ON THE LU FACTORIZATION OF SEQUENCES OF IDENTICALLY STRUCTURED SPARSE MATRICES WITHIN A DISTRIBUTED MEMORY ENVIRONMENT

BY

STEVEN MICHAEL HADFIELD

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1994

ACKNOWLEDGMENTS

First, I must express my sincere appreciation to my advisor, Dr. Timothy A. Davis for guiding, challenging, and encouraging me throughout this effort. His intelligence, patience, and concern have helped me establish high academic and personal goals. Also, I would like to thank my supervisory committee members, and especially Dr. Theodore Johnson, for their support and many helpful suggestions. In addition, I would like to express my appreciation to Professor Iain Duff of Harwell Laboratory, U.K. for reviewing and encouraging key aspects of this effort and Dr. Steve Zitney of Cray Research for providing the much needed matrix sequences for the evaluation of lost pivot recovery.

In additon, I recognize the National Science Foundation for their funding of the development of unsymmetric-pattern, multifrontal method (ASC-9111263, DMS-9223088). Furthermore, I appreciate the financial and professional support of the Air Force Institute of Technology.

Personally, I thank my parents for their continual support and encouragement (and especially Mom for her many hours of prayer). I would like to thank my children: Melissa, Andrew, and Christopher for the sacrifices they have had to endure during this program. Yet most of all, I thank my wonderful wife, Marissa, for her constant love and support. She makes it all worthwhile and helps me keep it all in perspective.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

ON THE LU FACTORIZATION OF SEQUENCES OF IDENTICALLY STRUCTURED SPARSE MATRICES WITHIN A DISTRIBUTED MEMORY ENVIRONMENT

By

Steven Michael Hadfield

April 1994

Chairman: Timothy A. Davis
Major Department: Computer and Information Sciences

This research effort studies the parallel LU factorization of sequences of identically structured sparse matrices using the unsymmetric-pattern, multifrontal method of Davis and Duff. Computational burden is reduced by using the computational structure (called an assembly directed acyclic graph: DAG) that results from analysis of the first matrix for the numerical factorization of the subsequent matrices. Execution time is reduced via exploitation of parallelism in the assembly DAG. The theoretical parallelism in the assembly DAG is investigated using both analysis and simulation. Achievable parallelism is evaluated using a simulation model based on the nCUBE 2 distributed memory multiprocessor.

A fixed pivot order parallel LU factorization routine is implemented on the nCUBE 2 and evaluated. Key subissues within this implementation are task scheduling, subcube allocation, subcube assignment, and data assignments to reduce communications and overall execution time. The resulting implementation is shown to be competitive with conventional techniques and demonstrates significant parallel performance with excellent scalability.

Of greatest significance is the theoretical development and implementation of a lost pivot recovery capability for the unsymmetric-pattern, multifrontal method. The

capability incorporates a lost pivot avoidance strategy with both inter- and intra-frontal matrix pivot recovery mechanisms. The inter-frontal matrix pivot recovery mechanisms migrate lost pivots to subsequent frontal matrices and accommodate corresponding side effects using relationships represented in the assembly DAG. Intra-frontal matrix pivot recoveries are done via a pipelined partial dense factorization kernel that handles unsymmetric permutations across processors. Evaluation of the lost pivot recovery capability using three matrix sequences from chemical engineering problems shows that significant execution time savings are afforded when executing on either a single processor or multiple processors.

CHAPTER 1
INTRODUCTION

With the advent and refinement of practical parallel processing platforms, there has been a flurry of research in concurrent algorithms that can efficiently utilize these resources. A primary emphasis of this research has been numerical algorithms, in general, and, in particular, algorithms for the direct solution (as opposed to iterative approximation) of systems of linear equations. Extensive research has been done on solving such systems when the coefficient matrix is dense. Such matrices are frequently encountered and have very regular structure that can be exploited in algorithm development.

When the coefficient matrix is sparse considerably more parallelism may be available, although it is more difficult to exploit. For sparse, symmetric, positive definite matrices, elimination tree structures can be derived to expose parallelism in the computations. Much of the parallel sparse matrix research focus to date as been on such matrices due to their frequent occurrence, the exposure of parallelism in their elimination tree, and the guarantee of numerically stable pivots on the diagonal.

Parallel algorithms for solving more general linear systems are more difficult as they are characterized by sparse matrices where the nonzero structure can be highly unsymmetric and the diagonal entries may not be numerically acceptable as pivots. Hence, parallelism is irregular and numerical pivoting considerations become important.

The most common direct technique for solving such systems is to factorize the coefficient matrix into the product of a lower triangular matrix, L, and an upper triangular matrix, U. Triangular solves (forward and backward substitutions) can then be used to solve the system. One approach to LU factorization that holds significant potential for parallel implementation is the unsymmetric-pattern multifrontal method of Davis and Duff [34, 35].

Like most sparse matrix factorization algorithms, the unsymmetric-pattern multifrontal method has two principal operations, analyze and factorize. The analyze operation selects matrix entries to act as pivots for the numerical factorization with objectives of reducing computations and maintaining numerical stability. The factorize operation then uses the selected pivots to perform the actual numerical factorization. Furthermore, there are many sparse matrix problems that require the factorization of sequences of identically structured sparse matrices. Such sequences are common to solution techniques for the systems of differential algebraic equations used in many application areas including circuit simulations, chemical engineering, magnetic resonance spectroscopy, and air pollution modeling [105, 144, 143, 146, 37, 82, 49, 18, 97, 108, 131, 43, 50]. When factorizing these sequences of identically structured sparse matrices, the computational burden can be eased by only doing the

analyze operation once for the entire sequence and using the selected pivots for the subsequent factorization of each matrix in the sequence.

1.1 Topic Statement

The focus of this research effort is to study and implement parallel LU factorization algorithms based on an unsymmetric-pattern multifrontal approach and targeted for a distributed memory environment. Specifically, the parallel algorithms developed will perform the numerical factorization of sequences of identically structured sparse matrices using a directed acyclic graph (DAG) structure produced by a sequential implementation of Davis and Duff's Unsymmetric Multifrontal Package (UMFPACK) [34, 35, 32]. This DAG structure, known as the *assembly DAG*, defines the necessary computations in terms of the partial factorizations of small, dense submatrices, which are known as *frontal matrices* and represented by nodes in the assembly DAG. The edges of the assembly DAG define the necessary data communication between frontal matrices, as well as the corresponding precedence relation. Thus, the assembly DAG can also serve as a task graph for the parallel computations.

The first objective is to determine how much parallelism is available and exploitable within the computational structure defined by the assembly DAG. Of interest within this objective is how much theoretical parallelism in inherent in the method itself and how much of this theoretical parallelism can be achieved on contemporary parallel architectures. The theoretical and practical parallelism will be investigated by analysis and simulation. Then the practical parallelism will be scrutinized further by implementing a parallel, distributed memory multifrontal refactorization algorithm and empirically measuring its achieved parallelism.

When factorizing a sequence of identically structured sparse matrices where additional properties such as symmetry, positive-definiteness, or diagonal dominance are not assumed, the changing numerical values of the various entries can cause the anticipated pivots to no longer be acceptable numerically. The capability to recover from these lost pivots and complete the factorization without having to recompute a new assembly DAG could significantly reduce the computation time required for factorization of the sequence of matrices. Therefore, the second objective is to develop a *lost pivot recovery* capability within the context of an unsymmetric-pattern multifrontal method. The actual implementation of the lost pivot recovery capability will be based on the parallel, distributed memory multifrontal algorithm from the first objective. Thus, both the sequential and parallel performance of lost pivot recovery may be assessed.

1.2 Overview

Chapters 2 through 7 of this document report the results of this research effort. Chapter 8 reviews the critical results and discusses possible future efforts.

In Chapter 2, pertinent background and the results of related efforts are presented. The LU factorization process and its stability and error bounds are described. A discussion follows of key sparse matrix concepts such as fill-in, the analyze and factorize phases, the relationship of sparse matrices to graph theory, scatter and gather operations, and block triangular form. Next the key concepts of the multifrontal approach

to LU factorization are illustrated. A mathematical framework is then shown for how sequences of identically structured sparse matrices arise in various solution techniques with references to specific applications provided. Some methods for parallel matrix computations are summarized for both dense matrices (such as the frontal matrices of a multifrontal method) as well as sparse matrices. Included in this discussion will be key results regarding parallelism from earlier efforts that were based on a symmetric-pattern, multifrontal method. Finally, some scheduling techniques for distributed memory parallel processing are summarized.

Chapter 3 describes the nCUBE 2 distributed memory parallel processor on which the algorithms are implemented. First, the key hardware features are presented followed by an overview of the software environment. Then the results of an empirical characterization of the nCUBE 2's performance are presented. This characterization will aid in assessing achievable parallelism and in making later design and implementation decisions.

The key results of this research effort begin in Chapter 4, which describes how the theoretical and achievable parallelism of the unsymmetric-pattern multifrontal method are assessed using both analysis and simulation. First, analytical models are used to determine how much parallelism is available within the computational structure provided by the assembly DAG with the various models corresponding to different sources of parallelism. These analytical models all assume an unbounded number of processors and an underlying parallel random access memory (PRAM) model and thus provide an assessment of theoretical parallelism. Next an evaluation of how much parallelism is achievable on bounded processor sets is done using simulation techniques. Finally, performance characteristics of the nCUBE 2 are used to extend the simulation models to represent a realistic distributed memory implementation.

With the understanding of the parallel characteristics of the unsymmetric-pattern multifrontal method gained from the analytical and simulation models, Chapter 5 begins the actual implementation. As the required computations are dominated by the partial factorization of frontal matrices, this chapter is devoted to the development of high performance parallel partial dense factorization kernels. Five distinct routines are developed and tested. The first two confine themselves to the given pivot ordering and do not check the numerical acceptability of the pivots. These routines are applicable to matrices that are known to be diagonally dominant, and thus the diagonal entries are assured to be acceptable as pivots. The second of these routines uses *pipelining* to overlap required communication with computation. The next two partial dense factorization kernels address numerical pivoting with alternative pivot selection done within the frontal matrix's pivot block. This can be difficult as the columns of the pivot block may be distributed across several processors. The second of these routines is pipelined to improve performance. The fifth kernel is designed to handle frontal matrices that have only a single potential pivot. Since single pivot frontal matrices occur frequently and do not require all the additional logic of the more general earlier routines, there is a performance advantage to be gained by a single pivot partial dense factorization routine. The performance of these kernels is

assessed, and analytical models of their execution times based on various processor sets are defined. These models will be useful in scheduling frontal matrix tasks.

With the partial dense factorization kernels available, Chapter 6 discusses the implementation of a full parallel refactorization capability based on the unsymmetric-pattern multifrontal method. Of the most critical issues to be addressed in this implementation are the scheduling and allocation of processors to frontal matrix tasks. Two distinct scheduling methods are developed. One focuses on minimizing required communication by overlapping the assignment of frontal matrix tasks to processor sets. The other method tries to reduce overall execution time by more sophisticated management of the hypercube topology with communication reduction a secondary concern. Allocation of processor sets to frontal matrix tasks is done using a proportional allocation scheme based on the predicted execution times of tasks and the tasks available to execute. A number of mechanisms are also developed for the distribution of frontal matrix columns to processors in hopes of further reducing required communication costs. All of the scheduling, allocation, and assignment activities are done as part of a sequential software interface that initiates and controls the parallel refactorization. The details of the parallel refactorization code are also discussed and include specifics on how frontal matrices are represented, how original matrix entries and contributions between frontal matrices are handled, and how frontal matrix tasks are synchronized. A distributed forward and back solve capability is also implemented and takes advantage of the distributed storage of the LU factors that results from the parallel refactorization. With the implementation complete, a detailed performance evaluation is accomplished. Both the achieved parallelism and competitiveness of the implementation are assessed. The effectiveness of the various scheduling, allocation, and assignment mechanisms is also addressed.

Perhaps the most significant of the results from this research effort is development of a robust lost pivot recovery capability for the unsymmetric-pattern multifrontal approach to LU factorization. Chapter 7 describes this capability. First an extensive theoretical development of the lost pivot recovery capability is done. This is followed by a description of the implementation that was done by extending the distributed memory parallel refactorization software of Chapter 6. Finally, a performance evaluation addresses the sequential effectiveness of the lost pivot recovery capability, its parallel execution time and speed-up characteristics, and its memory requirements and scalability.

CHAPTER 2
BACKGROUND AND RELATED EFFORTS

Prerequisite to pursuing the study and development of a parallel unsymmetric-pattern multifrontal method is the understanding of a number of key concepts related to linear algebra and sparse matrices. Specifically, the process of LU factorization and concepts of algorithm stability and error analysis are necessary. Furthermore, concepts and techniques for sparse matrices must be defined and described as they will be frequently referenced.

In addition to this general background, the specifics of the unsymmetric-pattern multifrontal method of Davis and Duff [34, 35] are necessary as this entire effort is based on that method. Mathematical techniques that give rise to sequences of identially structured sparse matrices together with specific applications justify the need to be able to factorize such matrix sequences.

Finally, a summary of previous results in both parallel matrix computations and multiprocessor scheduling help to define realistic goals for this effort and establish its significance.

2.1 LU Factorization

The most common method for solving general systems of linear equations in the form $A\bar{x} = \bar{b}$ (where A is an $n \times n$ matrix of full rank and $\bar{x}$ and $\bar{b}$ are vectors of length n) is LU factorization which is based on Gauss Elimination [48]. In this method, the matrix A is factorized into $A = LU$ where L is unit lower triangular and U is upper triangular. The problem $A\bar{x} = \bar{b}$ then becomes

$$LU\bar{x} = \bar{b}$$

and can be solved with two subsequent triangular solves. The first intermediate solve is the forward substitution

$$L\bar{y} = \bar{b}.$$

The result, $\bar{y}$, is then used in the second triangular solve that is a back substitution

$$U\bar{x} = \bar{y}.$$

In the actual factorization process, A can be overwritten with the factors LU with the unit diagonal of L stored implicitly. The actual process proceeds down the diagonal of the matrix using the diagonal entries as pivots. Row and sometimes column interchanges may be required to replace zero entries on the diagonal, and similar permutations are common in order to preserve the numerical stability of the algorithm. An illustrative version of the basic LU factorization algorithm is

presented in Figure 2.1 that assumes good pivots on the diagonal. The version of Gauss Elimination presented in Figure 2.1 is based on the most common ordering of the nested loops (i.e., the kij ordering). However, alternative orderings are also possible that correspond to all six of the possible permutations of the indices k, i, and j [78].

```
for k := 1 to n do
      for j := k to n do
            U(k,j) := A(k,j)
      endfor
      for i := k+1 to n do
            L(i,k) := A(i,k)/A(k,k)
            for j := k+1 to n do
                  A(i,j) := A(i,j) - L(i,k) * U(k,j)
            endfor
      endfor
endfor
```

Figure 2-1. LU Factorization Algorithm

When row and column permutations are introduced into the algorithm, the factorization becomes

$$PAQ = LU$$

where P provides the row permutations and Q provides the column permutations [48]. The equation $A\bar{x} = \bar{b}$ then becomes

$$PAQQ^T\bar{x} = P\bar{b}$$

with the following sequence of a matrix multiply, two triangular solves, and another matrix multiply required to complete the solution based on the factorization: $\bar{c} = P\bar{b}$, $L\bar{y} = \bar{c}$, $U\bar{w} = \bar{y}$, $\bar{x} = Q\bar{w}$. As P and Q are permutation matrices, they can be stored in $O(n)$ storage and the required multiplications can also be done in $O(n)$ time.

2.2 Algorithm Stability and Error Analysis

There are two types of error analysis typically used. The first, *forward error analysis*, provides bounds directly on the relative error between the computed and actual solutions. Such a technique is typically overly pessimistic and does not distinguish between error sources. Thus, this type of analysis is not as useful as the second type, which is backwards error analysis.

The concept of *backwards error analysis* considers the computed solution as the exact solution to a "nearby" problem. Analysis of the round off computing error associated with the algorithm in use provides bounds on the distance between the original problem and the nearby version for which the true solution has been computed. The bounds on this distance provide a measure of the algorithm's stability.

Perturbation theory is then used to relate the distance between the two problems to the distance between their respective true solutions. While the computing round off error is dependent upon the algorithm in use and is, to a certain extent, controllable by algorithmic alterations, the perturbations are strictly a function of the particular problem instance and cannot be controlled. Hence, backwards error analysis provides a means to distinguish between controllable and uncontrollable error sources.

Furthermore, backwards error analysis can be used in an a posteriori fashion. Here the actual deviations between the solved and original problems can be measured (not just bounded). These more accurate results are then combined with the perturbation theory to provide a refined error bound on the solution. Thus, backwards error analysis provides both the distinction of error sources and a refined error bound.

This section addresses both aspects of backwards error analysis. Computing round off errors and algorithm stability for LU factorization are first discussed. Means of insuring stability are emphasized. Then, perturbation theory is addressed. The effects of problem conditioning are discussed with the condition number defined as a means to quantify these effects. The integration of stability and conditioning is used to develop an overall error bound for LU factorization.

Computing Round Off Errors: The two primary sources of computing errors are the representation of real numbers by finite digit approximations and the corresponding loss of significant digits during finite digit arithmetic. When real numbers are stored and used in a computer, they are typically approximated by a floating point value represented in a finite bit storage cell and a bounded amount of relative error is introduced. Consider the real value represented by a. The corresponding floating point value would be

$$fl(a) = a(1+\epsilon),\ \epsilon \leq \mu$$

where ϵ is the relative error introduced by the finite bit representation, which is bounded by the machine precision, μ, of the particular processor [78].

In a similar fashion, finite arithmetic computations introduce additional errors as results of the computations are subject to the same finite storage restrictions. Particularly vulnerable to this effect is addition (subtraction) as order of magnitude differences (similarities) in the values of the two operands cause increasing losses in accuracy.

Combining the effects of inaccuracies due to floating point representations and computations, we find a cumulative effect as illustrated by

$$fl(a+b) = (fl(a) + fl(b))(1+\epsilon_1) = ((a(1+\epsilon_2)) + (b(1+\epsilon_3)))(1+\epsilon_1)$$

where ϵ_1, ϵ_2, and ϵ_3 are each bounded by the machine precision μ [78].

Algorithm Stability: Algorithm stability (or instability) is a measure of the effects of computing round off errors on the computation sequence dictated by the algorithm. The magnitude of this effect is measured by the difference between the factorized representation of the problem produced by the Gaussian Elimination algorithm ($\tilde{L}\tilde{U}$) and the true problem (A). This difference is accounted for in the matrix

term H where

$$\tilde{L}\tilde{U} = A + H.$$

Wilkinson [139] showed that the entrywise deviations accounted for by H in the absence of pivoting are bounded by

$$|h_{ij}| \leq 5.01n\mu\rho, \text{ where } \rho = \max_k |a_{ij}^{(k)}|$$

for each $h_{ij} \in H$ where $a_{ij}^{(k)}$ refers to the value of a^{ij} after the kth step of factorization. Extensive growth in the $a_{ij}^{(k)}$'s can thus have catastrophic effects.

Pivoting for Stability: *Partial Pivoting* is the typical mechanism used to control growth of the pivots. At each step, k, a new pivot is chosen from the kth column of the active submatrix such that

$$|a_{kk}^{(k)}| \geq |a_{ik}^{(k)}|, \ i > k.$$

With this approach the growth represented by ρ is limited to

$$\rho = 2^{n-1} \max_{i,j} |a_{ij}|.$$

While this is not a particularly good bound, in practice the results are consistently much better [139].

A stronger alternative to partial pivoting is *complete pivoting*. Complete pivoting chooses the kth pivot as the largest entry in the entire active submatrix $A^{(k)}$. In doing so the growth is limited to

$$\rho = f(n) \max |a_{ij}|$$

where $f(n)$ is a nearly linear function [138].

