathcal{A}_{1,2} \\
\mathcal{A}_{2,1} = \mathcal{L}_{2,1}\mathcal{U}_{1,1} \quad &\Rightarrow \quad \mathcal{L}_{2,1} = \mathcal{A}_{2,1}\mathcal{L}_{1,1}^{-1} \\
\mathcal{A}_{2,2} = \mathcal{L}_{2,1}\mathcal{U}_{1,2} + \mathcal{L}_{2,2}\mathcal{U}_{2,2} \quad &\Rightarrow \quad \mathcal{L}_{2,2}\mathcal{U}_{2,2} = \mathcal{A}_{2,2} - \mathcal{L}_{2,1}\mathcal{U}_{1,2}
\end{aligned} \tag{4.3}$$

Before we can proceed and prove the theorem that the **LU** factors of a block-diagonal-bordered (BDB) position symmetric sparse matrix are also in block-diagonal-bordered form, we must define additional matrix partitions in the desired form and prove a Lemma that the **LU** factors of a block-diagonal (BD) matrix are also in block-diagonal form. At this point, we must define additional partitions of **A** that represent the block-diagonal-bordered nature of the original **A** matrix:

$$\mathbf{A}_{BDB} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{1,2} \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{1,1} & & \mathbf{0} & & \mathbf{A}_{1,m+1} \\ & \mathbf{A}_{2,2} & & & \mathbf{A}_{2,m+1} \\ \mathbf{0} & & \ddots & & \vdots \\ & & & \mathbf{A}_{m,m} & \mathbf{A}_{m,m+1} \\ \mathbf{A}_{m+1,1} & \mathbf{A}_{m+1,2} & \cdots & \mathbf{A}_{m+1,m} & \mathbf{A}_{m+1,m+1} \end{pmatrix} \tag{4.4}$$

$$\mathbf{L}_{BDB} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathcal{L}_{2,1} & \mathcal{L}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{1,1} & & & & \\ & \mathbf{L}_{2,2} & & & \mathbf{0} \\ \mathbf{0} & & \ddots & & \\ & & & \mathbf{L}_{m,m} & \\ \mathbf{L}_{m+1,1} & \mathbf{L}_{m+1,2} & \cdots & \mathbf{L}_{m+1,m} & \mathbf{L}_{m+1,m+1} \end{pmatrix} \tag{4.5}$$

$$\mathbf{U}_{BDB} = \begin{pmatrix} \mathcal{U}_{1,1} & \mathcal{U}_{1,2} \\ \mathbf{0} & \mathcal{U}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{U}_{1,1} & & & \mathbf{0} & \mathbf{U}_{1,m+1} \\ & \mathbf{U}_{2,2} & & & \mathbf{U}_{2,m+1} \\ \mathbf{0} & & \ddots & & \vdots \\ & & & \mathbf{U}_{m,m} & \mathbf{U}_{m,m+1} \\ & & & & \mathbf{U}_{m+1,m+1} \end{pmatrix} \tag{4.6}$$

$$\mathcal{A}_{1,1} = \mathbf{A}_{BD} = \begin{pmatrix} \mathbf{A}_{1,1} & & \mathbf{0} & \\ & \mathbf{A}_{2,2} & & \\ \mathbf{0} & & \ddots & \\ & & & \mathbf{A}_{m,m} \end{pmatrix} \tag{4.7}$$

$$\mathcal{A}_{1,2} = \begin{pmatrix} \mathbf{A}_{1,m+1} \\ \mathbf{A}_{2,m+1} \\ \vdots \\ \mathbf{A}_{m,m+1} \end{pmatrix} \tag{4.8}$$

$$\mathcal{A}_{2,1} = \begin{pmatrix} \mathbf{A}_{m+1,1} & \mathbf{A}_{m+1,2} & \cdots & \mathbf{A}_{m+1,m} \end{pmatrix} \tag{4.9}$$

$$\mathcal{A}_{2,2} = A_{m+1,m+1} \tag{4.10}$$

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

Proof:

Let:

$$\mathbf{A}_{BD} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathcal{L}_{2,1} & \mathcal{L}_{2,2} \end{pmatrix} \begin{pmatrix} \mathcal{U}_{1,1} & \mathcal{U}_{1,2} \\ \mathbf{0} & \mathcal{U}_{2,2} \end{pmatrix} = \mathbf{L}_{BD}\mathbf{U}_{BD} \tag{4.11}$$

By applying the Shur complement to equation 4.11, we obtain:

$$\mathcal{A}_{1,2} = \mathcal{L}_{1,1}\mathcal{U}_{1,2} = \mathbf{0} \Rightarrow \mathcal{U}_{1,2} = \mathcal{L}_{1,1}^{-1}\mathbf{0} = \mathbf{0} \tag{4.12}$$

and

$$\mathcal{A}_{2,1} = \mathcal{L}_{2,1}\mathcal{U}_{1,1} = \mathbf{0} \Rightarrow \mathcal{L}_{2,1} = \mathbf{0}\mathcal{U}_{1,1}^{-1} = \mathbf{0} \tag{4.13}$$

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

$$\mathbf{A}_{BD} = \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{1,2} \\ \mathcal{A}_{2,1} & \mathcal{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathcal{L}_{2,2} \end{pmatrix} \begin{pmatrix} \mathcal{U}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathcal{U}_{2,2} \end{pmatrix} = \mathbf{L}_{BD}\mathbf{U}_{BD} \tag{4.14}$$

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

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

Proof:

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

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

1. Diagonal Blocks: $\mathcal{A}_{1,1} = \mathcal{L}_{1,1}\mathcal{U}_{1,1} \Rightarrow \begin{cases} \mathbf{A}_{1,1} = \mathbf{L}_{1,1}\mathbf{U}_{1,1} \\ \mathbf{A}_{2,2} = \mathbf{L}_{2,2}\mathbf{U}_{2,2} \\ \vdots \end{cases}$,
2. Lower Border: $\mathcal{A}_{2,1} = \mathcal{L}_{2,1}\mathcal{U}_{1,1} \Rightarrow \begin{cases} \mathbf{A}_{m+1,1} = \mathbf{L}_{m+1,1}\mathbf{U}_{1,1} \\ \mathbf{A}_{m+1,2} = \mathbf{L}_{m+1,2}\mathbf{U}_{2,2} \\ \vdots \end{cases}$,
3. Upper Border: $\mathcal{A}_{1,2} = \mathcal{U}_{1,2}\mathcal{L}_{1,1} \Rightarrow \begin{cases} \mathbf{A}_{1,m+1} = \mathbf{L}_{1,1}\mathbf{U}_{1,m+1} \\ \mathbf{A}_{2,m+1} = \mathbf{L}_{2,2}\mathbf{U}_{2,m+1} \\ \vdots \end{cases}$,

4. Last Block:

$$\mathcal{A}_{2,2} - \mathcal{L}_{2,1}\mathcal{U}_{1,2} = \mathcal{L}_{2,2}\mathcal{U}_{2,2} \Rightarrow \left\{ \mathbf{A}_{m+1,m+1} - \sum_{i=1}^{m} \mathbf{L}_{m+1,i}\mathbf{U}_{i,m+1} = \mathbf{L}_{m+1,m+1}\mathbf{U}_{m+1,m+1} \right.$$

.

If the matrix blocks $\mathbf{A}_{i,i}$, $\mathbf{A}_{m+1,i}$, and $\mathbf{A}_{i,m+1}$ $(1 \leq i \leq m)$ are assigned to the same processor, then there are *no* communications until the last block is factored. At that time, only sums of sparse matrix × sparse matrix products are sent to the processors that hold the appropriate data in the last block.

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

Figure 4.3 illustrates both the LU factorization steps and the reduction/substitution steps for a block-diagonal-bordered sparse matrix. In this figure, the strictly lower diagonal portion of the matrix is $\mathbf{L}$, and the strictly upper diagonal portion of the matrix is $\mathbf{U}$. This figure depicts four diagonal blocks, and processor assignments (P1, P2, P3, and P4) are listed with the data block. This figure would represent the block-diagonal-bordered form matrix and data distribution for the data represented in figures 4.1 and 4.2.

4.2 Available Parallelism in Block-Diagonal-Bordered Form Matrices for Iterative Methods

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 sharing 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 derive the formulation of the Gauss-Seidel algorithm for a block-diagonal-bordered matrix, the optimum data/processor assignments for an efficient parallel implementation are straightforward.

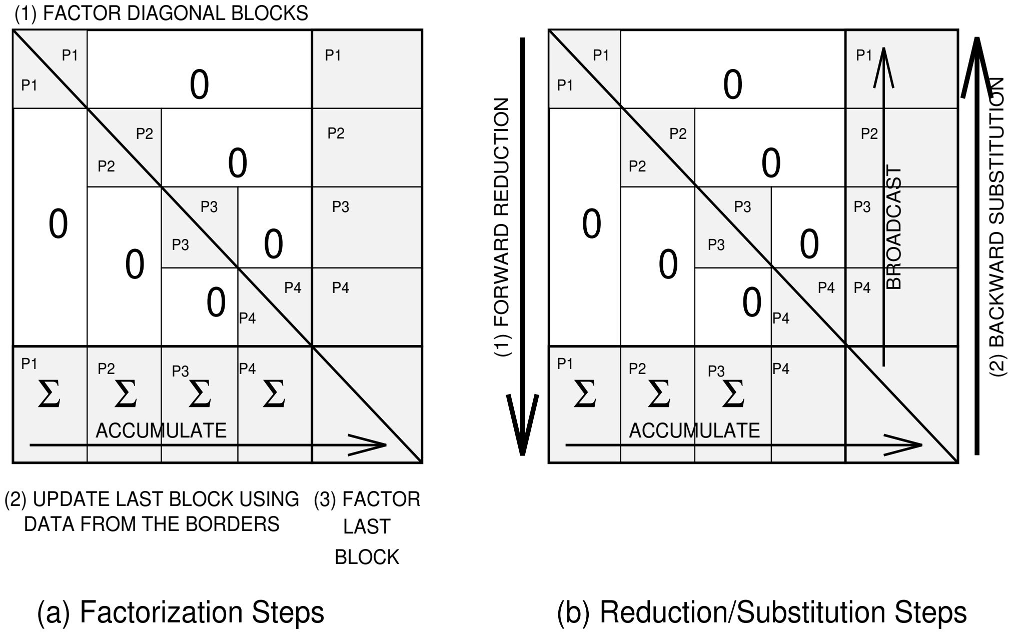

(1) FACTOR DIAGONAL BLOCKS
P1
P2
P3
P4
0
Σ
ACCUMULATE
(2) UPDATE LAST BLOCK USING DATA FROM THE BORDERS
(3) FACTOR LAST BLOCK
(1) FORWARD REDUCTION
BROADCAST
(2) BACKWARD SUBSTITUTION
(a) Factorization Steps
(b) Reduction/Substitution Steps

Figure 4.3: Block Bordered Diagonal Form Sparse Matrix Solution Steps

While much of the parallelism in this algorithm is made clearly visible as a result of 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, and there is no precedence when performing the calculations. 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.

4.2.1 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 3.9, 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.

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{0} & & \mathbf{A}_{1,m+1} \\ \mathbf{0} & \ddots & & \vdots \\ & & \mathbf{A}_{m,m} & \mathbf{A}_{m,m+1} \\ \mathbf{A}_{m+1,1} & \cdots & \mathbf{A}_{m+1,m} & \mathbf{A}_{m+1,m+1} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1^{(k)} \\ \vdots \\ \mathbf{x}_m^{(k)} \\ \mathbf{x}_{m+1}^{(k)} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_m \\ \mathbf{b}_{m+1} \end{pmatrix}. \tag{4.15}$$

Equation 3.9 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:

$$\mathbf{PAP}^T = \mathbf{D} + \mathbf{L} + \mathbf{U} \tag{4.16}$$

Derivation of the block-diagonal-bordered form of the $\mathbf{D}$, $\mathbf{L}$, and $\mathbf{U}$ matrices is straightforward. Equation 3.9 requires the calculation of $(\mathbf{D} + \mathbf{L})^{-1}$, which is simple to determine explicitly, because this matrix has block-diagonal-lower-bordered form. The diagonals in $(\mathbf{D} + \mathbf{L})^{-1}$ are of the form:

$$(\mathbf{D} + \mathbf{L})^{-1}_{i,i} = (\mathbf{D}_{i,i} + \mathbf{L}_{i,i})^{-1}, i = 1, \cdots, m+1 \tag{4.17}$$

and the only other non-zero terms are in the last row. These values are of the form:

$$(\mathbf{D} + \mathbf{L})^{-1}_{m+1,i} = (\mathbf{D}_{m+1,i} + \mathbf{L}_{m+1,i})^{-1}(-\mathbf{L}_{m+1,i})(\mathbf{D}_{i,i} + \mathbf{L}_{i,i})^{-1}, i = 1, \cdots, m. \tag{4.18}$$

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:

$$\mathbf{x}^{(k+1)} = \begin{pmatrix} (\mathbf{D}_{1,1} + \mathbf{L}_{1,1})^{-1} \left[\mathbf{b}_1 - \mathbf{U}_{1,1}\mathbf{x}_1^{(k)} - \mathbf{U}_{1,m+1}\mathbf{x}_{m+1}^{(k)}\right] \\ \vdots \\ (\mathbf{D}_{m,m} + \mathbf{L}_{m,m})^{-1} \left[\mathbf{b}_m - \mathbf{U}_{m,m}\mathbf{x}_m^{(k)} - \mathbf{U}_{m,m+1}\mathbf{x}_{m+1}^{(k)}\right] \\ (\mathbf{D}_{m+1,m+1} + \mathbf{L}_{m+1,m+1})^{-1} \left[\mathbf{b}_{m+1} - \sum_{i=1}^{m}(\mathbf{L}^{-1}_{m+1,i}\mathbf{x}_i^{(k+1)}) - \mathbf{U}_{m+1,m+1}\mathbf{x}_{m+1}^{(k)}\right] \end{pmatrix}. \tag{4.19}$$

We can identify the parallelism in the block-diagonal-bordered portion of the matrix by examining equation 4.19. 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 previous 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 × vector products, calculated in parallel, are involved in the communications phase. Furthermore, if we assign

$$\hat{\mathbf{b}} = \mathbf{b}_{m+1} - \sum_{i=1}^{m} \left(\mathbf{L}_{m+1,i}^{-1} \mathbf{x}_i^{(k+1)} \right) , \tag{4.20}$$

then the formulation of $\mathbf{x}_{m+1}^{(k+1)}$ looks similar to equation 3.9:

$$\hat{\mathbf{x}}^{(k+1)} = \mathbf{x}_{m+1}^{(k+1)} = (\mathbf{D}_{m+1,m+1} + \mathbf{L}_{m+1,m+1})^{-1} \left[\hat{\mathbf{b}} - \mathbf{U}_{m+1,m+1}\mathbf{x}^{(k)} \right] . \tag{4.21}$$

4.2.2 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 — nodes with the same color are independent and can be solved in parallel. We define the sub-partitioning as:

$$\mathbf{A}_{m+1,m+1} = \begin{pmatrix} \hat{\mathbf{D}}_{1,1} & \hat{\mathbf{A}}_{1,2} & \cdots & \hat{\mathbf{A}}_{1,c} \\ \hat{\mathbf{A}}_{2,1} & \hat{\mathbf{D}}_{2,2} & \cdots & \hat{\mathbf{A}}_{2,c} \\ \vdots & & \ddots & \vdots \\ \hat{\mathbf{A}}_{c,1} & \hat{\mathbf{A}}_{c,2} & \cdots & \hat{\mathbf{D}}_{c,c} \end{pmatrix} . \tag{4.22}$$

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:

$$\hat{\mathbf{x}}^{(k+1)} = \begin{pmatrix} \hat{\mathbf{x}}_1^{(k+1)} \\ \hat{\mathbf{x}}_2^{(k+1)} \\ \vdots \\ \hat{\mathbf{x}}_c^{(k+1)} \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{D}}_{1,1}^{-1} \left[\hat{\mathbf{b}}_1 - \sum_{j>1} \hat{\mathbf{A}}_{1,j} \hat{\mathbf{x}}_j^{(k)} \right] \\ \hat{\mathbf{D}}_{2,2}^{-1} \left[\hat{\mathbf{b}}_2 - \sum_{j<2} \hat{\mathbf{A}}_{2,j} \hat{\mathbf{x}}_j^{(k+1)} - \sum_{j>2} \hat{\mathbf{A}}_{2,j} \hat{\mathbf{x}}_j^{(k)} \right] \\ \vdots \\ \hat{\mathbf{D}}_{c,c}^{-1} \left[\hat{\mathbf{b}}_c - \sum_{j<c} \hat{\mathbf{A}}_{c,j} \hat{\mathbf{x}}_j^{(k+1)} \right] \end{pmatrix} . \tag{4.23}$$

Calculating $\hat{\mathbf{x}}_i^{(k+1)}$ in each sub-partition of $\hat{\mathbf{x}}^{(k+1)}$ does not require other 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 that require them, 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 $\hat{\mathbf{x}}^{(k+1)}$ is calculated. At

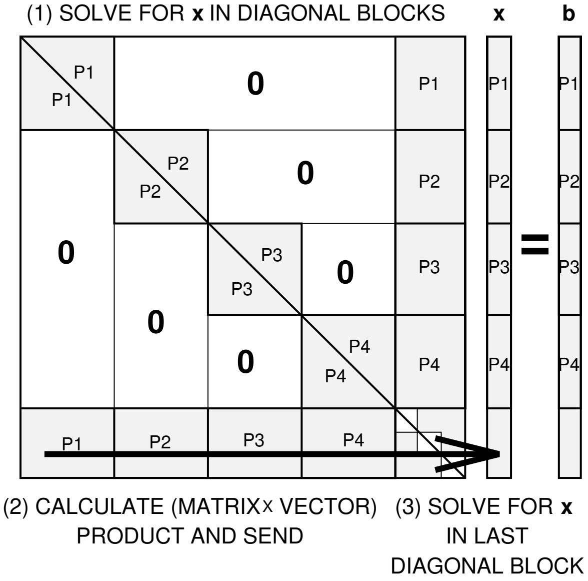
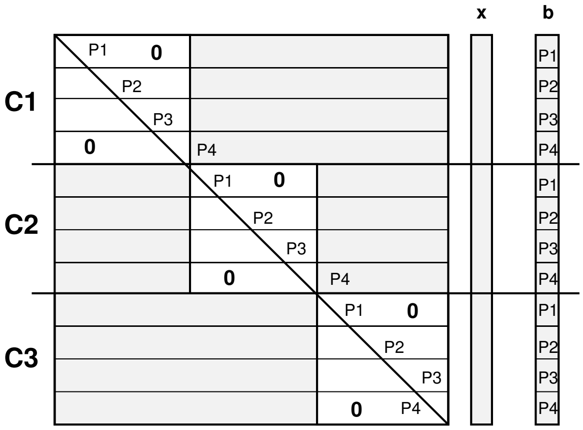

(1) SOLVE FOR x IN DIAGONAL BLOCKS
x
b
P1
P2
P3
P4
0
=
(2) CALCULATE (MATRIX× VECTOR) PRODUCT AND SEND
(3) SOLVE FOR x IN LAST DIAGONAL BLOCK

Figure 4.4: Block-Bordered-Diagonal Form Gauss-Seidel Method

that time, only matrix × vector products are sent to the processors that hold the appropriate data in the last diagonal block. Figure 4.4 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 4.5 illustrates the data/processor assignments in the last diagonal block.



(1) SOLVE FOR **x** WITHIN A COLOR

(2) BROADCAST NEW **x** VALUES

Figure 4.5: Multi-Colored Gauss-Seidel Method for the Last Diagonal Block

Chapter 5

The Preprocessing Phase

In the previous chapter, we developed the theoretical foundations for parallel sparse block-diagonal-bordered linear solvers, and we now will discuss the procedures required to generate the permutation matrices, **P**, to produce block-diagonal-bordered/multi-colored sparse matrices so that our parallel algorithms are efficient. To reach this goal, we must ensure that the parallel algorithms are not overwhelmed by load-imbalance-overhead and are as efficient as possible on an architecture. We must reiterate that all parallelism for our linear solvers is identified from the power systems network structure during this preprocessing phase. The specifics for the direct and iterative solvers are discussed separately below.

The preprocessing phase may be incorporated into an optimization framework that is used to produce matrix orderings with optimal overall performance for a particular version of the block-diagonal-bordered sparse matrix factorization algorithm. For this research, the optimization has been performed by hand. We have examined ordered matrices resulting from various values of the input parameter for the node-tearing algorithm, and we have identified the block-diagonal-bordered form sparse matrix with the best empirical run-time performance for each of five sample power system networks. These results are presented in sections 7.1 and 7.2.

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. The times for solving the linear power systems network matrices in parallel are measured in fractions of a second to seconds; however, the times for the preprocessing stage are at least an order of magnitude greater. The preprocessing phase for direct methods takes longer than the preprocessing phase for iterative methods, due to requirements for multiple minimum degree ordering. The preprocessing phase produces a permutation matrix with which to order the actual matrix, and defines what rows/columns are placed on what processors. The permutation matrix can be calculated with no more information than the graph interconnectivity of the power systems networks, which represent actual power systems generators, power lines, and distribution substations. The infrastructure underlying the power systems network matrices changes infrequently, so the effort to perform a single ordering can be amortized over days or even over weeks.

5.1 The Preprocessing Phase for Direct Methods

For parallel sparse block-diagonal-bordered direct linear solvers to be efficient when factoring irregular sparse matrices, the following three step preprocessing phase is required:

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

The first step determines the block-diagonal-bordered form and the ordering of nodes within diagonal blocks to minimize calculations; the second step determines the locations of fillin values for static data structures and also determines the number of calculations in independent blocks for the load balancing step; and the third step determines a mapping of data to processors for efficient implementation of the algorithm for the user specified data.

5.1.1 Ordering

The preprocessing phase ordering step must identify diagonal matrix blocks while also attempting to minimize the amount of fillin during factorization. Few matrices can be readily ordered into block-diagonal-bordered form with equal workload in each block. The exception to this rule are highly regular matrices from the structural analysis community, where the nested dissection ordering algorithm can produce balanced block-diagonal-bordered matrices [29]. Recursive spectral bisection can be used to partition irregular matrices [3, 45, 50], and subsequently, the coupling equations can be extracted. Unfortunately, this technique, as well as nested dissection, relies on dividing the matrix into equal sized partitions, without considering the number of coupling equations or considering the number of calculations in each diagonal block. Load-imbalance limits the potential for using recursive spectral bisection, because the number of calculations in a block for factorization or forward reduction/backward substitution are related to the sparsity of the subgraph, which can vary significantly for irregular power systems network matrices [37].

A third method to order a sparse matrix into block-diagonal-bordered form is referred to as node-tearing [12, 49], which is a specialized form of diakoptics [26]. This technique attempts to extract the natural structure in the matrix or graph, and generally produces many irregularly sized blocks, while minimizing the number of coupling equations or the size of the lower border and last diagonal block. Additional detail on the node-tearing form of diakoptics is presented in appendix B. Load balancing techniques must be used after the node-tearing matrix ordering step to distribute the processing load uniformly onto a multi-processor. As shown in chapter 4, diagonal blocks can be assigned to any processor without requirements for interprocessor communications to factor the diagonal block and associated portion of the lower border.

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

5.1.2 Pseudo-Factorization

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 multicomputer. 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 [21, 29, 56]. The number of floating point operations may not be proportional to the number of equations assigned to a processor because the number of calculations in an independent subgraph is a function of the number and location of non-zero matrix elements in that block — not the number of equations in a block. For dense matrices, the computational complexity of factorization is $O(N^3)$. Meanwhile, the computational complexity for factoring entire sparse power systems matrices used in later parallel algorithm performance studies is less than $O(N^{1.5})$, but greater than $O(N)$. Determining the actual workload requires a detailed simulation of all processing for the factorization phases, which we refer to as pseudo-factorization.

Pseudo-factorization is merely a replication of the numerical factorization process without actually performing the calculations. Numbers of calculations to factor the independent data blocks and numbers of calculations to update the last block using data from the borders are tallied.

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. Efficient parallel sparse matrix solvers require that any disparities in processor workloads be minimized in order to minimize load imbalance overhead, and consequently, to maximize processor utilization.

5.1.3 Load Balancing

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

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

Load balancing, as implemented for factorization of the diagonal blocks and the lower border, emphasizes the uniform distribution of the processing workload in the first two steps. The second factorization step, updating the last block using data in the borders, requires that the results of sparse matrix × sparse matrix products be sent to the processor that holds the data for an element in the last block. The independent nature of calculations in the diagonal blocks and the border permit a processor to start the second phase as soon as that processor has completed factoring the independent blocks. Consequently, the sum total of all calculations in the diagonal blocks and corresponding border sub-matrices can be used when load balancing. The parallel calculations in the last diagonal block are performed using a pipelined blocked kji-saxpy LU algorithm that does not require explicit load-balancing [61].

The metrics we use consider only the number of floating point operations and do not consider indexing overhead, which can be rather extensive when sparse matrices are stored in an implicit form. The data structure used in our solver has explicit links between non-zero values in a column and stores the data in any row as a sparse vector. This data structure should minimize indexing overhead at the cost of additional memory required to store the sparse matrix when compared with other sparse data storage methods [13]. The implementation of the parallel block-diagonal-bordered LU solver is discussed in greater detail in chapter 6.

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

5.2 The Preprocessing Phase for Iterative Methods

For parallel sparse block-diagonal-bordered iterative methods to be efficient when solving irregular sparse matrices, we must:

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.

After performing the first ordering step, we must:

- *pseudo-solve* to identify the workload for each diagonal block and corresponding portion of the borders,
- *load balance* to distribute the workload uniformly among processors.

As with the preprocessing step for direct methods, the metric for load balancing 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 a matrix partition is a function of the number of non-zero matrix elements in that block — not the number of equations in a block. For dense matrices, the computational complexity of the Gauss-Seidel algorithm is $O(N^2)$; however, the computational complexity of calculating $\mathbf{x}^{(k+1)}$ for sparse power systems matrices is only slightly greater than $O(N^1)$. Determining the actual workload requires calculating the number of multiply/addition operations in each matrix block, which we refer to as the pseudo-solution.

When performing the second ordering step, we must consider that there are a limited number of calculations in the last diagonal block and communications requirements for this portion of the algorithm are extensive. In the worst case, all values calculated on a processor must be broadcast to all other processors — resulting in significant communications overhead. Consequently, we load-balance this ordering step as a function of the amount of communications, and make this load balancing step integral with the graph multi-coloring algorithm.

5.2.1 Ordering

As with ordering matrices into block-diagonal-bordered form for direct methods, we are looking for an ordering technique that produces a permutation matrix **P** that transforms the irregular power systems matrix into block-diagonal-bordered form while balancing the workload in the diagonal blocks and also limiting the number of coupling equations in the last diagonal block. Minimizing the size of the last diagonal block in a parallel block-diagonal-bordered sparse matrix minimizes the amount of communications; however, we illustrate in section 7.2 with empirical data that there are trade-offs with minimizing the number of equations in the last diagonal block and balancing the workload in the block-diagonal portion of the matrix. If the size of the last block of the matrix can be adequately constrained, the amount of communications can be drastically reduced. When determining the optimal ordering for a sparse matrix, an ordering that yields a better load-balance in the highly parallel portion of the calculations may be traded for the size of the last diagonal block and the subsequent additional communications.

The method we have chosen to order the sparse power systems networks into block-diagonal-bordered form matrices for parallel Gauss-Seidel is a specialized form of diakoptics, referred to as node-tearing [12, 26, 49]. We have performed an analysis similar to that described above to order power systems matrices into block-diagonal-bordered form for direct methods, and have found that node-tearing nodal analysis has features that make it superior to recursive spectral bisection and nested dissection — techniques that attempt to load-balance on the number of rows/columns in a partition, rather than the actual workload in the irregular matrix. The node-tearing form of diakoptics analysis attempts to extract the natural structure in the matrix or graph, and generally produces many irregularly sized blocks, while minimizing the size of the lower border and last diagonal block. Additional detail on the node-tearing form of diakoptics is presented in appendix B. Load balancing techniques must be used in a separate step after the node-tearing ordering step to distribute the processing load uniformly onto a multi-processor.

When ordering power systems networks into block-diagonal-bordered form for the parallel Gauss-Seidel method, there is no concern for additional ordering of the diagonal blocks to minimize fillin, because iterative methods do not generate these additional non-zero values. However, additional ordering is required for the last diagonal block, because without ordering, the calculations in this portion of the matrix would be purely sequential, limiting the potential speedup of the algorithm in accordance to Amdahl's law. The last diagonal block represents the interconnection structure within the equations that couple the partitions found in the node-tearing-based ordering 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 $\hat{\mathbf{x}}^{(k+1)}$ 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, and barriers to synchronize parallel operations are only required between graph colors.

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 load-balancing for communications. The graph multi-coloring technique is discussed in greater detail in appendix D.

5.2.2 Pseudo-Solution

Efficient parallel sparse matrix solvers require that any disparities in processor workloads be minimized in order to minimize load imbalance overhead, and consequently, to maximize processor utilization. The metric for performing load balancing must be based on the actual workload required by the processors in a distributed-memory multi-computer. The metric employed when load-balancing the partitions for parallel Gauss-Seidel, is the number of floating point multiply/add operations, not simply the number of rows per partition. Determining the number of floating point operations in each partition for the parallel Gauss-Seidel is simple when compared to the pseudo-factorization required for parallel direct methods. Pseudo-solving for an iteration 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. It simply requires that the number of off-diagonal non-zero elements in all rows within a partition be summed. While this step is simple and can be performed with significantly less effort than pseudo-factorizations for parallel direct methods, it is an essential input to the load-balancing step for the block-diagonal portion of the matrix.

5.2.3 Load Balancing

The load balancing step of the preprocessing phase can be performed with the same simple pigeon-hole type algorithm described above for direct methods in section 5.1.3. The load-balancing algorithm uses a metric based on the numbers of floating point multiply/add operations in a partition, not simply the number of rows per partition. The independent nature of calculations in the diagonal blocks and the border permit a processor to start updating the last diagonal block as soon as that processor has completed calculating $\mathbf{x}^{(k+1)}$ in the diagonal blocks. The parallel calculations in the last diagonal block are load balanced separately, by dividing the rows within colors evenly among processors to minimize communications overhead. These metrics do not consider indexing overhead, which can be rather extensive when working with very sparse matrices stored in an implicit form.

We have found that load-balancing for the parallel block-diagonal-bordered Gauss-Seidel algorithm has been easier to perform than load-balancing for direct methods, because workload per partition has less variability for iterative methods than direct methods. Workloads within a partition for direct methods have a higher computational complexity than for iterative methods, so for the same partitioned matrix, there may be significantly different distributions of data to processors for the two methods.

Chapter 6

Parallel Sparse Matrix Solver Implementations

This chapter is divided into three sections. In the first section we describe methods to improve performance of the sequential portions of the linear solver implementations. In the other two sections, we describe in detail the implementations of the parallel sparse matrix solvers. Pseudo-code descriptions of the parallel algorithms are presented in appendix E.

The parallel implementation presented in this chapter has been developed as an instrumented proof-of-concept to examine the efficiency of each section of the algorithm. The host processor is used to gather and tabulate statistics on the multi-processor calculations. Statistics are gathered in a manner that do not impact the total empirical measures of timing data for factorization, forward reduction, or backward substitution in the implementation of the direct solvers, nor do statistics impact the total measures of performance for the parallel Gauss-Seidel.

6.1 Sequential Code Optimization

In this chapter, we describe the implementations of the parallel block-diagonal-bordered sparse matrix solvers that we have developed for the Thinking Machines CM-5 distributed-memory multiprocessor. All implementations have been developed with special concern for sequential code optimization, in addition to optimization of the more complex parallel code. Paramount to sequential code optimization is matching the data structures to the algorithm in order to maximize cache hits. This can be accomplished by attempting to always perform operations on *vectors* of consecutively stored data. The algorithm for the iterative Gauss-Seidel method makes this easy. The Gauss-Seidel algorithm is essentially a matrix × vector product with multiple vector × vector products as new

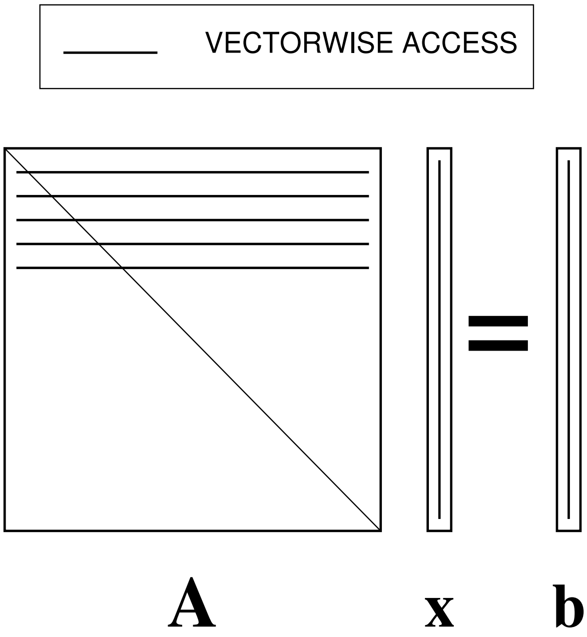


Figure 6.1: Optimal Row-Major Data Storage for Gauss-Seidel Algorithms

values of $x_i^{(k+1)}$ are calculated for the i^{th} row. The sparse implementation of Gauss-Seidel requires the calculation of a sparse matrix × a dense vector product. Figure 6.1 illustrates the optimal way to perform the matrix × vector product, with the (sparse) matrix stored in row-major form. In this figure, long horizontal lines depict sparse vectors and the vertical line representing $\mathbf{x}$ is a dense vector.

Algorithms for direct linear solvers have much more complicated access patterns. For dense direct methods, normal storage of data in conventional two-dimensional matrices always leaves the data in a pattern that is not accessible in one direction as a vector, and unless the software is written to optimize over strides in the matrix, there may be cache hit problems when performing dense factorization. Figure 6.2 illustrates four possible dense factorization implementations, two each for column-major storage and row-major storage [30]. In this figure, long lines depict dense vectors and dots represent scalar access.

For sparse direct linear solvers, we store the sparse matrices implicitly, providing rich opportunities for improving sequential program cache hits. It is simple to store data with separate data structures for the diagonal, lower triangular, and upper triangular portions of the matrix and have sparse vector × sparse vector products throughout the factorization calculations. This is illustrated

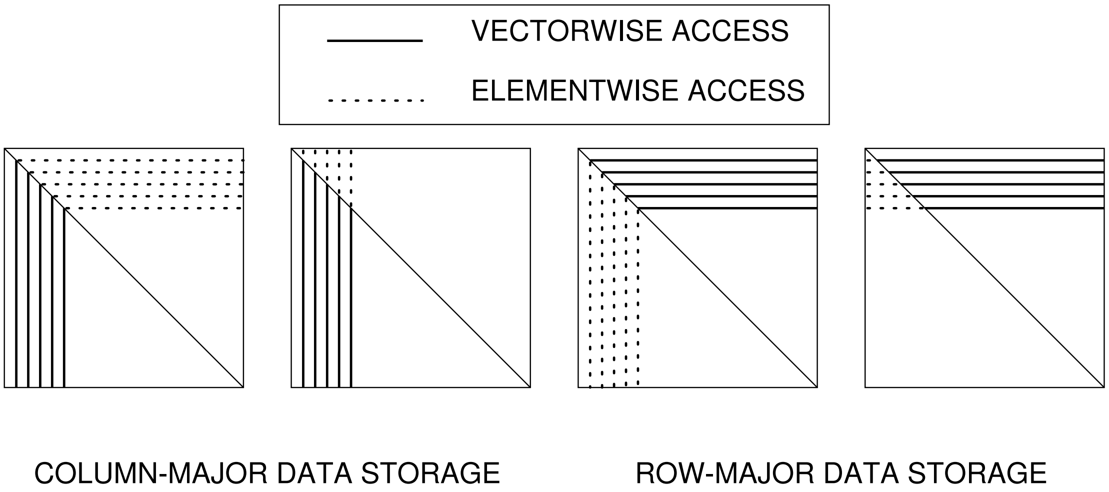


Figure 6.2: Optimal Data Storage for Dense Factorization

in figure 6.3 for a factorization algorithm with lower triangular matrix data stored in row-major order and upper triangular matrix storage in column-major order. Care must be taken to use forward reduction and backward substitution algorithms compatible with the data structures that optimize performance. This is easy to perform, because the loops to perform the triangular solutions can be performed in either order.