A problem with both partial and complete pivoting is that the rigid pivot selection rules can result in extensive fill-in (*fill-in* is the changing of zero entries into nonzero entries) when dealing with sparse matrices. *Threshold pivoting* provides an alternative that allows greater flexibility in pivot selection to aid in the preservation of sparsity [48]. Pivot selection in threshold pivoting requires the selected pivot to meet

$$|a_{kk}^{(k)}| \geq u|a_{ik}^{(k)}|, \ i > k, \ 0 \leq u \leq 1.$$

Here u is a parameter that is typically set between 0.001 and 0.1. When threshold pivoting is employed, the growth bounding function becomes

$$\rho \leq (1 + u^{-1})^{n-1} \max |a_{ij}|.$$

When LU factorization is applied to a matrix A and no zero pivots are encountered [78], the bound on H in $\tilde{L}\tilde{U} = A + H$ is

$$|H| \leq 3(n-1)\mu[|A| + |\tilde{L}||\tilde{U}|] + O(\mu^2).$$

Here $|M|$ refers to the largest entry in the matrix M.

Similar analysis for a triangular solve reveals that the computed solution, $\tilde{y}$, to the triangular solve problem $L\bar{y} = \bar{b}$ is the exact solution of

$$(L + F)\tilde{y} = \bar{b}$$

with

$$|F| \leq n\mu|L| + O(n^2).$$

Combining the LU factorization results with the results of the two required triangular solves reveals that the computed solution, $\tilde{x}$, to the problem $A\bar{x} = \bar{b}$ is the exact solution to some problem

$$(A + E)\tilde{x} = \bar{b}$$

for some E satisfying

$$|E| \leq n\mu(3|A| + 5|\tilde{L}||\tilde{U}|) + O(n^2).$$

When partial pivoting is included, the entries of $\tilde{L}$ can be bounded by 1 (and $\|\tilde{L}\|_\infty \leq n$) and the bound on E in the infinity norm becomes

$$\|E\|_\infty \leq n\mu(3\|A\|_\infty + 5n\|\tilde{U}\|_\infty) + O(n^2).$$

This bound for $\|E\|_\infty$ provides the measure of the distance of the problem for which $\tilde{x}$ is the exact solution from the original problem. This measure will be combined with the perturbation theory to provide a bound on the relative error of the computed solution.

Ill Conditioning: At the heart of perturbation theory is the concept of how the distance between two nearby problems relates to the distance between their solutions. When the distance between solutions is great relative to the distance between the problems, the problems are said to be *ill conditioned*. Distance here is measured in an appropriate matrix or vector norm. The conditioning of a problem can be quantified by a value known as the condition number. Analysis can then utilize this condition number to bound the relative error in the solution space based on the relative error in the problem space (distance between original problem and the problem exactly solved by the computed solution).

The condition number can be defined in terms of matrix norms as

$$\kappa_p(A) = \|A\|_p\|A^{-1}\|_p$$

where p designates the specific norm in use [48]. An alternative, but equivalent formulation based on singular values is

$$\kappa_2(A) = \sigma_1/\sigma_n$$

where σ_1 is the largest singular value and σ_n the smallest [78]. This later formulation illustrates that ill conditioning can be viewed as a measure of singularity as σ_n will be zero for a singular $n \times n$ matrix, and a relatively small value of σ_n (as compared to σ_1) will produce a large condition number.

Analysis of how conditioning affects the relative error in the computed solution produces the following relationship between the relative distance between the original problem $A\bar{x} = \bar{b}$ and the one exactly solved by the computed solution $(A + E)\tilde{x} = \bar{b}$. Specifically, the relative distance between the true ($\bar{x}$) and computed ($\tilde{x}$) solutions is

$$\frac{\|\tilde{x} - x\|_\infty}{\|x\|_\infty} \leq \frac{\kappa_\infty(A)}{1 - r} \frac{\|E\|_\infty}{\|A\|_\infty}$$

where $r = \|E\|\|A^{-1}\| < 1$ [78].

Prevention of ill conditioning may be addressed by analyzing if the process has the same sensitivities as the model. If not, identify the problem sources in the model and alter the model.

Overall Error Bound: The overall error bound on the relative error of the solution is obtained by combining the round off error analysis. Recall that the computed solution, $\tilde{x}$, solves a nearby problem $(A + E)\tilde{x} = \bar{b}$ where

$$\|E\|_\infty \leq n\mu(3\|A\|_\infty + 5n\|\tilde{U}\|_\infty) + O(n^2).$$

Also, the condition of the problem affects the relative error in the solution by

$$\frac{\|\tilde{x} - x\|_\infty}{\|x\|_\infty} \leq \left(\frac{\kappa_\infty(A)}{1 - r}\right) \frac{\|E\|_\infty}{\|A\|_\infty}$$

where $r = \|E\|\|A^{-1}\| < 1$. Combining these two results, we obtain the overall error bound of

$$\frac{\|\tilde{x} - x\|_\infty}{\|x\|_\infty} \leq \frac{\kappa_\infty(A)}{1 - r} \left[n\mu \left(3 + 5\frac{\|\tilde{U}\|_\infty}{\|\tilde{A}\|_\infty}\right)\right] + O(\mu^2).$$

Measuring Stability: A large relative error in the solution of the equation $Ax = b$ can be the result of either an unstable algorithm or a poorly conditioned problem or both. While the condition of the problem is typically not controllable, the stability of the algorithm is, and a large relative error would raise the question of whether or not it can be reduced by a more stable algorithm. Thus, there is a need to measure the stability of an algorithm. The relative residual, as defined below, provides such a measure [84].

$$\text{relative\_residual} = \frac{\|A\tilde{x} - b\|}{\|b\|}$$

In this equation, $\tilde{x}$ represents the computed solution. To understand how the relative residual measures stability, a simple derivation is useful. Recall that

$$A = \tilde{L}\tilde{U} + H \text{ and } \tilde{L}\tilde{U}\tilde{x} = b$$

where $\tilde{L}$ and $\tilde{U}$ are the computed LU factors. H then represents the differences between the original and actually solved versions of the problem. Hence we can multiply the first equation by $\tilde{x}$ and get the following derivation:

$$\begin{aligned} A\tilde{x} &= (\tilde{L}\tilde{U} + H)\tilde{x} \\ &= \tilde{L}\tilde{U}\tilde{x} + H\tilde{x} \\ &= b + H\tilde{x}. \end{aligned}$$

The vector b is then subtracted from each side, norms are taken, and the results divided by $\|b\|$ to get

$$\frac{\|A\tilde{x} - b\|}{\|b\|} = \frac{\|H\tilde{x}\|}{\|b\|}.$$

Thus the relative residual provides a relative measure of the size of H, which is controlled by the algorithm's stability. If the relative error is large but the relative residual small, then the trouble is with the conditioning of the problem and not the stability of the algorithm.

2.3 Sparse Matrix Concepts

The matrices used in many applications tend to have a very significant number of entries that are zero. This is particularly true and significant for very large matrices as the computations on such large matrices, as dictated by direct solution methods, would be too numerous to be done in a reasonable amount of time. The sparsity induced by the many zero entries allows special techniques to be employed that significantly reduce the required number of computations. Thus, the number of nonzero entries in a matrix becomes a very critical measure of the amount of computations required for a particular sparse matrix. Furthermore, as the number of nonzeros is so critical, special attention is paid to minimize/reduce the number of matrix entries that were originally zero and become nonzero due to the computations. This situation is commonly called *fill-in*.

The discussion of this section first relates sparse matrices to graph theory, which is an important tool used in structuring sparse matrix computations. Special techniques for storing sparse matrices are then addressed. These techniques focus on the efficient use of memory while preserving the ability to access the matrix entries efficiently. A typical preprocessing step of reduction to block triangular form is then described followed by a brief discussion of how the Markowitz Criterion can be used during the analyze phase to determine the pivot ordering.

Relationship to Graph Theory: Critical to the exploitation of sparsity in sparse matrices is the understanding and representation of the inherent matrix structure. Graph theory provides an excellent vehicle for this purpose [48, 70]. By representing the structure of a matrix with a graph construct, exploitable patterns can be more readily recognized. Furthermore, as operations take place on the matrix that alter its structure, corresponding changes can be made to the graph representation to track the structural dynamics of the process.

In this section, the graph representations of both unsymmetric and symmetric matrices are discussed, as well as the corresponding graph dynamics that occur due to Gauss Elimination and matrix permutations.

If each row and each column in an $n \times n$ matrix A is labeled using $1, .., n$ and each such label is associated with a node in a graph, the structure of the matrix can be represented with edges from node i to node j for each nonzero a_{ij} entry in the matrix A. These edges are directed and thus uniquely identify each nonzero entry in the structure of any matrix. The corresponding graph is called a *digraph* or *directed graph*. As an example, the matrix structure shown in Figure 2-2 is represented by the digraph structure shown in Figure 2-3.

$$A = \begin{pmatrix} X & & X \\ & X & \\ X & X & \end{pmatrix}$$

Figure 2-2. Sample Matrix Structure

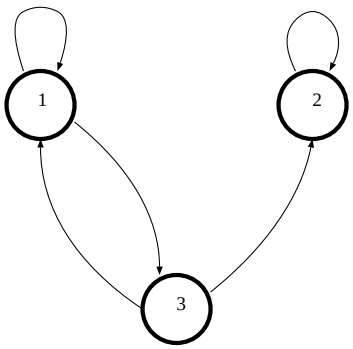

Figure 2-3. Sample Digraph for Matrix A

If the matrix has a symmetric pattern (that is, if for each nonzero a_{ij} entry, the a_{ji} entry is also nonzero), then each pair of nodes connected by a directed edge will have a second edge connecting them that is oriented in the opposite direction of the first edge. These two directed edges can be replaced by a single undirected edge, and thus matrices with a symmetric pattern can be represented with an undirected graph. A sample symmetric pattern matrix structure and the corresponding undirected graph are shown in Figures 2-4 and 2-5.

The undirected graph for matrix B (as shown in Figure 2-5) happens to form a tree structure with node 1 as the root. While this does not occur with all unsymmetric matrices, it is a very important special case that will be discussed in more detail later.

When Gauss Elimination is performed on a matrix to zero a column below the diagonal, the structure of the matrix changes and corresponding alterations occur within the graph representation [118]. In particular, assume the column below the

$$B = \begin{pmatrix} X & X & X \\ X & X & \\ X & & X \end{pmatrix}$$

Figure 2-4. Sample Symmetric Matrix Structure

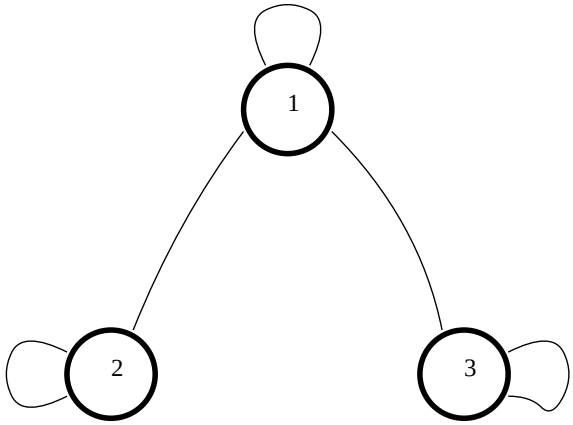

Figure 2-5. Sample Undirected Graph for Symmetric Matrix B

diagonal that corresponds to the node y in the graph is eliminated. Then, the node y may be removed from the graph, and for any pair of nodes x and z such that both (x, y) and (y, z) are existing edges, the edge (x, z) is added to the graph (assuming it does not already exist). The resulting graph structure corresponds to the active submatrix after this step of Gauss Elimination. This relationship is shown in the matrices of Figure 2-6 and the digraphs of Figure 2-7.

$$A = \begin{pmatrix} X & X & & X \\ & X & & X \\ X & & X & \\ X & X & & X \end{pmatrix}$$

$$A^{(1)} = \begin{pmatrix} X & X & & X \\ & X & & X \\ & X & X & X \\ & X & & X \end{pmatrix}$$

Figure 2-6. Matrix Structure Before and After Elimination

Notice that in the graphs of Figure 2-7 edge $(3, 4)$ was added due to the existence of $(3, 1)$ and $(1, 4)$. Likewise $(3, 2)$ was added due to $(3, 1)$ and $(1, 2)$. However, $(4, 2)$ was not added since it was already in the graph. Interestingly, the changes that take

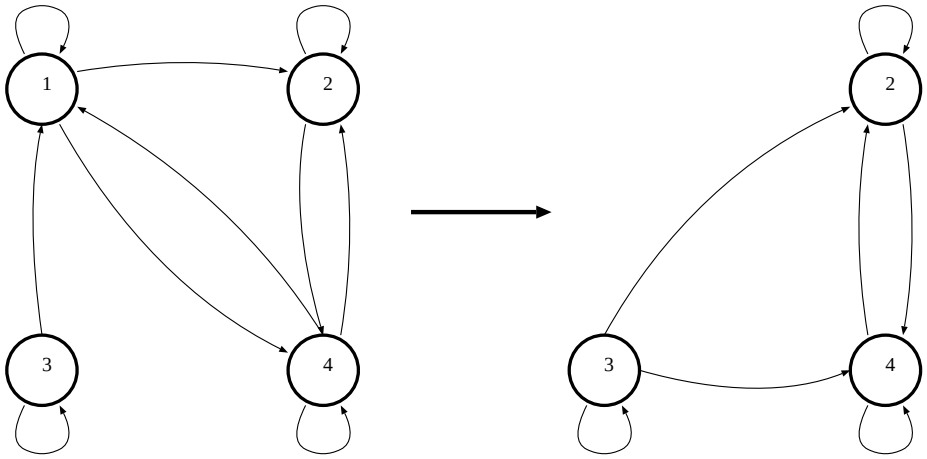

Figure 2-7. Graph Reduction Due to Gauss Elimination

place in the graph structure due to corresponding Gauss Elimination in the matrix are very similar to the graph theory process of computing transitive closures.

Permutations are another very important operation performed on matrices. If the permutation is symmetric (that is, both rows and columns are interchanged), then the corresponding graph alterations amount to simply relabeling the nodes. For example consider the symmetric permutation $B_{new} = PBP^T$ where B is as shown in Figure 3.3 and P is defined as

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

The resulting B_{new} will be

$$B_{new} = \begin{pmatrix} X & & X \\ & X & X \\ X & X & X \end{pmatrix}$$

and the corresponding graph will be as shown in Figure 2-8, which is simply a relabeling of the graph in Figure 2-5.

Storage Techniques: Typical storage mechanisms for dense matrices require $O(n^2)$ memory. The number of nonzeroes in a sparse matrix (referred to as τ) will frequently be several orders of magnitude less than n^2. Thus, more efficient storage mechanisms are required for simple efficiency as well as to make the algorithm being implemented tenable. Furthermore, due to phenomena such as fill-in during Gauss Elimination, it will frequently be necessary to alter the storage structure during processing. Thus, a static structure will not be suitable unless it preallocates sufficient room for growth. The alternative is a dynamic structure that can be easily altered during processing. While both static and dynamic storage techniques for sparse matrices will be addressed, the storage of sparse vectors is discussed first as a necessary and illustrative

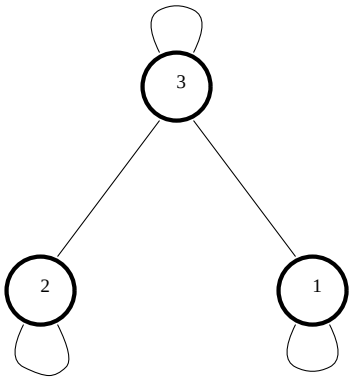

Figure 2-8. Graph After Symmetric Permutation

prerequisite. Much of this summary will follow that provided in Duff, Reid, and Erisman's text on sparse matrices [48].

A *full* storage scheme for a sparse vector requires $O(n)$ memory but has the advantage that individual vector components can be directly indexed. A more space efficient storage technique would be to hold each vector as a set of (value, index) pairs with one pair for each nonzero component in the vector. These pairs may be ordered by the index values or left unordered. The set of (value, index) pairs is typically called the *packed* storage form. A transformation from the full to the packed storage forms is called a *gather* operation. The reciprocating transformation from packed to full form is called a *scatter* operation. Some contemporary computers with extensive vector facilities, such as the Cray X-MP and the Fujitsu FACOM VP, provide hardware support for these operations.

Typical mathematical operations on vectors include dot products and the saxpy operation. The *saxpy* operation is defined as

$$z = \alpha x + y$$

where x, y, and z are n-vectors and α is a scalar. The name saxpy comes from the software mnemonic for "scalar alpha x plus y" [78]. As the packed forms of sparse vectors are typically unordered (done for efficiency), operations on these vectors will frequently follow the outline provided below:

- Scatter the nonzero entries of one of the vectors into a full length vector known to be initially zeroed.
- For each operation:
    1. Revise the full length vector by applying the operation to it (can be done by a traversal of the other vector operand in packed form for vector-vector operations).
    2. If an operation changes a zero to a nonzero, then update the corresponding packed structure index list.

- Gather the nonzeros out of the full length vector and back into packed form by using the (potentially updated) index list of the packed form and in the process zero out the full length vector entries used (facilitates later reuse of this vector).

When a vector facility is available, there is a greater tradeoff in performance between the full and packed formats. Especially if the vector facility does not accommodate indirect indexing, the storage savings from a packed format will result in a significant performance disadvantage by precluding the effective use of the vector hardware.

Moving on to sparse matrix storage techniques, there are four primary mechanisms that are coordinate schemes, collection of sparse vector schemes, linked lists, and clique schemes. Each mechanism has unique advantages and disadvantages, and a particular algorithm may use a number of different mechanisms throughout the different phases of its operation.

The coordinate schemes maintain each nonzero in the matrix as a distinct triple (a_{ij}, i, j). This technique is typically used as the general input and output format for a general purpose algorithm. While this format is difficult to work with internally since it cannot be efficiently accessed by either row or column, it provides an efficient and simple interface format. Contributing to its efficiency as an input format is the fact that it can be sorted into either the collection of sparse vectors or linked list formats in time that is $O(n) + O(\tau)$ where τ is the number of nonzeros in the matrix. The sorting technique is based on the bucket sort using either the row or column indices. Furthermore, a column ordered row oriented result from the sort is possible by first sorting by columns and using that result to sort by rows.

The collection of sparse vectors method uses a size n array of (pointer, count) pairs that index into the rows or columns that are held as sparse vectors. These sparse vectors are held in unordered, yet contiguous format. Such a static format is frequently allocated with extra space to accommodate growth. When the extra space has been exhausted, but prior entries have been freed, a compaction operation may be done to reclaim free entries. The advantages of this method are that row or column accesses are efficient depending on the orientation of the structure and that no memory is needed for links between vector entries. Disadvantages are primarily with the growth restrictions imposed by the static allocation.

The linked list representation offers a dynamic structure similar to the prior collection of sparse vectors. However, as each vector (row or column) is held as a linked list, it can easily be expanded or contracted. Furthermore, it is also now possible to easily maintain row vectors in column order (or column vectors in row order). While access to these structures is usually simple and short, there are some disadvantages. For one, it may be necessary to maintain doubly linked lists to accommodate insertions and deletions (unless access methods are limited to means that preserve pointers to prior list entries). Also, the linked list structures will exhibit less spatial locality than the collection of sparse vectors method. This could adversely affect performance if a hierarchical memory is in use.

A common drawback of both the linked list and collection of sparse vectors techniques is that they are either row or column oriented but not both. For operations

such as Gauss Elimination, both row and column accesses may be necessary, and thus one of these two access types will be inefficient. This problem can be remedied by developing a second structure that is oriented opposite to the first (i.e., if the primary structure is row oriented, the secondary structure would be column oriented). Furthermore, this secondary structure may only need to hold the structural components of the matrix (i.e., the row/column indices) so full duplication would not be necessary. Another possibility using linked lists would be to intermingle row- and column-oriented linked lists between the nonzero elements.

The final storage method is the clique scheme, which is derived from finite element models. In this scheme, the matrix, A, can be represented as the sum of a number of matrices, $A^{[k]}$, that each have only a few rows/columns with nonzero entries:

$$A = \Sigma_k A^{[k]}.$$

These $A^{[k]}$ matrices correspond to fully connected subgraphs within the graph for the matrix, which is from where the term clique is derived. By way of example, the matrix shown in Figure 2-9 can be represented as the sum of the three cliques that are obtained by using the following row/column patterns: $\{1,2,5\}$, $\{2,3\}$, and $\{3,4\}$.

$$A = \begin{pmatrix} X & X & & & X \\ X & X & X & & X \\ & X & X & X & \\ & & X & X & \\ X & X & & & X \end{pmatrix}$$

Figure 2-9. Clique Storage Matrix Example

While this method was originally developed for finite element problems that are directly derived as the sum of component matrices, this method is more widely applicable as any symmetric pattern matrix can be *disassembled* into a sum of cliques.

In order to improve the structure or maintain numerical stability, it is often necessary to perform permutations to the original matrix. Such permutations are algebraically represented as full matrices that are developed by interchanging rows or columns of the identity matrix. However, storage of the full matrix for a permutation is not required. A single n length vector can be used to represent any permutation and variations exist that allow the permutation to be performed in place for only an $O(n)$ additional memory requirement for the permutation.

Reduction to Block Triangular Form: Frequently the matrix A can be permuted into a block triangular form as shown in Figure 2-10 (a lower triangular version is also possible). This form of the matrix is significant as it can reduce the amount of computations required for factorizing and divides the overall problem into a number of independent subproblems. In particular, only the B_{ii} blocks need to be factorized. A back substitution process can then be used with a set of simple matrix-vector multiplications to evaluate the contributions of the off diagonal blocks [48]. The

equation,

$$A\bar{x} = \bar{b}$$

thus becomes

$$PAQ\bar{y} = P\bar{b}$$

where

$$\bar{x} = Q\bar{y}.$$

The back substitution then proceeds using

$$B_{ii}\bar{y}_i = (Pb)_i - \Sigma_{j=i+1}^{N} B_{ij}\bar{y}_j,\ i = N, N-1, .., 1$$

(where N is the number of diagonal blocks) followed by

$$\bar{x} = Q\bar{y}.$$

Notice with this technique fill-in is limited to the B_{ii} submatrices and row or column interchanges within these submatrices do not affect the rest of the structure. Furthermore, reduction to block triangular form can be done in $O(n) + O(\tau)$ time which is often more than offset by the savings in storage and computations of block triangular form.

$$PAQ = \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ & B_{22} & B_{23} & B_{24} \\ & & B_{33} & B_{34} \\ & & & B_{44} \end{pmatrix}$$

Figure 2-10. Block Triangular Form Matrix

As a final set of notes on block triangular form, it has been proved that the block triangular form for a nonsingular matrix is essentially unique (to the extent that other equivalent forms are merely a reordering of the diagonal blocks). Also, empirical studies have illustrated a high correlation between asymmetric structure and block triangular reducibility.

Once the block triangular form has been found, it can be exploited for the subsequent factorization that begins with an analysis of the sparsity structure in the case of a sparse matrix.

Analyze Phase: Not present in dense matrix factorization algorithms, the analyze phase of a sparse matrix algorithm determines how the sparsity structure of the matrix can be exploited to reduce the computations necessary for factorization. This phase manifests itself as the selection of a pivot ordering and a symbolic factorization to determine the pattern of the LU factors. The primary consideration in this phase is to minimize the amount of fill-in that occurs as factorization progresses. This in itself is a very difficult problem. In fact, a minimum fill-in pivot selection is NP-complete in the restricted case where the diagonal consists of all nonzero entries and all pivots are chosen from the diagonal [141]. However, there are also other

considerations. They include the selection of pivots to preserve numerical stability, as well as, parallel and vector processing considerations. Furthermore, the selection of a minimum fill-in pivot sequence may be so costly as to overshadow any potential savings in the factorization. As a result, a number of heuristic techniques have been developed for the analyze phase. When the matrix has an unsymmetric pattern, the techniques most frequently are based on the Markowitz Criterion.

Markowitz Criterion: When selecting the kth pivot for the factorizing of the $A^{(k)}$ active submatrix, the Markowitz Criterion chooses the $a_{ij}^{(k)}$ entry that is numerically acceptable and minimizes the product of the corresponding row and column degrees [106]. The *row degree* for an entry is simply the number of nonzeroes currently in the entry's row of the active submatrix. Likewise, the *column degree* is the number of nonzeroes in the entry's column of the active submatrix. These degrees shrink with the active submatrix as the effects of earlier pivot selections are taken into account, but they may also grow due to fill-in. The idea is that this strategy will modify the least number of coefficients in the remaining submatrix. Drawbacks of this approach are that it requires maintaining updated row and column degrees for each step of the factorization and that both row and column access to the matrix pattern is required.

When implementing the Markowitz Criterion, a typical approach is to order the current row and column degrees in an increasing order with all equal values in a single doubly linked list which is indexed by a pointer array. The search then commences in increasing order until an entry is found that meets the pivot criteria. Additional searches are done until it can be assured that no better entry can be found. It is also possible to break ties based on the entry with the best numerical characteristics. Importantly, the row and column degrees should only be calculated once at the beginning and then implemented to facilitate expedient incremental updates.

2.4 Multifrontal Methods

The multifrontal approach to sparse matrix factorization is based on a decomposition of the matrix into a sum of smaller dense submatrices. Factorization of these smaller matrices (called *elements* or *frontal matrices*) can be done using dense matrix operations that do not require the indirect addressing required of conventional sparse matrix operations. The multifrontal approach was first developed by Duff and Reid for symmetric, indefinite matrices [54] and then extended to unsymmetric matrices [55]. However, the multifrontal approach has been used most extensively for the Cholesky factorization of symmetric, positive-definite matrices [70, 65, 67]. Most recently, Davis and Duff have generalized the method to take advantage of unsymmetric-pattern matrices [34, 35].

One of the major advantages of a multifrontal method is that the regularity found in the dense matrix operations can be used to take advantage of advanced architectural features such as vector processors, hierarchical memories, multiprocessors with shared memories or distributed memories connected by regular pattern communication networks [5, 7]. Furthermore, when frontal matrices do not overlap in their pivot rows and columns, the factorization steps associated with those pivots can be done concurrently [46]. This provides an additional degree of parallelism that can be

exploited by multiprocessors. The combination of these two sources of parallelism, within dense frontal matrix operations and between frontal matrices, has shown to provide a significant amount of potential speed-up in several studies [83, 51, 121, 122].

Multifrontal methods for sparse matrices were originally developed for matrices resulting from finite element problems where the coefficient matrix is the sum of a number of dense symmetric matrices that correspond to the elements of the original model (hence the use of *element* to describe the frontal matrices). As rows and columns of the aggregate matrix are assembled from all the contributing elements, the factorization associated with that row/column pivot can be performed even though other rows and columns have not yet been fully assembled. Later, these techniques were extended to other matrices by artificially decomposing the matrix into a sum of frontal matrices. With finite element and other symmetric-pattern matrices, the frontal matrices are square. Furthermore, an assembly graph can be built with the frontal matrices as nodes using edges to describe overlaps between frontal matrices. The assembly graphs associated with symmetric matrices always form tree structures. Such a structure simplifies both the task mapping and scheduling problems when employing a multiprocessors on the method.

Recent efforts by Davis and Duff have focused on extension of the method to unsymmetric-pattern matrices [34, 35]. With such matrices, the frontal matrices are no longer guaranteed to be square and are typically rectangular. Furthermore, the assembly graph is no longer a tree. Instead, it becomes a directed acyclic graph (DAG). The rest of this section describes the multifrontal method applied to unsymmetric-pattern matrices.

Frontal Formation: A central issue of the multifrontal method is the decomposition of the matrix into frontal matrices. As the amount of computation is proportional to the size of the frontal matrix and as distinct memory is typically allocated for each frontal matrix, the primary objective is to minimize the size of the frontal matrices. Moreover, as the amount of fill-in is typically proportional to the size of the frontal matix, minimizing frontal matrix size will also tend to minimize fill-in. This is especially important in the earlier stages of the algorithm, as early fill-in will likely propagate into larger memory and computational requirements for the later frontal matrices.

The approach taken by Davis & Duff to minimize frontal size is to base frontal matrix determination on selection of a pivot with minimal Markowitz cost. In order to preserve numerical stability, threshold pivoting is often added to the pivot selection criteria.

Criteria stronger than the Markowitz cost, such as minimum actual fill-in, are typically too costly computationally and provide little additional benefit. As Markowitz cost requires that updated row and column degrees be maintained, approximate degrees are frequently used.

Frontal Factorization: Once a frontal matrix has been determined, the factorization associated with its pivot will be done. This involves the updating of the corresponding row in the U factor via a copy operation, the computation (via a divide by the pivot value) and copying of the pivot column into the corresponding

column of L, and the updating of the remaining frontal matrix entries. This updating process is referred to as the *Schur Complement* where each entry is computed as follows:

$$a_{ij}^{new} = a_{ij} - (a_{ik}/a_{kk})a_{kj}$$

which is equivalent to:

$$a_{ij}^{new} = a_{ij} - l_{ik}u_{kj}.$$

The $-l_{ik}u_{kj}$ contribution to the a_{ij} entry need not be applied immediately and, in fact, distinct updates to the same entry can be applied in any order. Hence only the pivot row and column are needed for the frontal matrix. The update terms can be calculated and saved until their row/column becomes pivotal or they can be applied to the entry earlier if convenient. This can significantly reduce the required amount of data sharing.

Assembly: The process of applying updates to an entry is called *assembly*. Consider the following example of Figures 2-11, 2-12, and 2-13 where each P_i represents a pivot, letters are nonzeros, and $\cdot$ are zeros.

$$A = \begin{pmatrix} P_1 & \cdot & a & b & \cdot \\ c & P_2 & d & \cdot & e \\ f & g & \cdot & \cdot & \cdot \\ \cdot & h & i & j & \cdot \\ k & \cdot & \cdot & \cdot & l \end{pmatrix}$$

Figure 2-11. Sample Sparse Matrix

The first frontal matrix of A in Figure 2-11 is E_1 and consists of rows 1, 2, 3, and 5 together with columns 1, 3, and 4 as shown in Figure 2-12.

$$E_1 = \begin{pmatrix} P_1 & a & b \\ c & d & \cdot \\ f & \cdot & \cdot \\ k & \cdot & \cdot \end{pmatrix}.$$

Figure 2-12. Frontal Matrix for First Pivot

With E_1, fill-in occurs in entries (2,4), (3,3), (3,4), (5,3), and (5,4). Furthermore, entry $a_{2,3}$ is updated to $\bar{d}$ and must be assembled into E_2 (shown in Figure 2-13) as it is in the pivot row of E_2. Furthermore, the fill-in occurring in entries (2,4) and (3,4) of E_1 will cause E_2 to be expanded by this column.

While the contributions to entries $a_{2,3}$ and $a_{2,4}$ are assembled into E_2, it would also be convenient to assemble E_1's contributions to entries $a_{3,3}$ and $a_{3,4}$. However, none of the other entries in the Schur Complement of E_1 can be assembled into E_2 and thus E_1 must be held until the rest of its contributions can be assembled into later frontal matrices. The inability for E_2 to accommodate all of E_1's contributions is referred to as *partial assembly* and only occurs in the unsymmetric-pattern version

$$E_2 = \begin{pmatrix} P_2 & \bar{d} & e \\ g & \cdot & \cdot \\ h & i & \cdot \end{pmatrix}$$

Figure 2-13. Frontal Matrix for Second Pivot

of the method. In the symmetric-pattern version all contributions can be passed together to a single subsequent frontal matrix.

Assembly DAG Edges: Whenever a data dependency exists between frontal matrices, an edge in the assembly DAG is used to represent the dependency. In the previous example of E_1 and E_2, there would be a directed edge (E_1, E_2). Furthermore, as the row pattern of E_1 contains the pivot row of E_2 and the column pattern of E_1 does not contain the pivot row of E_2, the relationship indicated by the edge (E_1, E_2) is referred to as an L relationship where E_1 is an *L child* of E_2 and E_2 is an *L parent* of E_1. The four matrix patterns of Figure 2-14 illustrate the four possible relationships between the frontal matrices that are induced by pivots P_1 and P_2.

$$\text{No relationship} \begin{pmatrix} P_1 & \cdot & \cdots \\ \cdot & P_2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \quad \text{L relationship} \begin{pmatrix} P_1 & \cdot & \cdots \\ X & P_2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$\text{U relationship} \begin{pmatrix} P_1 & X & \cdots \\ \cdot & P_2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \quad \text{LU relationship} \begin{pmatrix} P_1 & X & \cdots \\ X & P_2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

Figure 2-14. Possible Relationships between Frontal Matrices

The edges in the assembly DAG show not only data dependencies but also control dependencies. Transitive reductions in the DAG are possible to more efficiently describe the precedence relationships and to reduce the data passing.

Pivot Amalgamation: Up to this point in the discussion each frontal matrix has been built using a single pivot and updated via a single factorization step. When additional pivots are identified that have identical (or nearly identical) row and column patterns, the pivots can be amalgamated together into a single frontal matrix. This allows several pivots' contributions to be calculated and combined. In particular, consider the generalized example below of an $r \times c$ frontal matrix where F is the $k \times k$ pivot block that includes the k pivots on its diagonal. The matrix F will be factored into the product $L_1 U_1$ in the frontal matrix's factored form. C^T is a $k \times (c-k)$ submatrix that will be updated to become part of the U factor (specifically U_2 in the frontal's factored form). Likewise, L_2 will be formed from B which is a $(r-k) \times k$ submatrix. Finally, the remaining $(r-k) \times (c-k)$ submatrix D will be updated by

the Schur complement of the factorization into D'.

$$\begin{pmatrix} F & C^T \\ B & D \end{pmatrix} \Rightarrow \begin{pmatrix} L_1 \backslash U_1 & U_2 \\ L_2 & D' \end{pmatrix}$$

The update to D is formed as:

$$D' = D - (L_2 U_2)$$

which is a rank k update to D. In the single pivot case this is just a rank 1 or outer product update commonly found in BLAS 2 (Basic Linear Algebra Subroutines Level 2) routines. However, in this more general case, the update is a rank k update that can be implemented via matrix multiplication in BLAS 3 routines. These higher level BLAS routines have a richer computational structure that can be used to exploit the advanced architectural features of a particular system. Hence, amalgamation can significantly improve the speed of frontal matrix numerical factorizations.

As eluded to earlier, the row and column patterns of two pivots do not need to be identical for amalgamation to be applied. If they are nearly identical, amalgamation can still take place at the cost of a slightly larger frontal matrix with additional fill-in.

Separate Symbolic and Numerical Factorizations: When multiple matrices of identical nonzero structure are to be factored, it is possible to separate the symbolic and numerical factorization phases. When this is done the results of a single symbolic factorization of the matrices' pattern can be performed followed by a numerical factorization for each distinct matrix. The symbolic factorization will determine the pivot ordering, develop the assembly DAG, and determine the nonzero structures of the LU factors. (Symbolic factorization is essentially the same operation as was previously referred to as the analyze phase). The subsequent numerical factorizations will then make use of this information for the actual numerical factorizations.

Loss of Anticipated Pivots: An important and difficult problem exists however with the separation of symbolic and numerical factorizations. Specifically, the symbolic factorization will not be able to make use of numerical values for the selection of pivots. Hence, specified pivots in the ordering may become numerically unacceptable. When such an occurrence is detected during a numerical factorization, special recourses are necessary to recover. An example of this situation can be seen in Figure 2-15.

$$A_1 = \begin{pmatrix} 2 & 1 & 0 & 4 \\ 1 & 4 & 1 & 0 \\ 0 & 3 & 2 & 0 \\ 1 & 2 & 0 & 4 \end{pmatrix} \quad A_i = \begin{pmatrix} 2 & 8 & 0 & 4 \\ 1 & 4 & 1 & 0 \\ 0 & 3 & 2 & 0 \\ 1 & 2 & 0 & 4 \end{pmatrix}$$

Figure 2-15. Lost Pivot Example

In matrix A_1, the $a_{2,2}$ entry is a perfectly acceptable second pivot. However, as the values change in matrix A_i, the update to the $a_{2,2}$ entry, accomplished via:

$$a_{2,2}^{new} = a_{2,2}^{old} - (a_{2,1}/a_{1,1}) * a_{1,2}$$

causes $a_{2,2}$ to become zero and no longer acceptable as a pivot. Furthermore, a pivot's numerical value need only be altered to drop it below the threshold pivoting limits in place for it to become no longer acceptable. Also interesting to note is that the actual pivot entry did not need to change for it to fail to be acceptable. Other entries may affect the anticipated pivot's value during previous steps of elimination, as was the case in the example provided.

Davis and Duff suggest two possible approaches to dealing with the loss of anticipated pivots [34, 35]. They are:

- Force the amalgamation of the lost pivot with subsequent frontal matrices during the numerical factorization. This creates a larger pivot block from which permutations can be used to select alternative pivots. The drawback is that the row and column patterns of the amalgamated frontals could be quite different and result in catastrophic fill-in and loss of inter-frontal parallelism.

- Amalgamate the frontal matrix with the lost pivot with its first LU parent (assuming one exists). This will limit additional fill-in, however, such an LU parent is not guaranteed to exist in which case forced amalgamation will be necessary.

While the possibility of lost pivots complicates the factorization of sequences of identically structured sparse matrices, the unsymmetric-pattern method with separated analyze and factorize phases does provide significant potential benefits for dealing with such sequences. Yet, the question of where such sequences arise is still open. The next section provides general mathematical settings in which such sequences occur and provides a listing of sample application areas.

2.5 Factorization Sequences of Matrices

The factorization of a sequence of sparse matrices that maintain an identical nonzero structure occurs frequently in a variety of applications. The principal categories of such applications are those that use a Newton method-based approach to solving a system of nonlinear algebraic equations and those that require the solution to a system of ordinary differential equations by an implicit method. Both of these categories will be discussed in this section.

Systems of Nonlinear Algebraic Equations: Newton's method is a very common technique in numerical analysis that is used to find the zeros of a function by an approximation of the function that is based on a first order Taylor's series expansion [13]. For one dimensional real valued functions, the first order Taylor's series expansion is

$$f(x) = f(x_0) + f'(x_0)(x - x_0).$$

When solving for α such that $f(\alpha) = 0$ we replace x with an approximation x_1 of α and assume $f(x_1) = 0$. The result is

$$0 = f(x_0) + f'(x_0)(x_1 - x_0)$$

which can be solved for x_1 and generalized into the following iteration formula by replacing x_1 with x_{k+1} and x_0 with x_k:

$$x_{k+1} = x_k - f(x_k)/f'(x_k).$$

The method can be generalized from strictly real (or complex) functions to any mapping between Banach spaces by using the Frechet derivative [93]. A *Banach space* is a normed vector space that is complete (that is, all Cauchy sequences will converge to an element in the space). The *Frechet derivative* is a bounded linear operator L that satisfies:

$$L(x_0) = \lim_{y \to 0} \frac{\|f(x_0 + y) - f(x_0) - L\|}{\|y\|}$$

where f is a function mapping one Banach space to another and the $\|\cdot\|$'s are norms appropriate to the given Banach spaces.

Systems of nonlinear algebraic equations can be considered to be a mapping from an n-dimensional Euclidean space (R^n), which is a Banach space, back to itself defined by:

$$f(\bar{x}) = \begin{pmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{pmatrix}$$

for f_i mapping R^n to R^1. The Frechet derivative of such a mapping is simply the Jacobian of the function (denoted ∇f) which is an $n \times n$ matrix of partial derivatives where the ith row, jth column of the Jacobian (evaluated at a given $\bar{x}_0$) is

$$\frac{\partial f_i(\bar{x}_0)}{\partial x_j}.$$

The Newton iteration formula thus becomes

$$\bar{x}_{k+1} = \bar{x}_k - \nabla f(\bar{x}_k)^{-1} f(\bar{x}_k)$$

within this context. The problem with this formula is the requirement to compute the inverse of the Jacobian. In order to avoid this costly (and sparsity destroying) process we define

$$\Delta \bar{x}_k = \bar{x}_{k+1} - \bar{x}_k$$

and then manipulate the formula into

$$\nabla f(\bar{x}_k) \Delta \bar{x}_k = -f(\bar{x}_k)$$

and determine $\bar{x}_{k+1} = \bar{x}_k + \Delta\bar{x}_k$. As the Jacobian $\nabla f(\bar{x}_k)$ is a linear operator and $-f(\bar{x})$ is a vector, the computation now consists of simply solving a linear system of equations. The focus of this work is on instances where the Jacobian is sparse which is typical of most large systems. The nonzero entries of the Jacobian can change values with the various $\bar{x}_k$'s but the nonzero structure will remain constant.