Now we examine methods to optimize parallel direct and parallel iterative solver implementations.

6.2 Parallel Sparse Direct Solver Implementations

We have implemented a parallel version of a block-diagonal-bordered sparse LU solver and a similar Choleski solver in the C programming language for the Thinking Machines CM-5 multi-computer using message passing and a host-node paradigm. In order to have an implementation with extremely low-latency communications, we utilized the Connection Machine active message layer (CMAML) remote procedure call features [53, 59]. We also implemented versions of the algorithms using non-blocking, buffered communications. Substantial improvements in the performance of the algorithm have been observed for low-latency active messages, when compared to more traditional communications paradigms that use non-blocking communications functions in conjunction with packing data into communications buffers. Throughout this discussion of parallel direct sparse solvers, the active message communications paradigm is the means with which we implemented low-latency communications on the Thinking Machines CM-5.

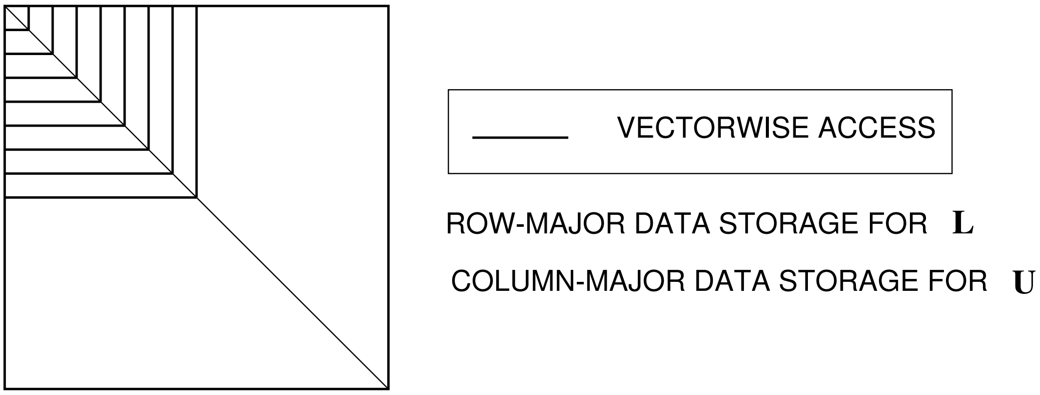


Figure 6.3: Optimal Data Storage for Sparse Factorization — Doolittle's Algorithm

A version of the software is available that runs on a single processor on the CM-5 to provide empirical speed-up data to quantify multi-processor performance. Empirical performance data has been gathered for a range of numbers of processors with real power systems sparse network matrices. Results based on empirical data collected in benchmarking trials are presented in the next chapter. Our block-diagonal-bordered sparse direct solvers have the following distinct segments which were derived in chapter 4:

1. LU factorization —
   - factor the mutually independent diagonal blocks and associated portions of the border — $\mathbf{A}_{i,i} = \mathbf{L}_{i,i}\mathbf{U}_{i,i}$, $\mathbf{A}_{m+1,i} = \mathbf{L}_{m+1,i}\mathbf{U}_{i,i}$ and $\mathbf{A}_{i,m+1} = \mathbf{L}_{i,i}\mathbf{U}_{i,m+1}$ for $(1 \leq i \leq m)$,
   - update the last diagonal block using the data in the borders — $\mathbf{A}_{m+1,m+1} = \mathbf{A}_{m+1,m+1} - \sum_{i=1}^{m} \mathbf{L}_{m+1,i}\mathbf{U}_{i,m+1}$,
   - factor the last diagonal block — $\mathbf{A}_{m+1,m+1} = \mathbf{L}_{m+1,m+1}\mathbf{U}_{m+1,m+1}$,
2. forward reduction —
   - calculate the $\mathbf{y}$ vector partition corresponding to the mutually independent diagonal blocks — $\mathbf{y}_i$ for $(1 \leq i \leq m)$,
   - update the $\mathbf{b}$ vector partition corresponding to the last diagonal block — $\mathbf{b}_{m+1} = \mathbf{b}_{m+1} - \sum_{i=1}^{m} \mathbf{y}_i\mathbf{L}_{m+1,i}$,
   - calculate the $\mathbf{y}$ vector partition corresponding to the last diagonal block — $\mathbf{y}_{m+1}$,
3. backward substitution —

- calculate the $\mathbf{x}$ vector partition corresponding to the last diagonal block — $\mathbf{x}_{m+1}$,
- calculate the $\mathbf{x}$ vector partition corresponding to the mutually independent diagonal blocks — $\mathbf{x}_i$ for $(1 \leq i \leq m)$.

The Choleski factorization algorithm is similar to LU factorization, with the block-diagonal-bordered Choleski algorithm having the same distinct sections as described above with the exception of $\mathbf{L}^T_{i,i}$ and $\mathbf{L}^T_{m+1,i}$ being substituted for $\mathbf{U}_{i,i}$ and $\mathbf{U}_{i,m+1}$ respectively.

6.2.1 The Hierarchical Data Structure

This block-diagonal-bordered sparse LU solver uses implicit hierarchical data structures based on vectors of C programming language structures to efficiently store and retrieve data for a block-diagonal-bordered sparse matrix. These data structures provide good cache hit access, because non-zero data values and row and column location indicators are stored in adjacent physical memory locations. For this work, we are assuming that there is no requirement for pivoting; consequently, we can use static data structures and the locations of all fillin are determined before memory is allocated for the data structures. We use explicit pointers to subsequent data locations in order to reduce indexing overhead. Row location indicators are explicitly stored as are pointers to subsequent values in columns that are required when updating values in the matrix. The use of additional memory in the data structures is traded for reduced indexing overhead. Modern distributed memory multi-processors are available with substantial amounts of random access memory at each node, and power systems network matrices are actually small relative to other matrices found in the academic and industrial communities. Consequently, this research examines data structures that are designed to optimize processing speed at the cost of increased memory usage when compared to other compressed storage formats. We compare the memory requirements for these data structures to the memory requirements for the more conventional compressed data structures below.

The hierarchical data structure is composed of eight separate parts that implicitly store a block-diagonal-bordered sparse matrix. The hierarchical nature of these structures store only non-zero values, especially in the borders where entire rows may be zero. Eight separate C language structures are employed to store the data in a manner that can efficiently be accessed with minimal indexing overhead. Static vectors of each structure type are used to store the block-diagonal-bordered sparse matrix. Figure 6.4 graphically illustrates the hierarchical nature of the data structure, where the distinct C structure elements presented in that figure are:

1. diagonal block identifiers,
2. matrix diagonal elements,
3. non-zero values in the lower triangular diagonal matrix blocks (stored as sparse row vectors),

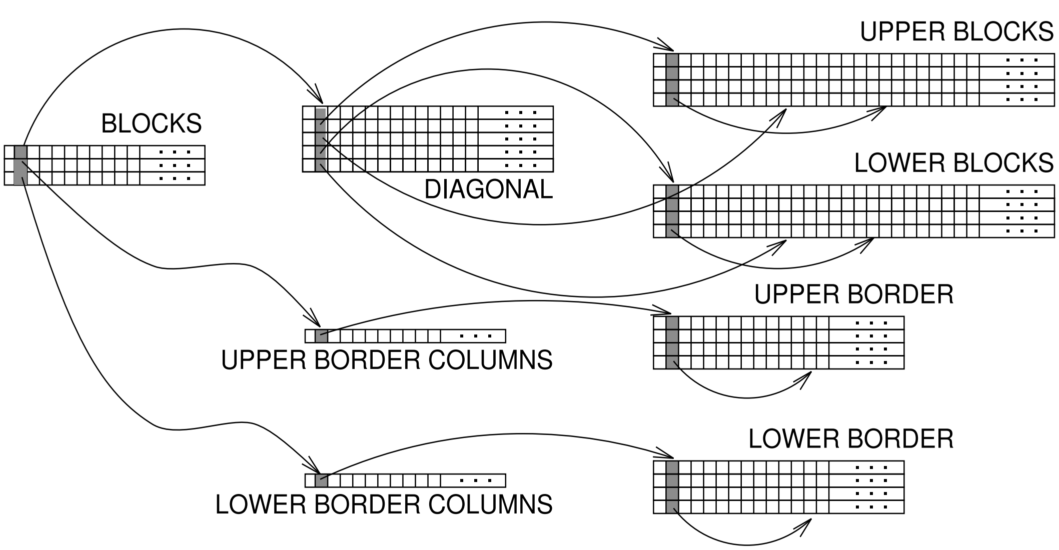


Figure 6.4: The Hierarchical Data Structure for Parallel LU Factorization

4. non-zero values in the upper triangular diagonal matrix blocks (stored as sparse column vectors),
5. non-zero row identifiers in the lower border,
6. non-zero column identifiers in the upper border,
7. non-zero values in the lower border (stored as sparse row vectors),
8. non-zero values in the upper border (stored as sparse column vectors).

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

Data in the strictly lower triangular portion of the matrix is stored as sparse row vectors; likewise, data in the strictly upper triangular portion of the matrix is stored as sparse column vectors. This data storage scheme minimizes the effort to find non-zero $A_{i,k}$ — $A_{k,j}$ pairs used to modify $A_{i,j}$ by consecutively storing values in lower triangular rows and upper triangular columns. However, Crout and Doolittle-based LU factorization algorithms require access to the next non-zero value in the same column or row for lower/upper triangular matrices, so pointers are used to permit direct access to those values without requiring searches for the data as is required in other compressed storage formats. This data structure provides the benefits of a doubly linked data structure in order to minimize indexing overhead. The value corresponding to any diagonal element has pointers to the first non-zero element in the lower triangular row and upper triangular column, and to the first non-zero elements in the lower and upper border. This data structure trades memory utilization for speed by storing indicators to all non-zero column values. In addition, the combination of adjacent storage of non-zero row values and the explicit storage of column identifiers, greatly simplify the forward reduction and backward substitution steps.

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

Conventional compressed data formats require less storage than this data structure; however, additional memory has been traded for reduced indexing overhead. Two reasons exist that justify the use of additional memory: large 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

$$S_c = (\lambda_{fp} + \lambda_{int}) \times \eta(A) + (\lambda_{int} \times N) \tag{6.1}$$

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

$$S_h = (\lambda_{fp} + (3 \times \lambda_{int})) \times \eta(A) + (\lambda_{int} \times N) + ((3 \times \lambda_{int}) \times N_{blocks}) + ((2 \times \lambda_{int}) \times N_{border}) \tag{6.2}$$

bytes to store the same matrix implicitly.

where:

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

N_{border} is the number of non-zero row and column segments in the borders.

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

While Crout and Doolittle factorization algorithms permit partial pivoting [30], 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 [12]. Block-diagonal-bordered sparse matrices offer the potential for an additional relaxed pivoting rule that limits pivoting choices to within a diagonal matrix block. For the present research, it is assumed that numerical pivoting will not be required, because the matrices for power systems distribution networks will be derived from matrices that are diagonally dominate or even positive definite.

6.2.2 Parallel Blocked-Diagonal-Bordered LU Factorization

Implementations for both parallel block-diagonal-bordered sparse LU and Choleski factorization have been developed in the C programming language for the Thinking Machines CM-5 multi-processor using a host-node paradigm with explicit message passing. Two versions of each parallel block-diagonal-bordered algorithm have been implemented: one implementation uses low-latency active messages to update the last diagonal block using data in the borders and the second implementation uses conventional high(er) latency non-blocking, buffered interprocessor communications. The communications paradigms for these two implementations differ significantly. The communications paradigm we used with active messages, is to calculate a vector × vector product and immediately send the value to the processor holding the corresponding value of $\mathbf{A}_{m+1,m+1}$. The communications paradigm for buffered communications is to perform the vector × vector products, store them in a buffer, and then have each processor send buffers to all other processors. The low-latency, active message communications paradigm greatly simplified development of the algorithm, and the empirical results presented in the next chapter, show that (not unexpectedly) the low-latency communication implementation is substantially faster.

The block-diagonal-bordered LU factorization algorithm can be broken into three component parts as defined in the derivation on available parallelism in chapter 4. Pseudo-code representations of each parallel algorithm section are presented separately in figures E.1 through E.4 in appendix E. In particular, each of these figures correspond to the following figure numbers:

1. factor the diagonal blocks and border — figure E.1,
2. update the last diagonal block —
    - low-latency communications paradigm — figure E.2,
    - buffered communications paradigm — figure E.3,
3. factor the last diagonal block — figure E.4.

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

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

This block-diagonal-bordered LU factorization implementation solves the last block using a sparse blocked kji-saxpy LU algorithm based on the dense kij-saxpy algorithm described in [61]. Here **k** is the index corresponding to the diagonal, **i** is the index corresponding to rows, and **j** is the index corresponding to columns. Our examinations of power systems network matrices showed that after partitioning these matrices into block-diagonal-bordered form, there is little additional available parallelism in the last diagonal block — insufficient inherent parallelism to be exploited in a load-balanced manner. Consequently, a pipelined algorithm has been used. The algorithm to factor the last diagonal block maintains the blocked format and pipelined nature of the dense kij-saxpy algorithm; however, special changes were made to the algorithm in order to minimize calculations and to minimize overhead when performing communications. The three significant changes to the kij-saxpy algorithm are:

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

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

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

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

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

Parallel block-diagonal-bordered Choleski factorization algorithms are similar to the LU factorization algorithms presented in figures E.1 through E.4 — with modifications to account for the symmetric nature of the matrices used in Choleski factorization. Choleski factorization has only about half of the calculations of LU factorization, and block-diagonal-bordered Choleski factorization has only half the number of updates to the last diagonal block. The parallel Choleski algorithm using the active message communications paradigm would see reduced communications when compared to LU factorization; however, there would be no reduction in the number of messages required in a buffered communications update of the last diagonal block. While the buffers would be shorter, there is still the requirement for each processor to communicate with all other processors. In most instances, the message start-up latency dominates, not the per-word transport costs. There would also be no reduction in the amount of communications when factoring the last block.

In other words, parallel Choleski factorization is a more difficult algorithm to implement efficiently than LU factorization, because there is a significant reduction in the amount of calculations without a similar reduction in communications overhead. The computation-to-communications ratio for Choleski factorization is $\frac{1}{2}$ to $\frac{1}{8}$ of LU factorization. Consequently, the results in section 7.1 will compare the performance of LU factorization and Choleski factorization to illustrate performance as a function of the amount of floating point operations versus the communications overhead. To get a better understanding of this trend, a version of the parallel block-diagonal-bordered LU factorization algorithm has been implemented for complex data. Complex math has four floating point multiplies and four subtracts/adds when compared to double precision floating point multiply/subtract operations. With these three implementations, we will be able to clearly illustrate the need for low-latency communications for algorithms to solve power systems network linear equations. These matrices are sufficiently sparse, that by increasing the number of floating-point operations by two or eight times that of Choleski factorization, there is a marked increase in the relative parallel speedup of these algorithms. Communications overhead remains constant and only the number of floating point operations increases. By considering the capabilities of the target parallel architecture, namely the computation-to-communications ratio or granularity, you can identify the communications capabilities required in target parallel architectures.

6.2.3 Forward Reduction and Backward Substitution

The remaining steps in the parallel algorithm are forward reduction and backward substitution. The parallel version of these algorithms take advantage of the fact that calculations can be performed in one of two distinct orders that preserve the precedence relation in the calculations [34]. The combination of these techniques is utilized to minimize communications times when solving for the last diagonal block. The forward reduction algorithm to operate with the parallel block-diagonal-bordered LU factorization algorithm can be broken into three component parts, similar to LU factorization. Pseudo-code representations of each parallel algorithm section are presented separately in figures E.5 through E.8 in appendix E. In particular, each of these figures correspond to the following figure numbers:

1. forward reduce the diagonal blocks and border — figure E.5,
2. update the last diagonal block —
    - low-latency communications paradigm — figure E.6,
    - buffered communications paradigm — figure E.7,
3. forward reduce the last diagonal block — figure E.8.

The backward substitution algorithm to operate with the parallel block-diagonal-bordered LU factorization algorithm can be broken into two component parts, back substitute the last diagonal block then back substitute the remaining upper triangular matrix. The only interprocessor communications required occurs when solving for $\mathbf{x}_{m+1}$ in the last diagonal block. The solution for $\mathbf{x}_{m+1}$ in this portion of the matrix broadcasts the values of $\mathbf{x}_{m+1}$ to all processors, so those values are available to the next step, solving for $\mathbf{x}_1$ to $\mathbf{x}_m$ in the remaining diagonal blocks. Pseudo-code representations of each parallel algorithm section are presented separately in figures E.9 and E.10 in appendix E, respectively for backward substitution of the last diagonal block and backward substitution of the diagonal blocks and border.

Forward reduction and backward substitution algorithms for Choleski factorization are similar to those for LU factorization, with one major difference. The factorization process generates only a single lower triangular matrix, $\mathbf{L}$. For the last diagonal block, one triangular solution step must occur in a manner that requires more communications than an optimally implemented triangular solution for LU factorization. The kji order for LU factorization in the last diagonal block has been selected to maximize performance for both forward reduction and backward substitution — by minimizing communications overhead. Meanwhile, for Choleski factorization, the optimal direct solver algorithm must use a column distribution for the data in the last block, require additional communications be incurred in the forward reduction of the last diagonal block, and then backward substitute the last diagonal block using an implicit transpose of $\mathbf{L}$. The final step ensures that all $\mathbf{x}_{m+1}$ values are broadcast to all processors, eliminating an extra, costly communications step. This combination of data distributions and algorithm specifics ensures the least number of calculations and the minimum amount of communications are performed and should offer the best opportunity for good parallel performance.

6.3 Parallel Sparse Iterative Solver Implementations

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 using explicit message passing and a host-node paradigm. In order to have an implementation with extremely low-latency communications, we utilized the Connection Machine active message layer (CMAML) remote procedure call features [53, 59]. We also implemented versions of the algorithms using non-blocking, buffered communications. 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. Substantial improvements in the performance of the algorithm were observed for low-latency active messages, when compared to more traditional communications paradigms that use non-blocking communications functions in conjunction with packing data into communications buffers. Throughout this discussion of parallel iterative sparse solvers, the active message communications paradigm is the means with which we implemented low-latency communications on the Thinking Machines CM-5.

The low-latency communications paradigm we use throughout this algorithm is to send a double precision or complex data value to the destination processor as soon as the value is calculated and the value is sent only to those processors that need the value. 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 or a similar complex value and 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.

A version of the software is available that runs on a single processor on the CM-5 to provide empirical speed-up data to quantify multi-processor performance. Empirical performance data has been gathered for a range of numbers of processors and real power systems sparse network matrices. Results based on empirical data collected in benchmarking trials are presented in section 7.2. This block-diagonal-bordered sparse Gauss-Seidel method has the following distinct segments which were derived in chapter 4:

1. calculate $\mathbf{x}_i^{(k+1)}$ in the diagonal blocks and upper border, $\mathbf{A}_{i,i}$ and $\mathbf{A}_{i,m+1}$ respectively, where $(1 \leq i \leq m)$,
2. update the values of $\hat{\mathbf{b}}$ using values in the lower border — $\hat{\mathbf{b}} = \mathbf{b}_{m+1} - \sum_{i=1}^{m} \left(\mathbf{L}_{m+1,i}^{-1} \mathbf{x}_i^{(k+1)} \right)$ — the actual implementations use values of $\hat{\mathbf{b}}$,

3. calculate $\hat{\mathbf{x}}^{(k+1)}$ using the values of $\hat{\mathbf{b}}$ and the last diagonal block $\mathbf{A}_{m+1,m+1}$.

6.3.1 The Hierarchical Data Structure

This block-diagonal-bordered sparse Gauss-Seidel method 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 hit access, because non-zero data values and column location indicators are stored in adjacent physical memory locations. The hierarchical data structure is composed of six separate parts that implicitly store the block-diagonal-bordered sparse matrix and the last block. The hierarchical nature of these structures store only non-zero values, especially in the lower border where entire rows may be zero. Six 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 6.5 graphically illustrates the relationships within the data structure, where the distinct C structure elements presented in that figure are:

1. diagonal block identifiers and matrix diagonal elements,
2. non-zero values in the diagonal blocks and upper border (stored as sparse row vectors),
3. non-zero row identifiers in the lower border,
4. non-zero values in the lower border (stored as sparse row vectors),
5. last diagonal block identifiers and matrix diagonal elements,
6. non-zero values in the last diagonal block (stored as sparse row vectors).

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, ρ, for the vector $\times$ vector product from this border row used in calculating values in the last diagonal block.

The hierarchical data structure for parallel Gauss-Seidel actually has three separate data structures that have pointers to corresponding data structures to store off-diagonal non-zero row vector elements. The first data structure of pointers to off-diagonal non-zero matrix elements represents the elements on the diagonal within blocks, excluding the last diagonal block. The last diagonal block is handled separately by the third set of data structures. The second data structure of pointers efficiently stores the non-zero values in the lower border. 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.

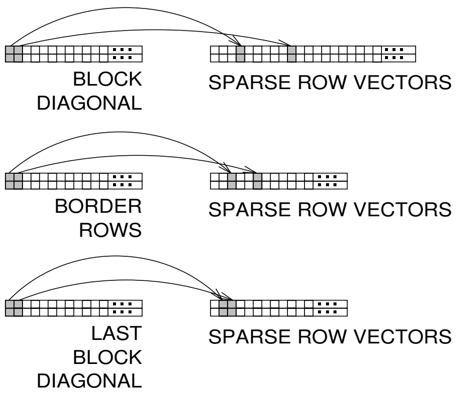


Figure 6.5: The Hierarchical Data Structure for Parallel Gauss-Seidel Methods

The data structures for the sparse block-diagonal-bordered Gauss-Seidel algorithm are much simpler than the data structures for the direct methods. Data structures for the parallel Gauss-Seidel method have one less layer than the data structures for direct methods — it is possible to eliminate the block identifiers without causing additional effort to search for values. In addition, separate data storage for the upper borders has been eliminated. These values are entered into sparse column vectors that are denoted by the block identifiers. The extra loops for blocks and the upper border must be eliminated in order to reduce indexing overhead that would occur in the parallel implementation but not in the sequential implementation. During algorithm development, we discovered that this indexing overhead significantly degraded parallel algorithm performance, so enhanced data structures were developed that required a minimum of **for** loop instantiations.

6.3.2 Parallel Blocked-Diagonal-Bordered Gauss-Seidel

Implementations for the parallel block-diagonal-bordered sparse Gauss-Seidel method have been developed in the C programming language for the Thinking Machines CM-5 multi-processor using a host-node paradigm with explicit message passing. Two versions of the parallel block-diagonal-bordered iterative algorithm have been implemented: one implementation uses low-latency active messages to update $\hat{\mathbf{b}}$ using data in the lower border and to distribute values of $\hat{\mathbf{x}}^{(k+1)}$ from the last diagonal block. The second implementation uses conventional high(er) latency non-blocking, buffered interprocessor communications. The programming paradigms for these two implementations differ significantly.

The programming paradigm we used with active messages, is to calculate a vector × vector product when updating the last diagonal block and immediately send the value to the processor holding the corresponding value of $\hat{\mathbf{b}}$. Likewise, when values of $\hat{\mathbf{x}}^{(k+1)}$ are calculated, they are sent immediately but to only those processors that require these values. The programming paradigm we used with buffered communications is to perform the vector × vector products, store them in separate buffers, and then have each processor send buffers to all other processors. Likewise, when values of $\hat{\mathbf{x}}^{(k+1)}$ are calculated, they are stored in a buffer at each processor and sent to all other processors. The active message communications programming paradigm greatly simplified development of the algorithm, and the empirical results, presented in section 7.2, show that (not unexpectedly) the low-latency, active message-based implementation is significantly faster.

The parallel block-diagonal-bordered sparse Gauss-Seidel algorithm can be broken into three component parts as defined in the derivation of available parallelism in chapter 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 × vector products in parallel,
3. solve for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block.

Pseudo-code representations of each parallel algorithm section are presented separately in figures E.11 through E.18 in appendix E. In particular, each of these figures correspond to the following figure numbers:

1. monitor convergence for the parallel Gauss-Seidel method — figure E.11,
2. solve for $\mathbf{x}^{(k+1)}$ in the diagonal blocks and upper border — figure E.12,
3. update $\hat{\mathbf{b}}$ for the last diagonal block —
    - low-latency communications paradigm — figure E.13,
    - buffered communications paradigm — figure E.14,
4. solve for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block —
    - low-latency communications paradigm — figure E.15,
    - buffered communications paradigm — figure E.16,
5. perform the convergence check —
    - low-latency communications paradigm — figure E.17,
    - buffered communications paradigm — figure E.18.

Figure E.11 provides the framework for the parallel block-diagonal-bordered sparse Gauss-Seidel implementation. In this implementation, multiple iterations are performed before a check is made on convergence. Convergence checks have computational complexity nearly as great as to solve for $\mathbf{x}^{(k+1)}$, so the algorithm performs a predefined number of iterations before the convergence check. To better understand the algorithm, timing statistics were collected for each portion of the algorithm.

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 simply places all data into one diagonal block without a border and includes the capability to gather convergence-rate data. The parallel implementation has been developed as an instrumented proof-of-concept to examine the efficiency of each section of the code described above. The host processor is used to gather and tabulate statistics on the multi-processor calculations. Statistics are gathered at synchronization points, so there is no impact on total empirical measures of performance. Empirical performance data is presented in section 7.2 for varied numbers of processors solving real power systems sparse network matrices.

The algorithm section that updates the values of $\hat{\mathbf{b}}$ calculates a sparse matrix × dense vector product by calculating individual sparse vector × dense vector products for lower border rows. These partial sums must be distributed to the proper processor holding the respective row data in the last diagonal block. Separate sparse vector × dense vector products are performed for each block of data on a processor. Only nonzero rows in the lower border are utilized when calculating vector × vector products to generate the required partial sum values to update the values of $\hat{\mathbf{b}}$. Examining only non-zero values significantly limits the amounts of calculations in this phase.

There has been no attempt at parallel reduction of the partial sums of updates from the borders. In the process of developing an implementation with optimal performance, we discovered that any attempt to consolidate updates to a value in the last diagonal block caused more overhead than was encountered by sending multiple update values to the same processor. There is more work required to sum update data than to calculate the sparse vector × dense vector products. Likewise, there has been no attempt at parallel reduction of the partial sums of updates from the borders.

In the parallel Gauss-Seidel, differences in programming paradigms resulting from the different interprocessor communications capabilities are no more significant than when calculating new values in the last diagonal block. With active message communications, only those communications that are required are performed. Lists of processors that require a particular value of $\hat{\mathbf{x}}^{(k+1)}$ are calculated a priori and used repeatedly. Empirical data presented in section 7.2 clearly illustrates the limited growth in the number of communications messages as the number of processors increases. When using buffered communications, the programming paradigm is to calculate all values on a processor within a color and then broadcast those values to all other processors. Development testing illustrated that the algorithm for this communications paradigm would be the most efficient way to implement the algorithm. There is little to be gained by attempting to determine a subset of processors. In almost all distributions of data to processors, one or more of the multiple values in a buffered message was required on every processor. As a result, every processor must broadcast data to all other processors after every color. While some communications overhead could be saved by limiting what data is sent to each processor — shorten the messages — it was determined that too much overhead would be required to sort the data and produce separate communications buffers for each destination processor.

Chapter 7

Empirical Results for Parallel Linear Solvers

Empirical results for the parallel block-diagonal-bordered direct and iterative methods are presented in this chapter along with comparisons of performance of direct versus iterative parallel linear solvers. We analyze the empirical performance of the parallel block-diagonal-bordered LU solvers in section 7.1, we analyze the empirical performance of the parallel block-diagonal-bordered iterative solver in section 7.2, and in section 7.3, we compare performance of the two algorithms.

7.1 Empirical Results for Parallel Direct Linear Solvers

A stated goal of this block-diagonal-bordered direct solver is to simplify the task organization of the parallel LU algorithm and have interprocessor communications significantly reduced and regular. The performance of this block-diagonal-bordered LU solver is dependent on the ability to order the real power systems sparse matrices into the desired form with uniformly distributed data in the diagonal blocks and a minimum number of equations in the lower border.

In section 7.1.1, we illustrate the ordering capabilities of the node-tearing nodal analysis by presenting pseudo-images of selected sparse power systems network matrices after we have applied both our node-tearing algorithm to partition the matrices into block-diagonal-bordered form and our pigeon-hole load-balancing algorithm. We provide additional information as to the overall performance of the three-step preprocessing phase, with special note to the amount of fillin in the matrices after ordering and to the total number of floating point operations required to factor the matrices. We then report on the performance of the block-diagonal-bordered sparse LU and Choleski solvers in section 7.1.2. Performance of these parallel block-diagonal-bordered direct linear solvers is dependent upon the ability of the preprocessing phase, in addition to the performance of the parallel implementations. The real performance test of the node-tearing algorithm occurs when the performance of the block-diagonal-bordered sparse LU solver is examined for real power system network matrices in section 7.1.2. In section 7.1.3, we compare the performance of low-latency, active message-based implementations and buffered communications-based implementations. In section 7.1.4, we present preliminary results when running the complex variate LU algorithms on the IBM SP1 and SP2 scalable parallel processors, and in section 7.1.5, we present our conclusions concerning the performance of our parallel direct implementations.

7.1.1 Ordering Power Systems Network Matrices into Block-Diagonal-Bordered Form

Critical to the efficient operation of these parallel block-diagonal-bordered direct sparse matrix solvers is the ability to order sparse power systems networks into block-diagonal-bordered form with equal workloads in all processors. In this section, we illustrate that it is possible to order power systems networks to the desired form, and later we present empirical data that show the load-balancing capabilities of the preprocessing phase.

To demonstrate the performance of the graph partitioning algorithm, we present pseudo-images that show the locations of the non-zero values in the sparse matrices, both the original non-zero values and those that would become non-zero due to fillin during factorization. In the following pseudo-images, original non-zero values are represented as black pixels and fillin values are represented by a lighter grey color. A bounding box has been placed around the sparse matrix. These pseudo-images clearly show the block-diagonal-bordered form of the power systems network matrices after the preprocessing phase.

We examine the performance of our parallel block-diagonal-bordered LU and Choleski solvers with five separate power systems network matrices:

- Boeing-Harwell matrix BCSPWR09 — 1,723 nodes and 2,394 graph edges [13],
- Boeing-Harwell matrix BCSPWR10 — 5,300 nodes and 8,271 graph edges [13],
- EPRI matrix EPRI6K matrix — 6,692 nodes and 10,535 graph edges [14],
- Niagara Mohawk Power Corporation operations matrix NiMo-OPS — 1,766 nodes and 2,506 graph edges,
- Niagara Mohawk Power Corporation planning matrix NiMo-PLANS — 9,430 nodes and 14,001 graph edges.

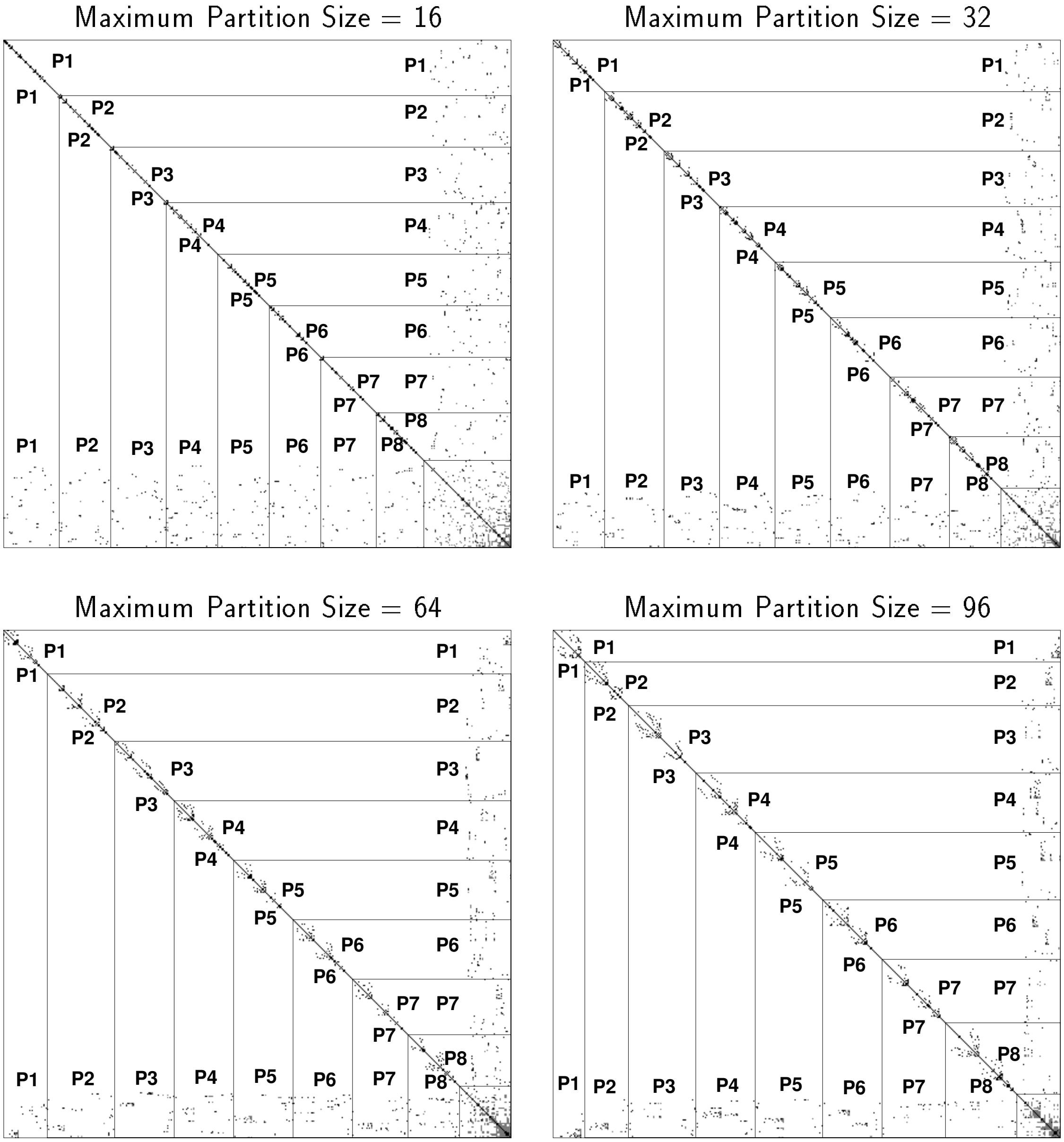
In order to provide a baseline with which to illustrate the performance of the node-tearing algorithm, we have provided pseudo-images of these original, unordered matrices in chapter 2.

Our parallel block-diagonal-bordered direct algorithms require that the power systems network matrix be ordered into block-diagonal-bordered form in a manner that yields a minimum of floating point operations and that has uniformly distributed workloads at all processors. A single specified input parameter, the maximum partition size, defines the shape of the matrix after ordering by the node-tearing algorithm, which in turn directly impacts the size of the borders and the last diagonal block, the number of floating point operations, and the efficacy of load-balancing. In order to illustrate the ability of the node-tearing-based ordering algorithm, we present a detailed analysis of graph partitioning for the BCSPWR09 power systems network in figure 7.1, with sample ordered matrices for maximum diagonal block sizes of 16, 32, 64, and 96 nodes. For the larger values of maximum partition size, the application of minimum degree ordering within a partition is evident in these figures. The upper left-hand corner of a diagonal block has fewer values than the lower right-hand corner.

Detailed statistics for the matrix partitionings are presented in table F.1 (appendix F) for the four example orderings of the BCSPWR09 matrix. This table includes the number of fillin, and the number of rows/columns in the borders and last diagonal block of the ordered matrix. Table F.1 shows that the ordering with maximum partition size of 32 has the least fillin, the fewest total operations, and the largest percentage of operations in the mutually independent matrix partitions. Empirical data collected when benchmarking the parallel software implementation on the CM-5 show that this partitioning has the best parallel direct linear solver performance for this power systems network.

The pseudo-images in figure 7.1 illustrate that the size of the borders and last diagonal block can be manipulated by varying the the maximum partition size. The number of rows/columns in the borders and last diagonal block of these ordered matrices vary from 277 to 131 for maximum partition size of 16 and 96 respectively. Each of these four figures has been load-balanced for eight processors, and the pseudo-images in figure 7.1 include additional markings to illustrate how this matrix would be distributed to the eight processors — P1 through P8. The metric for load-balancing is the number of operations and not the number of columns or rows assigned to a processor. The load balancing step is simply another permutation of the matrix that keeps rows/columns within partitions together in the same order. As the matrix is load-balanced for various numbers of processors, there is no change in the number of fillin nor in the total number of operations.

Figure 7.2 has families of curves that illustrate the relationship between maximum partition size and size of the borders and last diagonal block when partitioning each of the five power systems networks used in this analysis. The partitioning results for the BCSPWR09 network are very similar to the data for the Niagara Mohawk operations data, NiMo-OPS. These matrices are similar in size



Maximum Partition Size = 16

Maximum Partition Size = 32

Maximum Partition Size = 64

Maximum Partition Size = 96

Figure 7.1: BCSPWR09 — Block-Diagonal-Bordered Form — Load Balanced for 8 Processors

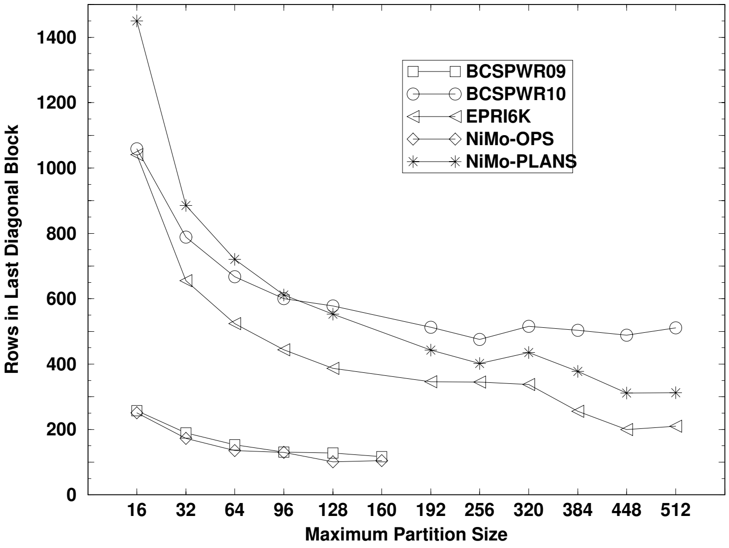


Figure 7.2: Last Diagonal Block Size after Partitioning

and have similar numbers of edges per node. Meanwhile, larger matrices have significantly greater numbers of rows in the border and last diagonal block. Also, these larger matrices have significantly greater variations between the number of rows in the last diagonal block. This empirical evidence suggests that there are fundamental differences between operational analysis networks and larger planning networks. Additional evidence of these differences is discussed below, both as we present orderings for these matrices and as we discuss the performance of the parallel direct linear solvers.

Note, that in figure 7.2, the maximum size of the diagonal blocks is inversely related to the size of the last diagonal block. This is intuitive, because as diagonal matrix blocks are permitted to grow larger, multiple smaller blocks can be incorporated into a single block. Not only will the two blocks be consolidated into the single block, but in addition, any elements in the coupling equations that are unique to those network partitions would also be moved into the larger block. Another interesting point with the relationship between maximum size of the diagonal block and the size of the last block, is that the percentage of non-zeros and fillin in the last diagonal block increases significantly as the size of the last block decreases. The empirical performance data for the parallel solvers show that the best parallel performance is closely correlated with minimum numbers of operations.

In tables F.2 through F.5 (appendix F), we present summary statistics for the remaining power systems networks used in this analysis. In each table, the maximum partition size that yielded the best parallel performance has been identified.

In figure 7.3, we provide an accompanying visual reference to the partitioning performance data presented in tables F.2 through F.5. For each power systems network, we present a representation of the matrix after partitioning and load-balancing for 8 processors. Partitioned graphs presented here have maximum partition size values that yielded the best empirical parallel block-diagonal-bordered direct linear solver performance. The pseudo-images of the block-diagonal-bordered matrices are highlighted to illustrate the manner in which each matrix would be distributed to eight processors — P1 through P8.

We want to reiterate that the block-diagonal-bordered matrix for the BCSPWR09 network has many similarities with the NiMo-OPS network. Also, the EPRI6K matrix has noticeable similarities with the NiMo-PLANS matrix. The BCSPWR09 and NiMo-OPS matrices are operational networks that are homogeneous and have very similar voltage distributions throughout. Meanwhile, the EPRI6K and NiMo-PLANS matrices are from planning applications, and one subsection of these networks includes some lower voltage electrical distribution lines. This matrix has enhanced detail in the local area, with less detail in areas distant from the power utility's own network. This causes additional rows/columns in the borders and the last diagonal blocks, but our parallel block-diagonal-bordered direct solvers appear to have little difficulty with efficiently solving these matrices. The small, highly connected graph section can be seen at the lower right-hand corner of the EPRI6K and NiMo-PLANS matrices in figure 7.3.

7.1.2 Parallel Direct Sparse Solver Performance

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

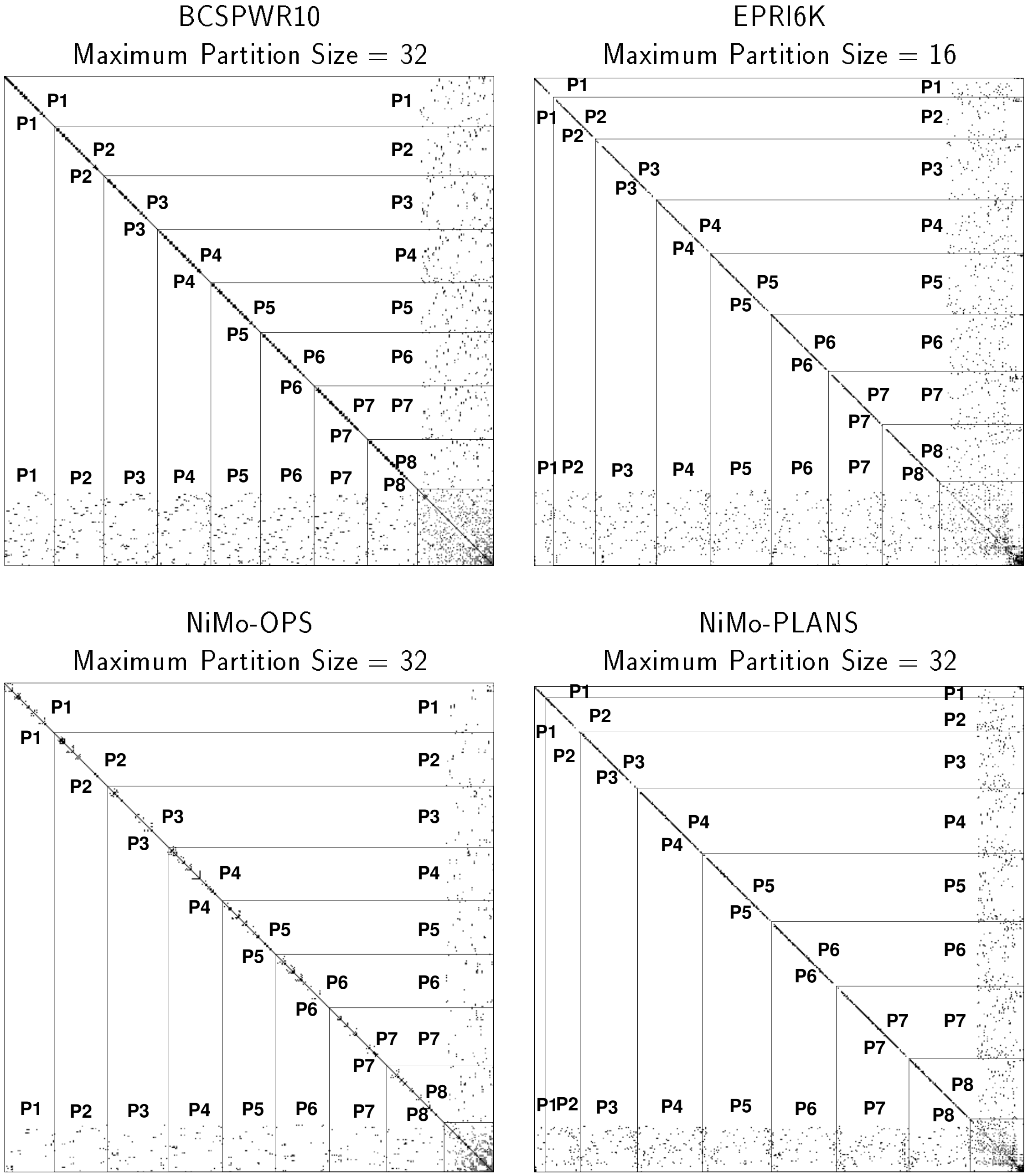
1. Choleski factorization and forward reduction/backward substitution for double precision variables,
2. LU factorization and forward reduction/backward substitution for double precision variables,
3. LU factorization and forward reduction/backward substitution for complex variables,

for each of two communications paradigms —

1. low-latency communications,
2. buffered communications,

for five separate power systems networks —

1. BCSPWR09,



BCSPWR10
Maximum Partition Size = 32

EPRI6K
Maximum Partition Size = 16

NiMo-OPS
Maximum Partition Size = 32

NiMo-PLANS
Maximum Partition Size = 32

Figure 7.3: Block-Diagonal-Bordered Form Matrices — Load Balanced for 8 Processors

2. BCSPWR10,
3. EPRI6K,
4. NiMo-OPS,
5. NiMo-PLANS,

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

For the three solver implementations, there are increasing amounts of floating point calculations in double precision Choleski factorization, double precision LU factorization, and complex LU factorization, with a relative workload of approximately 1:2:8. Choleski algorithms have only approximately one half the number of floating point operations of LU algorithms, and complex floating point operations require four separate multiplications and four addition/subtraction operations for a single complex multiply/add operation. While there are differing amounts of calculations in these algorithms, there are nearly equal amounts of communications, thus the granularity of the algorithm increases proportionally to 1:2:8. The empirical timing data presented below will illustrate just how sensitive the parallel sparse direct solvers for power systems networks are to communications overhead. This sensitivity is not totally unexpected, given the extremely sparse nature of power systems matrices.

Communications in block-diagonal-bordered Choleski or LU factorization occurs in two locations — updating the last diagonal block using data in the borders and factoring the last diagonal block. Because LU factorization requires the update of $\mathbf{L}_{m+1,m+1}\mathbf{U}_{m+1,m+1}$ versus only $\mathbf{L}_{m+1,m+1}$, there are twice as many calculations and twice as many values to distribute when updating the last diagonal block for LU factorization versus Choleski factorization. Meanwhile, there are equal amounts of communications for LU and Choleski factorization when factoring the last diagonal block. The parallel block-diagonal-bordered Choleski algorithm requires that data in $\mathbf{L}^T_{m+1,m+1}$ be broadcast to all processors in the pipelined algorithm that perform the rank 1 update of the sub-matrix. However, for the last diagonal block in the parallel block-diagonal-bordered LU factorization algorithm, only $\mathbf{U}_{m+1,m+1}$ must be broadcast during the parallel rank 1 update. For this research, we are assuming that the matrices are position symmetric, so $\mathbf{L}_{m+1,m+1}$ and $\mathbf{U}_{m+1,m+1}$ have equal numbers of non-zero values.

As we examine the empirical results, we first describe the selection process to identify the matrix partitioning with the best parallel empirical performance. This reduces the amount of data we must consider when examining the performance of the implementations.

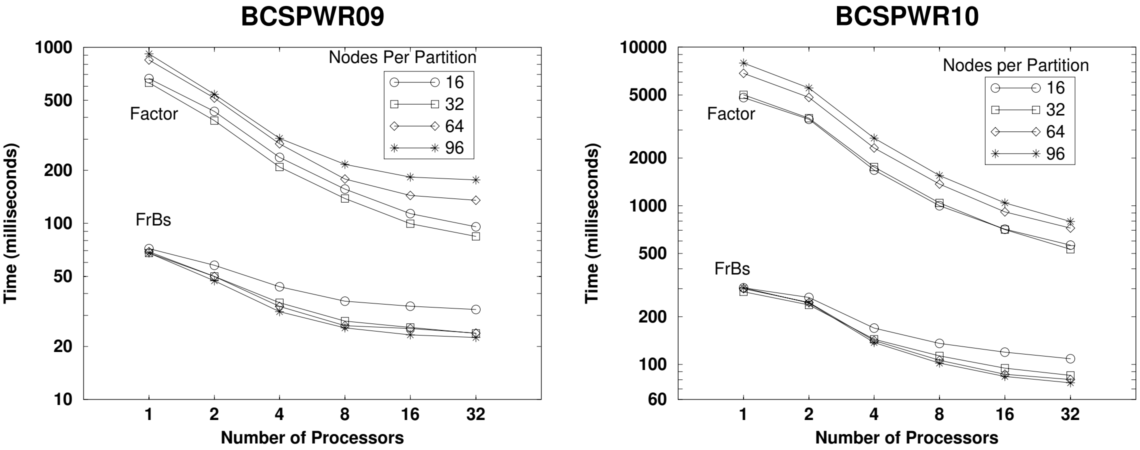


Figure 7.4: Parallel LU Factorization Timing Data — Double Precision

7.1.2.1 Selecting Partitioned Matrices with *Best* Parallel Solver Performance

The primary factors, that affect the performance of parallel direct sparse linear solvers, are available parallelism, load balancing, and communication overhead. Our choice to order the power systems matrices into block-diagonal-bordered form provides a means to significantly limit the task graph to factor the matrix and to make all communications regular. We have shown in section 7.1.1 that the node-tearing algorithm can partition the power systems network matrices into block-diagonal-bordered form and offer substantial parallelism in the diagonal blocks and borders.

The single input parameter to the node-tearing algorithm, the maximum partition size, when varied, affects the size of the diagonal blocks and the size of the borders and last diagonal block. When determining the partitioning with the best parallel direct block-diagonal-bordered sparse linear solver performance, we examined the empirical data collected from algorithm benchmark trials on the Thinking Machines CM-5 multi-computer. Graphs presented in figure 7.4 illustrate the performance for LU factorization and the combination of the forward and backward triangular solution steps for the Boeing-Harwell matrices: BCSPWR09 and BCSPWR10. Each graph has timing data for double precision LU factorization and for forward reduction/backward substitution. These graphs are on a log-log scale and show that for each power system network, a maximum of 32 nodes per partition yields the best overall performance for factorization.

We are considering software to be embedded within a more extensive power systems application, so we must examine efficient parallel forward reduction and backward substitution algorithms in addition to parallel factorization algorithms. Due to the reduced amount of calculations in the triangular solution phases, these algorithms are often ignored when parallel Choleski or LU factorization algorithms are presented in the literature. However, graphs presented in figure 7.4 show that the time to factor the matrix is only approximately an order of magnitude greater than the time required to perform forward reduction and backward substitution on a single processor. This is a direct result of the extremely sparse nature of power system network matrices. For a dense matrix, the number of calculation to factor a matrix is $O(N^3)$ and the number of calculations to triangular solve the matrix is $O(N^2)$. For dense matrices as large as these two matrices, there would be a significant difference in wall-clock time between factorization and triangular solutions, a difference that is not present here. As a result, we must also consider the performance of the triangular solution step, especially if there will be dishonest *(re)use* of a factored matrix as it is repeatedly *(re)used* for multiple triangular solutions. Meanwhile, this order of magnitude difference in performance erodes for large numbers of processors, because it will be shown that there is better relative speedup for the factorization algorithms than for forward reduction and backward substitution.

For the BCSPWR10 power systems network in figure 7.4, we must consider the performance of the forward reduction/backward substitution step in selecting the optimum network partitioning. Performance of the factorization algorithm are nearly similar, although the performance of the triangular solution step is significantly better for 32 nodes per partition than 16 nodes per partition.

7.1.2.2 Timing Performance Comparisons

For the three solver implementations, there are increasing amounts of floating point calculations in double precision Choleski factorization, double precision LU factorization, and complex LU factorization, with granularity proportional to the relative workload of approximately 1:2:8. While there are differing amounts of calculations in these algorithms, there are nearly equal amounts of communications. We present sample timing comparisons for the three solver implementations in figure 7.5 for two power system networks: BCSPWR09 and BCSPWR10. These graphs each have six curves — three each for factorization and for the triangular solution. These graphs illustrate the sensitivity that parallel sparse direct solvers for power systems networks exhibit relative to the amount of communications overhead. This sensitivity is not totally unexpected, given the extremely sparse nature of power systems network matrices. Performance is similar for the other sample power systems network matrices examined in this research.

These graphs plot the time in milliseconds that it takes to factor or calculate a triangular solution for these matrices. These graphs also show the relative number of floating point operations for the three implementations, when examining performance on a single processor. When comparing the empirical parallel performance data from the three implementations, the ratios of the times to factor the matrix decrease as additional processors are utilized. With nearly constant amounts of communications, this overhead has proportionally less of an effect when there are more calculations

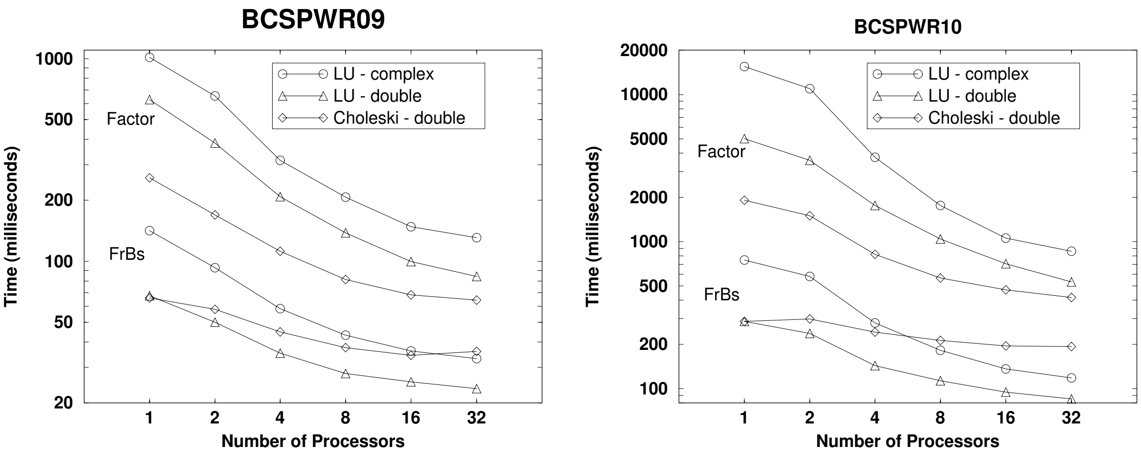


Figure 7.5: Parallel Choleski and LU Timing Comparisons

in the LU factorization algorithms, and it is also easier to hide communications behind calculations when more floating point operations are being performed.

These graphs also illustrate some important facts concerning parallel triangular solutions for Choleski factorization. First, while there is only half the calculations in the factorization step, there is no reduction in the number of calculations in the triangular solution step — both forward reduction and backward substitution steps must be performed. To calculate the triangular solution, every non-zero coefficient in $\mathbf{L}$ is used once during forward reduction and every non-zero coefficient in $\mathbf{L}^T$ must also be used once during backward substitution. While it is possible to avoid explicitly performing the matrix transpose, one of the triangular solutions will require additional communications overhead because the data will be oriented inconveniently. This solution phase must incur additional communications, proportional to the number of non-zeros in the last diagonal block as compared to the communications for LU factorization-based forward reduction, which would be proportional to the number of rows or columns. The number of non-zeros is greater than the number of rows/columns in the matrix, especially after considering the amount of fillin. For a single processor, there is the same amount of work when solving the factored equations for double precision LU and Choleski. However, the effect of the additional communications overhead has a noticeable effect on the slope of the curve representing the triangular solution for Choleski solvers. We observed this phenomenon for all five power systems networks.

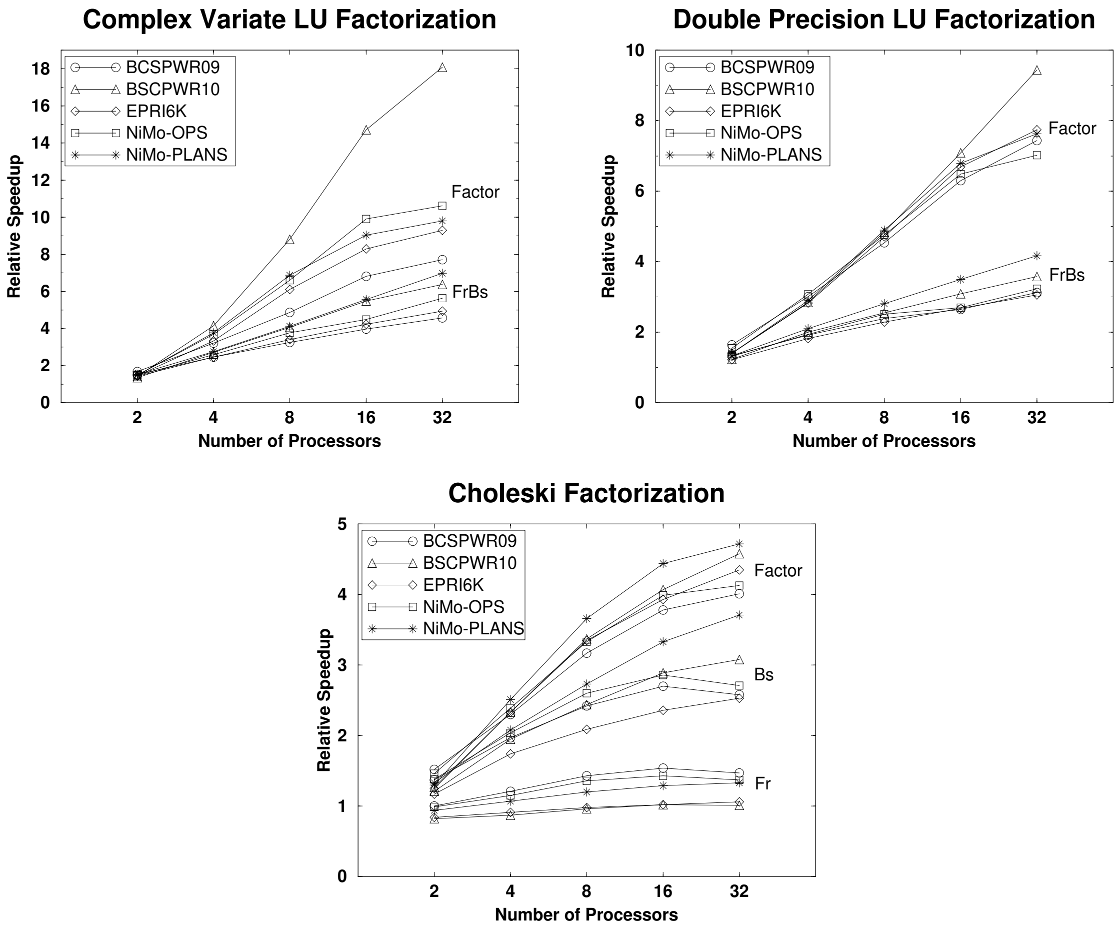


Figure 7.6: Relative Speedup — Parallel Direct Solvers

7.1.2.3 Examining Speedup

We present graphs of relative speedup calculated from empirical performance data in figure 7.6 for the three parallel direct solver implementations. The graphs for double precision and complex variate LU factorization each have two families of speedup curves that show speedup for the five power systems networks examined in this research with separate families of curves for both factorization and the triangular solution. The graph for Choleski factorization has three families of curves that show speedup for factorization, forward reduction, and backward substitution. Each curve plots relative speedup for 2, 4, 8, 16, and 32 processors,

The graph for complex variate LU factorization in figure 7.6 illustrates that parallel performance of the complex LU factorization algorithm can be as much as 18 on 32 processors for the BCSPWR10 power systems network. Meanwhile, factorization performance for the other data sets range from eight to nearly eleven. The BCSPWR10 matrix has the most calculations, and for a single processor, the empirical timing data for this matrix requires a greater relative increase in time from double precision LU factorization to complex LU factorization than other power systems networks. A significant increase in the time to factor the matrix on a single processor will cause significant increases in speedup. We believe that the unusually good performance of the parallel solver for the BCSPWR10 data is a result of caching effects — when the program is run on one or two processors, there is too much data on each processor to fit into the fast-access cache memory. When more processors are used, there is less data per processor, and the entire portion of the matrix assigned to each processor can fit concurrently into the fast-access cache, and as a result, the program runs considerably faster.

While there is a noticeable improvement in speedup performance for complex variate LU factorization of the BCSPWR10 matrix, complex triangular solutions for this matrix do not exhibit as significant an increase in performance over the other power systems network matrices. Complex variate triangular solutions provide speedups ranging between a low of four to a high of eight.

The graph for double precision LU factorization in figure 7.6 illustrates that parallel performance of the double precision LU factorization algorithm can be nearly 10 on 32 processors for the BCSPWR10 power systems network. Factorization performance falls between seven and eight for the other four networks. Likewise, double precision triangular solutions provide speedups ranging from a low of three to slightly greater than four.

Parallel Choleski factorization yields speedups that are less than similar LU algorithms. Empirical data for relative speedup varies between four and five for 32 processors, as illustrated in the graph for Choleski factorization in figure 7.6. This figure also presents empirical speedup data for forward reduction and backward substitution. Due to the significant differences in the implementation of these triangular solution algorithms, empirical data are presented for both.

Backward substitution is the simplest algorithm with the lowest communications overhead — limited to only the broadcast of recently calculated values in $\mathbf{x}_{m+1}$ when performing the triangular backward substitution on $\mathbf{L}^T_{m+1,m+1}$. Empirical relative speedup ranges from 2.5 to a high of 3.5. These speedups are only slightly less than speedups for backward substitution associated with LU factorization. Meanwhile, essentially no speedup has been measured for the forward reduction algorithm, due primarily to additional communications overhead for this implementation than either LU forward reduction or Choleski backward substitution. Communications are required to update the last diagonal block using data in the borders, and there are additional communications for reducing the last diagonal block. These additional communications occur because the data distribution forces interprocessor communications of partial updates when calculating values in $\mathbf{y}_{m+1}$, rather than broadcasting values in $\mathbf{y}_{m+1}$ as in the LU-based forward reduction. Interprocessor communications increase from being proportional to the number of rows/columns in the last diagonal block to being proportional to the number of non-zeros in the last diagonal block. After minimum degree ordering of the last diagonal block, the number of non-zeros may be significantly greater than the number of rows or columns. Due to the characteristics of Choleski factorization, it is inevitable that either forward reduction or backward substitution would have to deal with the problem of increased interprocessor communications [29].

The analysis in this thesis section has used relative speedup, where the sequential execution time was measured with a version of the parallel algorithm running on a single processor. The block-diagonal-bordered form matrices have additional fillin and additional calculations when compared to a matrix that has been ordered as a single large matrix. Tables F.1 through F.5 (appendix F), present summary statistics for the power systems networks used in this analysis and show that there are between 15% and 20% additional non-zeros due to fillin in the block-diagonal-bordered matrices when compared to general minimum degree ordered matrices.

Differences in run times would be expected to be a greater percentage because of the algorithm complexity is greater that $O(N)$. However, when preliminary tests were run with the NiMo-OPS matrix on single processor Sun Microsystems SPARCstations, the difference in performance ranged between 5% to 10%, depending on processor type. The general sparse sequential direct solver was a fan-out algorithm, that requires access to any memory location through out the remaining matrix as rank 1 updates are performed. We hypothesize that general sequential sparse algorithm performance was less than expected due to cache access difficulties. While the general sparse algorithm requires access to the entire matrix as updates are performed, the sequential block-diagonal-bordered direct solver performs operations on limited portions of the matrix at any instance.

Due to the limited difference in performance between the sparse block-bordered-diagonal solver and the general sparse fan-out solver, sequential timings were taken from single processor runs with the parallel solver. Relative speedups reported here should differ from speedup calculated with the *best sequential* algorithm by no more that 10%.

The sensitivities of these parallel algorithms to communications overhead is clearly apparent when comparing the relative speedups presented in figure 7.6. Communications overhead is nearly constant and the 1:2:8 relative workload of floating point operations result in relative speedups of 1:2:4 (4.5:9:18) for the BCSPWR10 power systems network. Consequently, if speedups of 18 were required for a Choleski factorizations algorithm embedded in a real-time application, one way to reach those design goals is to improve the processor/communications performance by a factor of eight to cause proportional reductions in the communications overhead. Another way that algorithm speedup could be achieved is by increasing the performance of the floating point capability of the processor, although, the ratio of computation-to-communications must stay equal to that in the

CM-5 to obtain similar parallel speedups [17].

7.1.2.4 Analyzing Algorithm Component Performance

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

1. factor —
    - diagonal blocks,
    - update last diagonal block,
    - last diagonal block,
2. forward reduction —
    - diagonal blocks,
    - update last diagonal block,
    - last diagonal block,
3. backward substitution —
    - last diagonal block,
    - diagonal blocks.

This detailed analysis of the parallel algorithm will demonstrate that the preprocessing phase can effectively load balance the matrix for as many as 32 processors and illustrate some of the limitations of the algorithm for certain classes of data sets. We present the data for two separate power systems networks: BCSPWR09 — a 1723 node network from an operations application; and BCSPWR10 — a larger, 5300 node network from a planning application. The operations network empirical performance data is presented in figure 7.7 and the planning network empirical performance data is presented in figure 7.8.

For factoring the operations network, the respective graph in figure 7.7 illustrates that factoring the diagonal blocks and updating the last diagonal block have no apparent load balancing overhead. Also, communications overhead is minimal when updating the last diagonal block. The curve representing the performance to factor the diagonal blocks is nearly straight, with a slope that denotes nearly perfect parallelism — relative speedup at each point is approximately equal to the number of processors. The curve representing the performance to update the last diagonal block is also nearly straight, although the slope of the curve shows that some overhead has occurred. On this log-log

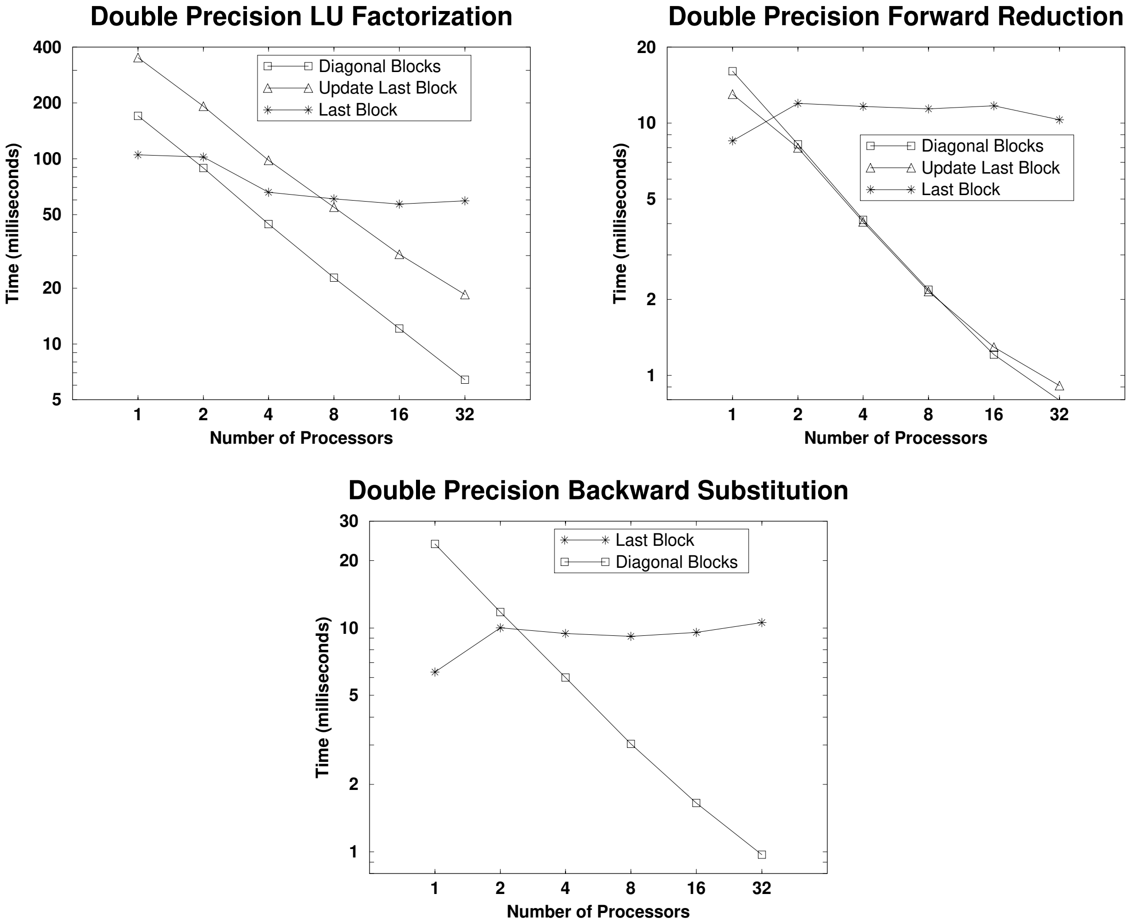


Figure 7.7: BCSPWR09 — Algorithm Component Timing Data

chart, the difference in slope is slight. Meanwhile, the curve representing the times to factor the last diagonal block show that this portion of the algorithm has poor performance — speedups are no more that 1.84 for sixteen processors and performance shows no improvement for 32 processors. Fortunately, the preprocessing phase was able to partition the network and generate matrices where the number of operations to factor the last diagonal block is significantly less than the number of operations to factor the diagonal blocks or update the last diagonal block.