One application area of such nonlinear algebraic equations is chemical engineering. Specifically, Zitney discusses the solution of such systems for distillation flowsheeting problems where the Jacobians are of an unsymmetric-pattern and fail to have favorable numerical properities such as diagonal dominance [144, 143].

Systems of Ordinary Differential Equations: Another major type of problems where the factorization of sequences of identically structured matrices is required is the solving of systems of ordinary differential equations using an implicit method. The predominant type of numerical methods for such tasks when initial values are provided are linear single- and multi-step techniques due mainly to their ease of implementation and analysis [107]. These techniques divide the independent axis into steps or mesh points across which the differential equations are integrated via a discrete summation method. A generic form of the problem would be:

$$\dot{x} = f(x)$$

with x a vector function of time (t), $\dot{x}$ the first time derivative of x, and the initial condition is $x(0) = x_0$. A typical linear single step technique would use

$$x_{n+1} + \alpha_1 x_n = h[\beta_0 f(x_{n+1}) + \beta_1 f(x_n)]$$

to solve for x_{n+1} where x_i denotes $x(t_i)$, t_i the time at time step i, h the time step size, and α_1, β_0, and β_1 constants. If $\beta_0 = 0$ then the method is *explicit*, that is x_{n+1} can be solved for directly. If $\beta_0 \neq 0$ then the method is *implicit* and an iterative method is required to solve for x_{n+1}.

Linear multi-step techniques are similar but use more of the previously predicted values x_i and function evaluations at those x_i's for the next prediction. In particular, they take the general form:

$$x_{n+1} + \Sigma_{i=1}^{p} \alpha_i x_{n-i+1} = h\beta_0 f(x_{n+1}) + h\Sigma_{i=1}^{p} \beta_i f(x_{n-i+1})$$

where $\beta_0 = 0$ again means an explicit method and $\beta_0 \neq 0$ means an implicit method. Specific methods vary in their value for p and the weights given by the α_i's and β_i's. Such methods include Adams-Bashford (explicit), Adams-Moulton (implicit), Nystrom, Newton-Cates, and the Backward Differentiation Formula [107, 124, 90].

For reasons of stability, implicit methods are preferred for "stiff" (that is, ill-conditioned) problems. In particular, explicit linear multi-step methods cannot meet A-stable criteria and can only meet A(α)-stable criteria if $p \geq 3$ [107]. These stability conditions relate the effectiveness of a method to the conditioning of the problem as evidenced by the eigenvalues of the corresponding matrix. Hence, unless only "non-stiff" (well-conditioned) problems are to be solved by a linear multi-step method, the

method should be implicit. (Other classes of alternative nonlinear methods will be discussed briefly later).

When using implicit methods, the linear multi-step method given generically above can be reorganized to accumulate only terms dependent upon x_{n+1} onto the left side as follows:

$$(I - h\beta_0 f)x_{n+1} = \Sigma_{i=1}^{p} \left[h\beta_i f(x_{n-i+1}) - \alpha_i x_{n-i+1}\right].$$

As the right hand side is constant within a time step, we can define it equal to b. Hence, our task within a time step becomes finding the solution x in

$$F(x) = b \text{ where } x \text{ replaces } x_{n+1} \text{ and } F(x) = (I - h\beta_0 f)(x).$$

For f defining a linear system of ODEs, this simply requires the solution of a linear system of equations. However, when the system is stiff, the coefficient matrix corresponding to F is ill-conditioned and direct solution methods can be too inaccurate. Thus Newton's method can be used to solve for the zero of the modified function:

$$G(x) = F(x) - b = 0.$$

This has proven useful in practice due to the quadratic convergence of Newton's method [107, 90]. The same approach is used when f is a nonlinear function (representing a system of nonlinear ODEs).

When Newton's method is applied within each time step as above, each Newton iteration requires solving a linearized system (as discussed in the previous subsection on systems of nonlinear algebraic equations). Specifically, the equation:

$$\nabla G(x_{(i)})\Delta x_{(i)} = -G(x_{(i)})$$

must be solved for $\Delta x_{(i)}$ where $x_{(i)}$ is the ith approximation to x_{n+1}, ∇G is the Jacobian of G, and $x_{(i+1)} = x_{(i)} + \Delta x_{(i)}$. The structure of ∇G remains constant both within and between time steps thus producing a sequence of matrices with identical structure that must be factorized and solved.

Other methods also exist and are particularly useful for solving stiff systems. These include nonlinear methods such as Certaine's method, Jain's method and the class of Runge-Kutta techniques [107]. Certaine's method is also an example of another class called *predictor/corrector* methods where an explicit technique is used to predict a first approximation to x_{n+1} and then an implicit method is used to correct (refine) the approximation. Furthermore, there are both implicit and explicit versions of the numerous Runge-Kutta techniques. Whenever implicit techniques are used, the factorization of a sequence of matrices is a likely subtask. However, with some of the techniques these matrices take on an already factored triangular or even diagonal structure. Yet, the bottom line is that the factorization of a sequence of general sparse identically structured matrices is a common subproblem to solving systems of ODEs via an implicit method. Furthermore, these systems are frequently encountered in applications that include nuclear magnetic resonance spectroscopy, computational

chemistry, and computational biology [147]. More specifically, Miranker provides examples of systems of nonlinear ODEs with the circuit simulation of tunnel diodes commonly used in high speed circuits, thermal decomposition of ozone, and the behavior of a catalytic fluidized bed [107]. Electronic circuit simulations are another common source of large systems of nonlinear, first order ODEs as evidenced in the modified nodal analysis approach used by Saleh and Kundert [131, 97] and the sparse tableau formulation of Hachtel [82].

2.6 Parallel Matrix Computations

A major emphasis of this research effort is to perform the numerical factorization of sequences of sparse matrices in parallel. Furthermore, the unsymmetric-pattern multifrontal approach provides dense submatrices which may also be factorized in parallel. This section summarizes some previous efforts that explored distributed memory implementations of both dense and sparse matrix factorizations.

Distributed Memory Factorization: In distributed memory environments, algorithms are heavily dependent upon the storage allocation scheme for the matrix. The two most common schemes are row oriented [22] and column oriented [64, 40, 27, 76]. Blocking schemes of rows or columns are frequently used and Dongarra and Ostrouchov [40] discuss such methods together with the need for adaptive blocking mechanisms. Choi, Dongarra, Pozo, and Walker explore the various data allocation schemes and associated algorithms and develop a generalized *block scattered* approach with data blocked and blocks scattered (wrapped) to processors [21]. By varying the parameters of this method both pure blocked and pure scattered (together with inbetween formats) can be achieved. A block-cyclic strategy has recently been proposed by Lichtenstein and Johnsson for dense linear algebra on distributed memory multiprocessors [100]. The block-cyclic strategy is applied to both rows and columns and is effectively applied to LU factorization and the subsequent triangular solves.

Furthermore, pivoting, if required, can be done in a row or column fashion. When partial pivoting is used, Geist and Heath [62] recommend inclusion of pipelining to offset the cost of pivoting. For efficiency of pivot determination, row pivoting is preferred with column storage and column pivoting with row storage. A simple illustrative example of a basic (non-pipelined) algorithm (taken from Gallivan, Plemmons, and Sameh [59]) is shown in Figure 2-16. It uses column oriented storage with a row pivoting scheme.

Sparse Matrix Computations: While considerable attention has been paid to implementation of dense matrix operations on parallel architectures, less has been given to sparse matrix operations and there exists significant open research to be accomplished in this area. Major reasons for delaying such attention are that sparse matrix algorithms are more complex, use more sophisticated data structures, require irregular memory referencing, and are thus much more difficult to efficiently implement.

Ironically, operations for sparse matrices will typically have a greater degree of potential parallelism than their dense counterparts. This additional parallelism comes in the form of large grain parallelism that may be found in the structure of nonzeros in sparse matrices. The available parallelism in sparse matrix factorization has been

Each processor executes the following:

```
do k = 0, n − 1
      if ( column k is local ) then
            determine pivot row
            interchange row k with pivot row
            do i = k + 1, n − 1
                  compute multipliers λ_ik = a_ik / a_kk
            enddo
            broadcast the column just completed with pivot index
      else
            receive the column just computed with pivot index
            interchange row k with pivot row
      endif
      for (all columns j > k that are local ) do
            do i = k + 1, n − 1
                  a_ij = a_ij − λ_ik a_kj
            enddo
      endfor
enddo.
```

Figure 2-16. Basic Column Fan-Out LU Factorization

studied using a variety of models. Wing and Huang developed a very fine grain model with each divide/update operation on a single entry considered a task [140]. With this model (which restricts the factorization to a given pivot ordering), the maximal parallelism available can be realized. A large grain model was developed by Jess and Kees [91] that defines a task as the factorizing and complete corresponding update of a single pivot. Liu [102] establishes a third, medium grain model with the various column operations associated with a single pivot defined as tasks with one task per column. Ostrouchov, Heath, and Romine [117] investigate whether the rather disappointing results seen so far with Cholesky factorization are due to an inherent lack of parallelism in the problems or to limitations on contemporary hardware. They conclude that parallelism is available, but the relatively high cost of communication (relative to computation) has limited performance. Specifically, on currently available distributed memory multiprocessors, only 20 to 25 percent of potential speed-ups are realized when using a medium grain of parallelization.

A most important issue in exploiting available parallelism in a parallel sparse matrix factorization is that of task allocation.

Task Allocation: Task allocation requires the careful balance of two opposing objectives. The first objective is to always have as much useful activity as possible going on in parallel. This typically requires use of finer grains parallelism to achieve

more potential concurrency. The second objective is to have as little interaction as possible between concurrently executing processors. This frequently contradicts fine grain parallelism that tends to require frequent interaction. For very regular computations, interactions can be orchestrated to minimize idle time due to processors waiting for an interaction. However, with sparse matrix routines, this regularity is typically lost due to variations in the pattern of nonzeros. Thus, task allocation becomes more challenging.

In a distributed memory environment task allocation is especially challenging. Due to the higher cost of communication, dynamically balancing loads can be expensive. Hence, a static allocation is frequently used and tightly coupled to the data allocation scheme. For Cholesky factorization, early column oriented algorithms allocated data and column tasks to processors in a wrapped fashion proceeding in a bottom up manner from the elimination tree [63, 71]. This did well to address the load balancing issue but caused excessive communication. Later schemes allocated subcubes to subtrees in the elimination tree and effectively reduced communication [63, 71]. This worked well for the well balanced trees produced from nested dissection orderings but not for the potentially unbalanced trees of minimum degree orderings. A further refined allocation scheme was developed by Geist and Ng to deal with unbalanced trees in a way that addresses both communication elimination and load balancing [63].

Additional advantages are typically found when using contemporary architectures by organizing tasks to use fewer, but larger, messages. Hulbert and Zmijewski propose such a method that combine update contributions into a single message [88]. While some granularity is lost, there are fewer message setup delays which tend to dominate much of the communication cost in contemporary architectures.

Numerical Factorization: As much of the work in sparse matrix factorization comes in the numerical factorization phase and as this work is typically the most structured (due both to the nature of the work and the advantages obtained from the earlier phases), this phase has the most exploitable parallelism and has received the most research attention. However, sparse numerical factorization is still much more difficult than its dense counterpart. Most implementations exploit only large and medium grains of parallelism.

Within a distributed memory environment, it is important to exploit locality of the data. Using data local to the processor as much as possible avoids the communication and synchronization overheads imposed by message passing between processors.

Chu and George [22] discuss a row oriented *dense* LU factorization routine for distributed memory environments that could be extended in concept to sparse matrices. This technique uses a *fan-in* operation to determine the next pivot row followed by a *fan-out* operation on the selected pivot that facilitates the updating of the active submatrix. Another key feature is the dynamic load balancing used to accommodate the use of partial pivoting.

Using a column oriented allocation scheme, pivot selection is handled by a single processor (the one owning the current pivot column). The factored pivot column can then be broadcasted (or in the sparse matrix context perhaps multicasting is more

appropriate) to the processors holding the other columns of the active submatrix to allow updating. This is the *fan-out* approach that has been used in initial Cholesky factorization routines. The excessive communication overheads of these methods limited their performance.

Another class of methods (used for sparse Cholesky factorization) are the *fan-in* methods [9, 117, 88]. These methods accumulate contributions to the updating of the active submatrix and send fewer but larger messages [9].

An increasingly popular approach to sparse matrix factorization are the symmetric-pattern, *multifrontal* methods [105, 76, 51, 121, 122, 77]. Large grain parallelism is available in these methods via independent subtrees in the assembly trees. However, as subtrees combine and parallelism at that level decreases, a switch is made to exploit parallelism at a finer grain within the factorization of a particular frontal matrix. This is especially appropriate as the later frontal matrices are typically much larger. Within a distributed memory environment, a column oriented storage scheme can be used with frontal matrices allocated to subcubes and frontal matrix factorization done in a fan-out implementation of dense factorization. Assembly involves sending of contributions to appropriate follow-on processors where they are added to the appropriate entries. While these distributed multifrontal approaches typically require more communication than a fan-in approach, their performance has been quite promising for the Cholesky factorization of symmetric, positive definite matrices [76, 119, 120]. Specifically, speed-ups of 5.9 to 21.4 were achieved on a 32 processor Intel iPSC/2 using Cholesky factorization on matrices varying in order from 1,824 to 16,129 [119, 120]. Lucas has implemented a distributed memory multifrontal LU factorization routine that assumes numerically acceptable pivots and a symmetric-pattern. Speed-ups of up to 10.2 were obtained on 16 processors of an iPSC/2 using electronic device simulation matrices of order 7225. Recent simulation studies of similar methods for the LU factorization of (assumed-)symmetric matrices [121, 122] have shown a reasonable parallel potential in distributed memory environments. Here speed-ups of up to 30-40 were predicted for 128 node iPSC/2 and iPSC/860 hypercube topology multiprocessors. Comparison of these results with other simulated architectures reveal that lower communication to computation ratios produced better results (as one would expect). The exploitation of parallelism both in the elimination structure and within the distinct pivot steps has been found to be critical to the success of all of these methods [76, 51, 121]

Recently, fine grain parallelism has been investigated by Gilbert and Schreiber for a variety of sparse Cholesky methods on a SIMD distributed memory environment [77]. The multifrontal approach has demonstrated the greatest potential for exploiting this level of parallelism.

A critical subissue in any parallel sparse matrix factorization routine is how to efficiently schedule the component tasks. This issue will be discussed in the next section.

2.7 Multiprocessor Scheduling

The precedence constrained multiprocessor scheduling problem has been shown to be NP-Complete in both shared and distributed memory environments [137]. Thus,

heuristic techniques have been the emphasis of much of the research in this area. Within shared memory environments, a list scheduling scheme is typically used. Under such schemes, tasks are assigned priorities and when made ready (predecessors in the partial order have all completed) are put into a single priority queue for scheduling by next available processor [133]. The most common and effective priority schemes are based on a *critical path* analysis of the precedence DAG with the priority of a task defined as the heaviest weighted path from that task's node to an *exit* node (node with no successors) in the DAG [23]. Variations within these schemes primarily deal with tie breaking with a *Most Immediate Successors First* criteria both popular and effective [94].

Within the context of a distributed memory environment, communication delays (represented by edge weights in the task precedence graph) can become quite significant. Three methods for dealing with such *Task with Communications Scheduling (TCS)* problems are presented in this section.

ETF – Earliest Task First: The Earliest Task First (ETF) heuristic [89] is based on an extension of the Rayward-Smith computing model [125]. This extended model uses message counts for edge weights in the task graph together with a transmission cost function that defines the communication costs between any two processors. Different network topologies can be represented by varying this transmission cost function. Task computation times are represented as node weights.

Rayward-Smith proposed a list scheduling (LS) approach based on a greedy strategy for his original unit execution time/unit communication time (UET/UCT) model [125]. The idea is that no processor should be left idle if there is an available task to execute. Under the UET/UCT assumption, a task T can be scheduled on processor P_i at time t if T has no immediate predecessors, or each immediate predecessor has been scheduled to start on P_i at time $t-1$ or on P_j $(i \neq j)$ at time $t-2$. The length of the schedule produced by this greedy schedule w_g satisfies

$$w_g \leq (3 - (2/n)) * w_{opt} - (1 - (1/n))$$

where w_{opt} is the length of the optimal schedule and n is the number of processors.

Hwang, Chow, Anger, and Lee first propose the Extended List Scheduling (ELS) algorithm that applies the LS approach assuming no communication delays and then adds the necessary communication delays to the initially produced schedule. They show that the ELS method produces schedules that are bounded by

$$w_{ELS} \leq (2 - (1/n)w_{opt} + \tau_{max} * \Sigma_{i=1}^{alledges} edgeweight_i$$

where τ_{max} is the maximum transmission cost between two processors. While this seems like a loose bound, the authors show that specific problem instances will asymptotically achieve it. Thus, we see their motivation for the ETF method.

Due to its complexity, a number of formalisms are necessary to describe the ETF algorithm. For tasks T and T', let $\nu(T, T')$ be the number of messages sent on the edge (T, T') in the task graph. Let $\tau(P_i, P_j)$ be the transmission cost of communication between processors P_i and P_j. Then, define $s(T)$, $f(T)$, and $p(T)$ as the start time,

finish time, and processor assigned for task T by the scheduling algorithm. With these definitions, the time the last message for task T arrives at processor P is $r(T,P)$ which will be zero if T has no predecessors otherwise

$$r(T,P) = \max_{T' \in Pred(T)} \{f(T') + \nu(T',T) * \tau(p(T'),P)\}.$$

The earliest starting time for an available task T is then

$$e_s(T) = \max\{CM, \min_{P \in Available} \{r(T,P)\}\}$$

where CM is the "current moment" of scheduling time. The ETF uses a greedy strategy and finds the minimum earliest starting time (e_s^{min}) across all available tasks and available processors.

Due to arbitrary communication delays, if $e_s^{min} > CM$ it is possible that another processor may be freed between CM and e_s^{min} that would reduce the next earliest starting time. To handle this possibility, ETF uses a second time variable called NM for "next moment". NM is maintained as the next time at which a processor will become free. If $NM > e_s^{min}$ then no earlier starting time is possible and the schedule is set. Otherwise the scheduling decision is postponed.

If two tasks come up with the same scheduled start time on the same processor, the task chosen can be arbitrary or according to some priority list scheme. One such priority list scheme that is particularly attractive is that given by critical path analysis. However, the authors base their subsequent analysis of ETF on an arbitrary resolution mechanism.

Careful analysis of the ETF algorithm reveals its time complexity to be $O(nm^2)$ where n is the number of processors and m is the number of tasks. Furthermore, they develop an upper bound on the schedule lengths produced by ETF of

$$w_{ETF} \leq (2 - (1/n))w_{opt} + C_{max}$$

where C_{max} is the maximum communication requirement of any path through the task graph (assuming the τ_{max} transmission cost). This is a significant improvement over the bound produced by ELS. Furthermore, a tighter bound is even possible that reduces the C_{max} component.

The Earliest Task First (ETF) seems to be a very natural and effective extension to the shared memory environment list scheduling methods.

SCST – Scheduling with Communication Successors' Priority Tracking: Another algorithm that addresses the scheduling of task graphs with nonzero communication costs is due to Yin [142]. This method defines the combinatorial level of a particular task node as the heaviest node and edge weighted path from that node to any accessible leaf node (exit node). The node weights represent task execution times and the edge weights reflect communication costs. The run priority of a particular task is then defined as the combinatorial level of that task plus the heaviest edge from any immediate predecessor (if one exists) to that node. Figure 2-17 gives a sample

task graph with communication costs. The combinatorial levels for T_1, T_2, T_3, T_4, and T_5 are 15, 14, 7, 8, and 3 respectively. The run priorities given in the same order of tasks are 15, 14, 11, 12, and 6.

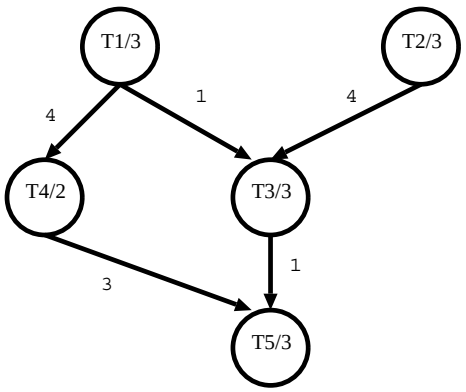

Figure 2-17. Task Graph with Communication Costs

The first objective is to execute the heaviest paths first. The second objective is to assign tasks to processors in a manner that minimizes the amount of necessary communication. The SSCT algorithm addresses these objectives by scheduling tasks from the ready queue with the highest run priority first (ties are broken using the dictionary ordering of the combinatorial level values of successor nodes). However, assignment of tasks to processors is accomplished to minimize the amount of communication necessary. That is, if task 4 is next to be scheduled, it should run on the same processor as task 1 so that the communication cost of 4 from task 1 to task 4 can be precluded (under the assumption that running on the same processor has zero communication overhead). Using this strategy, the SSCT algorithm produces the schedule shown in Figure 2-18.

Figure 2-18. Gantt Chart of Schedule with Communication Costs

Worst case complexity analysis of SSCT is $O(n^2)$ which reduces to $O(n)$ when the precedence graph is a tree. Simulations were used on 20 randomly generated task graphs with random edge weights to compare SSCT with an algorithm based on topological levels that disregards communication. In 70% of the cases SSCT was better and in 28% of the cases was SSCT worse.

MH – Mapping Heuristic: The Mapping Heuristic (MH) introduced by El-Rewini and Lewis takes both communication and contention into account when addressing the TCS problem [57]. The MH algorithm accepts a task graph description of the

parallel program and a description of the target machine topology and produces a Gantt chart that shows the allocation of tasks to processor elements and their execution time frames. El-Rewini and Lewis illustrate the method by first presenting it without considering contention and then separately presenting the extensions needed to account for contention in the communication links.

The MH algorithm proceeds in three steps:

1. A priority is assigned to each task in a fashion resembling the critical path method mentioned earlier. In particular, the heaviest weighted path from each task node to a leaf (exit node) in the task graph is determined using both execution times for node weights and communication time estimates for edge weights. (At this point communication time estimates assume a fully connected network with no contention). In the case of a priority tie, the task with the largest number of immediate successors is given the higher priority. If this does not break the tie, a random selection is used. An event list is initialized with tasks that have no predecessors and are thus initially "ready" for execution.

2. As long as the event list is not empty, entries are taken from the event list. These events will be either "task is ready" or "task is done".

   - For "task is ready" events, a processing element is selected such that the task cannot complete any sooner on any other processing element. This selection takes into account the machine topology and number of communication links that messages must be sent across prior to the initiation of the task's computation. The task is then scheduled on the selected processor and will signal a "task is done" event upon completion.
   - For "task is done" events, all immediate successors are notified and any successor that becomes "ready" as a result of the notification adds a "task is ready" event to the event list.

3. The second step is repeated until all of the tasks have been scheduled.

Notice that this algorithm does not treat communication cost uniformly. That is, the priority is assigned assuming a fully connected network but the actual task allocation is accomplished based on a specific (and typically less connected) topology. Thus multiple hops may be required that were not accounted for in the priority assignment. Furthermore, scheduling communicating tasks on the same processor assumes that communication will take zero time. This assumption is also not accounted for in the priority assignment.

The extension to the MH algorithm that considers contention is accomplished by maintaining a table of ongoing communication such that the best currently available communication path can be determined. Specifically, the tables record for each pair of processors the number of hops on the currently preferred path between these processors, the preferred first communication link, and the current delay associated with this path. Table maintenance is triggered by two additional event types for the event list. The "sending message" event causes the contention tables to be updated

to reflect the additional links made busy by the message. The "message received" event causes the tables to updated to reflect the links that are now free. Notice that this mechanism correlates to a virtual circuit type of communication. If the underlying communication is packet switching, the view presented by this method is too pessimistic. However, the authors opt against a lower level tracking mechanism for reasons of efficiency.

El-Rewini and Lewis ran a number of simulations of their MH algorithm and variants of it on both actual and randomly generated task graphs. Among their more interesting results is that priority assignments that neglect communication produced better schedules for "computation-intensive" task structures and priority assignments that consider communication produced better schedules for "communication-intensive" task structures. The delineation between "computation-intensive" and "communication-intensive" was a communication to computation ratio of one.

These three methods provide insight into how the task with communication scheduling problem might be addressed for scheduling an unsymmetric-pattern multifrontal method elimination DAG on a distributed memory multiprocessor.

Chapter Conclusion: This chapter established the necessary background for this research effort and summarized pertinent related efforts. To this end, the use of LU factorization to solve a system of linear equations was described and the issues of algorithm stability and error analysis addressed. Key sparse matrix concepts were then defined. Details of Davis and Duff's unsymmetric-pattern multifrontal method were provided as this entire research effort is based on this method. The focus on factorizing sequences of sparse matrices was then justified by presenting the mathematical formulation for two key types of problems that give rise to such sequences. Specific applications were also identified within each type of problem. Related efforts in parallel, distributed memory algorithms for both dense and sparse matrix factorizations were summarized with specific results providing performance goals for this effort. Finally, previous results in the key sub-topic of multiprocessor scheduling for distributed memory environments were discussed.

With this background established, the next chapter will highlight key hardware and software features of this effort's implementation platform.

CHAPTER 3
IMPLEMENTATION PLATFORM

A major thrust of this research effort will be to evaluate the performance of derived algorithms through experimental measurements. This will be accomplished on the nCUBE 2 distributed memory multiprocessor. The next two sections summarize key aspects of the nCUBE 2's hardware architecture and software environment. Information for these sections was obtained from the nCUBE 2 Systems Technical Overview [112], the nCUBE 2 Programmer's Guide [110], the nCUBE 2 Processor Manual [109], and the on-line *ndoc* documentation manager [113]. The third and final section describes an effort to experimentally characterize the performance of the nCUBE 2 multiprocessor.

3.1 Hardware Architecture

The nCUBE 2 supercomputer is a distributed memory, multiple instruction multiple data (MIMD) multiprocessor based on a binary hypercube interconnection topology. Configurations range from 8 to 8192 processors (3 to 13 dimension hypercubes). Each processor is a custom, 64 bit, VLSI (very large scale integration) processor with an integrated 64 bit floating point unit. Also integrated into the processor's design is an error correcting memory management unit with a 39 bit data path and full message routing circuitry. There are 14 bidirectional DMA (direct memory access) channels with 13 used for node interconnections and one for input/output. Processor node configurations contain from 4 to 64 Mbytes of memory. Processor options include both 20 MHz and 25 MHz versions of the custom processor. Performance of the 25 MHz processor is rated at 15 VAX MIPS, 4.1 Mflops (single precision), and 3.0 Mflops (double precision). Instruction execution times will vary however due to hits/misses in the 128 byte instruction buffer, data alignments in memory, and memory contention due to IO, refresh cycles, etc.

In addition to their use as node processors, additional custom 64 bit processor chips are also used in a separate, but connected, input/output array. These IO processors provide connections to disk arrays, tapes, graphics facilities, signal processors, and networks. The nChannel board connects up to 128 channels from the processor array to 16 serial channels. Each serial channel is controlled by a single IO processor. Eight of the serial channels are devoted to SCSI peripheral controllers that can handle up to seven SCSI peripherals. Other serial channels provide ethernet, analog/digital and digital/analog conversion, digital input and output, video, and tape interfaces. There is also an nCUBE HIPPI (High Performance Parallel Interface) board for interconnection to other supercomputers, UltraNet hubs, and mass storage devices. An nVision Real-Time Graphics Subsystem provides an integrated graphics and IO subsystem.

The processing capabilities are balanced by the very flexible, high performance hypercube interconnection network. Wormhole routing is employed in the network so that messages are not stored in intermediate nodes as they pass through the network. In the absence of path conflicts, messages pass directly from source to destination without delay so communication latencies are largely independent of path length in a practical sense. The primary manner in which longer paths will affect performance is the greater likelihood of path conflicts when the paths are longer. The Programmer's Guide suggests a simple performance model for messages less than 100 bytes [110]. The model anticipates a constant overhead for a message send or receive of 100 μsecs with a link delay time of 0 μsecs. (This simple model is refined experimentally later in the chapter). This model indicates a relatively low communication to computation ratio compared to other machines in the class [121] and is characteristic of a well balanced design.

In more detail, the nCUBE 2 uses an *e-code* default routing that progresses from least to most significant bit in the destination address. At each node on the path, the message is forwarded on the channel whose ID corresponds to the bit position of the least significant bit that differs between the destination node ID and the current node's ID. This scheme precludes the possibility of deadlocks if it is used exclusively.

While the default routing provides safe and reliable communication, it does not efficiently handle multiple destinations for a single message. Therefore, a broadcast/multicast facility is also available that will send a message to all nodes in the cube (or in a selected subcube) with a single transmission initiation and a single message fetch from memory. A mask is used to define the destination cube/subcube. If a bit in the mask is zero, the corresponding bit in the destination field must match for all receivers. If a mask bit is one, then the corresponding destination bit need not match and a fan-out duplication of the message occurs for each such bit setting. During such fan-out activities all necessary forwarding channels must be open or else the message will block. Hence, frequent point to point traffic on these channels can livelock the broadcast/multicast operation.

In particular, the broadcast/multicast routing will send the message out on all channels with ID's higher than the channel ID upon which the message was received such that the channel ID for forwarding corresponds to a "don't care" bit setting in the mask. For example, consider a broadcast from node 000 as shown in Figure 3.1.

More flexible routing is also possible using the hardware supported message forwarding mechanisms. These use multiple message addressing headers with forwarding bits to provide complete routing freedom.

Also, multiple messages from the same source and to the same destination may be aggregated into a single transmission to reduce setup overhead.

The nCUBE 2 hardware architecture is complimented by a powerful and robust software environment.

3.2 Software Environment

As the nCUBE 2 is a powerful second generation multiprocessor, the accompanying nCUBE Parallel Software Environment is an extensive, robust, and refined software environment. The environment supports a variety of programming models.

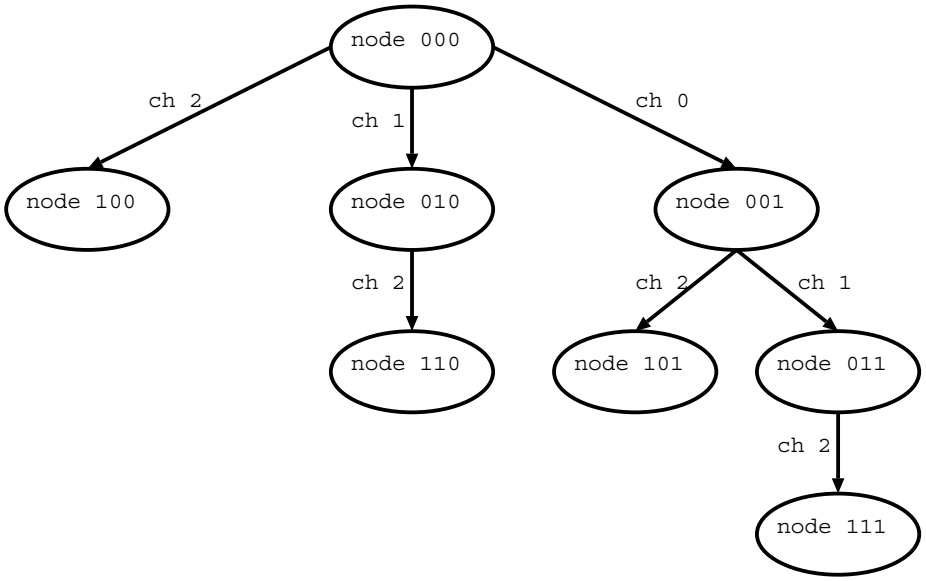

Figure 3-1. 3D Broadcast Example

The two primary models are (1) where node programs are launched directly from the host shell, and (2) where a host program launches the node programs. Within node programs, both the single program, multiple data (SPMD) and multiple program, multiple data (MPMD) models are provided. SPMD uses a single program that is loaded onto all the processors, but each instance of the program can have distinct data with data dependent branching. By contrast, the MPMD model allows different programs to be loaded onto distinct subsets of processors. The different programs can interact via message passing with data dependent branching still provided.

The environment includes the nCX microkernel executive that runs on the node and IO processors, the Host Interface Driver, a variety of standard programming languages, a powerful symbolic and assembly level debugger, extensive software libraries, and a sophisticated profiler.

The nCX microkernel is the node/IO processor resident executive. nCX supports process management, local memory management, message passing, multi-tasking, UNIX System V Release 4 system calls, and POSIX signals.

The nCUBE Host Interface Driver provides for the connection of the node processors to the host/front-end computer, which is usually a high performance workstation.

The nCUBE Parallel Software Environment supports FORTRAN 77, C, and C++ together with the Linda suite of hardware independent extensions for parallel processing and message processing.

A powerful symbolic and assembly level debugger, *ndb*, provides a full range of functions including:

- examination of variables, the stack, and processor registers

- multiple format data displays
- address map and source code displays
- the use of breakpoints
- single stepping through the code
- command and response logging, and
- other standard debugging operations

Extensive libraries provide for efficient usage of the features of the nCUBE 2 supercomputer. An extended Run-Time Library provides basic subcube management to include routines to identify a program's position in the subcube, its process ID, the host's process ID, and the dimension of the subcube. Also in this library are interprocessor communication routines such as an asynchronous message send *(nwrite)*, a blocking message receive *(nread)*, and routines to check if any or a selected range of messages types have been received (*ntest* and *nrange* respectively). These later routines provide for the development of a nonblocking message receive capability. Processor allocation routines are also included in this library and can be used to select processors in a subcube and launch programs on them.

The nCUBE Parallelization Library includes routines for 2 and 3 dimension fast Fourier transforms and matrix transposition; the min, max, and sum determination of either scalar or vector data items of various types; processor sequencing; grid-hypercube mappings; general purpose data allocation; and general purpose subcube broadcasting.

The nCUBE Math Libraries contain 375 routines callable from C or FORTRAN and are based on the Seismic Subroutine Standard (SSS) and Basic Linear Algebra Standard (BLAS). These are uniprocessor routines written in assembly language and highly tuned to the nCUBE 2's node processor. The IMSL libraries of FORTRAN subroutines are also available.

The nCUBE 2's profiler (PM) is composed of three tools: *xtool*, *ctool*, and *etool*. The xtool program profiles execution by functions on the various node processors by using a time interval sampling technique. The ctool emphasizes analysis of communication and for each node provides:

- time spent in calculation, communication, and IO;
- number of calls to each communication routine
- total number of all communication routine calls
- number of errors returned by communication routines
- distribution of sent and received message lengths
- number of objects broadcasted.

The etool facility provides for the profiling of user-defined events to include the time of each event, the type of event, and the value of a specified variable at the event's occurrence. Profiling results are displayed graphically for easy analysis. Profiling can be done for the entire program by setting an environment variable or can be limited to a specific part of the program by insertion of profiling routines into the code.

All together these facilities provide a powerful and robust environment for software development and performance-oriented execution.

3.3 Performance Characterization of the nCUBE 2

In order to achieve maximal performance on a specific multiprocessor, both the strengths and weakness of the target hardware must be identified, understood, and taken into account during algorithm development. This section summarizes a variety of experiments designed to quantify key performance characteristics of the nCUBE 2 architecture. These results will be used later in Chapter 4 to define a simulation model that will evaluate the achievable parallel performance of the unsymmetric-pattern multifrontal method. The results will also be used to justify subsequent algorithm design decisions in Chapters 5 and 6.

Specific areas of investigation include the rate of floating point operations, point-to-point message latencies, and message broadcast/multicast latencies.

Floating Point Operations: The design of the nCUBE 2 processor places a significant emphasis on fast floating point operations and even includes a combined multiply-add floating point instruction. According to the nCUBE 2 Processor Manual [109], a typical floating point operation (i.e., add, subtract, multiply, compare) takes 300-350 nanoseconds with more complex functions (like divide, square root, and remainder) taking 300-3450 nanoseconds. Additional timing variability occurs due to the variety of addressing modes (26), hits/misses in the 128 byte instruction cache, memory contention (with DMA activities, DRAM refresh cycles, and instruction fetches), and data misalignments.

In order to determine a typical floating point operations per second (flops) rate, two experiments were constructed. The first used a single loop that performed floating point divides (as would be typical of the calculation of multipliers in LU factorization). The second experiment used two nested loops that performed a multiply-add operation as would used in a typical pivot step of LU factorization. This second experiment was done with both possible loop orderings in an attempt to determine the affect of unit and nonunit memory strides. The experiments were coded in C and measurements taken with the *etool* user-defined event profiler.

The first experiment, using a single loop of divisions, produced results that fit well to a linear relationship. Specifically, for single precision divisions the equation is

$$time = 4.6 + 2.745n$$

where n is the number of divisions and time is in microseconds. Likewise, for double precisions divides the relationship is

$$time = 4.93 + 3.444n.$$

As arrays are stored by rows in the C language, unit stride is achieved with the row index in the outer loop. When this was done in the second experiment that used nested loops and a subtract-multiply operation, the performance was approximately 5% worse than was seen with the column index in the outer loop. This was a bit surprising given the auto-incrementing possible in the nCUBE 2 processor. The specific results are given below with all times in microseconds and i as the row index and j as the column index.

$$\text{Unit Stride (Single precision): } time = 17.24 + 1.56i + 4.88ij$$

$$\text{Unit Stride (Double precision): } time = 16.60 + 2.71i + 4.72ij$$

$$\text{Non-Unit Stride (Single precision): } time = 17.41 + 2.05j + 4.635ij$$

$$\text{Non-Unit Stride (Double precision): } time = 16.85 + 2.78j + 4.629ij$$

More efficient routines to perform these operations are available in the BLAS (Basic Linear Algebra Subroutines) library resident on the system, but these will be explored later. As these are highly tuned assembly language routines, their performance is expected to be significantly better than these initial experiments.

Point to Point Message Latency: Passing of contributions between frontal matrices will likely be done using point to point message passing. Hence, an understanding of the latencies associated with such activities will be important. As the nCUBE 2 uses wormhole routing (where messages passing through the network are not stored at intermediate nodes), very little dependency on path length is expected. However, message length is expected to be significant. Furthermore, we need to determine two distinct latencies. The first is the message transmission latency. This is the time it takes for a message of given length to be transmitted and received by a particular set of processors of given distance apart. The second time is the message initiation latency, which is how much time a sending processor takes to initiate the transmission before it can continue.

For the first experiment, dealing with the message transmission latency, a test program was written in C and profiled using *etool*. Within the program, processor node 0 sends a message of a specified byte length to the most distant node in the executing cube/subcube. It then does a blocking receive waiting for an equal length acknowledgment message from the other node. The distant node performs a blocking receive and then sends the acknowledgment message. The test was repeated 1000 times for each configuration. For each configuration, message lengths were set at 10, 100, 500, 1000, 5000, and 10000 bytes. Cube/subcube dimensions varied from 1 to 5.

A linear regression run on the resulting timings revealed a very strong linear relationship with both message length and cube/subcube dimension (with message length providing a much larger contribution however). The resulting linear equation (given in microseconds with message length l in bytes and d being the dimension of the subcube) is

$$time = 170.32 + 4.54d + 0.573l.$$

The second experiment was designed to determine the message initiation latency. Again using a C program and *etool*, a set of experiments using the same configurations of message length and cube/subcube size as were used in the first experiment was conducted. This time, however, node 0 just sent one message after another to the distant node without waiting for an acknowledgment. A total of 1000 messages were sent in each configuration.