For the triangular solutions, the respective graphs in figure 7.7 show that we were able to get no speedup when performing the triangular solutions in the last diagonal block. Both triangular solution algorithms suffered load imbalance overhead, which was slight and not unexpected. We distributed the data to processors as a function of balanced computational load for factorization.

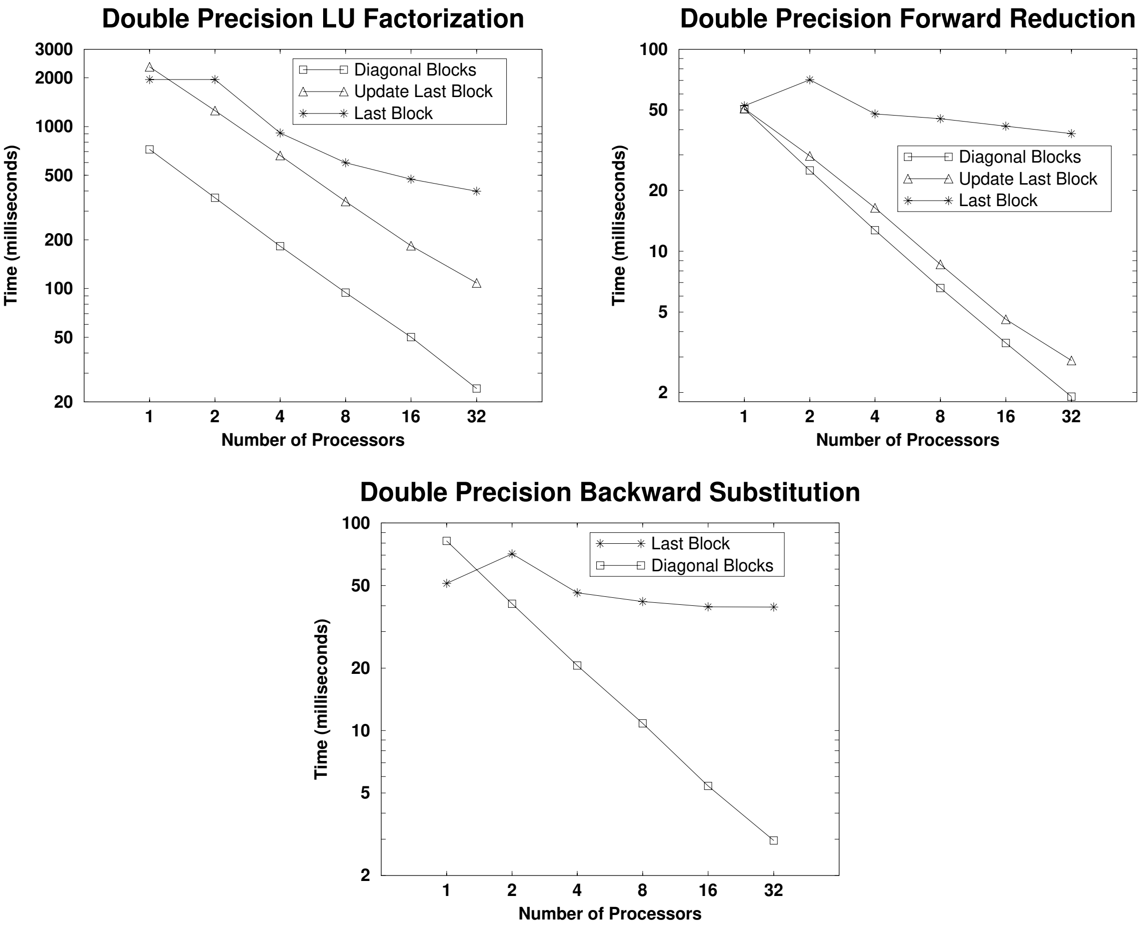


Figure 7.8: BCSPWR10 — Algorithm Component Timing Data

The sparse matrices associated with these power systems networks have significantly lower orders of computational complexity for the two components; however, factorization still has more calculations per row than triangular solves. As a result, some load imbalance overhead has been encountered in these algorithms.

We next examine parallel algorithm performance for a larger power systems network, BCSPWR10, that has four times the number of rows/columns and over eleven times the number of floating point operations. The graph for LU factorization in figure 7.8 illustrates that the performance of factoring the diagonal blocks and updating the last diagonal block have little apparent load balancing overhead and communications overhead is minimal when updating the last diagonal block. Relative speedups are 29.9 for factoring the diagonal blocks on 32 processors and 21.6 for updating the last diagonal block on 32 processors. Performance for factoring the last diagonal block shows great improvement for this planning matrix when compared to the small operations matrix, BCSPWR09. While there is no measurable speedup for two processors due to the pipelined-nature of the algorithm, parallel performance improves respectably for larger numbers of processors. The timing data for LU factorization in figure 7.8 correspond to speedups of 4.9 for factoring the last diagonal block on 32 processors. The extensive amount of operations to update and factor the last block make it imperative that good speedups have been obtained in these algorithm sections — in spite of the fact that both algorithm sections contain communications. The relative speedup obtained for factoring this matrix is 9.4 for 32 processors.

Performance of the triangular solvers on this larger, planning matrix is more promising than for the operations matrix. The respective graphs in figure 7.8 show that we were able to get limited speedup when performing the triangular solutions in the last diagonal block. Both triangular solution algorithms suffered nearly no load imbalance overhead for this larger power systems network, in spite of the fact that we distributed the data to processors as a function of balanced computational load for factorization.

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

- Power systems networks can vary greatly — not only are planning networks larger than operations networks, they also have different characteristics than power systems operations networks. Planning matrices are likely to have adequate workload in the last diagonal block — as a result, this portion of the algorithm will yield good speedups. Little speedup appeared possible when factoring the last diagonal block in operations matrices: however, this generates minimal cause for concern, because there is very little work involved in factoring that matrix relative to the other algorithm sections. For planning matrices, the last block was larger, with a larger percentage of calculations; however, better speedups were achieved with these matrices, allowing improved relative speedups.

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

- There are limitations to the number of processors that can be used to solve linear systems derived from these power systems networks due to the extreme sparsity of these matrices. We have shown that all algorithm components have good performance for as many as 32 processors, except those algorithm components working with the last diagonal block. There is overhead to fill pipelines, and it is questionable whether or not there are adequate floating point operations

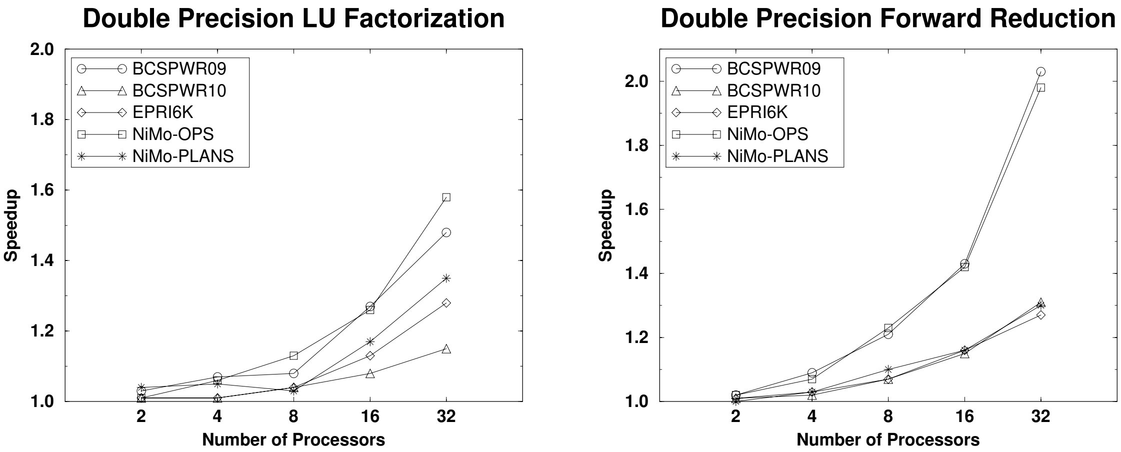


Figure 7.9: Speedup — Low-Latency versus Buffered Communications

to keep the pipeline full for algorithms that factor or solve the last diagonal block. Increasing the size of the last block results in significant performance improvements, but no power system network examined here was large enough that the algorithm could get speedups greater than ten for parallel double precision block-diagonal-bordered LU factorization. Performance improved when the number of floating point operations increased for a complex-variate version of the algorithm and worsened when the number of floating point operations was reduced in a Choleski implementation.

7.1.3 Comparing Communications Paradigms

We have developed two versions of this parallel block-diagonal-bordered sparse linear solver, one version uses a low-latency, active message-based communications paradigm and the other uses a buffered communications paradigm. These communications paradigms significantly modified the respective algorithms as seen in chapter 6. For all power systems networks examined, the largest relative workload when factoring or forward reducing the matrix is to update the last diagonal block. Increases in communications overhead in this portion of the algorithm could significantly affect parallel algorithm performance. In figure 7.9, we present graphs of the speedups obtained using low-latency communications versus buffered communications on the CM-5 for factoring and reducing the matrices. These graphs show that overall speedups can be as great as 1.6 for factoring the operations matrices with an algorithm based on the low-latency communications paradigm, but speedups are less for the larger planning networks. Speedups are also greatest for the smaller operations matrices when updating the last diagonal block during forward reduction.

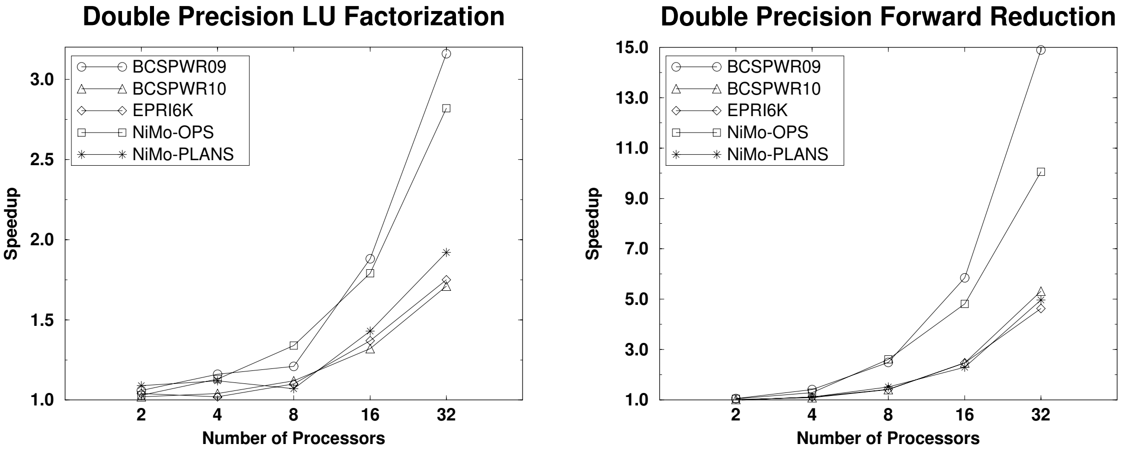


Figure 7.10: Speedup — Low-Latency versus Buffered Communications — Update Last Block

The speedups reported in the two graphs in figure 7.9 are relative to the entire time required to factor or reduce the matrices in parallel for the respective number of processors. While updating the last block has the most relative workload for a single processor, for larger numbers of processors, this relative workload changes significantly. The last diagonal block algorithm section assumes a larger relative portion of the workload because this algorithm section is less efficient. To provide a better understanding of the algorithm speedup at the component level, we present speedup graphs for active messages versus buffered communications in figure 7.10. Speedups for the factorization algorithm component are as great as 2.8, while speedups for the reduction component are as much as 14.8. Low-latency communications have their greatest impact when there are fewer operations to offset the greater communications overhead of the buffered communications. This is most evident when comparing speedups for factorization versus forward reduction in figure 7.10.

It has been sufficiently difficult to obtain usable speedups for the triangular solutions with the low-latency communications paradigm, and performance reductions of 1.2 to 2.0 for buffered communications would have a significant impact on the usability of this algorithm. For complex-variate implementations of these LU algorithms, the effect was somewhat less than for double precision LU algorithms. Conversely, the Choleski implementation saw more pronounced speedups from active messages due to the reduced workload. There are the same number of buffered communications messages in all algorithms, with less data sent for the Choleski algorithms.

7.1.4 Algorithm Performance for the IBM SP1 and SP2

We have ported this parallel block-diagonal-bordered direct solver to the IBM scalable parallel processors (SPPs), the IBM SP1 and SP2. These multi-computers are based on workstation clusters with switched network communications. The available communications on the IBM parallel machines required the use of a non-blocking, buffered communications paradigm. Our communications language of choice for this architecture has been the Message Passing Interface (MPI), because it is being developed as a communications standard for multi-processors with strong emphasis on optimizing message-passing performance. In table 7.1, we present empirical performance data for both the IBM SP1 and IBM SP2 using MPI, and the IBM SP2 using standard Transmission Control Protocol (TCP)/Internet Protocol (IP) based communications through the embedded communications switch. In addition to providing the timing data for factorization and the triangular solutions of the EPRI6K data set, this table also provides the relative speedups for factorization. This table shows that we measured no speedup in these benchmarks for forward reduction and backward substitution. Meanwhile, speedup for factorization on the SP2 was a maximum of approximately 3.2 for eight processors.

This preliminary performance data from the IBM SP1 and SP2 SPPs also illustrate the following:

- From the time that the empirical data was collected on the Cornell Theory Center SP1 and SP2 running MPI, there was a processor upgrade that increased processor capability by nearly 50%.
- Relative speedup for the Cornell Theory Center IBM SP2 using MPI and the NPAC SP2 using TCP/IP-based communications were similar, although the version of the processors in the machines were different. Granularity remained constant, as the lower performance communications were offset by the lesser capable processors.
- Communications latency becomes a significant factor for the triangular solutions, with no speedup observed for the MPI implementations and serious performance problems observed for the TCP/IP-based implementations.

The latency for the IBM SP2 is approximately 30 μseconds, and is lower than the 86 μsecond latency encountered with the Thinking Machines CM-5. Consequently, the limited speedup measured on this architecture denotes a greater communications overhead, most likely a result of processor to communications performance ratio. A more detailed performance comparison is presented in chapter 8, as we predict algorithm performance for future architectures.

Table 7.1: EPRI6K — IBM SP1 and SP2 Performance Data — Complex Variate LU Solver

Number of Processors	Cornell Theory Center IBM SP1 using MPI			
	Factorization		Forward Reduction (milliseconds)	Backward Substitution (milliseconds)
	(milliseconds)	Relative speedup		
1	1450.0		50.0	50.0
2	1165.0	1.2	50.0	45.0
4	827.5	1.8	45.0	40.0

Number of Processors	Cornell Theory Center IBM IBM SP2 using MPI			
	Factorization		Forward Reduction (milliseconds)	Backward Substitution (milliseconds)
	(milliseconds)	Relative speedup		
1	980.0		40.0	30.0
2	610.0	1.6	40.0	40.0
4	400.5	2.5	50.0	40.0
8	320.0	3.1	40.7	40.0

Number of Processors	NPAC IBM SP2 using TCP/IP			
	Factorization		Forward Reduction (milliseconds)	Backward Substitution (milliseconds)
	(milliseconds)	Relative speedup		
1	1460.0		80.0	80.0
2	865.0	1.7	165.0	115.0
4	607.5	2.4	175.0	180.0
8	460.0	3.2	203.7	212.5

7.1.5 Conclusions

We have extensively analyzed the performance of parallel linear solvers for power systems applications on the Thinking Machines CM-5. We have shown that the node-tearing-based partitioning algorithm can yield matrices in block-diagonal-bordered form with balanced workloads; and we have shown that the performance of our parallel block-diagonal-bordered sparse direct linear solvers can yield good speedups for LU factorization. Power system matrices are so sparse that we were able to show that relative speedups for parallel Choleski factorization and complex-variate LU factorization can differ by factors of four for an eight-fold increase in the number of calculations. Matrix sparsity has an even greater effect on the triangular solution steps as it does on the factorization. Communications overhead when reducing or substituting in the last diagonal block is so great that there is no available speedup, so the performance of these algorithms becomes limited by Amdahl's law for both the Thinking Machines CM-5 architecture and the IBM SPPs.

7.2 Empirical Results for the Parallel Iterative Linear Solver

We have developed the parallel block-diagonal-bordered sparse Gauss-Seidel algorithm in order to examine the performance of parallel iterative methods and compare performance with parallel direct methods for power systems networks. In this section, we discuss the performance of the parallel sparse Gauss-Seidel linear solver implementation, and in section 7.3, we compare the performance of parallel direct and parallel iterative methods.

Overall performance of our parallel Gauss-Seidel linear solver is dependent on both the performance of the preprocessing phase to order the matrix and the performance of the parallel Gauss-Seidel implementation. Because these two components of the parallel Gauss-Seidel algorithm are inextricably related, the best way to assess the potential of this parallel iterative algorithm is to measure the empirical performance using matrices from real power systems networks. But first, in section 7.2.1, we illustrate the ordering capabilities of the node-tearing nodal analysis for the Gauss-Seidel algorithm by presenting pseudo-images of selected sparse power systems network matrices after we have applied both our node-tearing algorithm to partition the matrices into block-diagonal-bordered form and our pigeon-hole load-balancing algorithm. We next describe the performance of the parallel block-diagonal-bordered sparse Gauss-Seidel method in section 7.2.2, and we also present data to illustrate the performance of the pigeon-hole load-balancing performed in the preprocessing phase.

Iterative linear solvers must be concerned with the rate of convergence for the intended applications, because iterative solutions only converge in the limit to the solution [23]. We have chosen the Gauss-Seidel method because it converges better than similar iterative techniques such as the Gauss-Jacobi, and we have been able to develop a parallel sparse Gauss-Seidel algorithm that can maintain the strict precedence relations while having no inherently sequential calculations. We discuss the convergence of Gauss-Seidel techniques for power systems network applications in section 7.2.3, we compare the performance of low-latency, active message-based implementations and buffered communications-based implementations in section 7.2.4, and in section 7.2.5, we present our conclusions concerning the performance of our parallel iterative implementations.

7.2.1 Ordering Power Systems Network Matrices into Block-Diagonal-Bordered Form

Critical to the efficient operation of this parallel block-diagonal-bordered direct sparse matrix solvers is the ability to order the sparse power systems networks into block-diagonal-bordered form with equal workload in all processors. In this section, we show the results of ordering power systems networks to the desired block-diagonal-bordered form, and later we present empirical data that illustrate the load-balancing capabilities of this algorithm. We demonstrate the performance of the graph partitioning algorithm with pseudo-images that show the location of the non-zero values in the sparse matrices with black pixels. A bounding box has been placed around the sparse matrices. The pseudo-images clearly show both the block-diagonal-bordered form of the power systems network matrices after ordering with diakoptic techniques, and the additional multi-colored ordering of the last diagonal block.

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

- BCSPWR09 — 1,723 nodes and 2,394 graph edges [13],
- BCSPWR10 — 5,300 nodes and 8,271 graph edges [13],
- EPRI6K — 6,692 nodes and 10,535 graph edges [14],
- NiMo-OPS — 1,766 nodes and 2,506 graph edges,
- NiMo-PLANS — 9,430 nodes and 14,001 graph edges.

Pseudo-images of the matrices without ordering have been presented in chapter 2 to provide a baseline that illustrates the irregular sparse nature of these power systems network matrices. In order to illustrate the utility of the node-tearing ordering algorithm described in appendix C and the graph multi-coloring algorithm described in appendix D, we present a detailed analysis of graph partitioning for the parallel block-diagonal-bordered sparse Gauss-Seidel algorithm using the BCSPWR09 power systems network.

Our parallel block-diagonal-bordered Gauss-Seidel algorithm requires that the power systems network matrix be ordered into block-diagonal-bordered form in a manner that yields uniformly distributed workloads at all processors. The algorithm also requires that the last block be multi-colored in order to have parallelism in this portion of the algorithm. A single specified input parameter, the maximum partition size, defines the shape of the matrix after ordering by the node-tearing algorithm. Examples of applying the node-tearing algorithm to the BCSPWR09 matrix are presented in figure 7.11, with separate pseudo-images for maximum diagonal block sizes of 32, 96, and 160 nodes. These three sparse matrices in figure 7.11 have been load balanced for 32 processors, for sixteen processors, and for eight processors respectively.

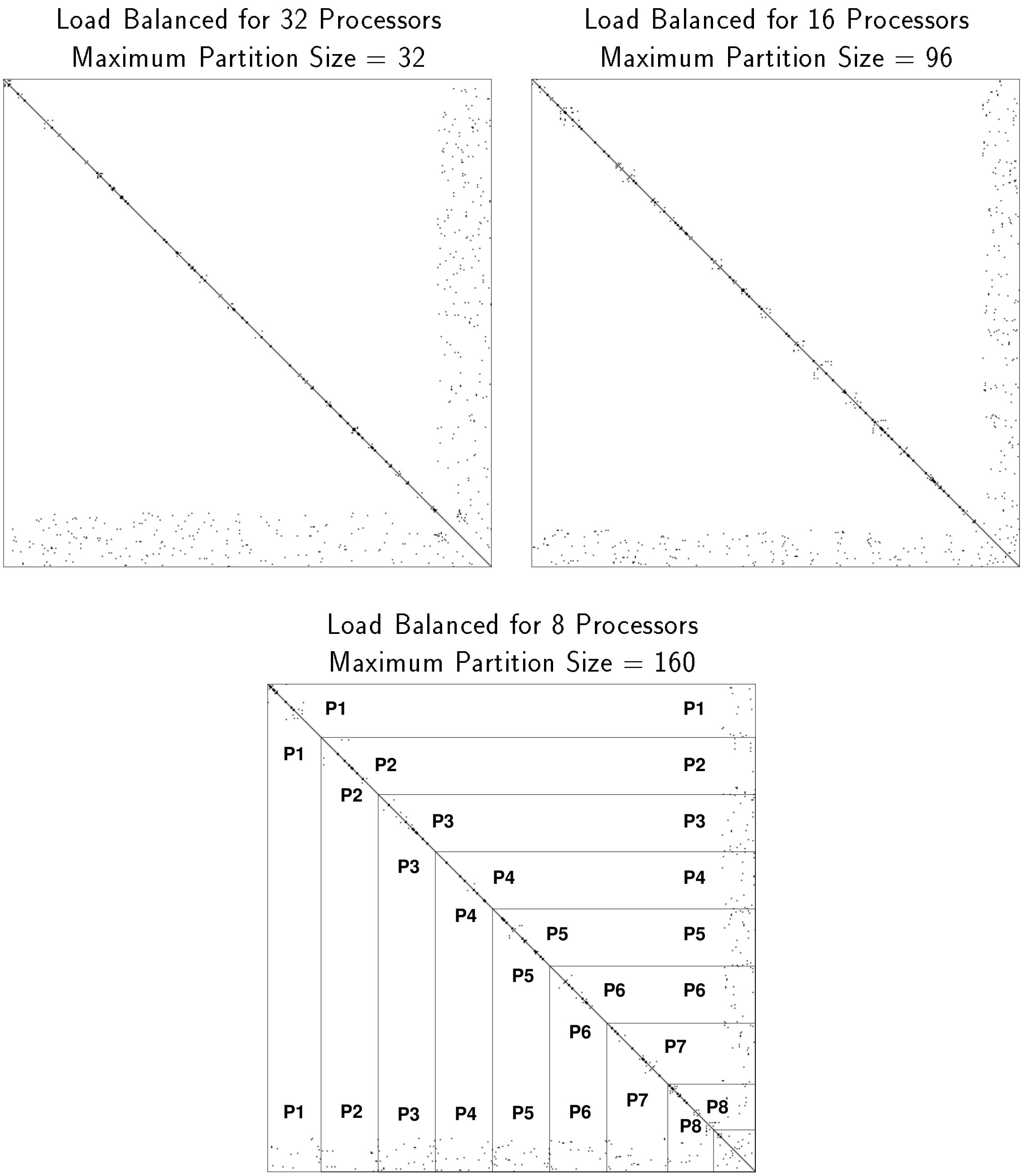
The pseudo-image for maximum block size of 160 nodes in figure 7.11 includes additional markings to illustrate how the blocks and border of this matrix would be distributed to eight processors — P1 through P8. The metric for load-balancing is the number of operations and not the number of columns assigned to a processor. Due to the amount of clutter within the pseudo-images, no processor assignments are listed for the pseudo-images partitioned with a maximum of 32 and 96 nodes.

These pseudo-images represent the actual matrix orderings that yielded the best performance for the respective number of processors. We have found in this work, that there are two significant areas that must be considered for optimal parallel Gauss-Seidel performance for a particular number of processors:

- load-balancing the workload for the diagonal blocks and lower border,
- reducing the size of the last block in order to minimize the amount of communications in this step.

The pseudo-images in figure 7.11 illustrate that the size of the borders and last diagonal block can be manipulated by varying the value of the maximum partition size. The number of rows/columns in the borders and last diagonal block of these ordered matrices vary from 190 to 117 for maximum partition size of 32 and 160 respectively. Statistics are presented in table F.6 (appendix F) for six example orderings of the BCSPWR09 matrix. The data presented in this table is for maximum partition sizes that have varied from 16 to 160. For each ordering, only three separate colors were required for the last diagonal block, and as many as 96.8% of floating point operations are in the diagonal blocks and border. There is a detailed listing of the number of floating point operations in this table for each ordering of the BCSPWR09 power systems network.

The pseudo-images in figure 7.12 provide an accompanying visual reference to the partitioning performance data presented in tables F.7 through F.10. We present a figure for each power systems network that illustrates the ordering after partitioning, multi-coloring, and load-balancing for eight



Load Balanced for 32 Processors
Maximum Partition Size = 32

Load Balanced for 16 Processors
Maximum Partition Size = 96

Load Balanced for 8 Processors
Maximum Partition Size = 160

Figure 7.11: BCSPWR09 — Block-Diagonal-Bordered Form for Parallel Gauss-Seidel

processors — P1 through P8. Partitioned graphs presented here have values of the maximum partition size that yielded the best empirical parallel block-diagonal-bordered Gauss-Seidel performance.

It is important to note that the block-diagonal-bordered matrices for the BCSPWR09 network has many similarities with the NiMo-OPS network. Also the EPRI6K matrix and the NiMo-PLANS matrix have many notable similarities. The BCSPWR09 and NiMo-OPS matrices are for operations networks that are homogeneous and have very similar voltage distributions throughout. Meanwhile, the EPRI6K and NiMo-PLANS matrices are from planning applications, and one subsection of these networks includes some lower voltage distribution lines.

Planning networks have enhanced detail in the area that represents the local power systems grid, with less detail in areas distant from the power utility's own network. Depending on the size of this highly connected portion of the graph and the maximum partition size, the size and interconnectivity of the last diagonal block can be affected. As a result, the last diagonal block for planning matrices may require more than four colors when this portion of the matrix is ordered. Algorithmic efficiency is severely hampered when solving the last diagonal block with eight or more colors on 32 processors, as will be shown below. The highly interconnected graph section can be seen at the upper left-hand matrix corner and in the last diagonal block after coloring for the EPRI6K and NiMo-PLANS power systems network matrices in figure 7.12.

When using additional processors, ordering with lesser values of maximum nodes per partition give better performance because they trade better load balance for additional rows/columns in the last diagonal block. The dense partition in the upper left-hand corners of these sparse matrices are *torn* into smaller partitions with the coupling equations being directed to the last diagonal block.

7.2.2 Parallel Sparse Gauss-Seidel Performance

We have collected empirical performance data for the parallel block-diagonal-bordered sparse Gauss-Seidel method running on the Thinking Machines CM-5 multi-computer for two implementations —

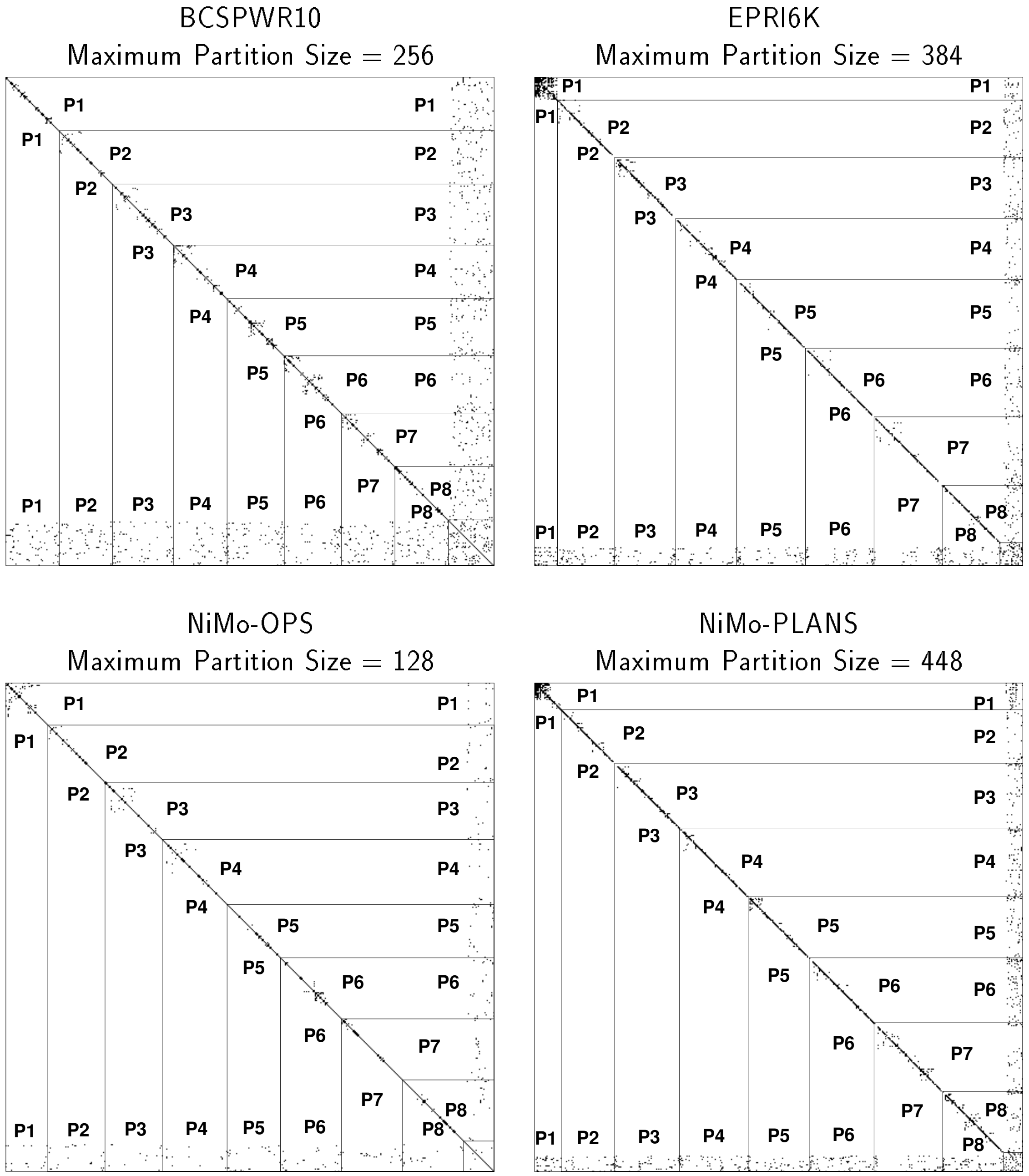
1. parallel sparse Gauss-Seidel for double precision variables,
2. parallel sparse Gauss-Seidel for complex variables,

for each of two communications paradigms —

1. low-latency communications,
2. buffered communications,

for five separate power systems networks —

1. BCSPWR09,



BCSPWR10
Maximum Partition Size = 256

EPRI6K
Maximum Partition Size = 384

NiMo-OPS
Maximum Partition Size = 128

NiMo-PLANS
Maximum Partition Size = 448

Figure 7.12: Block-Diagonal-Bordered Form for Parallel Gauss-Seidel — Load Balanced for 8 Processors

2. BCSPWR10,
3. EPRI6K,
4. NiMo-OPS,
5. NiMo-PLANS,

for 1, 2, 4, 8, 16, and 32 processors, and for as many as nine matrix partitions with the maximum graph nodes per partition varying from 16 to 512.

As we examine the empirical results, we first describe the selection process to identify the matrix partitioning that yielded the best parallel empirical performance for each number of processors. This reduces the amount of data we must consider when examining the performance of the implementations. For the two iterative solver implementations, there are increasing amounts of floating point calculations with the relative workload on a single processor of approximately 1:4, for double precision versus complex variate versions of the algorithms. Complex floating point operations require four separate multiplications and four addition/subtraction operations for a single complex precision multiply/add operation. While there are differing amounts of floating point calculations in these algorithms, there are equal amounts of communications, thus the granularity of the algorithm increases proportionally to 1:4.

We will present timing comparisons that illustrate the effect that partition size has on this parallel sparse Gauss-Seidel algorithm. We next examine relative speedup for the two solver implementations, and then examine the performance of the load-balancing step by examining the timing data for each component of the algorithm. Lastly, we discuss the performance improvements achieved by using low-latency communications and the corresponding simplifications to the algorithm that were possible using the low-latency communications paradigm. The low-latency communications paradigm permits an important implementation difference. The low-latency, active message-based implementation has only the minimal necessary communications when calculating values for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block, which greatly improves the performance of the low-latency implementation when compared to the buffered communications implementation.

7.2.2.1 Selecting Partitioned Matrices with *Best* Parallel Solver Performance

The primary factors affecting the performance of this parallel sparse Gauss-Seidel algorithm are available parallelism, load balancing overhead, and communications overhead. Our choice to order the power systems matrices into block-diagonal-bordered form and color the last diagonal block provides the means:

1. to make all parallelism easy to visualize,

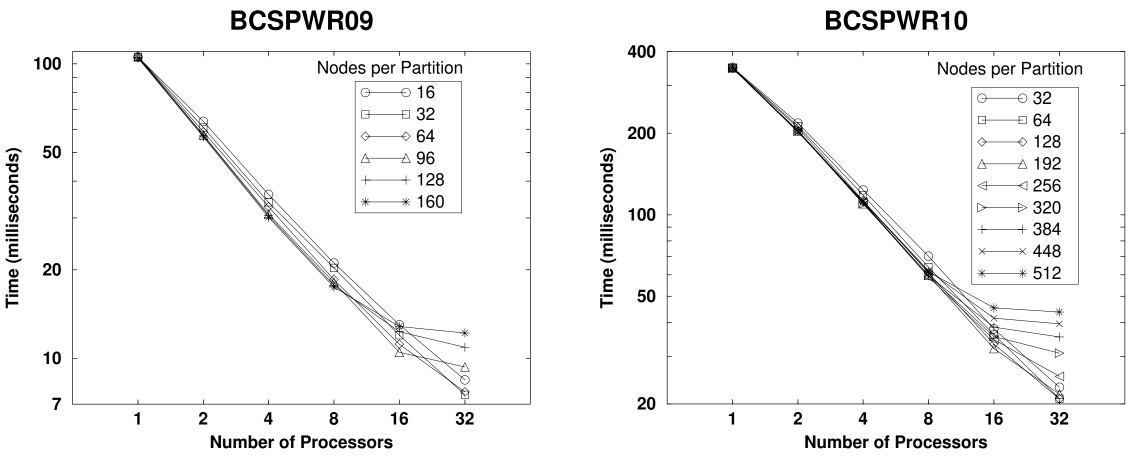


Figure 7.13: Parallel Gauss-Seidel Timing Data — Double Precision

2. to significantly limit the task graph when performing an iteration on the matrix,
3. to minimize the amount of communications for the low-latency implementation or at least make all communications regular for the buffered communications implementation.

We have shown in section 7.2.1 that the node-tearing algorithm can partition power systems network matrices into block-diagonal-bordered form and offer substantial parallelism in the diagonal blocks and borders while the multi-coloring algorithm can provide parallelism in this portion of the calculations.

The single input parameter to the node-tearing algorithm, the maximum partition size, when varied, affects the size of the diagonal blocks and the size of the borders and last diagonal block. Tables F.6 through F.10 (appendix F) illustrate that the percentage of floating point operations in the diagonal blocks and borders are a function of the maximum partition size. To determine the partitioning with the best parallel block-diagonal-bordered sparse Gauss-Seidel performance, we examined the empirical data collected from algorithm benchmark trials on the Thinking Machines CM-5 multi-computer. Graphs presented in figure 7.13 illustrate the performance of four Gauss-Seidel iterations and one convergence check for the Boeing-Harwell matrices: BCSPWR09 and BCSPWR10. For each power system network, a maximum of 32 nodes per partition yields the best overall performance.

The graph for the Boeing-Harwell BCSPWR09 matrix in figure 7.13 presents families of curves for the empirical performance of the double precision Gauss-Seidel for six values of the maximum number of nodes per partition. Timing performance is reported in milliseconds. All partitionings have roughly the same performance for a single processor; however, performance becomes quite variable for 32 processors. We are interested in the fastest times for a partitioning at the number of processors. Performance data is plotted on log-log scales, so for every doubling of the number of processors, the run time would be halved for perfect speedup and each curve would appear as a straight line. However, load imbalance is evident for some partitionings when the curves show no performance improvement for increasing from sixteen to 32 processors. Nevertheless, there are orderings that yield scalable performance even for 32 processors.

For small numbers of processors, performance is best with large partitions; however, smaller partitions yield the best performance for large numbers of processors. The size of the last diagonal block, and the communications overhead associated with solving that portion of the matrix is inversely related to the maximum partition size. The best performance at a number of processors is a trade-off of the amount of parallel work in the diagonal blocks and the border versus the amount of communications in the last diagonal block. Consequently, the best performance for a power systems network will be dependent both on the number of processors and the partition size.

The graph for the Boeing-Harwell BCSPWR10 matrix in figure 7.13 presents families of curves for the empirical performance of the double precision Gauss-Seidel for nine values of the maximum number of nodes per partition. This matrix has both more rows/columns and a slightly higher average number of non-zeros per row than the BCSPWR09 matrix. As a result, there are more partitions that have shown only limited effects of workload imbalance for 32 processors.

The performance for the complex variate parallel sparse Gauss-Seidel algorithm is similar to the double precision algorithm performance for these two matrices.

7.2.2.2 Timing Performance Comparisons

We present timing comparisons in figure 7.14 for two sample power system networks, BCSPWR09 and BCSPWR10, with curves for both implementations — double precision and complex. These graphs illustrate the sensitivity of the parallel sparse power systems network Gauss-Seidel solvers to load imbalance overhead for 32 processors. Meanwhile, the graphs also illustrate the relative insensitivity of the low-latency parallel Gauss-Seidel implementations to communications overhead.

For 16 and 32 processors, smaller partitions are required to minimize load imbalance overhead. The empirical data for both complex and double precision implementations on each graph include the maximum number of nodes per partition at each plotted point. The general trend in the performance data is for significantly smaller partitions for 16 and 32 processors, causing a trade-off in small increases in communications overhead in return for (possible significant) decreases in load-imbalance overhead. This trade-off yields significant results for the BCSPWR09, BCSPWR10, and NiMo-OPS power systems networks, although load-imbalance overhead dominates the results for 32 processors with the EPRI6K and NiMo-PLANS power systems network planning matrices.

There are increasing amounts of floating point calculations in the double precision and complex

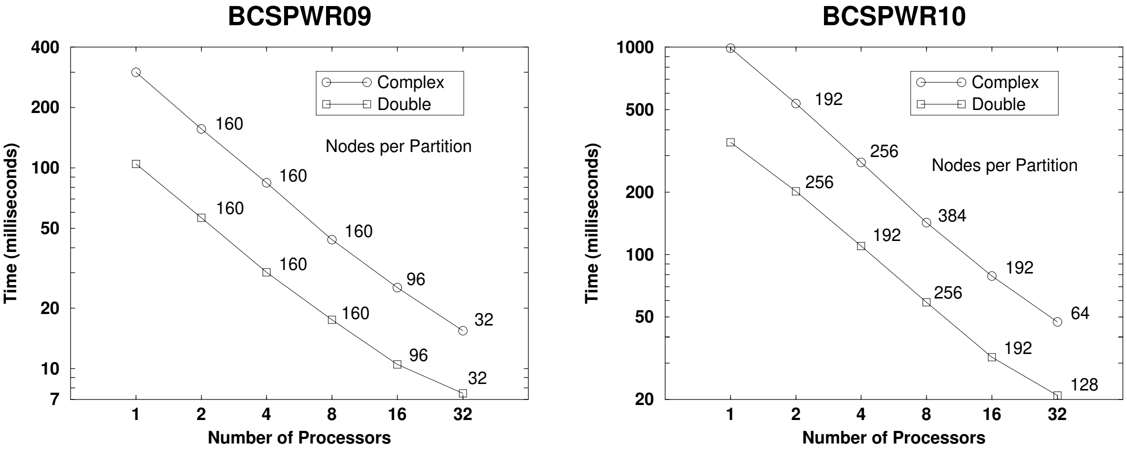


Figure 7.14: Best Empirical Performance for Parallel Gauss-Seidel Implementations

Gauss-Seidel, with a relative workload of approximately 1:4. While there are differing amounts of calculations in these algorithms, there are equal amounts of communications. Additional floating point operations can do little to mitigate load imbalance; however, they can improve the computation-to-communications ratio or granularity within the algorithm, and as a result, the additional computations can have a positive effect on implementation performance. The graphs in figure 7.14 illustrate that there is some improvement in performance due to computation-to-communications granularity, although the performance improvement is minimal. The difference in performance can be seen by examining the relative slopes of the curve splines between 16 and 32 processors. For the complex variate implementation, this spline always has a steeper negative slope, but the slope differences are still minimal. As a result, we can conclude that the low-latency parallel block-diagonal-bordered Gauss-Seidel implementation is not as sensitive to granularity as the parallel block-diagonal-bordered direct solvers. Communications overhead does not significantly detract from nearly perfect parallel speedups.

7.2.2.3 Examining Speedup

Graphs of relative speedup calculated from empirical performance data are provided in figure 7.15 for the two parallel iterative solver implementations. Each figure has a family of speedup curves for the five power systems networks examined in this research. Each curve plots relative speedup for 2, 4, 8, 16 and 32 processors. This analysis of parallel block-diagonal-bordered Gauss-Seidel speedup has used the *best sequential* algorithm to collect sequential execution performance data time. Research showed that there was a significant difference in the performance of a general sequential Gauss-Seidel

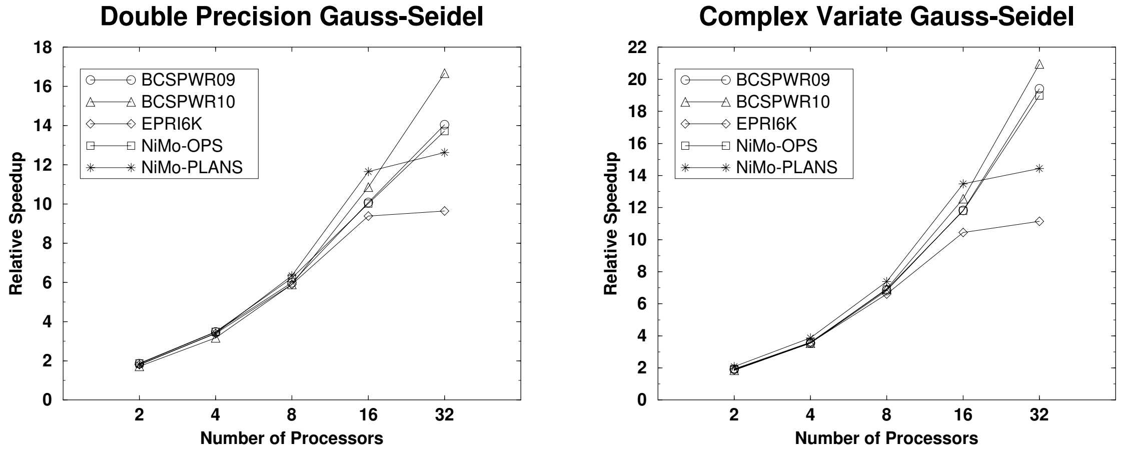


Figure 7.15: Relative Speedup — Parallel Gauss-Seidel

algorithm and the first version of the parallel block-diagonal-bordered sparse Gauss-Seidel solver. Modifications to the first parallel Gauss-Seidel algorithm yielded an algorithm that is performance competitive with the best sequential Gauss-Seidel algorithm.

These figures illustrate that parallel performance of the double precision parallel Gauss-Seidel implementation can be as much as 17 for 32 processors and 21 for the complex variate implementation. The best speedups were obtained with the BCSPWR10 power systems network, and nearly as good relative speedups were obtained for the BCSPWR09 and NiMo-OPS networks. The effects of load-imbalance overhead, described in the previous section, clearly affect the relative speedup for the EPRI6K and NiMo-PLANS networks. Performance for all networks is similar for two through sixteen processors; however, there is a radical change in the rate of increase for relative speedup with 32 processors for these two planning matrices.

7.2.2.4 Analyzing Algorithm Component Performance

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

1. calculate $\mathbf{x}^{(k+1)_i}$ in the diagonal blocks,
2. update $\hat{\mathbf{b}}$ using values in the lower border,
3. calculate $\hat{\mathbf{x}}^{(k+1)}$ using the values of $\hat{\mathbf{b}}$ and the last diagonal block $\mathbf{A}_{m+1,m+1}$,
4. perform a convergence check.

Detailed parallel algorithm analysis will demonstrate that the preprocessing phase can effectively load balance the matrix for as many as sixteen processors for all networks examined and can effectively load balance the matrix for as many as 32 processors for certain classes of data sets. We present graphs that illustrate algorithm component performance in figure 7.16. Each graph has four curves that show parallel Gauss-Seidel component performance for a single iteration.

These figures corroborate the results of the previous two sections that identified load imbalance for the two planning networks — EPRI6K and NiMo-PLANS. The graphs with performance data from the BCSPWR09, BCSPWR10, and NiMo-OPS power systems matrices show good load balancing for the diagonal blocks and lower border; however, the graphs for the EPRI6K and NiMo-PLANS data show degraded performance for 32 processors. Load imbalance is evident when empirical performance data for calculating $\mathbf{x}^{(k+1)}$ in the diagonal blocks does not yield a straight line. Load imbalance is also the likely cause that the slope of individual curve splines, both for updating $\hat{\mathbf{b}}$ in the last diagonal block and for performing convergence checks, do not have constant slope. Previous graphs showed little effect by increasing the computation-to-calculations granularity, so any degraded performance would be due to sources of overhead other than communications overhead.

The times to calculate $\hat{\mathbf{x}}^{(k+1)}$ in the multi-colored last diagonal block are always the least of the four operations for all five power systems networks, and for all but the planning matrices, the time to solve for $\hat{\mathbf{x}}^{(k+1)}$ is monotonically decreasing. Communications overhead, if it exists, would occur in this algorithm component as the number of processors increases.

We draw the following conclusions from this detailed examination of the parallel Gauss-Seidel algorithm components:

- The low-latency, active message-based implementations are able to obtain good performance improvements for all algorithm components as the number of processors increase, even when solving for $\hat{\mathbf{x}}^{(k+1)}$ for small operations matrices.

- Power systems networks can vary greatly — planning networks may be larger than operations networks, and these matrices have different characteristics. Planning matrices are likely to have the poorest performance for 32 processors due to load imbalance. For operations matrices, performance times for every component are monotonically decreasing, illustrating good load balance. For planning matrices, performance at 16 and 32 processors show the limitations of our preprocessing phase to order these matrices for the parallel Gauss-Seidel algorithm. Nevertheless, this parallel block-diagonal-bordered algorithm can get speedups of over 20 for 32 processors with large power systems networks with homogeneous voltage lines throughout the entire matrix. Operations matrices demonstrate performance that is nearly as good.

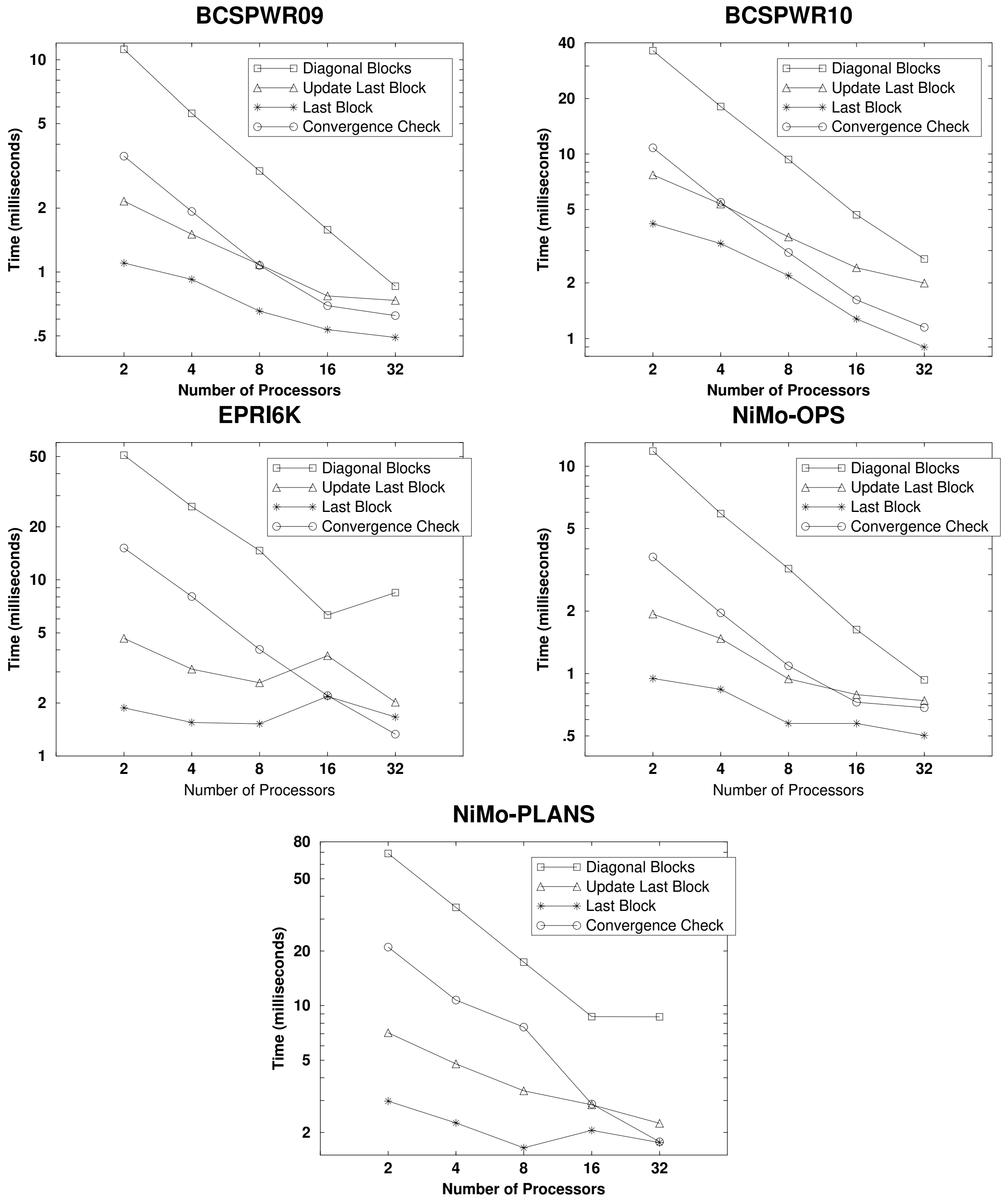


Figure 7.16: Algorithm Component Timing Data — Double Precision Gauss-Seidel

Table 7.2: Convergence for the BCSPWR09 Power Systems Network

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	1571.468461828037	0.00224373	0.33095152
2	81.288218360425	0.00037820	0.09514502
3	3.853385295799	0.00020072	0.08294093
4	0.174783477277	0.00020007	0.08292770
5	0.007786745594	0.00020006	0.08292727
6	0.000348455903	0.00020006	0.08292725
7	0.000015535582	0.00020006	0.08292725
8	0.000000688160	0.00020006	0.08292725
9	0.000000030701	0.00020006	0.08292725
10	0.000000001391	0.00020006	0.08292725
11	0.000000000064	0.00020006	0.08292725
12	0.000000000003	0.00020006	0.08292725

7.2.3 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 sparse Gauss-Seidel solver to sample positive definite matrices with the sparsity pattern from actual power systems networks and random values for the entries. We have examined convergence for various matrices and various matrix orderings. Samples of the measured convergence data are presented in tables 7.2 and 7.3 for the BCSPWR09 and BCSPWR10 power systems networks respectively. These tables present 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 zero.

In both tables 7.2 and 7.3, convergence is rather rapid, and after twelve iterations, total error is less than 1×10^{-10}. Consequently, only eight iterations are required for six decimal place accuracy with these data sets. In a positive definite matrix, the maximum values in the matrix fall on the diagonal. In this generated data, the magnitude of the diagonals were set equal to the number of non-zeros in the row plus a uniformly distributed random number between zero and one while the off-diagonal values were set equal to a uniformly distributed randomly number between zero and one. The values of $\mathbf{b}$ were set equal to one plus a uniformly distributed random number between

Table 7.3: Convergence for the BCSPWR10 Power Systems Network

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	4758.441355684042	0.00258651	0.55232051
2	303.405396555044	0.00140069	0.11749015
3	15.449365377376	0.00108438	0.09064293
4	0.782082340337	0.00102508	0.09046819
5	0.040509560300	0.00101835	0.09045822
6	0.002134338000	0.00101787	0.09045768
7	0.000114933652	0.00101784	0.09045765
8	0.000006358080	0.00101784	0.09045765
9	0.000000362744	0.00101784	0.09045765
10	0.000000021373	0.00101784	0.09045765
11	0.000000001301	0.00101784	0.09045765
12	0.000000000082	0.00101784	0.09045765

zero and one. If the relative magnitude of the diagonals with respect to the off-diagonals is larger, convergence will be even faster

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 networks, this solver would be used within an iterative non-linear solver, so even better estimates of starting points for each solution will be readily available, especially for transient stability simulations where differential-algebraic equations are solved for small time increments.

7.2.4 Comparing Communications Paradigms

We have developed two versions of this parallel block-diagonal-bordered sparse linear solver: one version uses a low-latency, active message-based communications paradigm and the other uses a buffered communications-based paradigm. These communications paradigms significantly modified the respective algorithms as seen in section 6.3.

The graphs in figure 7.17 illustrate direct comparisons of relative speedup for the low-latency, active message-based communications implementation and the buffered communications implementations for two power systems network matrices: BCSPWR09 and BCSPWR10. Performance

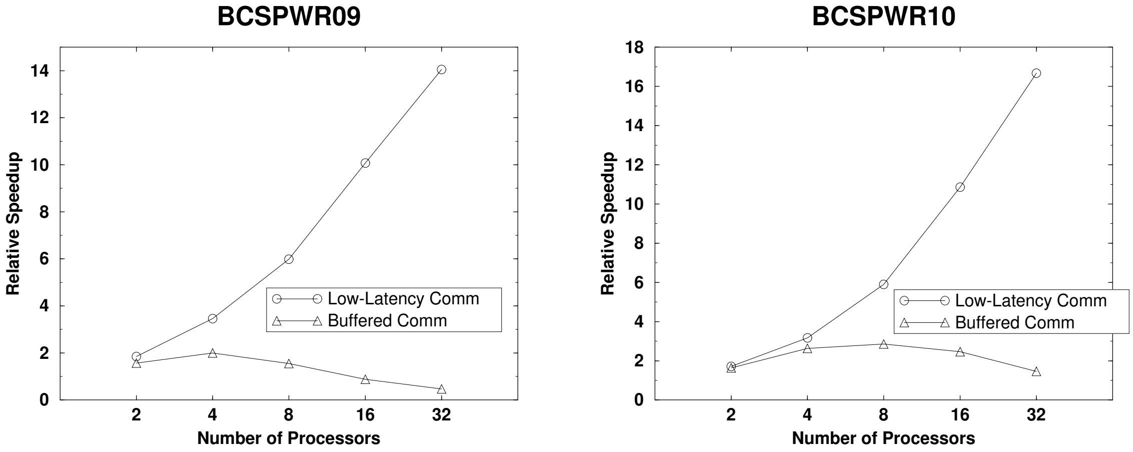


Figure 7.17: Relative Speedup — Double Precision Parallel Gauss-Seidel

for the other data sets were similar. These figures clearly illustrate the superior performance of the low-latency communications paradigm for the parallel block-diagonal-bordered sparse Gauss-Seidel solver. The low-latency implementations are always faster, even for two processors, and clearly faster for 32 processors. For the algorithm based on a more traditional send and receive paradigm, performance quickly becomes unacceptable as the number of processors increases. With the buffered communications-based implementation *no* speedup was measured for 16 and 32 processors. Meanwhile, speedups of as great as fourteen were measured for the double precision low-latency communications-based implementation. The remainder of this section discusses the reasons for the drastic differences in algorithm performance as a function of interprocessor communications paradigms.

For the low-latency communications-based parallel Gauss-Seidel algorithm, the amount of communications is greatly reduced by only sending values of $\hat{\mathbf{x}}^{(k+1)}$ to those processors that actually need them when solving for an iteration in the last diagonal block. Figure 7.18 illustrates the number of low-latency messages required to distribute the values of $\hat{\mathbf{x}}^{(k+1)}$ calculated, while figure 7.19 presents the percentage of low-latency messages required to distribute these values. Families of curves in figure 7.18 show that the number of low-latency messages increases apparently at linear rates (with a $\log(N_{procs})$ horizontal axis) for three of the five power systems networks. This implies that the number of low-latency messages increases at a rate proportional to $\log(N_{procs})$. Meanwhile, figure 7.18 illustrates the percentages of data actually sent with the low-latency communications paradigm relative to the maximum possible for a broadcast. For 32 processors, only 10% to 18% of the broadcast values are actually required.

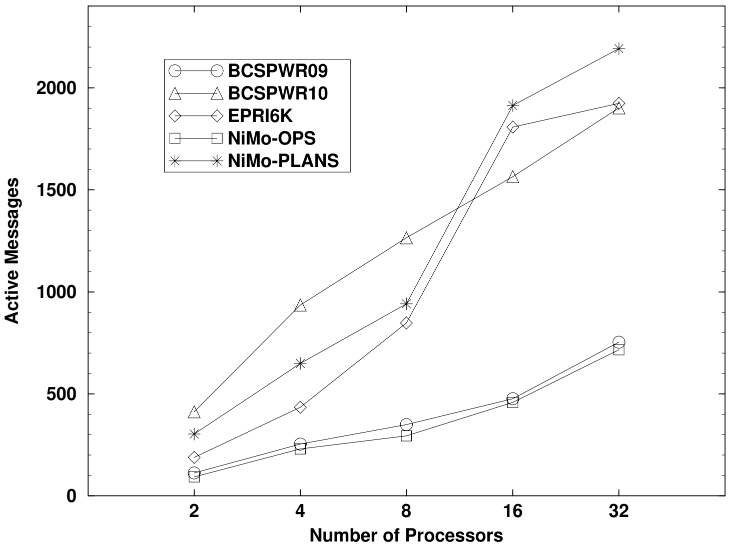


Figure 7.18: Number of Low-Latency Messages Required to Distribute $\hat{\mathbf{x}}^{(k+1)}$

In figures 7.20 and 7.21, we explore this phenomenon further for the BCSPWR09 and BCSPWR10 power systems networks. These figures present families of histograms of the number of low-latency messages to distribute the values of $\hat{\mathbf{x}}^{(k+1)}$. Each histogram shows the distribution of the number of required low-latency messages, and is labeled to emphasize the maximum number of messages, $(N_{procs} - 1)$. For the BCSPWR09 network, the maximum number of processors requiring any single value of $\hat{\mathbf{x}}^{(k+1)}$ is only eleven. Likewise, for the BCSPWR10 network, the maximum number of processors requiring any single value of $\hat{\mathbf{x}}^{(k+1)}$ is only eight. Significantly reducing the amount of communications in this component of the algorithm makes a corresponding improvement in overall parallel Gauss-Seidel algorithm performance. As a result, we are able to attain speedup even for an algorithm component that could have been sequential, and would have limited the overall algorithm performance as a function of Amdahl's law.

The effect of reduced communications overhead can be clearly seen as performance of the low-latency algorithm is compared to performance of an algorithm with more traditional buffered communications. In the portion of the algorithm that solves for values of $\hat{\mathbf{x}}^{(k+1)}$, the buffered communications implementation must broadcast the values of $\hat{\mathbf{x}}^{(k+1)}$ to all other processors before the next color can proceed. The number of communications messages is $O(N_{procs}^2)$ for traditional interprocessor communications. Consequently, as the number of processors increases, the number of

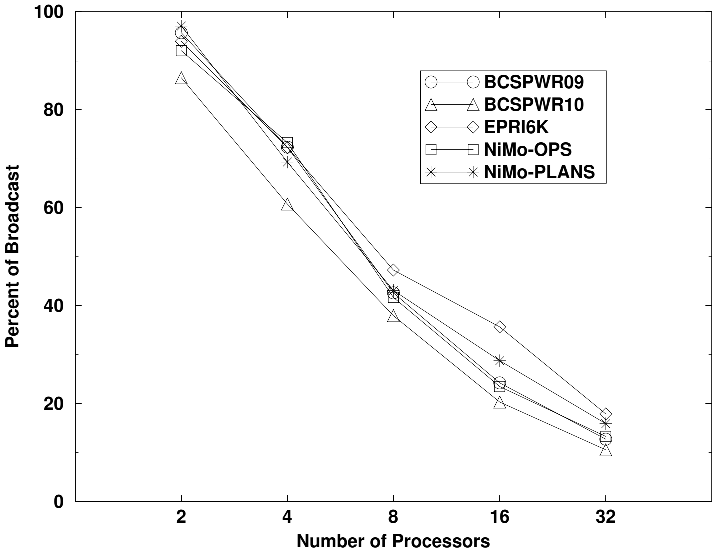


Figure 7.19: Percentage of Broadcast Values Required to Distribute $\hat{\mathbf{x}}^{(k+1)}$

messages increase but the buffered communications messages size decreases. For traditional message passing paradigms, the cost for communications increases dramatically as the number of processors increases, because each message incurs the same latency regardless of the amount of data sent. The cost of non-blocking, buffered communications is 86 μseconds latency and .12 μseconds per word or four bytes of buffered data on the Thinking Machines CM-5. Meanwhile, with the low-latency paradigm, the cost of an active message is only 1.6 μseconds to transfer four words of data.

In figure 7.22, we present speedup comparisons of the low-latency communications-based algorithm and the buffered communications-based algorithm. This series of graphs includes a comparison of overall speedup of the low-latency communications paradigm versus the buffered communications paradigm, and separate graphs of speedup for each of the three portions of the algorithm that have interprocessor communications. Overall, the small operations matrices have speedups for low-latency communications of greater than 28. The speedups for the larger matrices are smaller, because buffered communications are more efficient — more data can be sent in each message, improving performance for this message type. Nevertheless, for 32 processors, speedups from between nine and thirteen have been measured for four iterations and a convergence check on these power systems network matrices.

The other graphs in figure 7.22 illustrate the speedups for the algorithm components:

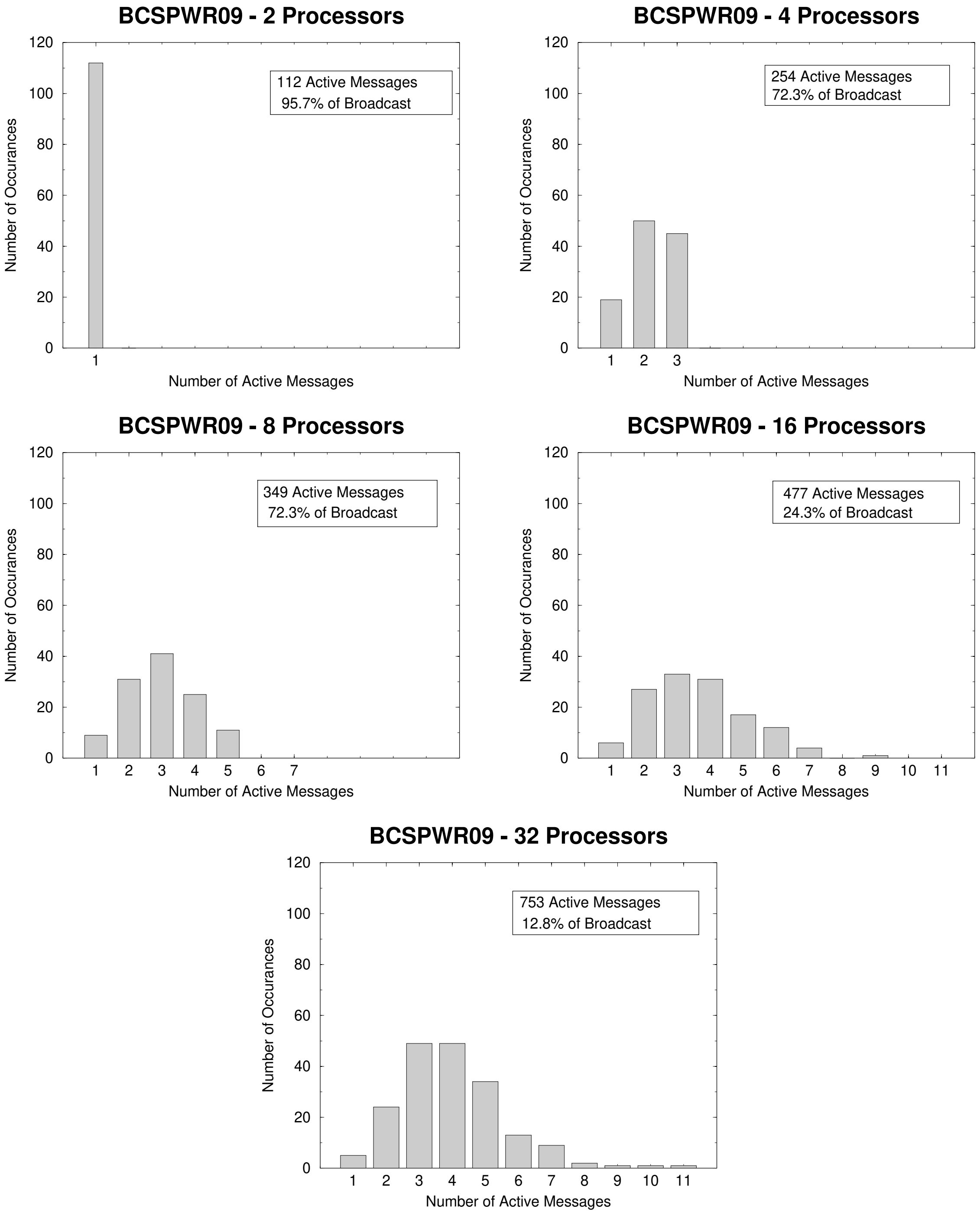
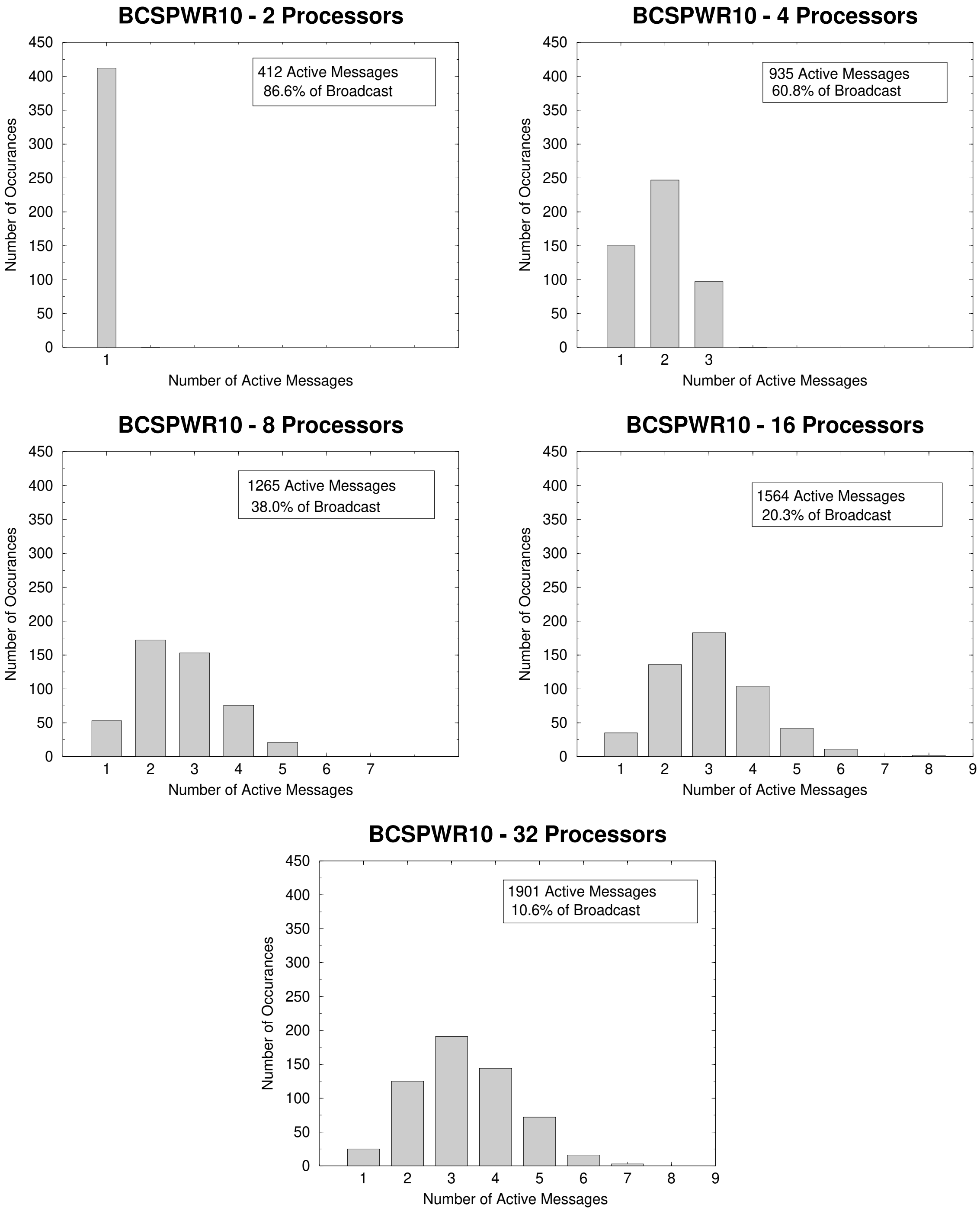


Figure 7.20: BCSPWR09 — Histograms of the Number of Messages to Distribute $\hat{\mathbf{x}}^{(k+1)}$



BCSPWR10 - 2 Processors

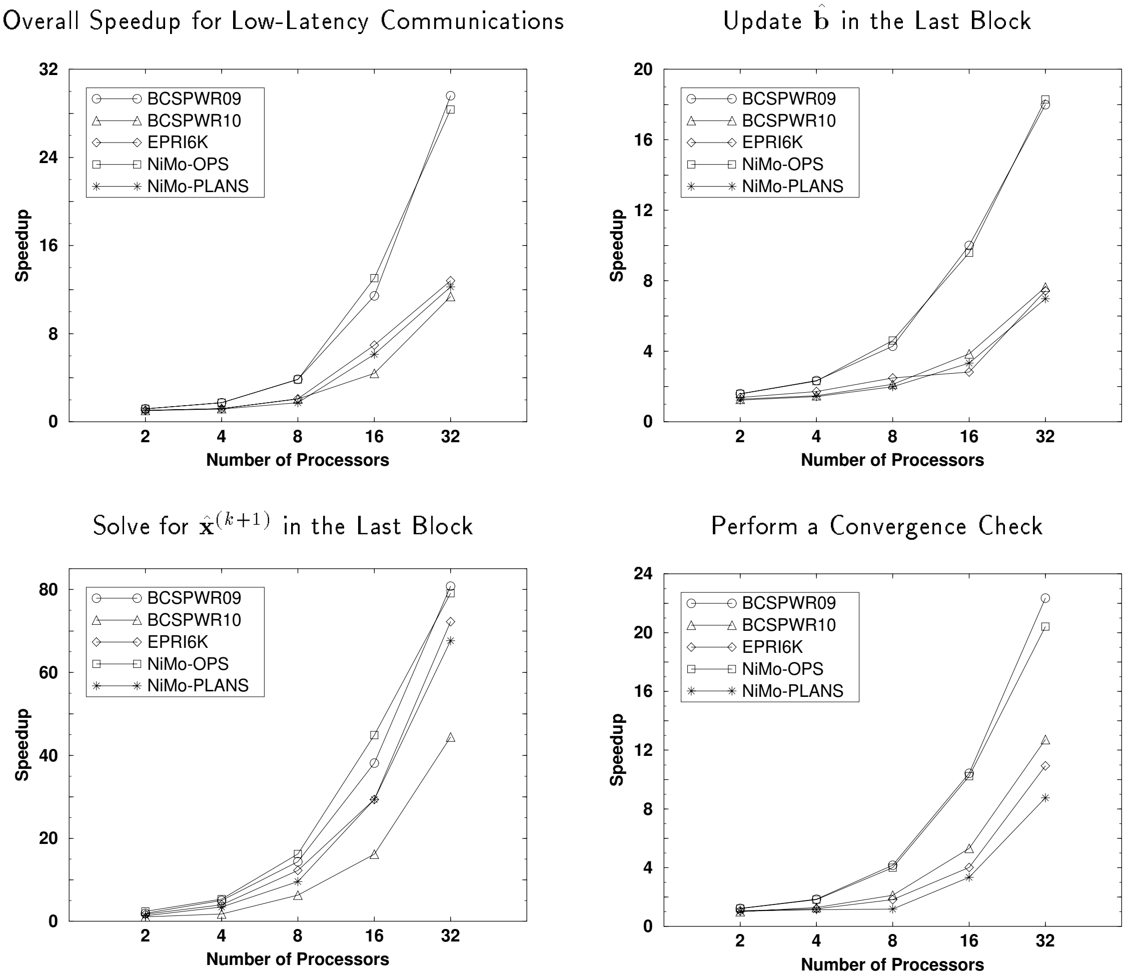
BCSPWR10 - 4 Processors

BCSPWR10 - 8 Processors

BCSPWR10 - 16 Processors

BCSPWR10 - 32 Processors

Figure 7.21: BCSPWR10 — Histograms of the Number of Messages to Distribute $\hat{\mathbf{x}}^{(k+1)}$


Overall Speedup for Low-Latency Communications

Update $\hat{\mathbf{b}}$ in the Last Block

Solve for $\hat{\mathbf{x}}^{(k+1)}$ in the Last Block

Perform a Convergence Check

Figure 7.22: Low-Latency Communications Speedup — Parallel Gauss-Seidel

- empirical performance when updating $\hat{\mathbf{b}}$ before solving the last diagonal block yields speedups from seven to nineteen for the low-latency message implementation,
- empirical performance when solving for $\hat{\mathbf{x}}^{(k+1)}$ yields speedups from 45 to greater than 80 for the low-latency message implementation,
- empirical performance when checking convergence yields speedups from eight to 22 for the low-latency message implementation.