For this second experiment, cube/subcube dimension had no influence on the initiation latency. There is a strong linear correlation with message length which is expressed by the following equation (again given in microseconds with message length (l) in bytes)

$$time = 79.94 + 0.4498l.$$

The regression for this second experiment however did not use the results for the 10 byte message configurations, as initiation latencies for messages of 10 to 100 had very similar initiation times. That is, the performance curve flattened in this region.

For both of the point to point message experiments, the standard *nwrite* and *nread* functions were used. These functions copy the message from a user buffer to a separate communication buffer (and vice-versa for *nread*) where the actual message transmission occurs from the communication buffer. Alternative functions are available that use only the communication buffers, but these were not tested. The message broadcast tests to be mentioned next also use the separate user and communication buffers.

Message Broadcast/Multicast Latencies: From the description of the message broadcast/multicast capabilities of the nCUBE 2 provided in the nCUBE 2 Processor Manual [109], definite performance benefits are anticipated when using the broadcast/multicast mechanisms as opposed to repeated point to point message passing. Such data distribution will be required in the dense matrix kernels that will factorize distinct frontal matrices in parallel.

To this end, another set of experiments was developed to quantify both the full message broadcast latency and the broadcast initiation latency. Again, C programs were written and run with the *etool* profiler. Configurations used message lengths of 100, 500, 1000, 5000, and 10000 bytes and cubes of dimensions 1 to 5.

In the full message broadcast latency test, node 0 initiates message broadcasts to all nodes in the active cube and then waits for an equal size point-to-point acknowledgment message from the most distant node in the cube. (The description of the broadcast function in the nCUBE 2 Processor Manual, which was described in the preceding chapter, assures that this most distant node is the last to receive the broadcast). All nodes receive the broadcast and the most distant node generates the acknowledgments. With the point-to-point message latency well defined by the earlier experiments, their contributions to the overall latency are subtracted to obtain the latency due only to the broadcast.

The results show strong correlations with both message length and cube dimension together with a very significant product term of message length times cube dimension. The resulting formula (given in microseconds with message length (l) in bytes and d

the subcube's dimension) is

$$time = 195.22 + 0.122l + 50.02d + 0.45ld.$$

Message broadcast initiation latencies were also investigated using a C program that had node 0 initiate 1000 broadcasts back to back and then wait for a single acknowledgment from the most distant node. (Again the well defined acknowledgment message time was subtracted off the overall latency). The same configurations as in the first experiment were used.

The initiation latency results also correlated with message length, cube dimension, and the product of the two. The resulting equation (given in microseconds) is

$$time = 186.65 + 0.0624l + 54.09d + 0.45ld.$$

Comparison of the point to point and broadcast/multicast results indicate that, for cubes of dimension greater than one, the broadcast/multicast mechanism is to be preferred over successive point-to-point transmissions.

Chapter Summary: In this chapter, the implementation platform of this research effort has been discussed. Specifically, this platform is the nCUBE 2 distributed memory multiprocessor. First, the features of the hardware architecture were described. Then, an overview of the key software features was provided. In the last section, a series of experiments were designed and conducted to characterize the performance of the nCUBE 2. The resulting characterization will be used in the next chapter to define a realistic simulation model to evaluate the achievable parallelism in the unsymmetric-pattern multifrontal method for the LU factorization of sparse matrices. This characterization will also be used in Chapters 5 and 6 when a parallel version of the unsymmetric-pattern multifrontal method is designed and implemented on the nCUBE 2.

CHAPTER 4
EVALUATION OF POTENTIAL PARALLELISM

In this chapter, the parallelism available in the unsymmetric-pattern multifrontal method [34, 35] is investigated. The assembly DAG is the principle data structure of this method and describes the data dependencies that exist between the various dense submatrices, called *frontal matrices*. This assembly DAG is also significant as it exposes parallelism between the frontal matrices. Moreover, parallelism can be exploited within frontal matrices also, as the factorization of these dense submatrices consists of many regular and independent computations. The models developed in this chapter will study both sources of parallelism as used in different combinations and at different levels of granularity.

The first objective is to determine the amount of theoretical parallelism available. This is first done by a series of analytical models that assume an unbounded number of available processors. Then, the analytical models are expanded into simulation models where a bounded number of processors are assumed. These simulation models are used to determine the minimum number of processors needed to actually achieve the theoretical speed-ups.

With the theoretical parallelism determined by the models of the first objective, the second objective is to determine how much of the theoretical parallelism can actually be achieved on existing multiprocessors. This objective is addressed by revising the simulation models to correspond to the performance characteristics of the nCUBE 2 multiprocessor (as determined in Chapter 3).

4.1 Unbounded Parallelism Models

The first objective in the investigation of parallelism within the unsymmetric-pattern multifrontal method is to assess the availability of theoretical parallelism inherent in the assembly DAGs produced by the method. This is done using analytical models that assume an unbounded number of available processors.

An underlying multiple instruction, multiple data (MIMD) shared memory (SM) model of computation is assumed, although several of the models are also realizable on single instruction, multiple data (SIMD) model. The shared memory is based on a parallel random access machine (PRAM) model, which is initially assumed to allow concurrent reads and concurrent writes with concurrently written results summed (CRCW-SUM). Later models will require only a concurrent read, exclusive write (CREW) shared memory. Instruction counts are abstracted to only the floating point operations (flops) required for the assembly and numerical factorization processes.

The analysis of the unbounded parallelism models is based on the representations of assembly DAGs built from traces produced by a sequential version of the unsymmetric-pattern multifrontal method on matrices from real applications [34, 35].

These assembly DAGs are analyzed and then used as input for the five unbounded parallelism models. The resulting speed-up, processor set, and utilization estimates are then analyzed.

4.1.1 Assembly DAG Analysis

A sound characterization of the assembly DAG is the cornerstone of the unbounded models and will be fundamental to fully understanding the results of the bounded models. In order to obtain this characterization, the assembly DAGs are topologically sorted and partitioned into a data structure that identifies all the nodes in a given topological level. Node weights are assigned in accordance with the model definitions defined later. Using these node weights, the heaviest weighted path through the assembly DAG is determined for each model. The weight of this path provides the lower bound on the parallel execution time.

Other characteristics of the assembly DAGs are also determined. In particular, the number of nodes and edges, the depth, and the number of leaf nodes are determined. Since all the assembly DAGs tend to be significantly wider at their leaf level (Level 0), the next widest level is also determined. The total weight of all nodes is divided by the number of nodes to obtain an average node weight. The depth is divided by number of nodes to give a depth ratio. Low values will correspond to DAGs that have the most significant course grain parallelism. Various methods of exploiting the available parallelism are investigated with the different models defined next.

4.1.2 Model Definitions

A model of the sequential version of the algorithm is provided as reference and will be used to establish the speed-up calculations. In this and all the subsequent models there are two levels. The first (higher) level consists of the outer sum that counts processing steps for the frontal matrices (nodes) that make up the overall sparse matrix. This summation will either be across all nodes of the assembly DAG, which represents a sequential processing of the frontal matrices, or along the heaviest path, which represents the critical path for the processing. The heaviest path summation is used to characterize parallelism exploited between the nodes in the assembly DAG. With the assumption of an unbounded number of processors, processing of nodes with dependencies not on the heaviest path (critical path) can be done during the critical path processing and not affect the overall completion time.

The second level of the models describes the processing and parallelism of factorizing a particular frontal matrix (i.e., a node in the assembly DAG). Each of these descriptions are presented in four terms. The first term describes the processing needed to complete the assembly of contribution blocks from predecessors in the assembly DAG. The second term characterizes the processing needed for the selection of a pivot for numerical stability. The third term describes the calculation of multipliers, which results in a column of the lower triangular matrix L. Finally, the fourth term defines the updating of the active submatrix, which is essentially the carrying out of the necessary row operations needed to reduce the matrix.

These models are very similar to an earlier effort by Duff and Johnsson [51] that focused on the assembly trees produced by symmetric-pattern, multifrontal methods.

Each of the models use the following terms:

- A_j - number of matrix entries assembled into the frontal matrix represented by $node_j$.
- S_j - number of children assembling entries into $node_j$.
- P_j - number of pivots by which the frontal matrix at $node_j$ is reduced.
- c_j - number of columns in the $node_j$ frontal matrix.
- r_j - number of rows in the $node_j$ frontal matrix.

With these definitions, the model for the sequential version of the algorithm is defined as follows:

$$\Sigma_{j=1}^{allnodes}[A_j + \Sigma_{i=1}^{P_j}(r_j - i + 1) + \Sigma_{i=1}^{P_j}(r_j - i) + 2\Sigma_{i=1}^{P_j}(r_j - i)(c_j - i)]$$

Model 0 (Concurrency Between Frontal Matrices Only): Model 0 depicts a version of the algorithm that only exploits parallelism across nodes in the assembly DAG. Each frontal matrix is factorized sequentially. This model requires an underlying MIMD architecture. The formula describing this model is:

$$\Sigma_{j=1}^{heaviest\_path}[A_j + \Sigma_{i=1}^{P_j}(r_j - i + 1) + \Sigma_{i=1}^{P_j}(r_j - i) + 2\Sigma_{i=1}^{P_j}(r_j - i)(c_j - i)]$$

In the case of a dense matrix, this model degenerates to a single, one processor tasks that requires $O(n^3)$ time.

Model 1 (Block Concurrency Within Frontal Matrices Only): Model 1 assumes that nodes of the assembly DAG are processed sequentially in an order that preserves the dependencies expressed by the assembly DAG. Within each node, the frontal matrix is factorized with a block level of parallelism. Assembly, multiplier computation, and updates to the active submatrix are done for all rows in parallel moving sequentially through the columns. For the assembly of a row that includes contributions from more than one child in the assembly DAG, the assumption of a CRCW-SUM shared memory insures correctness. In this model selection of numerical pivots is assumed to be sequential.

The method described by this model and the next lend themselves easily to either a SIMD or a MIMD architecture.

$$\Sigma_{j=1}^{allnodes}[c_j + \Sigma_{i=1}^{P_j}(r_j - i + 1) + P_j + 2\Sigma_{i=1}^{P_j}(c_j - i)]$$

In the case of a dense matrix, this model requires n processors and takes $O(n^2)$ time.

Model 2 (Full Concurrency Within Frontal Matrices Only): Concurrency within a frontal matrix is extended by Model 2. In this model, very fine grain parallelism is exploited. Assembly is done for each element in parallel. However, in order to maintain consistency with the earlier work of Johnsson and Duff [51], a CREW memory

is used by this model and the assembly of contributions to an entry from multiple children is done via a parallel prefix computation using the associative operator of addition.

Numerical pivot selection is handled in this model with another parallel prefix computation this time using the maximum function as the associative operator. Multiplier computation and updating of the active matrix are done with parallelism at the matrix entry level, which is easily realizable with the CREW memory model. This model is also oriented to either a SIMD or MIMD architecture.

$$\Sigma_{j=1}^{allnodes}[\lceil \log_2 S_j \rceil + \Sigma_{i=1}^{P_j} \lceil \log_2 (r_j - i + 1) \rceil + P_j + 2P_j]$$

In the case of a dense matrix, this model requires n^2 processors and takes $O(n \log n)$ time.

Model 3 (Block Concurrency Within and Between Frontal Matrices): The block-oriented, node level parallelism of Model 1 is augmented by parallelism across nodes of the assembly DAG in Model 3. This is done by changing the outer summation to be over the heaviest path instead of across all nodes. With this model, a MIMD architecture is required.

$$\Sigma_{j=1}^{heaviest\_path}[c_j + \Sigma_{i=1}^{P_j}(r_j - i + 1) + P_j + 2\Sigma_{i=1}^{P_j}(c_j - i)]$$

With a dense matrix, this model is equivalent to Model 1.

Model 4 (Full Concurrency Within and Between Frontal Matrices): The suite of models is completed by extending the full node level concurrency of Model 2 to include parallelism across nodes in the assembly DAG. This is done in the same fashion as with Model 3 and a MIMD architecture is likewise assumed.

$$\Sigma_{j=1}^{heaviestpath}[\lceil \log_2 S_j \rceil + \Sigma_{i=1}^{P_j} \lceil \log_2 (r_j - i + 1) \rceil + P_j + 2P_j]$$

With a dense matrix, this model is equivalent to Model 2.

4.1.3 Matrices Analyzed

A set of five matrices was selected for this study. They represent a cross section of the type of assembly DAGs that are generated. The matrices are from the *Harwell-Boeing set* and each matrix comes from a real application [49]. Table 4-1 briefly describes these matrices:

4.1.4 Assembly DAG Analysis Results

The first objective of this part of the effort was to characterize the assembly DAGs of the test matrices. Table 4-2 below provides these characterizations that were produced with the analysis program written for the effort. The DEPTH entries are the number levels in the assembly DAG as determined by a topological sort. The width of the DAG is presented in two entries. LEVEL 0 refers to the leaf nodes of the assembly DAG and WIDTH gives the widest level other than the leaf level (Level 0). It is important to remember that the assembly DAGs are typically NOT a

Table 4-1. Test Matrix Descriptions

MATRIX	ORDER	DESCRIPTION
MAHINDASB	1258	economic modeling
GEMAT11	4929	optimal power flow problem
GRE_1107	1107	computer system simulation
LNS_3937	3937	compressible fluid flow
SHERMAN5	3312	oil reservoir, implicit black oil model

single connected component and that the many small, distinct connected components frequently show up on these lowest levels of the DAG's topological order. The DEPTH RATIO is the DEPTH divided by the number of nodes in the assembly DAG. This statistic will be smaller for the short and wide DAGs that have the greatest inherent parallelism.

Table 4-2. Assembly DAG Characterizations

MATRIX	MAHINDASB	GEMAT11	GRE_1107	LNS_3937	SHERMAN5
# NODES	238	1015	511	1636	664
# EDGES	507	1,219	1,366	3,724	1,496
DEPTH	68	25	40	296	46
LEVEL 0	86	530	233	565	359
WIDTH	55	189	106	248	99
AVG WEIGHT	1,887	1,195	9,998	67,856	42,093
DEPTH RATIO	0.29	0.02	0.08	0.18	0.07

The statistics presented in Table 4-2 reveal that the assembly DAGs tend to be short and wide (which is the desired shape for parallelism). However, they do not provide the full picture. The true nature of the assembly DAGs was better revealed with an abstraction of the DAG provided by the analysis program. This abstraction illustrates the topologically ordered assembly DAG with an example provided for the GEMAT11 matrix in Figure 4-1. For each level the number of elements in that level is provided as well as the minimum, average, and maximum node weights. (The node weights are expressed as common logs to save space and since their magnitude is all that is of interest). These abstractions revealed that the assembly DAGs were very bottom heavy in terms of the number of nodes but that the average node size tended to increase dramatically in the upper levels. The assembly DAG for the GEMAT11 matrix had the nicest shape for parallelism (as suggested by its depth ratio). The other assembly DAGs were taller and relatively more narrow at their base.

Speed-Ups: The most interesting results of the unbounded models are the speed-ups available via parallelism. These results are presented in Table 4-3.

```
L(Level): Count (Min,Avg,Max)

L(  25):     1 (5.3, 5.3, 5.3)                    *
L(  24):     1 (4.8, 4.8, 4.8)                    *
L(  23):     1 (3.7, 3.7, 3.7)                    *
L(  22):     1 (3.7, 3.7, 3.7)                    *
L(  21):     1 (3.7, 3.7, 3.7)                    *
L(  20):     1 (4.7, 4.7, 4.7)                    *
L(  19):     1 (3.6, 3.6, 3.6)                    *
L(  18):     2 (3.6, 4.1, 4.3)                    *
L(  17):     2 (3.4, 3.8, 4.0)                    *
L(  16):     2 (3.3, 4.0, 4.2)                    *
L(  15):     2 (3.0, 4.0, 4.2)                    *
L(  14):     2 (3.0, 3.4, 3.7)                    *
L(  13):     3 (3.0, 3.4, 3.6)                    *
L(  12):     4 (2.6, 3.7, 4.2)                    *
L(  11):     6 (3.0, 3.5, 3.9)                    *
L(  10):     4 (2.1, 3.9, 4.5)                    *
L(   9):     4 (3.0, 3.3, 3.7)                    *
L(   8):     7 (2.6, 3.5, 3.9)                    *
L(   7):    10 (2.6, 3.4, 3.9)                    *
L(   6):    16 (2.5, 3.6, 4.5)                    **
L(   5):    24 (2.2, 3.5, 4.2)                    **
L(   4):    35 (2.2, 3.2, 4.2)                   ***
L(   3):    58 (1.6, 3.0, 3.8)                  *****
L(   2):   108 (1.6, 2.9, 3.6)                 ********
L(   1):   189 (1.6, 2.8, 3.5)               **************
L(   0):   530 (0.7, 2.5, 3.3)   ************************************
```

Figure 4-1. Assembly DAG Abstraction for GEMAT11

The speed-ups for Model 0 are disappointing but not unexpected after what was seen in the previous section on the shape of the assembly DAGs. However, significant speed-ups were seen as parallelism was introduced within the nodes (Models 1 and 2) and within and between nodes (Models 3 and 4). It is interesting to note that a significant synergism is taking place when parallelism is utilized both within and between nodes. One's intuition might suggest that both levels of parallelism would produce a resulting speed-up that is close to the product of the speed-ups achieved by using levels of parallelism separately. However, the speed-ups obtained are actually much better than this expectation. This is consistent with the results of the earlier study by Duff and Johnsson [51].

Table 4-3. Unbounded Parallelism Speed-Up Results

MATRIX	MAHINDASB	GEMAT11	GRE_1107	LNS_3937	SHERMAN5
MODEL 0	1.17	2.88	1.26	1.30	1.74
MODEL 1	10.50	9.31	41.21	103.77	78.36
MODEL 2	114.25	39.35	575.21	3,490.76	1,877.05
MODEL 3	15.96	68.79	70.15	167.94	166.86
MODEL 4	237.74	743.80	1,847.68	11,425.40	6,868.42

Another interesting, but expected, observation is that the speed-ups for models 0, 3, and 4 (that use parallelism between frontal matrices) correlate strongly with the depth ratio presented earlier. Likewise, models 1, 2, 3, and 4 correlate strongly with average node weight. The underlying principle is that concurrency within the node is most exploitable by larger frontal matrices and that concurrency across nodes is most exploitable with short and wide assembly DAG structures.

Processors Sets and Utilization: Also determined by the unbounded models was an upper bound on the number of processors that could be employed on a given matrix and the associated processor utilization. In particular, Model 0 assumed that any frontal matrix would have only one processor allocated to it. For the block level concurrency of Models 1 and 3, a frontal matrix would have a processor set equal to the number of rows in the frontal matrix. The full concurrency of Models 2 and 4 would use processor sets equal to the row size times column size of the frontal matrix. The maximum processor usage would then be the largest processor set for any particular frontal matrix in Models 1 and 2 and the largest sum of processors for any one level in the assembly DAG for Models 0, 3, and 4. This strategy results from the use of concurrency across nodes in Models 0, 3, and 4.

Utilizations were then calculated by dividing total sequential time by the product of total parallel time times the number of processors used. The results by matrix and model are provided in Table 4-4. The top value in each entry is the size of the processor set and the lower value is the utilization.

In all cases the size of the processor sets seem quite high and the utilization quite low, but these estimates are very crude upper bounds and it is likely that significantly better results are realizable.

4.1.5 Conclusions and Observations

A significant amount of theoretical parallelism is achievable with the unsymmetric-pattern multifrontal method. However, very little of the parallelism comes directly from the assembly DAG as is evidenced from the Model 0 results. Yet a synergism occurs when the parallelism from the assembly DAG is combined with the concurrency available within the factorizing of particular frontal matrices. The resulting speed-ups, as seen in Models 3 and 4, are very promising.