7.2.5 Conclusions

We have extensively analyzed the performance of parallel solvers for power systems applications on the Thinking Machines CM-5. We have shown that the node-tearing-based partitioning algorithm can yield matrices in block-diagonal-bordered form with balanced workloads for power systems networks with homogeneous voltage distribution lines; and we have shown that the performance of our parallel block-diagonal-bordered sparse iterative linear solvers can yield good speedups for Gauss-Seidel methods for those networks with balanced workloads. Not unexpectedly, low-latency communications paradigms greatly improve the performance of the algorithm, because of both improved communications performance and significantly simpler implementations.

7.3 Comparing Parallel Direct and Iterative Algorithms

We have developed both parallel direct and iterative methods in this research, with parallel direct implementations yielding speedups of nearly ten for double precision LU factorization and even greater speedups for complex variate LU factorization with 32 processors. Speedups for parallel block-diagonal-bordered Choleski factorization were less than for LU factorization, only 4.5 for 32 processors, and there are formidable problems implementing forward reduction due to the distribution of data to processors in the last diagonal block. Meanwhile, we were able to obtain significantly greater speedups, 17 to 21 for 32 processors, with our parallel block-diagonal-bordered sparse Gauss-Seidel solver implementations, although the only matrix types where there is assurance of convergence for Gauss-Seidel methods are diagonally dominant and positive definite matrices. Choleski factorization is limited to either of these symmetric matrix types. When comparing the performance of parallel sparse Choleski solvers and parallel sparse Gauss-Seidel algorithm, we show that there is potential for significant algorithmic speedup with the use of the iterative solver.

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

$$T_s(\delta) = T_f + (\delta \times T_{\triangle}), \tag{7.1}$$

where:

$T_s(\delta)$ is the total time for a single factorization and δ triangular solutions.
T_f is the time for (parallel) factorization.
δ is the number of dishonest *(re)uses*:.
$T_{\triangle}$ is the time for a (parallel) triangular solution.

It is important to note that both the sequential and parallel implementations of Gauss-Seidel yield the same convergence rate. The parallel block-diagonal-bordered form Gauss-Seidel solver maintains the same strict precedence relation in the calculations as does the sequential algorithm.

7.3.1 Parallel Choleski versus Parallel Gauss-Seidel

We compare the performance of parallel Choleski solvers with parallel iterative Gauss-Seidel solvers by determining the number of iterations for the parallel Gauss-Seidel given a number of *(re)uses*. Families of curves plotting the number of iterations versus the number of *dishonest (re)uses* are presented in figure 7.23 for one through ten reuses and one through 32 processors for power systems networks: BCSPWR09 and BCSPWR10. The shape of the curves show that the largest number of iterations possible for a constant time solution occur for a single use of the factored matrix. As the factorization is *(re)used*, the cost to factor the matrix is amortized over the additional *(re)uses*. For large numbers of factorization *(re)uses*, the curve becomes asymptotic to

$$y = \frac{T_{\triangle Ch}}{T_{GS}}, \tag{7.2}$$

where:

$T_{\triangle Ch}$ is the time for a single (parallel) Choleski triangular solution.
T_{GS} is the time for a single (parallel) Gauss-Seidel iteration.

For the other power systems network matrices examined in this research, performance is similar to these graphs.

The graph for the BCSPWR09 operations matrix in figure 7.23 illustrates that on a single processor, 12 Gauss-Seidel iterations take as much time as a single factorization and triangular solution. Meanwhile, only four iterations per solution would equal the time for 10 *dishonest (re)uses*.

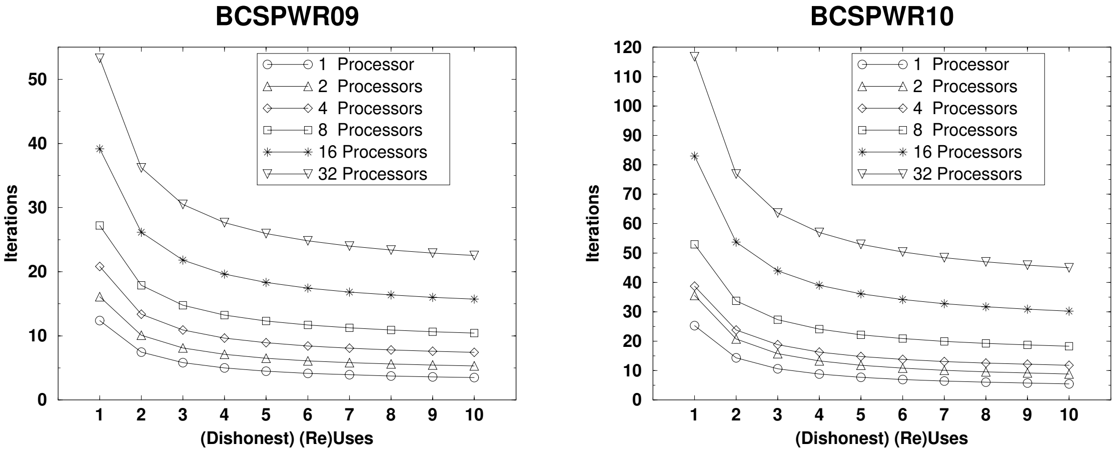


Figure 7.23: Gauss-Seidel Iterations as a Function of Dishonest Reuses of the Choleski $\mathbf{LL}^T$ Matrix

However, when 32 processors are utilized, 54 Gauss-Seidel iterations could be performed in the same time as a single direct solution, and 24 iterations per solution for 10 *dishonest (re)uses*. The graph for the BCSPWR10 operations matrix in figure 7.23 illustrates even greater numbers of iterations — nearly 120 Gauss-Seidel iterations could be performed in the same time as a single direct solution for 32 processors, and 55 iterations per solution for 10 *dishonest (re)uses*. These comparisons are for a convergence check every four iterations. Previous discussions on Gauss-Seidel convergence in section 7.2.3. have concluded that after twelve iterations, total error is less than 1×10^{-10}. Only eight iterations are required for six decimal place accuracy with data sets generated for actual sparse power systems networks. Given that there are good starting points for each successive iterative solution, there is a strong possibility that the use of parallel Gauss-Seidel should yield significant algorithmic speedups for diagonally dominant or positive definite sparse matrices. For these two cases, such speedups could be as high as a factor of ten for large data sets.

7.3.2 Parallel LU Solvers versus Parallel Gauss-Seidel

LU factorization has none of the limitations on matrix characteristics that define Choleski factorization. As a result, there is no guarantee that the iterative solution will converge. Nevertheless, we extend the performance comparison of parallel direct solvers with parallel iterative Gauss-Seidel solvers from the previous section. Again we compare direct and iterative solver performance by determining the number of iterations for the parallel Gauss-Seidel given a number of *(re)uses*. Families of curves plotting the number of iterations versus the number of *dishonest (re)uses* for a double precision parallel block-diagonal-bordered LU-based solver are presented in figure 7.24 for one through

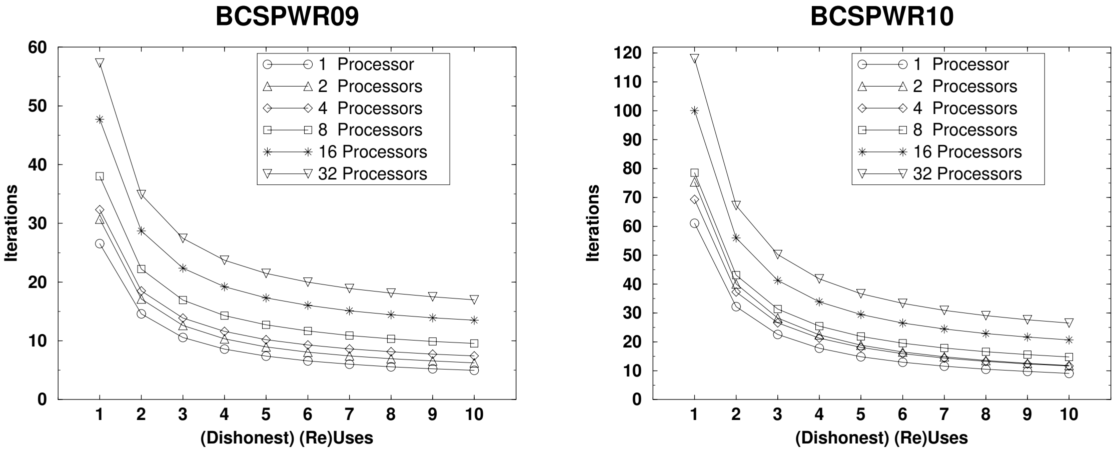


Figure 7.24: Gauss-Seidel Iterations as a Function of Dishonest Reuses of the Double Precision **LU** Matrix

ten reuses and one through 32 processors for the BCSPWR09 and BCSPWR10 power systems networks. Likewise, we present similar families of curves in figure 7.25 for parallel complex variate LU-based and Gauss-Seidel solvers. The shape of the curves show that the largest number of iterations possible for a constant time solution occur for a single use of the factored matrix. As the factorization is *(re)used*, the cost to factor the matrix is amortized over the additional *(re)uses*. For large numbers of factorization *(re)uses*, the curve becomes asymptotic to

$$y = \frac{T_{\triangle LU}}{T_{GS}}. \tag{7.3}$$

where:

$T_{\triangle LU}$ is the time for a single (parallel) LU triangular solution.
T_{GS} is the time for a single (parallel) Gauss-Seidel iteration.

Graphs in figures 7.24 and 7.25 illustrate that for one use of the factored matrix, the number of Gauss-Seidel iterations per dishonest reuse of the factored matrix are more for LU factorization than for parallel Choleski solvers. We have shown in section 7.1 that the time to factor a matrix into **LU** versus $\mathbf{LL}^T$ is greater because there are twice as many calculations in the LU factorization than Choleski factorization. However, the time to perform the triangular solutions are less for parallel LU solvers than parallel Choleski solvers — Choleski solvers must perform one of the triangular solutions for the last diagonal block with less than optimal communications. As a result, while the number of iterations for a single reuse of the double precision solver may be greater, the number of available

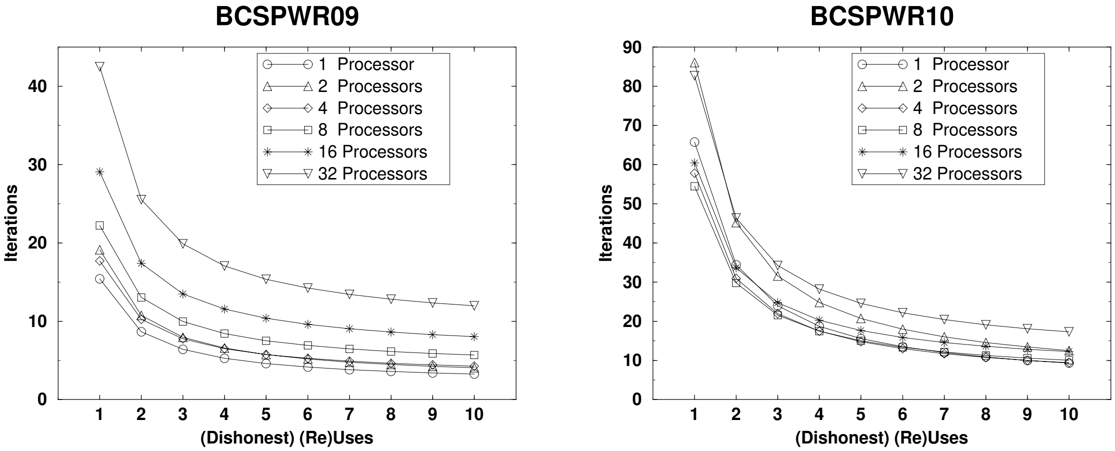


Figure 7.25: Gauss-Seidel Iterations as a Function of Dishonest Reuses of the Complex **LU** Matrix

iterations for multiple dishonest reuses actually decreases when compared to the Choleski/Gauss-Seidel comparison curves. The effects of the higher cost in the triangular solution phase can be clearly seen when comparing graphs in figures 7.23 and 7.24. For ten reuses with 32 processors, there would be time for 45 iterations per Choleski solution for the BCSPWR10 data set, meanwhile, there would only be time for 25 parallel Gauss-Seidel iterations per parallel double precision LU solution.

Parallel performance of the complex LU solver increases significantly when compared to the parallel Choleski and substantially when compared to the double precision LU solver. Meanwhile, there is only a small improvement in performance for the added calculations in parallel complex variate Gauss-Seidel versus the double precision version of this solver. As a result, parallel complex Gauss-Seidel would offer less potential for improved performance than for parallel double precision Gauss-Seidel.

7.3.3 Conclusions

Due to the guaranteed convergence of the Gauss-Seidel algorithm for positive definite or diagonally dominate matrices and the relative performance for the parallel Choleski solver, there is potential for significant speedup by selecting the parallel Gauss-Seidel method for solving those power systems network matrices that would normally require double precision Choleski factorization. For those applications that require LU factorization, more information concerning the iterative solver convergence would be required in addition to the rate of convergence before decisions could be made concerning the selection of direct versus iterative methods.

Chapter 8

Algorithm Performance on Future SPP Architectures

We design and implement algorithms on existing hardware; however, for industrial applications such as power systems network analysis, it is equally important to predict algorithm performance for future architectures. Performance predictions for future architectures will help determine whether or not it will be cost-effective to port critical software to parallel architectures now or to simply wait and get speedup in the future from faster single processor computers.

This analysis is a good case in point — performance for the parallel block-diagonal-bordered sparse solvers developed here is rather good on the Thinking Machine CM-5 for moderate number of processors (2–32). For Choleski solver applications, the parallel block-diagonal-bordered Gauss-Seidel algorithm yields good speedups and offers substantial algorithmic speedup when compared with parallel block-diagonal-bordered direct solvers. However, in this section we show that the superb computation-to-communication ratio available on the CM-5 using low-latency active messages will probably not be equaled in future architectures where processor performance increases significantly. Performance of our parallel Gauss-Seidel algorithm is latency dependent, due to the large number of small messages. Meanwhile, performance of our parallel direct algorithm is bandwidth dependent, due to the limited number of moderate size messages.

We show in this chapter that while the bandwidth-dependent parallel sparse block-diagonal-bordered direct solvers may port to future architectures with equal or better performance, the latency-dependent parallel sparse block-diagonal-bordered Gauss-Seidel solvers may not. While future architectures will have greater bandwidth than

the Thinking Machines CM-5, they will not have a comparable reduction in communications latency. Any algorithmic performance gains possible with the parallel Gauss-Seidel algorithm would not be realized on future architectures that do not have the computation-to-communication ratio available on the CM-5.

We open this chapter by discussing future computing architectures and the requirements of the power utility industry in section 8.1. We introduce overhead-based performance estimates, in section 8.2, that we developed to predict algorithm performance on future high-performance computing architectures. We apply these estimation techniques to both sparse parallel block-diagonal-bordered direct and iterative solvers developed in this research in sections 8.3 and 8.4. Due to the poor performance of the parallel iterative solver on future SPP architectures, we include comments on improving the latency performance of SPP communications in section 8.5, and in section 8.6, we reiterate the significant conclusions for porting our parallel linear solvers to future SPP architectures.

8.1 Parallel Computing for the Power Utility Industry

We believe that a power utility's interests in future parallel architectures will be in scalable parallel processors (SPPs) rather than massively parallel processors (MPPs), because:

1. the compatibility of SPP nodes with networked desktop computing resources contributes to reduced business overhead costs,
2. small to midsized SPPs offer an improved cost/performance ratio when compared to small MPPs.

We can expect future SPP architectures to be similar to the IBM SP-series with 2-64 processors interconnected by a non-blocking, high-bandwidth, switched network [16]. Internode communications performance may soon approach that of the Cray T3D massively parallel computer (1 μsecond latency and 300 megabytes per second bandwidth) [43]. When comparing the single processor performance of the CM-5 (a 33 MHz Sparc microprocessor from Sun Microsystems) [6] with a node of the Cornell Theory Center SP1 or Northeast Parallel Architectures Center (NPAC) SP2 (a 62.5 MHz IBM RS/6000 model 370 four command superscalar microprocessor), we have shown in section 7.1, that the IBM RS/6000 microprocessor in the SP1 is 6.6 times faster than the 33 MHz Sparc microprocessor when comparing empirical data from our algorithm run on a single processor. The speed for the microprocessor in the Cornell Theory Center SP2 is even 50% faster than the SP1 RS/6000 microprocessor. In the near-future, it will be feasible to get four times the individual processor power that is now available on the SP1, so it is conceivable that the future generation of SPP microprocessors will be 25 times as fast as those used in the Thinking Machines CM-5. Some of this processing power may come from placing multiple shared-memory processors per SPP node [16],

If SPP node processor capability increases by a factor of 25 relative to the Thinking Machines CM-5, communications capabilities must improve by at least as much if parallel sparse direct linear solver performance for power systems applications is to have equal or better multiple processor speedup. In other words, the computation-to-communications ratio for the SPP must remain constant or improve in order that SPP speedup remains constant or improves.

8.2 Overhead-Based Performance Estimates

For many concurrent algorithms, overhead associated with the concurrent algorithm appears more critical than the inherent percentage of sequential operations in the algorithm. In these situations, parallel overhead provides a better preliminary estimate of the potential speedup in a concurrent algorithm than Amdahl's Law. In these instances, the parallel execution time for an algorithm can be defined as

$$T_p \equiv \frac{T_{seq}}{p}(1 + f_t) \quad [17], \tag{8.1}$$

where:

T_p is the parallel execution time.

T_{seq} is the sequential execution time.

p is the number of processors — $p = N_{procs}$.

f_t is the total parallel overhead.

For predicting parallel algorithm performance with future architectures, we assume that the algorithm will be applied to a matrix that can be partitioned into block-diagonal-bordered form with no load-imbalance, and we assume that the same algorithm is being implemented on both architectures. As a result, we can assume that the only component of interest in f_t, the total parallel overhead, is f_c, the communications overhead, and we can rewrite equation 8.1 as

$$T_p \equiv \frac{T_{seq}}{p}(1 + f_c). \tag{8.2}$$

This can be further rewritten as

$$f_c = \left(\frac{pT_p}{T_{seq}} - 1\right) = \left(\frac{p}{S_p} - 1\right), \tag{8.3}$$

by substituting relative speedup, S_p for $\left(\frac{T_{seq}}{T_p}\right)$. Relative speedup is defined in equation A.2.

Communications overhead, f_c, is a measure of the additional workload incurred in a parallel algorithm as a result of interprocessor communications [17, 25, 28], and is dependent on the computation-to-communications ratio, not just the amount of communications

$$f_c \sim \frac{t_{comm}}{t_{calc}}, \tag{8.4}$$

or

$$f_c = \left(\alpha \times \frac{t_{comm}}{t_{calc}}\right), \tag{8.5}$$

where t_{calc} is the metric describing the computational capability of a single processor [17], t_{comm} is the metric describing the communications characteristics, and the constant α is an algorithm dependent coefficient of proportionality. The quantity $\frac{t_{calc}}{t_{comm}}$ is often referred to as the computation-to-communications ratio and is related to the granularity of the parallel algorithm. For traditional buffered interprocessor communications, t_{comm} is a linear combination of latency, bandwidth, and message size

$$t_{comm} \equiv t_{latency} + (\frac{4}{\mathcal{B}_{bytes}} \times N_{words}). \tag{8.6}$$

$t_{latency}$ is the communications latency or startup time, $\mathcal{B}_{bytes}$ is the bandwidth measured in bytes per second, and N_{words} is the number of words or four byte units of data. For active messages, t_{comm} is the product of the message latency and number of messages, N_{RPC},

$$t_{comm} \equiv (t_{latency} \times N_{RPC}). \tag{8.7}$$

We can determine α, the algorithm dependent constant of proportionality by combining formulas 8.3 and 8.5 to yield

$$\alpha = \left(\frac{t_{calc}}{t_{comm}}\right)\left(\frac{p}{S_p} - 1\right). \tag{8.8}$$

We can calculate estimates of speedup for a new parallel architecture without detailed simulation of an algorithm by using available empirical timing data to calculate the algorithm dependent constant of proportionality, and combine it with the number of processors and parallel architecture characteristics. Let $\tilde{f}_c$ be the communication overhead for the new architecture, then

$$\tilde{f}_c = \alpha\left(\frac{\tilde{t}_{comm}}{\tilde{t}_{calc}}\right) = \left(\frac{t_{calc}}{t_{comm}}\right)\left(\frac{p}{S_p} - 1\right)\left(\frac{\tilde{t}_{comm}}{\tilde{t}_{calc}}\right). \tag{8.9}$$

Terms can be reordered to yield

$$\tilde{f}_c = \left(\frac{t_{calc}}{\tilde{t}_{calc}}\right)\left(\frac{\tilde{t}_{comm}}{t_{comm}}\right)\left(\frac{p}{S_p} - 1\right). \tag{8.10}$$

The value of $\tilde{f}_c$ can be used to calculate $\tilde{S}_p$, the speedup for the new architecture, using

$$\tilde{S}_p = \frac{p}{(1 + \tilde{f}_c)} = \frac{p}{\left(1 + \left(\frac{t_{calc}}{\tilde{t}_{calc}}\right)\left(\frac{\tilde{t}_{comm}}{t_{comm}}\right)\left(\frac{p}{S_p} - 1\right)\right)}. \tag{8.11}$$

In the next section, we present an analysis that predicts performance for future architectures using these formulas.

8.3 Performance Predictions for Direct Solvers

We implemented two versions of the parallel block-diagonal-bordered sparse direct solver on the Thinking Machines CM-5 and the notable differences between the two implementations are the communications paradigms when updating the last diagonal block in the matrix. One communications paradigm uses low-latency, active message-based communications, and the other uses buffered communications. Active message-based communications on the CM-5 has latency of 1.6 μsecond to send four words, while the buffered communications version of the algorithm utilizes the traditional CMMD communications library, which has 86 μsecond latency and 0.12 μsecond per word communications costs [6]. Both versions of the algorithm utilized the active message s-copy-based buffered communications for factoring the last diagonal block. S-copy communications has 23 μsecond latency and 0.12 μsecond per word communications costs [6]. The CM-5 has a multi-tiered communications network with 40 megabyte-per-second bandwidth at the lowest layer [6].

8.3.1 Model Validation

In order to validate our formulas for comparing speedup on different architectures, we have used empirical performance data from section 7.1 and developed performance comparisons for complex variate LU factorization on the SP1 and SP2. We have extended the number of processors beyond the number actually utilized to collect the empirical data from the SPP architectures. Our implementations on the SP1 and SP2 used the Message Passing Interface (MPI), because it is being developed as a communications standard for multi-processors with strong emphasis on optimizing message-passing performance. The IBM SP2 has a 30 μsecond latency and 30 megabyte-per-second bandwidth in present configurations [16]. In figure 8.1, we present actual and predicted speedup values for the complex LU factorization algorithm with the EPRI6K power systems network for

1. empirical speedup data from the CM-5 implementation using buffered communications,
2. empirical speedup data from the SP1 implementation using MPI,
3. empirical speedup data from the SP2 implementation using MPI,
4. predicted speedup for the SP1 (6.6× processor speedup),
5. predicted speedup for the SP2 (9.9× processor speedup),

In this figure, we plot predicted values for both the SP1 and the SP2, where the single processor performance of the SP1 is 6.6 times the single processor performance of the Thinking Machines CM-5 and the SP2 is 50% faster than the SP1. When comparing the actual data to the predicted data, it may be possible that this simplistic technique has some difficulties in vertical displacement.

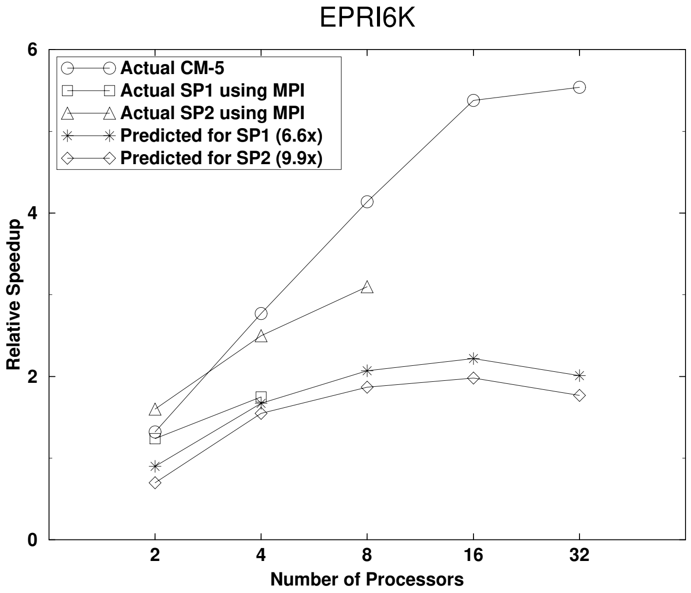


Figure 8.1: Performance Validation for Parallel Complex LU Factorization

While the slope of the curve splines for the predicted data and the curve splines for the empirical data are similar for the SP2, the predicted curve appears to underestimate the speedup by a nearly constant amount. These predictions have been made using a very simple model, and should only be interpreted as in indication of possible future performance. Better estimates are possible by carefully simulating the algorithms using models with more variables and greater detail.

8.3.2 Performance Predictions

In the near-future, we expect interprocessor communications for SPPs to improve significantly, with latency for buffered communications decreasing to levels that are available in MPPs like the Cray T3D today. We anticipate that buffered communications latency for SPPs, in the near future, will be only 1 μsecond, with 100 megabyte-per-second bandwidths between individual processors [43]. Per-word communications costs for this architecture should be less than 0.04 μsecond. In figure 8.2. we present actual and predicted speedup values for the complex LU factorization algorithm with the BCSPWR10 and EPRI6K power systems networks for

1. empirical speedup data from the CM-5 implementation using buffered communications,

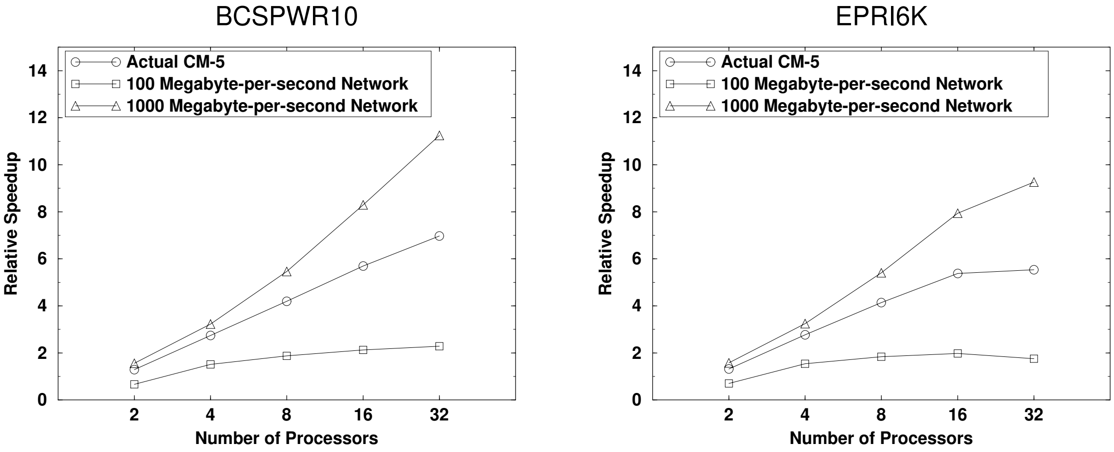


Figure 8.2: Performance Predictions for Parallel Complex LU Factorization

2. predicted speedup for 25× processor speeds and communications networks with 1 μsecond latency and 100 megabyte-per-second bandwidth,

3. predicted speedup for 25× processor speeds and communications networks with 1 μsecond latency and 1000 megabyte-per-second bandwidth.

The graphs in this figure show that we may see reduced speedup performance with this algorithm for 25× processor speeds and communications networks with 1 μsecond latency and 100 megabyte-per-second bandwidth. However, we may see improved speedup performance for a network that is 10 times faster. The slower network has a lower computation-to-communications ratio than the CM-5, but the faster network has a greater computation-to-communications ratio than the CM-5. The predicted performance for future architectures as seen in the two graphs presented in figure 8.2 is similar to the predicted performance that has been observed in all data sets, and is a direct result of the scaling imposed by the constant of proportionality defined as a function of the processor and communications performance in equation 8.10.

To fully understand the predicted speedup values, we will analytically examine how changes in the computation-to-communications ratio theoretically affect speedup on a future architecture. In order to be thorough, we will examine both sections of the parallel block-diagonal-bordered direct solver that have interprocessor communications:

1. update the last diagonal block using the data in the borders — $\mathbf{A}_{m+1,m+1} = \mathbf{A}_{m+1,m+1} - \sum_{i=1}^{m} \mathbf{L}_{m+1,i}\mathbf{U}_{i,m+1}$,

2. factor the last diagonal block — $\mathbf{A}_{m+1,m+1} = \mathbf{L}_{m+1,m+1}\mathbf{U}_{m+1,m+1}$,

and when appropriate, we will examine both implementations:

1. low-latency communications,
2. buffered communications.

We first examine updating the last diagonal block. For the low-latency communications version of the algorithm, we can't expect any improvement in the computation-to-communications ratio when updating the last diagonal block — we expect individual processor performance to yield decreases in run times by a factor of 25 and t_{comm} would decrease only to 1.16 μsecond from 1.6 μsecond. Thus the computation-to-communication ratio would decrease, accentuating the effect of communications overhead. Meanwhile, for buffered communications, most communications messages are of moderate size, approximately 500 words, so we can expect that communications performance in this section of the algorithm would improve substantially, by as much as a factor of seven ($\frac{(86+.12\times 500)}{(1+.04\times 500)}$). Due to the limited number of moderate sized messages, performance in this portion of our parallel direct algorithm using buffered communications is bandwidth dependent. If communications latency decreases as significantly as we anticipate and the communications bandwidth increases as expected, the version of the algorithm to update the last diagonal block that would yield the best performance would be the buffered communications algorithm. Nevertheless, in spite of the dramatic communications improvement, t_{comm} would not keep pace with the performance improvement of individual SPP processors, t_{calc}.

The communications in the section of the CM-5 program that factors the last diagonal block uses active message s-copy commands, which have 23 μsecond latency and 0.12 μsecond per word communications costs [6]. Messages are also of moderate size, several hundred words, so we can expect that communications performance would improve by a factor of nearly five ($\frac{(23+.12\times 200)}{(1+.04\times 200)}$). This is also significantly less than the factor of 25 improvement in single processor performance. Due to the limited number of moderate sized messages, performance in this portion of our parallel direct algorithm is also bandwidth dependent.

If we combine the three portions of the speedup analysis: improvements of a factor of 25 for the processor speed, and improvements of five to seven in the communications speeds, it may not be possible to sustain the parallel speedup that we have obtained in this example program. Performance may be limited for 32 processors; however, strong performance with fewer processors may be sustainable, because communications overhead is not as great with fewer processors. Consequently, we should be able to obtain significant speedups with a single processor due to increased processor performance, while additional speedup due to parallelism will be less than we obtained in this research.

If communications bandwidths between individual processors for our future machine improved an order of magnitude, to a gigabyte-per-second, the prognosis for this algorithm would change because of bandwidth dependency in this algorithm. For gigabyte-per-second networks, communications to update the last-diagonal block could improve by a factor of 48 ($\frac{(86+.12\times500)}{(1+.004\times500)}$) and communications to factor the last diagonal block could improve by a factor greater than 25 ($\frac{(23+.12\times200)}{(1+.004\times200)}$). As a result, the computation-to-communications ratio would be preserved, if not improved, and similar or better parallel speedups could be expected. This is clearly evident in the two graphs in figure 8.2.

8.4 Performance Predictions for Iterative Solvers

As stated above, we expect interprocessor communications for SPPs to improve significantly in the near-future, with latency for buffered communications decreasing to 1 μsecond, with 100 megabyte-per-second bandwidths between individual processors. Per-word communications costs for this architecture should be less than 0.04 μsecond. In figure 8.3. we present actual and predicted speedup values for the complex LU factorization algorithm with the BCSPWR10 and EPRI6K power systems networks for

1. empirical speedup data from the CM-5 implementation using low-latency communications,
2. predicted speedup for 25× processor speeds and communications networks with 1 μsecond latency and 100 megabyte-per-second bandwidth,
3. predicted speedup for 25× processor speeds and communications networks with 1 μsecond latency and 1000 megabyte-per-second bandwidth.

The two graphs in this figure show that we may see significantly reduced speedup for this algorithm with either future architecture. For the BCSPWR10 data set, with 25× processor speeds and communications networks with 1 μsecond latency and 100 megabyte-per-second bandwidth, speedups would be less than three for 32 processors and only slightly better, four, with a network that is 10 times faster. The computation-to-communications ratio for both network options are less than for the Thinking Machines CM-5 with low-latency, active message-based communications — t_{comm} would decrease only to 1.16 μsecond and 1.016 μsecond from 1.6 μsecond respectively for the two anticipated communications capabilities. This improvement in communications is small in comparison to the 25× improvement anticipated for t_{calc}. Performance of this parallel Gauss-Seidel implementation, is (not unexpectedly) highly dependent on communications latency, due to the large number of small messages. Similar poor performance is predicted for future architectures running the EPRI6K data set.

In figure 8.4. we present actual and predicted speedup values for the complex Gauss-Seidel algorithm solving applications using the BCSPWR10 and EPRI6K power systems networks for

1. empirical speedup data from the CM-5 implementation using buffered communications,

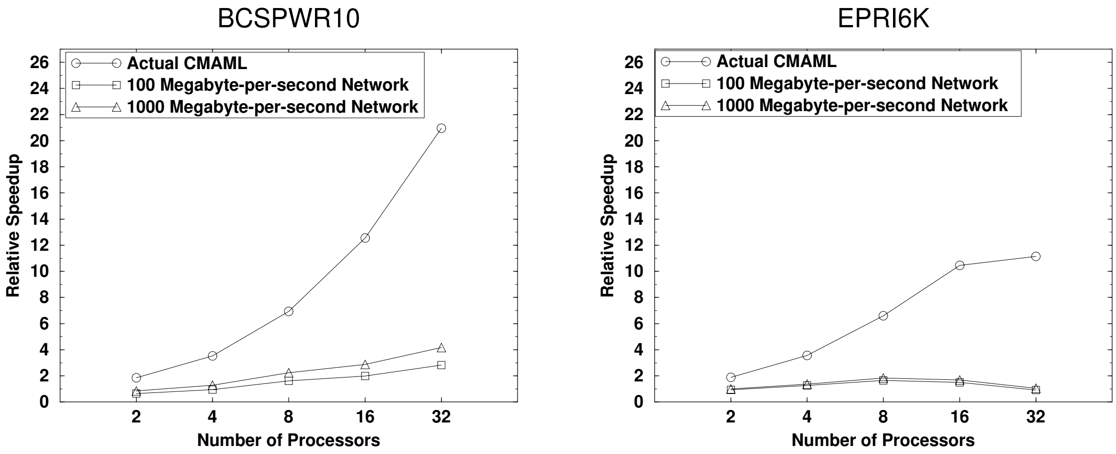


Figure 8.3: Performance Predictions for Parallel Complex Gauss-Seidel — Low-Latency Communications Paradigm

2. predicted speedup for 25× processor speeds and communications networks with 1 μsecond latency and 100 megabyte-per-second bandwidth,
3. predicted speedup for 25× processor speeds and communications networks with 1 μsecond latency and 1000 megabyte-per-second bandwidth.

The graphs in this figure show that we may see slightly improved speedup for this algorithm for both future architectures with respect to the empirical data collected on the Thinking Machines CM-5; although, performance is not scalable to 32 processors. This lack of scalability is due primarily to the parallel software overhead required to set up the buffers. For the BCSPWR10 data set, with 25× processor speeds and communications networks with 1 μsecond latency and 100 megabyte-per-second bandwidth, speedups would be greater than eight for 16 processors and slightly better, ten, with a network that is 10 times faster. The computation-to-communications ratio for both network options are both greater than for the Thinking Machines CM-5 with buffered communications — t_{comm} would increase by a factor greater than 62 ($\frac{(86+.12\times10)}{(1+.04\times10)}$) for 100 megabyte-per-second bandwidth communications and greater than 83 ($\frac{(86+.12\times10)}{(1+.004\times10)}$) for the faster proposed network. These communications performance improvements compare favorably to the anticipated 25× improvement anticipated for t_{calc}. Similar improved performance is predicted for future architectures running the EPRI6K data set; although, peak performance improvement is not as great as for the BCSPWR10 data set.

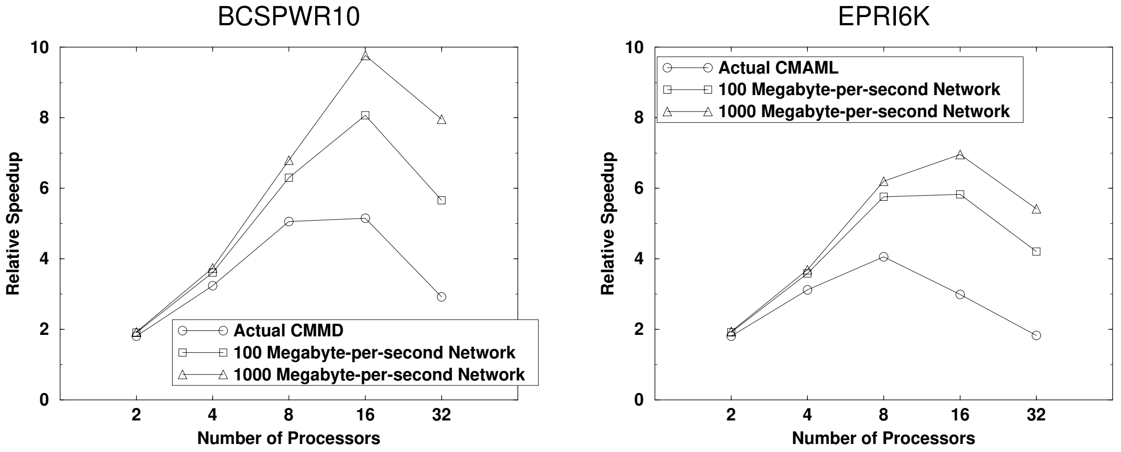


Figure 8.4: Performance Predictions for Parallel Complex Gauss-Seidel — Buffered Communications Paradigm

8.5 Low-Latency Communications Considerations

This research was inspired, in part, by the low-latency communications made possible using active messages on the Thinking Machines CM-5. The parallel block-diagonal-bordered Gauss-Seidel algorithm, as many others, would benefit from extremely low-latency communications, especially for short messages. As a result, future SPP architectures may provide low-latency communications for short messages because there are many classes of parallel algorithms that can only be implemented efficiently with this type of interprocessor communications support. SPP hardware developers recognize that low-latency communications increase the utility of their multi-computer and, consequently, improve market potential. There are, however, limits to possible reductions in latency.

Network latency can be viewed as a linear combination of several factors:

- physical network size,
- software latency,
- processing latency.
- switching latency,

Physical network size contributes to latency as a function of the distance signals must travel. At the speed-of-light eleven inches equals a nanosecond, or $\frac{1}{1000}$ of a μsecond. In order to limit latency, physical network size will be a concern in future SPPs — extremely low-latency communications will not be possible in spatially diverse networked workstations. The other three latencies are more difficult to control, but can be reduced proportionally to processor speeds.

Active messages on the CM-5 are able to significantly limit software latency and processing latency by forcing the user to assume all responsibilities of identifying what data to send and to identify the handler-function at the receiving processor. As a result, no more than 50 machine cycles are required for these operations. If active message-style communications were implemented on future, faster processors, the implementations should scale with processor speed.

The final factor that contributes to latency is the time required to send signals through switch(s) in the interconnection network. Switch latencies will decrease as faster components are used or as data parallel switching implementations are utilized. Given these contributions to latency, it may be possible to continue to decrease latency with faster components, although latency will always be bound by physical size of the network and hardware speeds.

If communications capabilities can *improve* significantly more than the performance of individual processors, additional classes of parallel algorithms can be implemented effectively on new multi-processor architectures. Nevertheless, improving communications capabilities proportional to computational performance increases will prove sufficiently challenging.

8.6 Conclusions

We have shown that the superb computation-to-communication ratio available on the CM-5 using low-latency active messages will probably not be equaled in future SPP architectures where processor performance increases significantly. Performance of our parallel Gauss-Seidel algorithm is latency dependent, due to the large number of small messages. Meanwhile, performance of our parallel direct algorithm is bandwidth dependent, due to the limited number of moderate sized messages.

We have shown that while the parallel sparse block-diagonal-bordered direct solvers may port to future architectures with equal or better performance, the parallel sparse block-diagonal-bordered Gauss-Seidel solvers may not. While future architectures will have greater bandwidth than the Thinking Machines CM-5, they will not have a comparable reduction for communications latency. Any algorithmic performance gains possible with the parallel Gauss-Seidel algorithm would not be realized on future architectures that do not have the computation-to-communication ratio available on the CM-5.

Chapter 9

Conclusions

In this thesis, we have presented research into parallel linear solvers for block-diagonal-bordered sparse matrices. The block-diagonal-bordered form identifies parallelism that can be exploited for both direct and iterative linear solvers. *Direct methods* obtain the exact solution for $\mathbf{Ax} = \mathbf{b}$ in a finite number of operations, whereas *iterative methods* calculate sequences of approximations that may or may not converge to the solution. In order to compare performance for parallel sparse direct and iterative linear solvers for power systems network applications, we have developed parallel block-diagonal-bordered sparse direct methods based on LU factorization and Choleski factorization algorithms, and we have developed a parallel block-diagonal-bordered sparse iterative linear solver based on the Gauss-Seidel method. We are examining parallel sparse linear solvers for embedded power systems applications, so our direct solver implementations also require parallel forward reduction and backward substitution algorithms. The parallel block-diagonal-bordered sparse linear solvers for power systems network applications have proven to be rather sensitive to computation-to-communications ratio or granularity.

This research has focused on block-diagonal-bordered form matrices, that are generated by *tearing* networks into mutually independent partitions by using diakoptic techniques. We have described how the power-systems-analysis-oriented diakoptic node-tearing techniques relate to the state-of-the-art in parallel sparse matrix algorithms. Using the node-tearing-based matrix ordering techniques, we have been able to partition power systems network matrices into highly parallel subgraphs that can be further ordered to balance workloads and to provide parallelism throughout all segments of the calculations.

9.1 Direct Methods

We have developed parallel block-diagonal-bordered sparse direct linear solver algorithms that have been optimized for the special irregular sparse matrices originating in the electrical power systems community. Available parallelism in the block-diagonal-bordered matrix structure has shown promise for simplified implementation and also provides a simple decomposition of the problem into clearly identifiable sub-problems. Parallel block-diagonal-bordered direct linear solvers require a three step preprocessing phase and the ordered matrix is reusable for many sparse linear solutions. The matrix is ordered into block-diagonal-bordered form, pseudo-factored to identify the location of all fillin and to obtain operations counts in the mutually independent diagonal blocks and corresponding portions of the borders, and load-balanced to distribute workload uniformly throughout all processors.

We developed an implementation that offered speedups on 32 processors of nearly ten for double precision LU factorization and even greater speedups for complex variate LU factorization. Speedups for parallel block-diagonal-bordered Choleski factorization were less than for LU factorization, and there are formidable problems implementing forward reduction due to last diagonal block data distributions. These parallel block-diagonal-bordered direct solvers address the most difficult power systems applications to implement on a multi-processor — solutions to linear equations corresponding to only power system networks. Load-flow has the smallest matrices and the fewest calculations due to symmetry and lack of requirements for pivoting to ensure numerical stability. LU factorization of network equations for decoupled solutions of differential-algebraic equations has additional calculations, but often is solved without numerical pivoting. These parallel direct algorithms are very sensitive to communications overhead, and the capabilities of the particular parallel architecture. We have been able to quantify the effects of granularity on implementation performance and we have shown that by simply increasing the granularity by a factor of eight, parallel speedup of the algorithm improves significantly.

9.2 Iterative Methods

We have also developed a parallel block-diagonal-bordered sparse Gauss-Seidel algorithm that also has been optimized for the same very sparse, irregular matrices encountered in electrical power system applications. We have developed this parallel Gauss-Seidel iterative method in order to compare the performance of parallel iterative algorithms with similar parallel direct algorithms. Block-diagonal-bordered matrix structure offers promise for simplified implementation and also offers a simple decomposition of the problem into clearly identifiable sub-problems. 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 often color the last block with only 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. Like the direct methods we have developed in this research, the parallel block-diagonal-bordered Gauss-Seidel algorithm requires a three step preprocessing phase that is reusable for static matrices. The matrix is ordered into block-diagonal-bordered form, pseudo-solved to obtain operations counts in the mutually independent diagonal blocks and corresponding portions of the borders, and load-balanced to distribute floating-point operations uniformly throughout all processors.

We have extensively analyzed the performance of parallel solvers for power systems applications on the Thinking Machines CM-5. We have shown that the node-tearing-based partitioning algorithm can yield matrices in block-diagonal-bordered form with balanced workloads for power systems networks with homogeneous voltage distribution lines; and we have shown that the performance of our parallel block-diagonal-bordered sparse iterative linear solvers can yield good speedups for Gauss-Seidel methods for those networks with balanced workloads. We have measured speedups in excess of 20 for 32 processors with this parallel sparse Gauss-Seidel algorithm.

The parallel Gauss-Seidel algorithm has proven quite sensitive to the interprocessor communications paradigm. The low-latency communications paradigm, greatly improved the performance of the algorithm, because of both reduced latency and significantly simplified implementation. The low-latency communications paradigm permits values to be sent to other processors in an asynchronous manner as soon as the values are calculated. This implementation greatly reduces the parallel software overhead, no buffers must be maintained, and also greatly reduces the amount of data required to distribute the values calculated in the last diagonal block. With total control over individual messages in the last diagonal block, as few as 10% of values that normally would be broadcast require distribution for 32 processors. As a result, we have measured good relative speedups for the low-latency communications paradigm implementation of parallel Gauss-Seidel, while the buffered communications version of this algorithm offers little or no speedup.

9.3 Comparisons of Direct and Iterative Methods

We are able to get substantially better speedups with the parallel Gauss-Seidel algorithm than with the parallel direct methods for power systems networks, although the only matrix types where there is assurance of convergence for Gauss-Seidel solvers are diagonally dominant and positive definite matrices. Choleski-based linear solvers encountered the worst parallel performance, and are limited to matrix forms where Gauss-Seidel convergence is assured. Due to guarantees of convergence for the Gauss-Seidel algorithm for positive definite or diagonally dominate matrices and the relative performance for the parallel Choleski solver, there is potential for significant algorithmic speedup by selecting the Gauss-Seidel for solving those power systems network matrices that could also use Choleski factorization. We have shown that the applications provide good estimates of starting points for the iterative methods, so convergence appears to be sufficiently rapid that algorithmic speedups of as great as ten may be possible. For those applications that require LU factorization, more information concerning iterative solver convergence would be required in addition to the rate of convergence for real power systems applications data before decisions could be made concerning the selection of direct versus iterative methods.

9.4 Future Architectures

We design and implement algorithms on existing hardware; however, for industrial applications such as power systems network analysis, it is equally important to predict algorithm performance for future architectures. Performance predictions for future architectures will help determine whether or not it will be cost-effective to port critical software to parallel architectures now or to simply wait and get speedup in the future from faster single processor computers.

This analysis is a good case in point — performance for the parallel block-diagonal-bordered sparse solvers developed here is rather good on the Thinking Machine CM-5 for moderate number of processors (2–32). For Choleski solver applications, the parallel block-diagonal-bordered Gauss-Seidel algorithm yields good speedups and offers substantial algorithmic speedup when compared with parallel block-diagonal-bordered direct solvers. However, the superb computation-to-communication ratio available on the CM-5 using low-latency active messages will probably not be equaled in future architectures where processor performance increases significantly.

While the bandwidth-dependent parallel sparse block-diagonal-bordered direct solvers may port to future architectures with equal or better performance, the latency-dependent parallel sparse block-diagonal-bordered Gauss-Seidel solvers may not. While future architectures will have greater bandwidth than the Thinking Machines CM-5, they will not have a comparable reduction in communications latency. Any algorithmic performance gains possible with the parallel Gauss-Seidel algorithm would not be realized on future architectures that do not have the computation-to-communication ratio available on the CM-5.

9.5 Future Research Opportunities

The parallel block-diagonal-bordered direct solvers we implemented in this research, address the most difficult power systems applications to implement on multi-processor architectures — solutions pertaining only to power system networks. Parallel block-diagonal-bordered sparse linear solver algorithms can readily be extended to applications that have power systems networks as a small portion of a larger matrix, for example, the entire system of linearized differential-algebraic equations (DAEs) encountered in transient stability analysis or small-signals analysis applications. These applications add many natural blocks of linearized differential equations that significantly increase the size of the matrix and the data density. The linearized differential equations are less-sparse than the network equations and may require pivoting to ensure numerical stability. Pivoting for this matrix would be limited to within diagonal-blocks to place limits on fillin, but the efficient static data structures would need to be replaced by less-efficient dynamic linked-list-based data structures. Any of these modifications would increase computational workload — work that does not require interprocessor communications. As a result, any modifications to algorithms to include these additional features would improve problem granularity and parallel speedup on the Thinking Machines CM-5 and on future SPPs.

In addition to simply implementing a version of these parallel block-diagonal-bordered linear solvers for transient stability or small signals analysis Jacobian solutions, there is a rich area for research to incorporate the concepts of these parallel linear solvers more closely with DAE solvers used to solve the differential equations and non-linear algebraic equations found in power systems simulations [34]. This research has examined general parallel block-diagonal-bordered sparse linear solvers; meanwhile, there are research opportunities to examine a more tight coupling of parallel block-diagonal-bordered sparse linear solvers with DAE solvers in power systems applications.

The parallel block-diagonal-bordered Gauss-Seidel solver we implemented in this research, does not include a preconditioner to speed convergence of the iterative solver. Future research could examine effective and efficient ways to add preconditioners such as incomplete LU factorization to improve overall iterative linear solver performance.

Bibliography

[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 for Partitioning Unstructured Problems. Technical Report RNR-92-033, NASA Ames Research Center, November 1992.

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

[5] G. Brassard and P. Bratley. *Algorithmics — Theory and Practice.* Prentice-Hall, 1988.

[6] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. *Proceedings of the 1994 International Parallel Processing Symposium*, 1994.

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

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

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

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

[11] 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.

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

[13] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I). Technical Report TR/PA/92/86, CERFACS, 1992.

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

[15] 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.

[16] G. Fox. . private correspondance, December 1994.

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalablility of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology: Systems and Applications*, 1(3):12–22, August 1993.

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

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

[27] N. Hartsfield and G. Ringel. *Pearls in Graph Theory — A Comprehensive Introduction.* Acedemic Press, Inc., 1990.

[28] K. A. Hawick. High Performance Computing and Communicaitons Glossary — Release 1.3a. Technical report, Northeast Parallel Architectures Center (NPAC), July 1994. available at http://www.npac.syr.edu/nse/hpccgloss/index.html.

[29] 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.

[30] 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.

[31] 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.

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

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

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

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

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

[37] 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.

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

[39] 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.

[40] D. P. Koester, S. Ranka, and G. C. Fox. Parallel Direct Methods for Block-Diagonal-Bordered Sparse Matrices. NPAC Technical Report SCCS 679, Northeast Parallel Architectures Center (NPAC), Syracuse University, December 1994.

[41] V. Kumar and A. Gupta. Analyzing Scalablility of Parallel Algorithms and Architectures. Technical Report TR 91-18, Department of Computer Science University of Minnesota, Minneapolis, MN, November 1991.

[42] D. W. Matula, G. Marble, and J. D. Isaacson. *Graph Coloring Algorithms*. Academic Press, New York, 1972.

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

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

[45] 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.

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

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

[48] 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.

[49] 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.

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

[51] 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.

[52] X. H. Sun and D. T. Rover. Scalability of Parallel Algorithm-Machine Combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, June 1994.

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

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

[63] Y. Wallach. *Parallel Processing and Ada.* Prentice-Hall, 1991.

[64] 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.

Appendix A

Nomenclature

A.1 Parallel Processor Architectures

Scalable Parallel Processors (SPP): Multicomputers with relatively small numbers of processors, 2 – 64, generally interconnected by switched networks. SPPs are often compatible with workstations, and may be composed of networked workstations.

Highly Parallel Processors (HPP): Multicomputers with 64 – 1024 processors.

Massively Parallel Processors (MPP): Multicomputers with greater than 1024 processors.

A.2 Parallel Computing Analysis

Speedup — S_{a_1}: Given a single problem with two algorithms that exhibit execution times of T_{a_1} and T_{a_2} with $T_{a_1} < T_{a_2}$, speedup for algorithm a_1 is defined as

$$S_{a_1} \equiv \frac{T_{a_2}}{T_{a_1}}. \tag{A.1}$$

This simple, intuitive definition will be expanded in order to compare the performance of sequential and parallel algorithms [51].

Parallel Execution Time — T_p: The time to run a parallel algorithm on p processors [17, 25].

Sequential Execution Time — T_{seq}: The time to run a sequential algorithm as a single process $T_{seq} = T_1$ [17, 25].

Relative Speedup — S_p: 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

$$S_p \equiv \frac{T_{seq}}{T_p} \quad [51]. \tag{A.2}$$

Relative Efficiency — E_p: Relative efficiency is defined as

$$E_p \equiv \frac{S_p}{p} \quad [51]. \tag{A.3}$$

Efficiency can be viewed as the speedup-per-node [17].

Amdahl's Law: A rule formalized by Gene Amdahl, which defines limits on parallel program speedup as a function of the sequential portion of the overall calculations. Given T_1, the time to solve a problem on a single processor, then $\hat{T}_p$ can be parameterized in $\alpha \in [0,1]$ by

$$\hat{T}_p \equiv \alpha T_1 + (1-\alpha)\frac{T_1}{p}, \tag{A.4}$$

where α is the inherently sequential fraction of computations [17, 25, 28, 51]. The aforementioned estimate of $\hat{T}_p$ can be used when estimating the speedup $\hat{S}_p$ [51] by

$$\hat{S}_p = \frac{p}{1+(p-1)\alpha} = \frac{1}{\alpha+(1-\alpha)/p} \leq \hat{S}_\infty \equiv \alpha^{-1}. \tag{A.5}$$

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. According to Amdahl's law, a task with 10% sequential operations could obtain no more than a speedup of 10, regardless of the number of processors applied to the problem.

Total Parallel Overhead — f_t: The sum of all overhead incurred in the parallel calculations by all processors. f_t includes overhead due to communications costs, load-imbalance, costs for additional software replicated on each processor, and non-optimal algorithms for the parallel processor [17, 25]. f_t is defined as

$$f_t \equiv (p \times T_p) - T_1. \tag{A.6}$$

When f_t is calculated using empirical data, it can be either a non-negative or negative quantity. Negative overhead signifies that speedup for the problem has not been bounded by p, and thus *superlinear* speedup has occurred. Superlinear speedup can result from cache effects as a problem is divided onto multiple processors, the amount of memory required on each processors is reduced, generally proportionally to the number of processors. The entire problem may not fit in fast cache for one or several processors, but the problem may fit into fast cache memory as the number of processors increases. Consequently, doubling the number of processors may more than double the measured speedup.

Communications Overhead — f_c**:** A measure of the additional workload incurred in a parallel algorithm as a result of interprocessor communications [17, 25, 28]. f_c is dependent on the ratio of communications to calculations, not just the amount of communications

$$f_c \sim \frac{t_{comm}}{t_{calc}}, \tag{A.7}$$

where t_{calc} is a metric describing the computational capability of a single processor [17], and t_{comm} represents the communications characteristics. The quantity $\frac{t_{calc}}{t_{comm}}$ is often referred to as the *computation-to-communications ratio.* For traditional buffered interprocessor communications, t_{comm} is a linear combination of *latency*, *bandwidth*, and *message size*

$$t_{comm} \equiv t_{latency} + \left(\frac{4}{\mathcal{B}_{bytes}} \times N_{words} \right). \tag{A.8}$$

$t_{latency}$ is the communications latency or startup time, $\mathcal{B}_{bytes}$ is the bandwidth measured in bytes per second, and N_{words} is the number of words or four byte units of data.

Load-Imbalance Overhead — f_l**:** Parallel speedup is limited by the time of the slowest processor in the calculations. f_l is the sum of idle time for processors waiting for the slowest processor [17, 28] .

Parallel Software Overhead — f_s**:** Parallel algorithms may require additional calculations that must be replicated at each processor, such as additional index calculations — the overhead to startup loops must be replicated in spite of the fact that the loops may have fewer applications, due to work being distributed amongst multiple processors [17].

Parallel Algorithmic Overhead — f_p**:** Efficient parallel algorithms may require additional calculations that are not present in a sequential algorithm. f_p denotes the additional work performed in the parallel algorithm not required in the sequential algorithm [17].

Overhead-Based Performance Estimate — $\check{T}_p$**:** Amdahl's Law gives one preliminary estimate of the potential speedup in a concurrent algorithm; however, for some concurrent algorithms, overhead associated with the concurrent algorithm appears more critical than the inherent percentage of sequential operations in a concurrent algorithm. In these instances, the time for a parallel algorithm can be defined as

$$\check{T}_p \equiv \frac{T_{seq}}{p}(1 + f_t). \tag{A.9}$$

This can be rewritten as

$$\check{S}_p = \frac{p}{1 + f_t}, \tag{A.10}$$

and

$$\check{E}_p = \frac{1}{1+f_t} \approx 1 - f_t, \tag{A.11}$$

which yields an estimate for f_t,

$$f_t = \frac{1}{\check{E}_p} - 1. \tag{A.12}$$

This measure of parallel algorithm performance, along with Amdahl's law, can be used to generate estimates of potential performance for algorithms on other existing or even future parallel architectures [17].

Computation-to-Communications Ratio: The computation-to-communications ratio denotes the relative number of calculations on a processor compared to the amount of communications. This quantity is related to *granularity*, and also is related to the inverse of f_c, communications overhead [17, 28].

Granularity: The amount of operations performed by a process between interprocessor communications events. A *fine-grain* process performs only a few operations between requisite communications, while a *coarse-grain* process performs many operations between interprocessor communications [17, 28].

Appendix B

Node-Tearing Nodal Analysis

Node-tearing nodal analysis partitions a power systems network graph into independent subgraphs and a coupling network, which corresponds to determining the diagonal blocks and lower border in a block-diagonal-bordered form matrix. We have selected node-tearing nodal analysis to partition power systems networks and form block-diagonal-bordered matrices because this algorithm examines the natural structure in the matrix while providing the means to minimize the number of coupling equations. Tearing here refers to breaking the original problem into smaller sub-problems whose partial solutions can be combined to give the solution of the original problem. Node-tearing nodal analysis is a specialized form of diakoptic analysis [26] that was developed especially for power system network analysis [49]. In general, node-tearing analysis is superior to conventional diakoptic analysis because node-tearing simply orders the network graph and does not generate new nodes in the power distribution network graph. For power systems networks, the corresponding ordered admittance matrices retain their symmetry and positive definite nature. Examples in reference [49] illustrate that the technique also has validity for general structural analysis matrices.

B.1 The Node-Tearing Algorithm

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

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

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

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

$$\mathcal{G}(\mathcal{S}) \equiv (\mathcal{S}, E_{\mathcal{S}}), \tag{B.1}$$

where: $\mathcal{E}_{\mathcal{S}} \equiv \{\varepsilon \in \mathcal{E} \mid \varepsilon$ is incident only with $\mathcal{S}\}$ [49].

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

$$\begin{array}{c} \mathcal{G}(\mathcal{N}_1^1) \equiv (\mathcal{N}_1^1, \mathcal{E}_1^1) \\ \mathcal{G}(\mathcal{N}_1^2) \equiv (\mathcal{N}_1^2, \mathcal{E}_1^2) \\ \vdots \\ \mathcal{G}(\mathcal{N}_1^m) \equiv (\mathcal{N}_1^m, \mathcal{E}_1^m). \end{array} \tag{B.2}$$

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

$$\begin{array}{c} \mathcal{N}_1 \equiv \cup_{i=1}^{m} \mathcal{N}_1^i \\ \\ \mathcal{N}_2 \equiv \mathcal{N} - \mathcal{N}_1 \end{array} \tag{B.3}$$

where:

$\mathcal{N}_1$ is the set of nodes in the mutually independent subgraph
$\mathcal{N}_2$ is the set of nodes in the coupling equations

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

The edge-coupling condition simply requires that the edges in $\mathcal{E}_1^i$ are not connected to edges in $\mathcal{E}_1^j$ $\forall\, i \neq j$ and $i, j = 1, 2, \ldots, m$. Consequently, $\mathcal{G}(\mathcal{N}_1^i)$ has no edges in common with $\mathcal{G}(\mathcal{N}_1^j)$, $\forall i \neq j$, 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 $\mathcal{G}(\mathcal{N}_1^i)$ and $\mathcal{G}(\mathcal{N}_1^j)$, $\forall i \neq j$, is not direct and must go through nodes in $\mathcal{N}_2$. Reference [49] contains the straight forward proof that these conditions yield a block-diagonal-bordered form matrix when the corresponding graph $\mathcal{G}$ is ordered by node-tearing.

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

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

Figure B.1: Sample Contour Tableau for the k^{th} Diagonal Block

1. the topological condition holds,
2. the edge coupling condition holds,
3. $\mid \mathcal{N}_1^k \mid \leq max_{DB}$, $k = 1, 2, \ldots, m$.

The last constraint for the tearing optimization problem permits some control of the maximum size of diagonal blocks, max_{DB}, which can prove quite useful when tearing a graph for factoring on multi-processors. By modifying this parameter, control can be exercised over the shape of the ordered sparse matrix — yielding narrow bandwidth block-diagonal-bordered form matrices when max_{DB} is small and limiting the size of the borders in a block-diagonal-bordered matrix when max_{DB} is large. The effects of varying the value of max_{DB} is illustrated in sections 7.1 and 7.2. This optimization problem belongs to the family of NP-complete problems [49]. We expect to apply node-tearing to order large sparse matrices into block-diagonal-bordered form, so the use of an exact exponentially-bounded-complexity algorithm is not feasible, and the following efficient heuristic algorithm has been developed,

The technique chosen to solve the graph optimization problem is based on examining the contour of the graph [49], and developing a contour tableau to identify independent subgraphs. A contour-tableau consists of three columns as illustrated in figure B.1 and a separate contour-tableau is developed for each diagonal block. The leftmost column contains the iterating sets or the potential elements of a set of nodes in the subgraph $\mathcal{N}_1^k$. The middle column contains the adjacency set, which contains the set of nodes adjacent to, but not including any elements in the corresponding iterating set. The remaining column contains the contour number or the cardinality of the corresponding adjacency set.

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

$$\mathcal{I}_{(i+1)}^k = \mathcal{I}_i^k \cup \{\nu_{(i+1)}\}. \tag{B.4}$$

The adjacency set is updated by the formula:

$$\mathcal{A}^k_{(i+1)} = \mathcal{A}^k_i \cup \Lambda(\nu_{(i+1)}) - \{\nu_{(i+1)}\}, \tag{B.5}$$

and

$$c^k_{(i+1)} = \mid \mathcal{A}^k_{(i+1)} \mid . \tag{B.6}$$

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

$$\Lambda(\nu_{(i+1)}) = \min_{\forall\, \upsilon \in \mathcal{N} - \mathcal{I}^k_i} \Lambda(\upsilon) \tag{B.7}$$

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

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

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

$$\mathcal{N}_2 = \mathcal{N}_2 \cup \mathcal{A}^k_i \tag{B.8}$$

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

B.2 An Example of Node-Tearing

An example illustrating node-tearing nodal analysis is presented in figures B.2 through B.6. The example graph, presented in figure B.2, has two distinct portions connected at node ν_4. Node ν_1

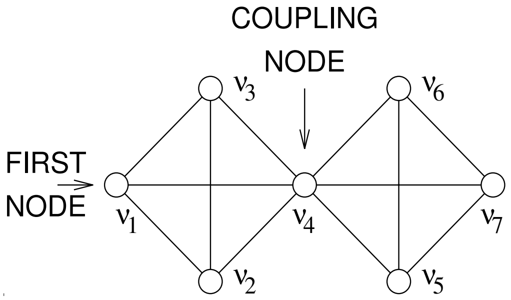


Figure B.2: Graph for a Node-Tearing Example

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

Figure B.3: Example Contour Tableau

meets the selection criteria for the first node, and the contour tableau is presented in figure B.3. There is a distinct local minimum in the contour number at c_3 which identifies node ν_4 as the node that couples the two mutually independent graph partitions. Figure B.4 presents the ordered graph — note that only the labels on the modes have changed from figure B.2. To illustrate the effect of ordering the matrix, the matrix sparsity structure for the original and ordered graphs are presented in figure B.5. In these figures, original data values are represented with + symbols while fillin are denoted with F characters. Within the subgraphs, the values would be ordered with a minimum-degree ordering algorithm. For this sample matrix, minimum degree ordering for the entire matrix would yield the same results.

Figure B.6 completes this example by illustrating the elimination tree for this example. For the unordered matrix, there are no branches in the elimination tree; consequently, if calculations are performed in the original order, there would be no factorization operations that could be performed in parallel. However, after ordering there is available parallelism denoted by the multiple leaf supernodes in the elimination tree.