Table 4-4. Processor Set and Utilization Results

MATRIX	MAHINDASB	GEMAT11	GRE_1107	LNS_3937	SHERMAN5
MODEL 0	86	530	233	565	359
	1.36%	0.54%	0.54%	0.23%	0.48%
MODEL 1	52	68	126	258	189
	20.19%	13.70%	32.71%	40.22%	41.46%
MODEL 2	3380	4624	19782	107874	47060
	3.38%	0.85%	2.91%	3.24%	3.99%
MODEL 3	304	3788	1161	2961	3827
	5.25%	1.82%	6.04%	5.67%	4.36%
MODEL 4	4161	30127	19782	179364	80031
	5.71%	2.47%	9.34%	6.37%	8.58%

Looking beyond the speed-ups, the large processor sets and low utilizations are of both concern and interest. In particular, the very bottom heavy assembly DAGs suggest that better distributions of the processing are possible. Such processing distributions could be achieved by appropriate task scheduling on bounded sets of processors. The next section explores these possibilities using trace-driven simulation techniques.

4.2 Bounded Parallelism Models

The results of the unbounded parallelism models illustrated that significant speed-ups are possible with parallel implementations of the unsymmetric-pattern multifrontal method. However, the efficient use of processor sets and the speed-ups achievable on smaller processor sets are still open questions. This section addresses these issues using the results of trace-driven simulations run on the five models defined in the previous section. With each model sixteen different processor sets and three different scheduling schemes were used. The processor sets were powers of 2 from 2^1 to 2^{16} (65,536). The scheduling schemes correspond to orderings of the work queue and are described in a subsequent paragraph.

The results of the bounded parallelism models will indicate that the speed-ups seen in the unbounded models are achievable will reasonable size processor sets and with significant improvements in efficiency.

This section is a summary of the bounded parallelism models. For a full description of the models and their revisions, see the Hadfield and Davis' technical report [83].

4.2.1 Initial Models

The initial bounded parallelism models follow directly from the unbounded parallelism models. The critical difference is that a limited processor set is assumed so tasks that are ready for execution may have to wait for available processors. This

can be appropriately modeled via a trace-driven simulation run against the representation of the assembly DAGs obtained from the traces produced by the sequential version of the unsymmetric-pattern multifrontal method [34, 35]. Critical issues with the simulation will be how to allocate processors when the number available is less than that required and how to order the work that is ready to be done.

4.2.2 Trace-Driven Simulation

The trace-driven simulations used to implement the bounded parallelism models were accomplished with a Pascal program that was built as an extension to the analysis program used for the unbounded parallelism models. The simulation uses the topologically sorted representation of the assembly DAG as input. All the nodes (frontal matrices) on the leaf level (Level 0) are initially put into a work queue. Work is scheduled from this queue based on processor availability. The amount of work and required processors for each node is determined based on the specific model being simulated. These models are initially those defined for the unbounded parallelism models. When the model calls for a sequential ordering of the nodes, only one node may be executing at any particular time, otherwise multiple nodes may execute concurrently based on processor availability and the allocation strategy in use. Upon completion of a node's execution, the edges to dependent nodes are removed. When all such edges to a particular node have been removed, that node is put into the work queue as it has become available for execution. The simulation continues until all nodes have completed execution.

Speed-up and processor utilization are then calculated per the following formulas:

$$Speedup = \frac{Time_{sequential}}{Time_{parallel}}$$

$$Utilization = \frac{\Sigma_{j=1}^{allnodes}(Time_j * ProcessorsUsed_j)}{(Time_{finished} * ProcessorSetSize)}$$

4.2.3 Processor Allocations

Allocation of processors to the assembly and factorizing of a frontal matrix is a critical issue for the realization of the algorithm on a bounded set of processors. As was seen in the results of the initial models, the allocation scheme can greatly effect the performance of the algorithm. Since the five unbounded models use three different approaches to factorization of a particular frontal, there are three distinct allocation schemes.

The allocation scheme required by Model 0 is trivial since each frontal matrix in this model is allocated only a single processor.

Models 1 and 3 use a block concurrency within a frontal matrix. As this block concurrency is row-oriented, a processor set equal to the row size is required. The difficulty arises when there are processor sets available but they are smaller than that required (in fact the total available number of processors can be less than the row size). In this eventuality the total amount of work is evenly distributed across the available processors. This is a fairly crude allocation scheme but is not too unrealistic given the nature of the processing within a frontal matrix.

This strategy was initially followed blindly and lead to some terrible schedulings as large frontal matrices were allocated relatively small processors sets. This greatly extended their execution times and delayed the subsequent frontal matrices that were dependent upon them. Revisions to this allocation strategy were to (a) check if waiting for more available processors would improve a frontal's execution time and scheduling appropriately and (b) wait for more processors if the required amount is not available. Both strategies corrected the problem and produced surprisingly similar results. Strategy (b) will be reported in the results of this discussion.

Models 2 and 4 use a full concurrency within each frontal matrix and requires a number of processors that is equal to the number of entries in the frontal matrix (row_size times col_size: $r_j \cdot c_j$). A more sophisticated allocation scheme is thus required. Prior to formally defining this scheme, we recall the definition of the time complexity for a frontal matrix using this model:

$$work_j = \lceil \log_2 S_j \rceil + \Sigma_{i=1}^{P_j} \lceil \log_2 (r_j - i + 1) \rceil + P_j + 2P_j$$

Using this definition and a similar assumption on the ability to distribute work on smaller processor sets, the allocation scheme is formally defined by the algorithm in Figure 4-2.

```
if processors_available ≥ (col_size * row_size) then
      Schedule work on (col_size * row_size) processors
else
      if processors_available ≥ row_size then
            Schedule (work + 3 * pivots * col_size ) on
                  (row_size) processors
      else
            if processors_total ≥ row_size then
                  Wait for more available processors
            else
                  Schedule (work * col_size * row_size) /
                        processors_available
                        on processors_available
            endif
      endif
endif
```

Figure 4-2. Processor Allocation For Fine Grain Parallelism

Notice that the deepest nested else block provides a fairly crude over-approximation of the amount of work to do. However, this case is only used for the small total processor set sizes, which are not the target processor sets for the parallelism of Models 2 and 4. Thus, this over-estimate is reasonable.

Furthermore, this allocation strategy tended to produce a very step-like performance function as processor set sizes grew to meet the various cut-off points. A smoother performance curve was achieved by using a technique called *vertical partitioning*, which subdivides a frontal matrix task into a series of subtasks each requiring substantially fewer processors. This produced much more efficient schedulings and significant performance improvements. Results for both the initial allocation scheme and vertical partitioning will be reported.

4.2.4 Scheduling Methods

There are three scheduling methods employed on the work queue. These methods correspond to how the work ready to be executed is ordered within the work queue. The three corresponding orderings are first-in, first-out (FIFO); largest calculation first; and heaviest path first.

The FIFO method is self-explanatory and included because of its simplicity and efficiency.

The largest calculation first order uses the node weight estimate of the work to be done and schedules the largest such nodes first. This method is designed as an approximation of the heaviest path first method. The largest calculation first requires a priority queue implementation based on a set of priorities that is quite variable. As such there would be some significant penalties to address in implementing this method.

The last method, heaviest path first, uses the heaviest path determined by the analysis portion of the software. Any node on this path would be the next scheduled. Other nodes are put in a FIFO ordering. Notice that since the heaviest path is essentially a path through the dependency chart, only one node at most from the heaviest path will ever be in the work queue at any one time.

Very little difference was found using these scheduling methods and they will not be discussed further in this document (for a detailed description of these results, see [83]).

4.2.5 Simulation Results

The results of the simulations are summarized for each model. Whenever the results for all five matrices are presented, the following line types will be used to represent the different matrices:

```
-------   MAHINDASB
-*-*-*-   GEMAT11
-+-+-+-   GRE_1107
-O-O-O-   LNS_3937
- - - -   SHERMAN5
```

All the results presented in this section are based on the heaviest path first scheduling.

Model 0 (Concurrency Between Frontal Matrices Only): Model 0 takes advantage of concurrency only between frontal matrices. The speed-up and utilization results from Model 0 are shown in Figures 4-3 and 4-4, respectively. These indicate that the speed-ups seen in the unbounded models are achievable with significantly fewer processors and corresponding higher utilizations. Table 4-5 compares the number of processors used in the bounded versus the unbounded models to achieve maximum speed-up.

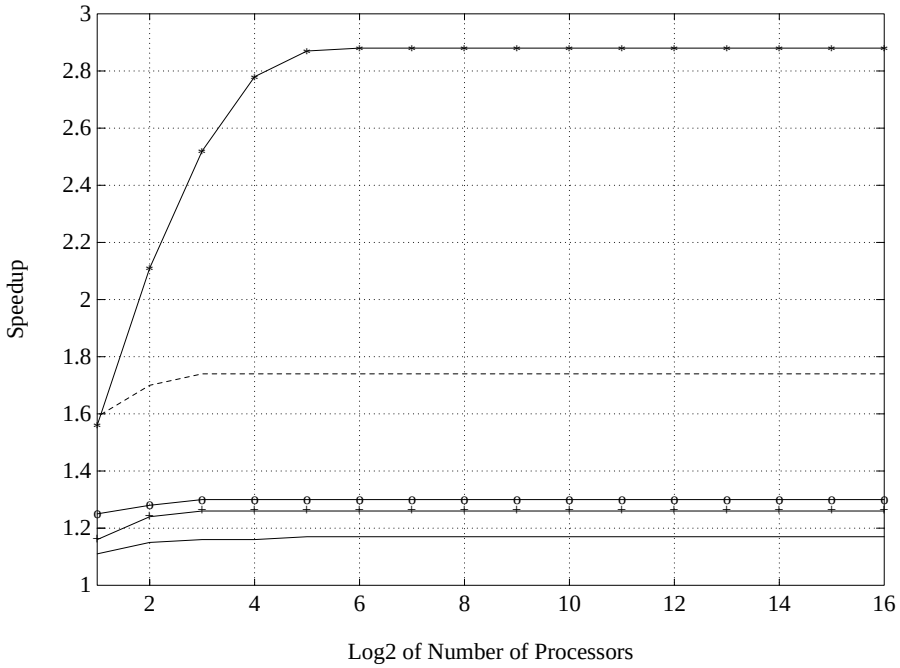

Figure 4-3. Model 0 Speed-Up Results

Table 4-5. Required Processors Comparison

MATRIX	BOUNDED MODEL	UNBOUNDED MODEL
MAHINDASB	32	86
GEMAT11	64	530
GRE_1107	8	233
LNS_3937	8	565
SHERMAN5	8	359

Model 1 (Block Concurrency Within Frontal Matrices Only): Model 1 only takes advantage of concurrency within frontal matrices and does so in a block-oriented fashion. The speed-up and utilization results are shown in Figures 4-5 and 4-6,

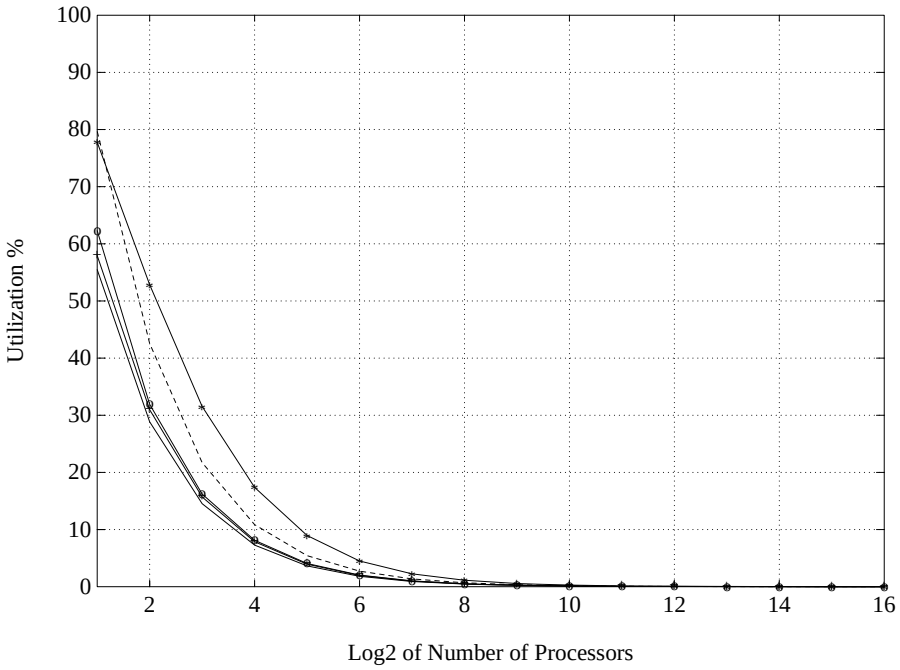

Figure 4-4. Model 0 Utilization Results

respectively. The method employed by this model seems well suited for processor sets up to about 256 processors. After that point there is no further advantage to additional processors for any of the test matrices.

Model 2 (Full Concurrency Within Frontal Matrices Only): The block concurrency of Model 1 is extended to full concurrency in Model 2 but is still restricted to concurrency only within the frontal matrix. The speed-up and utilization results for Model 2 using both the initial allocation scheme and vertical partitioning are shown in Figures 4-7, 4-8, 4-9, and 4-10, respectively. A very definite step behavior is evident in the speed-up curves for the initial allocation scheme. This behavior directly correlates to the allocation scheme in use that bases allocation on the row size and column size of the frontal matrices. While the speed-ups are significant, the step behavior severely limits scalability. Vertical partitioning resolves this problem.

Model 3 (Block Concurrency Within and Between Frontal Matrices): Model 3 combines the block concurrency within a frontal of Model 1 with the concurrency between frontal matrices used in Model 0. The speed-up and utilization results for this model are shown in Figures 4-11 and 4-11, respectively. These results were produced using the allocation strategy that waits if sufficient processors are not available.

Model 4 (Full Concurrency Within and Between Frontal Matrices): The fifth and final model uses concurrency between nodes and full concurrency within nodes. The speed-up and utilization results for this model are included for both the initial allocation scheme and vertical partitioning. These are presented in Figures 4-13, 4-14, 4-15, and 4-16.

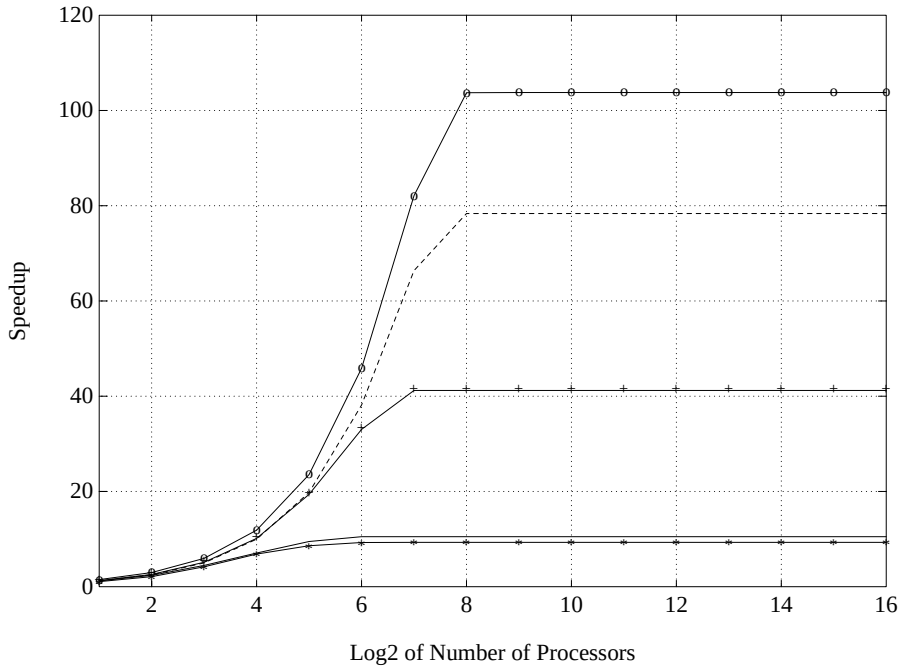

Figure 4-5. Model 1 Speed-Up Results

A step-like speed-up and corresponding utilization irregularities are evident in the original allocation scheme and very similar to the results for Model 2. Again, vertical partitioning corrects this problem.

4.2.6 Conclusions and Observations

Several conclusion and observations are apparent from the results of the bounded parallelism models.

1. There is an excellent potential for parallelism under both SIMD and MIMD implementations.

2. The speed-ups indicated in the unbounded models are achievable with practical size processor sets.

3. The benefits of the different models are realized within distinct ranges of the number of processors.

4. Scheduling method has little significant effect on performance for all five of the models. However, the critical path method was not tested in these initial runs and may have a significant effect.

5. Processor allocation schemes can have very significant effects on performance.

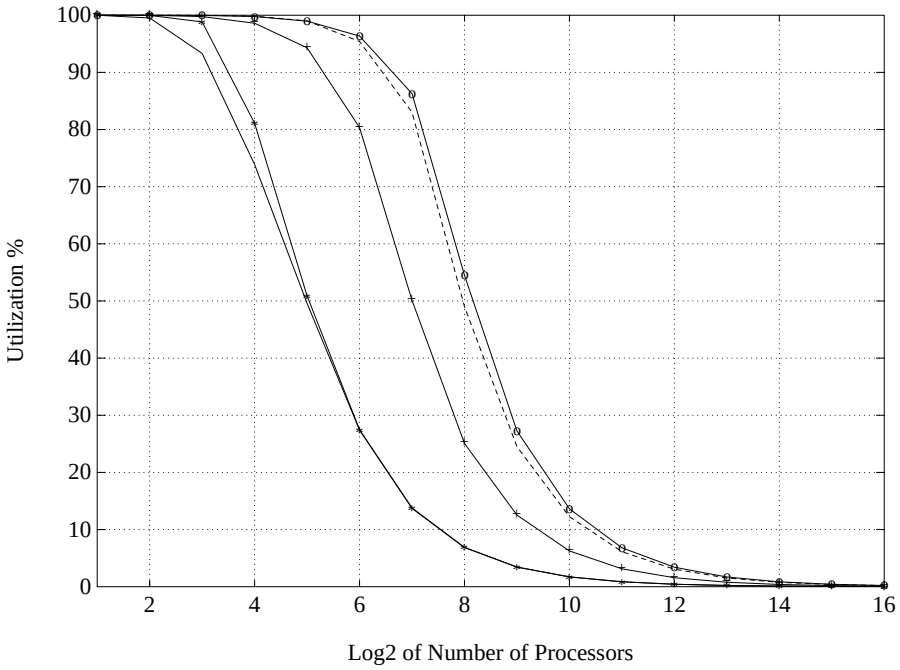

Figure 4-6. Model 1 Utilization Results

With this establishment of the theoretical parallelism in the unsymmetric-pattern multifrontal method, the next step is to use the characterization of an existing multiprocessor (the nCUBE 2) from Chapter 3 to revise the simulation model and study the parallelism that is actually achievable.

4.3 Distributed Memory Multifrontal Performance Potential

With the characterization of the nCUBE 2's performance achieved in the Chapter 3, the bounded parallelism PRAM simulation models are refined to the estimate the achievable parallelism of the unsymmetric-pattern multifrontal method. The performance characteristics of the nCUBE 2 are used in the revised simulation models so that predicted performance can be later compared to that actually achieved as the method is later implemented on the nCUBE 2.

However, one important prerequisite remains before the simulation model is updated. This prerequisite is to specifically characterize the performance of the dense matrix factorization routine(s) to be used within the unsymmetric-pattern multifrontal method. Recall that a primary motivation of multifrontal methods is to replace sparse matrix operations with more efficient dense submatrix operations. Hence, such a characterization of full and partial dense matrix factorization is critical to development of a realistic simulation.