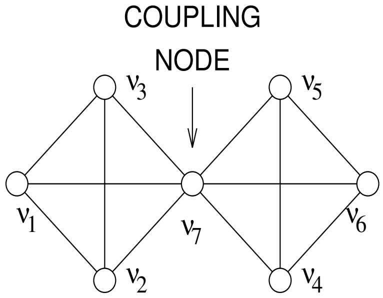
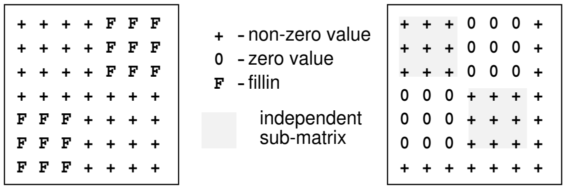


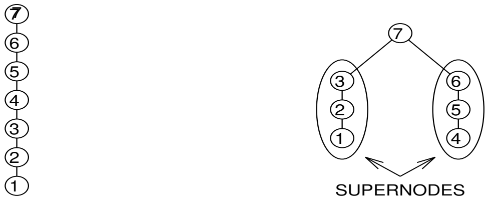
Figure B.4: Relabeled Example Graph



$$\begin{array}{ccccccc}
+ & + & + & + & F & F & F \\
+ & + & + & + & F & F & F \\
+ & + & + & + & F & F & F \\
+ & + & + & + & + & + & + \\
F & F & F & + & + & + & + \\
F & F & F & + & + & + & + \\
F & F & F & + & + & + & +
\end{array}
\qquad
\begin{array}{ccccccc}
+ & + & + & 0 & 0 & 0 & + \\
+ & + & + & 0 & 0 & 0 & + \\
+ & + & + & 0 & 0 & 0 & + \\
0 & 0 & 0 & + & + & + & + \\
0 & 0 & 0 & + & + & + & + \\
0 & 0 & 0 & + & + & + & + \\
+ & + & + & + & + & + & +
\end{array}$$

(a) Original Matrix (b) Ordered Matrix

Figure B.5: Matrix Representation of the Example Graphs



(a) Original Matrix (b) Ordered Matrix

Figure B.6: Elimination Trees for the Example Graphs

B.3 The Node-Tearing Implementation

The software implementation to perform node-tearing nodal analysis utilizes the basic concept of building a contour tableau to identify independent sub-matrices and the coupling equations in an undirected graph representing a sparse matrix. In our implementation, the search for the local minimum of the contour number is limited to within the range $(\alpha \times max_{DB}) \leq i \leq max_{DB}$, $0 < \alpha < 1$. When an independent sub-matrix is found, this iterating set is moved into a set $\mathcal{N}_1^k$, where $\mid \mathcal{N}_1^k \mid = i$. After the sets $\mathcal{N}_1 = \{\mathcal{N}_1^1, \ldots, \mathcal{N}_1^m\}$ and $\mathcal{N}_2$ are determined, the equations corresponding to the sets $\mathcal{N}_1^1, \ldots, \mathcal{N}_1^m$ and $\mathcal{N}_2$ are further ordered independently using the multiple minimum-degree ordering algorithm.

Figure B.7 illustrates the major steps in the node-tearing ordering algorithm that produces block-bordered-diagonal form matrices with minimum fillin. The algorithm examines all nodes essentially once, where the size of the independent subgraphs are limited to max_{DB}. The computational complexity of this algorithm is

$$O(\max_{\forall i} \mid \mathcal{A}_i^k \mid \times N) \tag{B.9}$$

due to the fact that all nodes in the graph must be examined, and for each element in the contour tableau — all elements of the adjacency set must be examined for the next node. The value of $\max_{\forall i} \mid \mathcal{A}_i^k \mid$ 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, so the computational complexity of the algorithm is substantially less than $O(N^2)$.

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

Figure B.7: The Node-Tearing Algorithm

Appendix C

Minimum-Degree Ordering

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

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

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

$$r_i^{(k)} = \min_t r_t^{(k)}, \tag{C.1}$$

where:

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

When factoring the k^{th} row, the row with the minimum number of variables is selected, moved by elementary row and column exchange rules to the k^{th} row, and then factored. Algorithms to implement this iterative formula are best described using the graph theoretical explanation of fillin presented in figure 3.7. Let G be an undirected graph and ν a node in G, then let $\Lambda_G(\nu)$ describe the set of nodes adjacent to ν and let $|\Lambda_G(\nu)|$ represent the degree of node ν. The last concept required to develop a concise minimum-degree algorithm is the concept of an *elimination graph* [22]. Given a graph G, the elimination graph G_ν is the resulting graph after the node ν is factored. Elimination

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

Figure C.1: The Minimum-Degree Algorithm

graphs get their name because of the close relationship of LU factorization and Gaussian elimination. The rudimentary minimum-degree algorithm used throughout this work is presented in figure C.1. The outer loop examines each node in the graph, and the inner loop searches through all remaining nodes in the present graph to select a node with the minimum degree. After a minimum-degree node is selected, the edges at adjacent nodes must be updated to reflect factorization. As illustrated in figure 3.7, the addition of new edges in the elimination graph G_ν is limited to those nodes in $\Lambda_G(\nu)$. For $v \in \Lambda_G(\nu)$, then

$$\Lambda_{G_\nu}(v) = (\Lambda_G(v) \cup \Lambda_G(\nu)) - \{\nu, v\}. \tag{C.2}$$

Given the two nested loops that can examine all nodes in the original sparse graph, the computational order of this algorithm is $O(N^2)$, although a significant portion of the workload is required to calculate the elimination graph G_ν [22]. As stated above, in formula 3.4, the total amount of calculations in the loop to update the elimination graph G_ν is bounded by the binomial coefficient of *the number of edges at a node choose 2* or $|\Lambda_G(\nu)|$ *chose 2*. See equation 3.4 for details on calculating the binomial coefficient. It is important to note that the location of all fillin can be determined when using this classical implementation of minimum degree ordering.

This version of the minimum-degree algorithm has been used in our research in a two-fold manner: to order symmetric power systems admittance matrices to provide baseline orderings with which to compare the performance of other ordering techniques, and to order the independent sub-matrices obtained with node-tearing ordering techniques. We apply the algorithm independently to each graph partition and the coupling equations, so we are actually implementing a version of the multiple minimum degree ordering algorithm.

Appendix D

Graph Coloring

To describe graph-coloring in rigorous mathematical terms, let the set $\mathcal{N}$ denote the nodes of an undirected graph $\mathcal{G}$ and let $\mathcal{E}$ denote the edges in $\mathcal{G}$, or $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Graph edges are tuples (i, j), where $i, j \in \mathcal{E}$ and $i \neq j$. The two nodes are defined to be *neighbors* if the tuple $(i, j) \in \mathcal{E}$. Given a graph $\mathcal{G}$, we define a *coloring of* $\mathcal{G}$ to be an assignment of colors to the nodes $\mathcal{N}$ of $\mathcal{G}$ such that no two adjacent nodes are given the same color. This can be restated as a *coloring of* $\mathcal{G}$ is a mapping $\sigma : \mathcal{N} \rightarrow \{1, 2, \ldots, \hat{\chi}\}$ such that $\sigma(i) \neq \sigma(j)\ \forall (i, j) \in \mathcal{E}$. The color of node i is $\sigma(i)$ and $\hat{\chi} \geq \chi_{\mathcal{G}}$. The minimum possible value for $\hat{\chi}$ is known as the *chromatic number* of $\mathcal{G}$, which we denote as $\chi_{\mathcal{G}}$ [32].

D.1 The Basic Greedy Graph Coloring Heuristic

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 [27, 32, 42]. A greedy polynomial time heuristic can yield an optimal ordering if the nodes are visited in the correct order, although the same heuristic may be arbitrarily bad for a different order [5]. The basic greedy graph coloring heuristic is presented in figure D.1. The only aspect of the basic heuristic that is normally modified, as variations on this algorithm are developed, is the rule to select the node $\hat{\nu}$ from the remaining uncolored nodes [32]. In adding load-balancing, we have also modified the selection of the smallest available consistent color.

```
$\hat{\mathcal{N}} \leftarrow \mathcal{N}$
while $\hat{\mathcal{N}} \neq \phi$ do
    select a node $\hat{\nu}$ from $\hat{\mathcal{N}}$
    $\sigma(\hat{\nu}) \leftarrow$ the smallest available consistent color
    $\hat{\mathcal{N}} \leftarrow \hat{\mathcal{N}} \backslash \{\hat{\nu}\}$
end while
```

Figure D.1: The Graph Multi-Coloring Algorithm

D.2 The Saturation Degree Ordering Heuristic

We selected the saturation degree ordering heuristic [32] to color the last diagonal block in our preprocessing phase when generating matrices in block-diagonal-bordered form for parallel Gauss-Seidel methods, but we have modified saturation degree ordering to include load-balancing. The saturation degree ordering heuristic selects a node in the graph that has the largest number of differently colored neighbors and is defined as follows. Suppose that nodes $\hat{\nu}_1, \ldots, \hat{\nu}_{i-1}$ have been selected and colors $\sigma(1)$ to $\sigma(i-1)$ assigned to these nodes. The next node, $\hat{\nu}_i$, selected for coloring is a node with the most adjacent colored nodes in the set $\{\hat{\nu}_1, \ldots, \hat{\nu}_{i-1}\}$. When there are multiple nodes that meet this selection criteria, a node is selected at random.

We have added a load-balancing capability to the saturation degree ordering algorithm that selects the color for a node and attempts to equalize the number of nodes with a particular color in a manner similar to the pigeon-hole load balancing algorithm. Whenever possible, a color is selected that will reduce the disparity in the number of nodes assigned a certain color. Our version of the saturation degree ordering algorithm simply selects the consistent color with the fewest occurrences. A consistent color for the node $\hat{\nu}_i$ is an element of the set $\{c_1, \ldots, c_k\}$, where $c_k < \hat{\chi}$ and $\forall j$ such that $(i,j) \in \mathcal{E}$, $\sigma(j) \ni \{c_1, \ldots, c_k\}$. We define a function $\gamma(m)$ that determines the cardinality of the set of nodes with the color m — $\gamma(m) =\mid \{\hat{\nu}_i \mid \forall i(\sigma(i) = m)\} \mid$. With our modifications to the saturation degree algorithm, a consistent color for the node $\hat{\nu}_i$ is an element of the set $\{c_1, \ldots, c_k\}$, where $\forall j$ such that $(i,j) \in \mathcal{E}$, $\sigma(j) \ni \{c_1, \ldots, c_k\}$ and $\{c_{lb} \mid \gamma(c_{lb}) \leq \gamma(m) \, \forall m \in \{c_1, \ldots, c_k\}\}$. This added selection condition helps to break ties when there are multiple consistent colors in the traditional saturation degree algorithm.

We present our version of the saturation degree ordering-based graph multi-coloring algorithm in figure D.2. 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

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

Figure D.2: The Graph Multi-Coloring Algorithm for the Last Diagonal Block of a Block-Diagonal-Bordered Matrix

are very sparse, generally with no more than three nodes adjacent to any single node. Some graphs representing the last diagonal blocks in real power systems networks after node-tearing ordering proved to be bipartite, while many graphs required only three colors. The results of graph coloring the last diagonal block when preprocessing the networks for the parallel Gauss-Seidel are presented in section 7.2.

Appendix E

Implementation Pseudo-Code

This appendix contains the pseudo-code listing of the parallel block-diagonal-bordered iterative solvers that have been developed in the C programming language for the Thinking Machines CM-5 multi-processor using a host-node paradigm with message passing. Pseudo-code listings are presented both for direct and iterative methods. Detailed discussions of the implementations are provided in chapter 6.

E.1 Parallel Blocked-Diagonal-Bordered LU Factorization

Implementations for both parallel block-diagonal-bordered sparse LU and Choleski factorization have been developed during this research. The pseudo-code presented in this section focuses on LU factorization, although Choleski factorization implementations are similar to these algorithms with modifications to account for the symmetric nature of the matrices used in Choleski factorization. The block-diagonal-bordered LU factorization algorithm can be broken into three component parts as defined in the derivation on available parallelism in chapter 4. Pseudo-code representations of each parallel LU factorization algorithm section are presented separately in figures E.1 through E.4. In particular, each of these figures correspond to the following figure numbers:

1. factor the diagonal blocks and border — figure E.1,
2. update the last diagonal block —
    - low-latency communications paradigm — figure E.2,
    - buffered communications paradigm — figure E.3,
3. factor the last diagonal block — figure E.4.

The remaining steps in the parallel algorithm are forward reduction and backward substitution. The forward reduction algorithm to operate with the parallel block-diagonal-bordered LU factorization algorithm can be broken into three component parts, similar to LU factorization. Pseudo-code representations of each parallel algorithm section are presented separately in figures E.5 through E.8. In particular, each of these figures correspond to the following figure numbers:

1. forward reduce the diagonal blocks and border — figure E.5,
2. update the last diagonal block —
    - low-latency communications paradigm — figure E.6,
    - buffered communications paradigm — figure E.7,
3. forward reduce the last diagonal block — figure E.8.

The backward substitution algorithm to operate with the parallel block-diagonal-bordered LU factorization algorithm can be broken into two component parts, back substitute the last diagonal block then back substitute the remaining upper triangular matrix. Pseudo-code representations of each parallel algorithm section are presented separately in figures E.9 and E.10, respectively for backward substitution of the last diagonal block and backward substitution of the diagonal blocks and border.

E.2 Parallel Blocked-Diagonal-Bordered Gauss-Seidel

Implementations for the parallel block-diagonal-bordered sparse Gauss-Seidel method have been developed during this research. Pseudo-code representations of each parallel algorithm section are presented separately in figures E.11 through E.18. In particular, each of these figures correspond to the following figure numbers:

1. monitor convergence for the parallel Gauss-Seidel method — figure E.11,
2. solve for $\mathbf{x}^{(k+1)}$ in the diagonal blocks and upper border — figure E.12,
3. update $\hat{\mathbf{b}}$ for the last diagonal block —
    - low-latency communications paradigm — figure E.13,
    - buffered communications paradigm — figure E.14,
4. solve for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block —
    - low-latency communications paradigm — figure E.15,

```
Node Program
/* factor the independent blocks and corresponding borders */
for those independent blocks $l$ assigned to this processor
    for all elements $k$ along the diagonal in block $l$
        for each $i \in [k, N_l]$
            for each $j \in [1, k-1]$ such that $A_{i,j} \neq 0$ and $A_{j,k} \neq 0$
                $A_{i,k} \leftarrow A_{i,k} - (A_{i,j} * A_{j,k})$
            endfor
        endfor
        for each $i \in [k+1, N_l]$
            $A_{k,j} \leftarrow (A_{k,j} / A_{k,k})$
        endfor
        for each $j \in [k+1, N_l]$
            for each $i \in [1, k-1]$ such that $A_{k,i} \neq 0$ and $A_{i,j} \neq 0$
                $A_{k,j} \leftarrow A_{k,j} - (A_{k,i} * A_{i,j})$
            endfor
        endfor
    endfor
endfor
```

Figure E.1: Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Diagonal Blocks and Border

  - buffered communications paradigm — figure E.16,

5. perform the convergence check —
  - low-latency communications paradigm — figure E.17,
  - buffered communications paradigm — figure E.18.

Figure E.11 provides the framework for the parallel block-diagonal-bordered sparse Gauss-Seidel implementation. In this implementation, multiple iterations are performed before a check is made on convergence.

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

/* update the last block with active message RPC handler function Update_FAC_LB */
Function Update_FAC_LB($\sigma, i, j$)
           $A_{i,j} \leftarrow A_{i,j} - \sigma$
End Update_FAC_LB
```

Figure E.2: Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Update the Last Diagonal Block — Low-Latency Communications Paradigm

```
Node Program
/* calculate updates to the last block */
$\sigma_{*,*,*} = 0$
for those independent blocks $l$ assigned to this processor
    for all non-zero columns $j$ in the upper border
        for all non-zero rows $i$ in the lower border
            $p \leftarrow \mathcal{P}(i,j)$ /* map the $(i,j)$ tuple to the processor location $p$ */
            for each $k$ such that $L_{i,k}$ and $U_{k,j} \neq 0$
                $\sigma_{i,j,p} \leftarrow \sigma_{i,j,p} + (L_{i,k} * U_{k,j})$
            endfor
        endfor
    endfor
endfor

/* update the last diagonal block using data calculated on this processor */
for all data in the buffer
    $A_{i,j} \leftarrow A_{i,j} - \sigma_{i,j}$
endfor

/* update the last diagonal block using data calculated on other processors */
$p_{send} \leftarrow p_{receive} \leftarrow local\_proc\_num$
$p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
$p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
for all processors around a conceptual ring
    Asynchronously send $\sigma_{*,*,p_{send}}$ to processor $p_{send}$
    Asynchronously receive $\tilde{\sigma}_{*,*}$ from processor $p_{receive}$
    /* update the last diagonal block on this processor using
    data from processor $p_{receive}$ */
    for all data in the received buffer
        $A_{i,j} \leftarrow A_{i,j} - \tilde{\sigma}_{i,j}$
    endfor
    $p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
    $p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
endfor
```

Figure E.3: Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Update the Last Diagonal Block — Buffered Communications Paradigm

```
Node Program
/* factor the last block — all communications are sent around a conceptual ring */
$p_{factor} \leftarrow p_{start} \leftarrow 0$
if $p_{factor} = local\_proc\_num$
  factor the first block
  send the panel of values to processor 1
end if
for all blocks on this processor
    if $p_{factor} \neq local\_proc\_num$
        receive the next panel
        if $p_{start} \neq local\_proc\_num$
            $p_{send} \leftarrow (local\_proc\_num + 1) \bmod N_{procs}$
            send the next panel of values to processor $p_{send}$
        end if
    end if
    $p_{factor} \leftarrow (p_{factor} + 1) \bmod N_{procs}$
    if $p_{factor} = local\_proc\_num$
        update only the next block with the next panel
        factor the next block
        if not the last block
            $p_{send} \leftarrow (local\_proc\_num + 1) \bmod N_{procs}$
            send the panel of values to processor $p_{send}$
            $p_{start} \leftarrow p_{send}$
        end if
    end if
    update all remaining blocks with the next panel
endfor
```

Figure E.4: Parallel Block-Diagonal-Bordered Sparse LU Factorization Algorithm — Last Diagonal Block

```
Node Program
/* reduce the independent blocks */
for all independent blocks $l$ assigned to this processor
    for all elements $k$ along the diagonal in block $l$
        for each $j \in [1, k-1]$ such that $L_{k,j} \neq 0$
            $b_j \leftarrow b_j - (y_j * L_{k,j})$
        endfor
        $y_k \leftarrow b_k$
    endfor
endfor
```

Figure E.5: Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Diagonal Blocks and Border

```
Node Program
/* calculate updates to the last block */
for all independent blocks $l$ assigned to this processor
    for all non-zero rows $i$ in the lower border of this block
        $\sigma = 0$
        for each $j$ such that $L_{i,j} \neq 0$
            $\sigma \leftarrow \sigma + (y_j * L_{i,j})$
        endfor
        $p \leftarrow \mathcal{P}(i)$ /* map the row value ($i$) to the processor location $p$ */
        Send an active message RPC to the handler function
                Update_FR_LB($\sigma, i$) on processor $p$
    endfor
endfor

/* update the last block with active message RPC handler function Update_FR_LB */
Function Update_FR_LB($\sigma, i$)
        $b_i \leftarrow b_i - \sigma$
End Update_FR_LB
```

Figure E.6: Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Update the Last Diagonal Block — Low-Latency Communications Paradigm

```
Node Program
/* calculate updates to the last block */
for all independent blocks $l$ assigned to this processor
    for all non-zero rows $i$ in the lower border of this block
        $p \leftarrow \mathcal{P}(i)$ /* map the row value ($i$) to the processor location $p$ */
        for each $j$ such that $L_{i,j} \neq 0$
            $\sigma_{i,p} \leftarrow \sigma_{i,p} + (y_j * L_{i,j})$
        endfor
    endfor
endfor

/* update the last diagonal block using data calculated on this processor */
for all data in the buffer
    $b_i \leftarrow b_i - \sigma_i$
endfor

/* update the last diagonal block using data calculated on other processors */
$p_{send} \leftarrow p_{receive} \leftarrow local\_proc\_num$
$p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
$p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
for all processors around a conceptual ring
    Asynchronously send $\sigma_{*,p_{send}}$ to processor $p_{send}$
    Asynchronously receive $\tilde{\sigma}_*$ from processor $p_{receive}$
    /* update the last diagonal block on this processor using
                data from processor $p_{receive}$ */
    for all data in the received buffer
        $b_i \leftarrow b_i - \tilde{\sigma}_i$
    endfor
    $p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
    $p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
endfor
```

Figure E.7: Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Update the Last Diagonal Block — Buffered Communications Paradigm

```
Node Program
/* reduce the last block */
$p_{reduce} \leftarrow p_{start} \leftarrow 0$
if $p_{reduce} = local\_proc\_num$
  forward reduce the first block
  send the values of y from this block to processor 1
end if
for all blocks on this processor
    if $p_{reduce} \neq local\_proc\_num$
      receive the next values of y
      if $p_{start} \neq local\_proc\_num$
        $p_{send} \leftarrow (local\_proc\_num + 1) \bmod N_{procs}$
        send the next values of y on to processor $p_{send}$
      end if
    end if
    $p_{reduce} \leftarrow (p_{reduce} + 1) \bmod N_{procs}$
    if $p_{reduce} = local\_proc\_num$
      forward reduce only the next block with the next values of y
      if not the last block
        $p_{send} \leftarrow (local\_proc\_num + 1) \bmod N_{procs}$
        send the values of y from this block to processor $p_{send}$
        $p_{start} \leftarrow p_{send}$
      end if
    end if
    forward reduce all remaining blocks with the next values of y
endfor
```

Figure E.8: Parallel Block-Diagonal-Bordered Sparse Forward Reduction Algorithm — LU Factorization — Last Diagonal Block

```
Node Program
/* backward substitute the last block */
$p_{sub} \leftarrow p_{start} \leftarrow p_{last\_block}$
if $p_{sub} = local\_proc\_num$
  backward substitute the last block
  $p_{send} \leftarrow (local\_proc\_num + N_{procs} - 1) \bmod N_{procs}$
  send the values of $x$ from this block to processor $p_{send}$
end if
for all blocks on this processor
    if $p_{sub} \neq local\_proc\_num$
      receive the next values of $x$
      if $p_{start} \neq local\_proc\_num$
        $p_{send} \leftarrow (local\_proc\_num + N_{procs} - 1) \bmod N_{procs}$
        send the next values of $x$ on to processor $p_{send}$
      end if
    end if
    $p_{sub} \leftarrow (p_{sub} + N_{procs} - 1) \bmod N_{procs}$
    if $p_{sub} = local\_proc\_num$
      backward substitute only the next block with the next values of $x$
      if not the first block
        $p_{send} \leftarrow (local\_proc\_num + N_{procs} - 1) \bmod N_{procs}$
        send the values of $x$ from this block to processor $p_{send}$
        $p_{start} \leftarrow p_{send}$
      end if
    end if
    backward substitute all remaining blocks with the next values of $x$
endfor
```

Figure E.9: Parallel Block-Diagonal-Bordered Sparse Backward Substitution Algorithm — LU Factorization — Last Diagonal Block

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

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

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

Figure E.10: Parallel Block-Diagonal-Bordered Sparse Backward Substitution Algorithm — LU Factorization — Diagonal Blocks and Border

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 and upper border
            update $\hat{\mathbf{b}}$ for the last diagonal block
            solve for $\hat{\mathbf{x}}^{(k+1)}$ in the last diagonal block */
        endfor
        check convergence
endwhile
```

Figure E.11: The Iterative Framework for the Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm

Node Program

```
/* solve for $\mathbf{x}^{(k+1)}$ in the diagonal blocks and upper border */
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
```

Figure E.12: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Diagonal Blocks

```
Node Program
/* update $\hat{\mathbf{b}} = \hat{\mathbf{x}}$ for the last diagonal block */
for all rows $i$ in the last block assigned to this processor
    $\tilde{x}_i \leftarrow x_i$
    $\hat{x}_i \leftarrow b_i$
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
    Send an active message RPC to the handler function
        Update_x̂$(\sigma, i)$ on processor $\rho_i$
    Poll for active message RPCs
endfor

/* update $\hat{\mathbf{x}}$ with active message RPC handler function Update_x̂ */
Function Update_x̂$(\sigma, i)$
        $\hat{x}_i \leftarrow \hat{x}_i - \sigma$
End Update_x̂
```

Figure E.13: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Update $\hat{\mathbf{b}}$ — Low-Latency Communications Paradigm

```
Node Program
/* update $\hat{\mathbf{b}} = \hat{\mathbf{x}}$ for the last diagonal block */
$\sigma_{*,*} = 0$
for all rows $i$ in the last block assigned to this processor
    $\tilde{x}_i \leftarrow x_i$
    $\hat{x}_i \leftarrow b_i$
endfor
for all non-zero rows $i$ in the lower border of this block
    for each $j$ such that $a_{ij} \neq 0$
        $\sigma_{i,\rho_i} \leftarrow \sigma_{i,\rho_i} - (a_{ij} * x_j)$
    endfor
endfor

/* update $\hat{x}_i$ on this processor using data calculated on this processor */
for all data in the buffer
    $\hat{x}_i \leftarrow \hat{x}_i - \sigma_i$
endfor

/* update $\hat{x}_i$ on this processor using data calculated on other processors */
$p_{send} \leftarrow p_{receive} \leftarrow local\_proc\_num$
$p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
$p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
for all processors around a conceptual ring
    Asynchronously send $\sigma_{*,p_{send}}$ to processor $p_{send}$
    Asynchronously receive $\tilde{\sigma}_*$ from processor $p_{receive}$
    /* update $\hat{x}_i$ on this processor using data from processor $p_{receive}$ */
    for all data in the received buffer
        $\hat{x}_i \leftarrow \hat{x}_i - \tilde{\sigma}_i$
    endfor
    $p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
    $p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
endfor
```

Figure E.14: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Update $\hat{\mathbf{b}}$ — Buffered Communications Paradigm

```
Node Program
/* 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 $a_{ij} \neq 0$
            $\hat{x}_i \leftarrow \hat{x}_i - (a_{ij} * x_j)$
        endfor
        $x_i \leftarrow \hat{x}_i / a_{ii}$
        for all processors $\rho$ requiring $x_i$
            Send an active message RPC to the handler function
                Store_x($x_i, i$) on processor $\rho$
            Poll for active message RPCs
        endfor
    endfor
    Poll for active message RPCs
endfor

/* store $x_i$ with active message RPC handler function Store_x */
Function Store_x($\bar{x}, i$)
        $x_i \leftarrow \bar{x}_i$
End Store_x
```

Figure E.15: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Last Diagonal Block — Low-Latency Communications Paradigm

```
Node Program
/* 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 $a_{ij} \neq 0$
            $\hat{x}_i \leftarrow \hat{x}_i - (a_{ij} * x_j)$
        endfor
        $x_i \leftarrow \hat{x}_i / a_{ii}$
    endfor
    /* update $x_i$ on this processor using data calculated on other processors */
    $p_{send} \leftarrow p_{receive} \leftarrow local\_proc\_num$
    $p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
    $p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
    for all processors around a conceptual ring
        Asynchronously send $\mathbf{x}$ from this color to processor $p_{send}$
        Asynchronously receive $\mathbf{x}$ from this color from processor $p_{receive}$
    endfor
    $p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
    $p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
endfor
```

Figure E.16: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Last Diagonal Block — Buffered Communications Paradigm

```
Node Program
/* check convergence */
$\hat{\epsilon} \leftarrow 0$
for all rows $i$ assigned to this processor
    $\hat{\epsilon} \leftarrow \hat{\epsilon} + abs(\tilde{x}_i - x_i)$
endfor
$\epsilon \leftarrow \hat{\epsilon}$
$p_{send} \leftarrow p_{receive} \leftarrow local\_proc\_num$
$p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
$p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
for all processors $\rho$ around a conceptual ring
    Send an active message RPC to the handler function
        Update_$\epsilon(\hat{\epsilon})$ on processor $\rho$
endfor

/* update $\epsilon$ with active message RPC handler function Update_$\epsilon$ */
Function Update_$\epsilon(\hat{\epsilon})$
        $\epsilon \leftarrow \epsilon + \hat{\epsilon}$
End Update_$\epsilon$
```

Figure E.17: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Convergence Check — Low-Latency Communications Paradigm

```
Node Program
/* check convergence */
$\hat{\epsilon} \leftarrow 0$
for all rows $i$ assigned to this processor
    $\hat{\epsilon} \leftarrow \hat{\epsilon} + abs(\tilde{x}_i - x_i)$
endfor
$\epsilon \leftarrow \hat{\epsilon}$

/* update $\epsilon$ on this processor using data calculated on other processors */
$p_{send} \leftarrow p_{receive} \leftarrow local\_proc\_num$
$p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
$p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
for all processors around a conceptual ring
    Asynchronously send $\hat{\epsilon}$ to processor $p_{send}$
    Asynchronously receive $\hat{\epsilon}$ from processor $p_{receive}$
    $\epsilon \leftarrow \epsilon + \hat{\epsilon}$
    $p_{send} \leftarrow (p_{send} + 1) \bmod N_{procs}$
    $p_{receive} \leftarrow (p_{receive} + N_{procs} - 1) \bmod N_{procs}$
endfor
```

Figure E.18: Parallel Block-Diagonal-Bordered Sparse Gauss-Seidel Algorithm — Convergence Check — Buffered Communications Paradigm

Appendix F

Network Ordering Performance Statistics

This appendix contains tables of statistics to support the manual optimization process to select the best matrix orderings for each of the five matrices used through out this research. Separate sections are provided for direct and iterative solvers. Detailed discussions of the empirical performance results are provided in chapters 7.1 and 7.2.

F.1 Parallel Direct Solver Statistics

Statistics are presented in tables F.1 through F.5 for four orderings of each of the five matrices used through out this research. In this table, N_{LB} is the number of rows/columns in the borders and last diagonal block of the ordered matrix and N_{FILLIN} is the number of fillin. The matrix partitionings that yielded the best empirical performance during benchmarking the parallel software implementations on the Thinking Machines CM-5 are labeled.

F.2 Parallel Iterative Solver Statistics

Statistics are presented in tables F.6 through F.10 for multiple orderings of each of the five matrices used through out this research. In this table, N_{LB} is the number of rows/columns in the borders.

Table F.1: BCSPWR09 — LU and Choleski Factorization Ordering Statistics

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

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

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

Table F.2: BCSPWR10 — LU and Choleski Factorization Ordering Statistics

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

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

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

Table F.3: EPRI6K — LU and Choleski Factorization Ordering Statistics

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

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

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

Table F.4: NiMo-OPS — LU and Choleski Factorization Ordering Statistics

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

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

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

Table F.5: NiMo-PLANS — LU and Choleski Factorization Ordering Statistics

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

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

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

Table F.6: BCSPWR09 — Gauss-Seidel Ordering Statistics

number of nodes	1723
number of edges	2394
number of non-zeros	6511
percent non-zeros	0.22%

Maximum Partition Size	N_{LB}	**% Operations in Blocks and Borders**	**Colors**	**Floating Point Operations (×−)**		
				Diagonal Blocks	**Borders**	**Last Block**
16	258	92.0%	3	4951	1040	520
32	190	94.3%	3	5245	896	370
64	153	95.6%	3	5397	825	289
96	131	96.3%	3	5538	734	239
128	128	96.1%	3	5544	715	252
160	117	96.8%	3	5610	694	207

Table F.7: BCSPWR10 — Gauss-Seidel Ordering Statistics

number of nodes	5300
number of edges	8271
number of non-zeros	21842
percent non-zeros	0.08%

Maximum Partition Size	N_{LB}	**% Operations in Blocks and Borders**	**Colors**	**Floating Point Operations (×−)**		
				Diagonal Blocks	**Borders**	**Last Block**
32	789	92.9%	3	16907	3384	1551
64	668	93.9%	4	17476	3032	1334
128	578	95.0%	3	18037	2705	1100
192	513	95.7%	3	18394	2509	939
256	476	95.8%	3	18613	2319	910
320	516	95.5%	4	18311	2555	976
384	504	95.5%	3	18364	2504	974
448	489	95.7%	3	18492	2411	939
512	511	95.4%	3	18275	2562	1005

Table F.8: EPRI6K — Gauss-Seidel Ordering Statistics

number of nodes	6692
number of edges	10535
number of non-zeros	27762
percent non-zeros	0.06%

Maximum Partition Size	N_{LB}	**% Operations in Blocks and Borders**	**Colors**	**Floating Point Operations (×−)**		
				Diagonal Blocks	**Borders**	**Last Block**
32	655	89.7%	12	21260	3641	2853
64	524	92.0%	11	22308	3220	2234
128	387	94.1%	9	23189	2938	1635
192	346	95.2%	9	24088	2350	1324
256	345	93.3%	12	23084	2691	1849
320	338	93.0%	11	23383	2651	1652
384	256	98.1%	6	25374	1872	516
448	200	98.7%	4	25931	1473	358
512	210	98.4%	6	25760	1562	440

Table F.9: NiMo-OPS — Gauss-Seidel Ordering Statistics

number of nodes	1766
number of edges	2506
number of non-zeros	6778
percent non-zeros	0.22%

Maximum Partition Size	N_{LB}	**% Operations in Blocks and Borders**	**Colors**	**Floating Point Operations (×−)**		
				Diagonal Blocks	**Borders**	**Last Block**
16	251	92.3%	4	5218	1041	519
32	173	94.9%	3	5562	867	349
64	136	95.5%	4	5708	766	304
96	130	96.1%	4	5763	753	262
128	101	97.4%	3	5986	619	173
160	105	97.4%	3	5945	658	175

Table F.10: NiMo-PLANS — Gauss-Seidel Ordering Statistics

number of nodes	9430
number of edges	14001
number of non-zeros	37432
percent non-zeros	0.04%

Maximum Partition Size	N_{LB}	**% Operations in Blocks and Borders**	**Colors**	**Floating Point Operations (×−)**		
				Diagonal Blocks	**Borders**	**Last Block**
32	886	92.1%	11	29653	4683	3096
64	721	92.2%	11	30311	4138	2983
96	612	94.0%	10	31361	3831	2240
128	553	93.8%	12	31423	3510	2499
192	443	96.5%	9	33113	2986	1333
256	402	97.3%	8	33589	2809	1034
320	436	95.5%	11	32516	3240	1676
384	378	97.4%	8	33764	2670	998
448	312	98.6%	5	34670	2228	534
512	313	98.0%	7	34313	2348	771

Biographical Data

Name:	David P. Koester
Date and Place of Birth:	28 April 1955 Aurora, Illinois USA
High School:	East Aurora High School Aurora, Illinois USA Graduated 1973
Undergraduate Degree:	Augustana College Rock Island, Illinois USA B.A., Mathematics, Computer Science, and German, 1977
Masters Degree:	The Ohio State University Columbus, Ohio USA Teaching Assistantship: 1977–78 M.A.S., Applied Statistics, 1978
Graduate Work:	Syracuse University Syracuse, New York USA NPAC Graduate Research Assistant: 1992–1995

¡

To Close on the Lighter Side ...

In a forest, a ferret bumps into a little mouse, and says, "Hi, Junior, what are you up to?
"I'm writing a dissertation on how mice eat ferrets," said the mouse.
"Come now, friend mouse, you know that's impossible!"
"Well, follow me and I'll show you." They both go inside the mouse's dwelling and after a while the mouse emerges with a satisfied expression on his face.
Comes along a badger. "Hello, what are you doing these days?"
"I'm writing the second chapter of my thesis, on how mice devour badgers.'
"Are you crazy? Where is your academic honesty?"
"Come with me and I'll show you." As before, the mouse comes out with a satisfied look on his face and a diploma in his paw.
Finally, the camera pans into the mouse's cave and, as everybody should have guessed by now, we see a mean-looking, huge Fox sitting next to some bloody and furry remnants of the ferret and badger.

The moral: It's not the contents of your thesis that are important — it's your Ph.D. advisor that really counts.

The UNIX Fortune Cookie Routine[‡]

[‡] Names have been changed here, not to protect the innocent, but rather to make this classic story a bit more relevant to my academic career at Syracuse University... Never underestimate the effect of a variable transform! ;)