4.3.1 Dense Matrix Factorization Routine

The characterization of the dense matrix factorization routine is developed by actually implementing such a routine and measuring its performance. A model of its

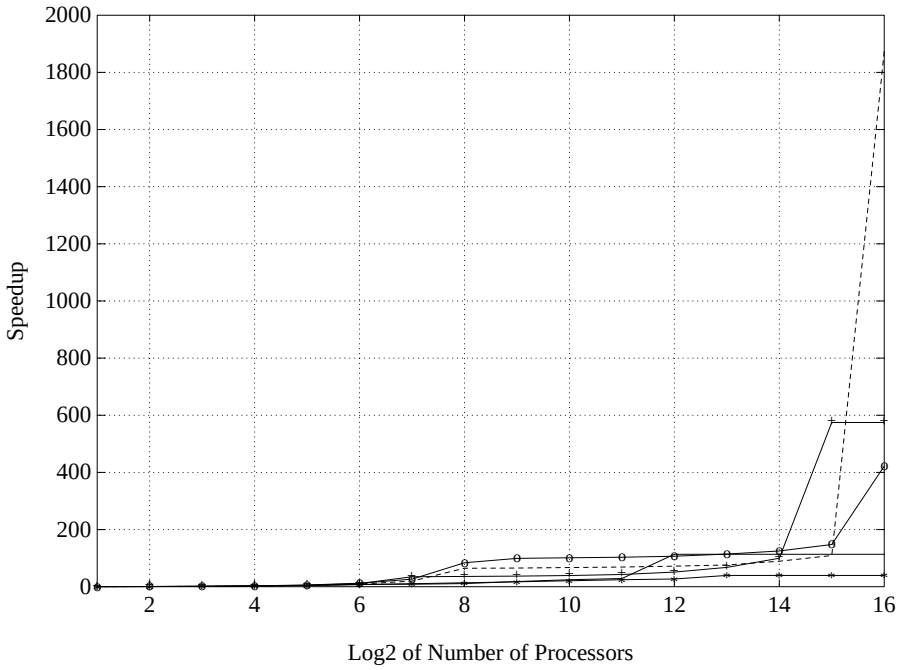

Figure 4-7. Model 2 Original Speed-Ups

performance is also developed and the actual measured results are compared to the results predicted by the model.

Choice of Algorithm

The algorithm chosen for dense matrix factorization is a column-oriented, fan-out method. This model was chosen because it is easy to analyze and implement and also because this was the approach taken in successful distributed memory implementations of multifrontal Cholesky factorization [105, 76].

Specifically, columns of the matrix are stored in a scattered fashion across the processors. For added simplicity, diagonal dominance is assumed, so no numerical pivoting will take place. An outer indexed loop provides one iteration for each pivot with which the matrix will be partially factorized. If the pivot column is owned by the processor, the multipliers (values for the corresponding column of L) are calculated by dividing by the pivot entry. Then, this column of multipliers is broadcasted to the other processors. Following the broadcast, the originating processor will use the multipliers to update any remaining columns that it owns. If a processor is not the owner of the current pivot column, then it will simply wait for the broadcasting of the multipliers and then update its columns with the received multipliers. This algorithm is outlined in Figure 4-17.

Performance Model

An analytical model of the performance of this algorithm can be easily derived by looking at a time line of processor activity. For this time line, the assumption is

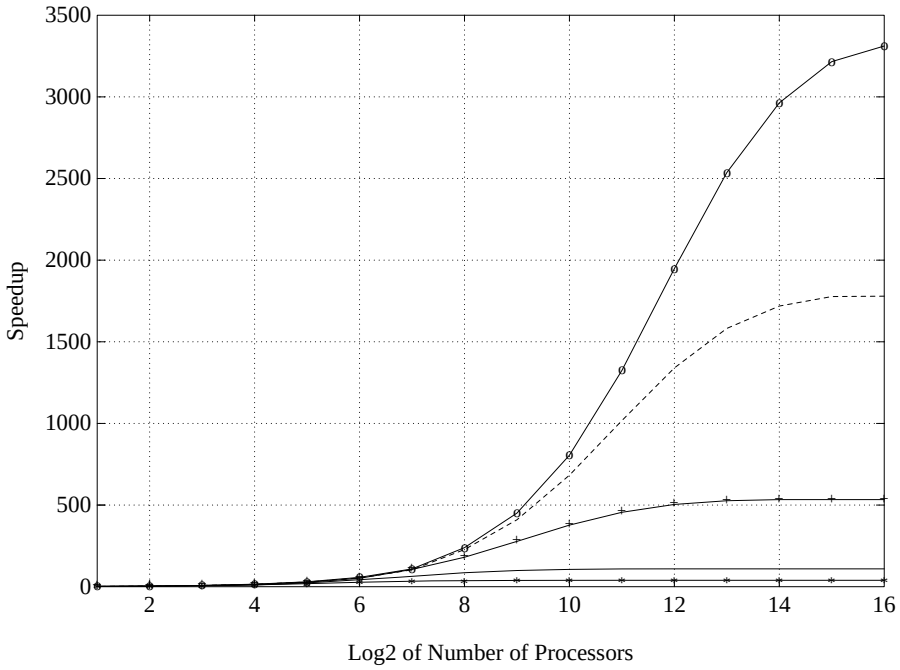

Figure 4-8. Model 2 Vertical Partitioning Speed-Ups

made that a broadcast ties up all involved processors for a fixed period of time and that all processors are available to receive the broadcast when it is initiated. (As was seen in the previous chapter, near neighbors will actually finish the broadcast reception earlier than other, but the difference is not deemed to be significant). The produced time line is shown in Figure 4-18. In this figure, each processor's activities are represented within a distinct column (four processors are shown in this figure). Solid lines indicate ongoing computation and dashed lines indicate communication and their associated latencies.

Using this model and assuming that the column update times are approximately equal for each involved processor, we see that calculation of the multipliers is not overlapped with any other useful activity and can be modeled as a sequential activity in time. (This observation is the motivation for more efficient pipelined versions of the method). Hence, as only updating of the columns is done in parallel, the execution time model is:

$$t_{exec} = t_m + t_b + \frac{t_u}{P}$$

where t_m is the time to compute the multipliers, t_b is the time required for the broadcasts, t_u is the time for all updates, and P is the number of processors being used. Each of the component times can be defined in terms of the matrix parameters using formulas similar to those derived from the experiments on the nCUBE 2. Thus, using r, c, and k for the number of rows, columns, and pivots for the matrix, the

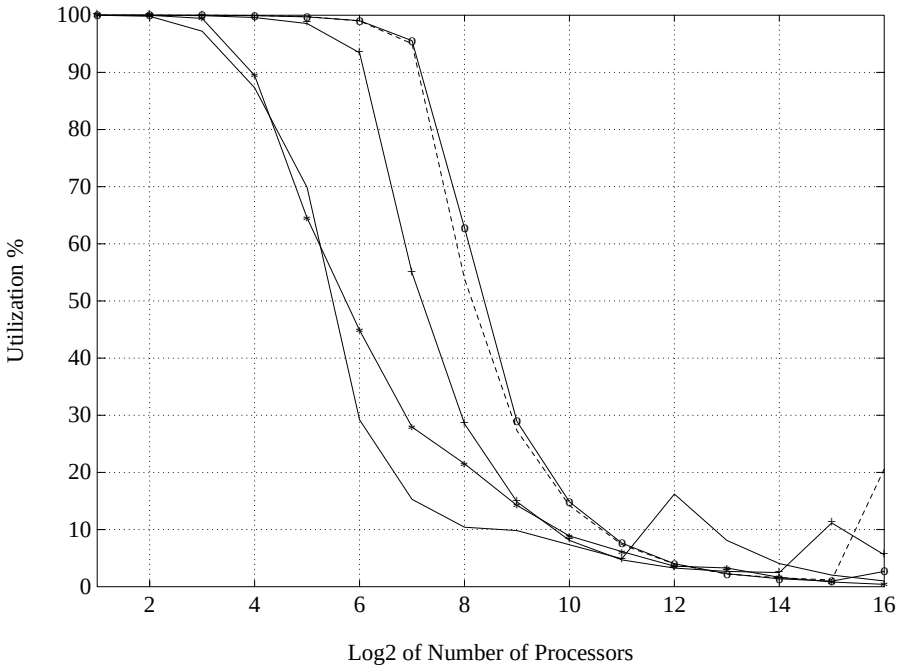

Figure 4-9. Model 2 Original Utilizations

following formulas can be derived:

$$t_m = 4 * \Sigma_{i=1}^{k}(c-i) + 5 * k$$

$$t_b = \Sigma_{i=1}^{k}[195.22 + 50.02 * \log_2(P) + 4 * 0.122 * (r-i) + 4 * 0.45 * \log_2(P) * (r-i)]$$

$$t_u = 5 * \Sigma_{i=1}^{k}(c-i)(r-i) + 17 * k + 3 * k * c.$$

Similarly, if numerical pivoting were to be introduced, it would be handled in a way very similar to that in which the multipliers are done and could be defined as:

$$t_p = 4 * \Sigma_{i=1}^{k}(c-i+1) + 5 * k$$

with execution time becoming:

$$t_{exec} = t_m + t_p + t_b + \frac{t_u}{P}.$$

Furthermore, a sequential version of the algorithm can be simply modeled with:

$$t_{exec} = t_m + t_u$$

or

$$t_{exec} = t_m + t_p + t_u$$

if numerical pivoting is incorporated.

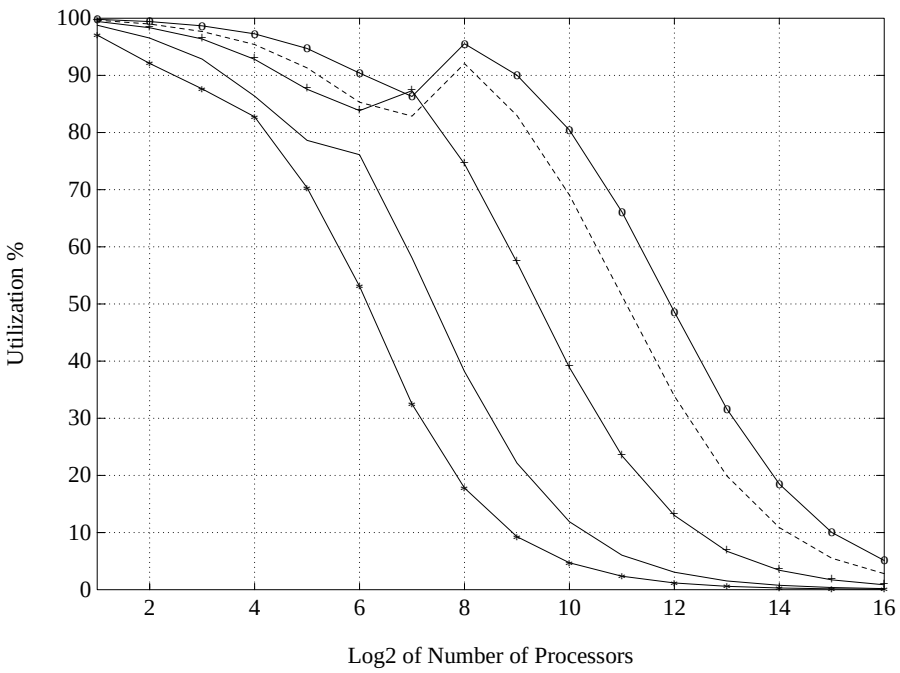

Figure 4-10. Model 2 Vertical Partitioning Utilizations

Dense Matrix Factoring Results

Timings of the dense matrix factorization algorithm were obtained using *etool* for a variety of matrix sizes, shapes, and number of pivots. The results were analyzed and several trends identified. Then, the same parameters were fed into the analytical models for validation purposes.

Representative empirical results from the factorization of several matrix configurations are provided below (Table 4-6) for hypercubes of dimensions 0 to 5. These results are provided in terms of speed-ups by dividing the multiprocessor results (dimensions 1 to 5) by the uniprocessor (dimension 0) results.

Several conclusions are readily apparent from this data:

1. Speed-up (and efficiency) decrease with more pivots. This occurs as there are fewer computations with later pivots with which to amortize the communication latencies.

2. There is a point at which additional processors add to overall latency (reduce speed-ups). This happens as broadcast times grow significantly with larger dimension hypercubes and can be seen for the 32 X 32 X 32 configuration with dimension 4 and 5 hypercubes in Table 4-6.

3. High efficiencies are achieved only with very limited numbers of processors and relatively large problems.

Other interesting observations are possible by looking at the results for rectangular matrices (which are important to the unsymmetric-pattern multifrontal method).

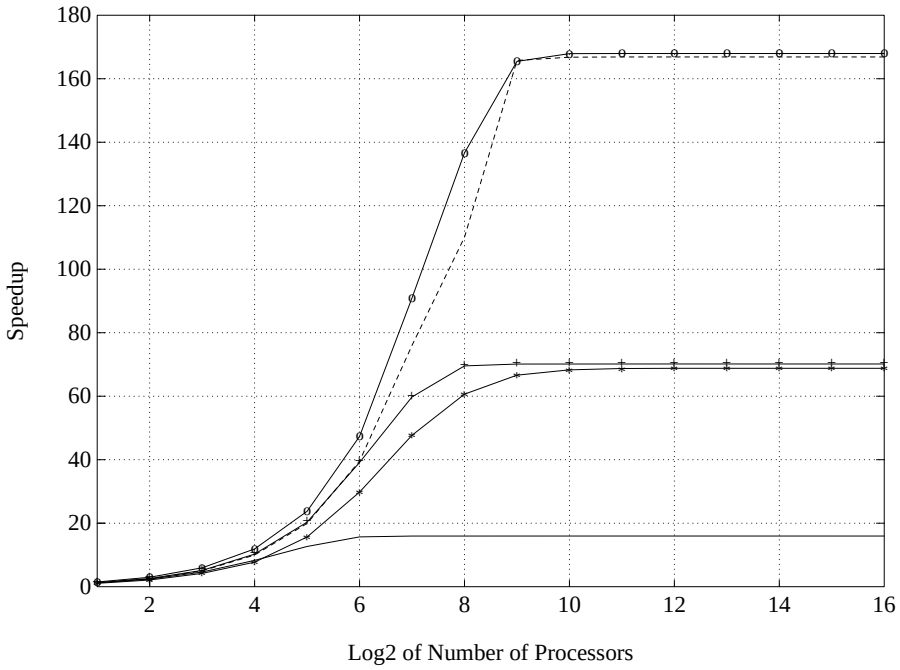

Figure 4-11. Model 3 Speed-Up Results

For example, by observing the results for the relatively tall and narrow (512 X 32) and short and wide (32 X 512) matrices of Table 4-7, one can see a striking difference in performance. This is easily explained as the column-oriented method in use broadcasts columns of data that are much larger in the 512 X 32 case and since the essentially sequential multiplier determination time (t_m) is proportional to the row size of the matrix. This suggests the possibility that row-oriented versions of the algorithm would be desirable for the tall and narrow frontal matrices and motivates consideration of alternative dense factorization methods within an overall distributed memory multifrontal implementation.

Finally, the actual timing obtained through experimentation is compared with the results of the corresponding analytical models. This will be done in two ways. First the actual observed and predicted times are compared and second the speed-ups based strictly on observed times are compared to those based strictly on predicted times.

The times observed from the experiments and those produced by the analytical models are provided in Table 4-8 for several matrix configurations. (Results from the experiments are annotated with "(e)" and those from the analytical models with "(m)" and all times are in milliseconds).

The corresponding speed-ups for the same configurations are provided in Table 4-9.

From these results it is readily apparent that the analytical model under-predicts execution time especially in the smaller configurations and over-predicts speed-up. Due to this combination, there is some overhead in the actual computations that is

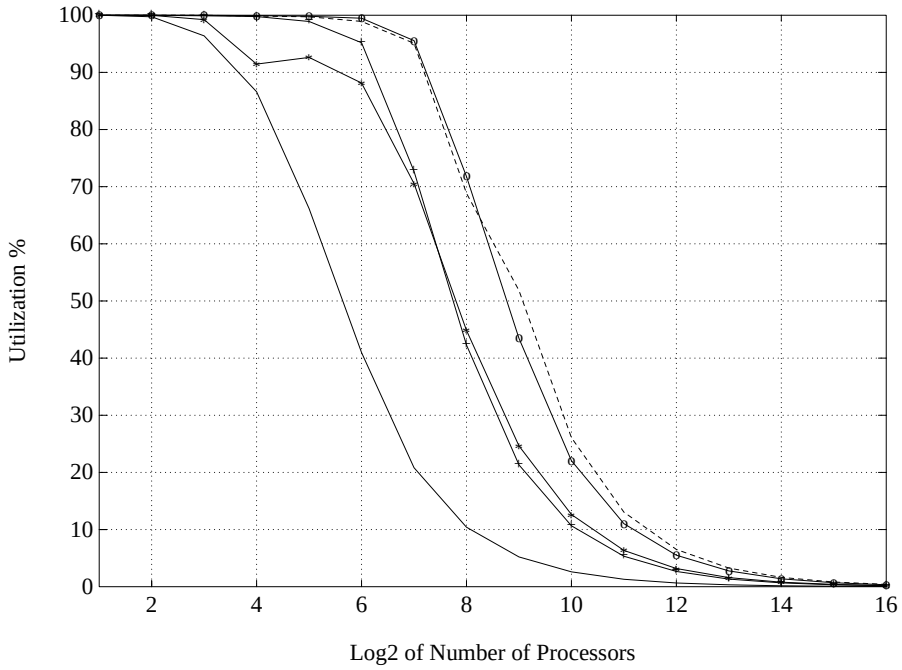

Figure 4-12. Model 3 Utilization Results

not being properly accounted for in the analytical model. However, the speed-ups are not too far off and the analytical model should provide fairly accurate results when integrated into the simulation model.

A common observation throughout this section is that the dense matrix factorization algorithm is not very efficient for hypercubes of dimension greater than 3 on the matrix configurations shown. Alternative algorithms will be explored as the efficiency and performance of a distributed memory, unsymmetric-pattern multifrontal method should be very closely tied to the performance of the dense matrix operations.

4.3.2 Distributed Memory Multifrontal Simulation Model

With a reasonable characterization of the dense matrix factorization process achieved in the last section, the bounded parallelism PRAM simulation model described earlier can be revised into a distributed memory implementation model with performance characteristics similar to those of the nCUBE 2. In order to revise the simulation model, the following specific changes will be required:

- Replace the node weight (frontal task's execution time) with the execution time obtained from the analytical model of the previous discussion.

- Account for passing of contribution blocks between frontal matrices with edge weights in the graph. Specific edge weights will be defined using the performance equations derived from experimentation on the nCUBE 2's point to point message passing capabilities.

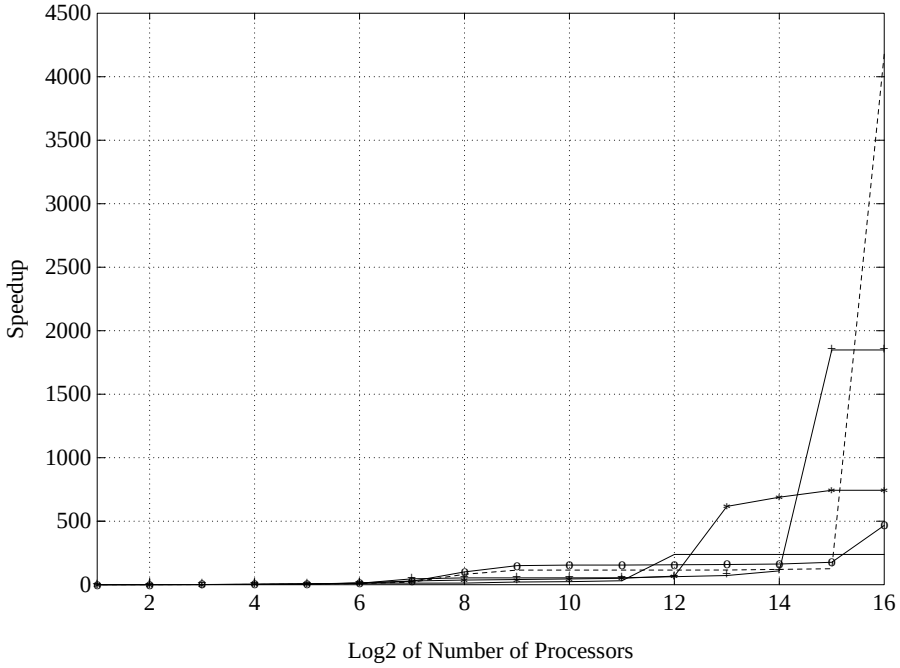

Figure 4-13. Model 4 Original Speed-Ups

- Modify the processor allocation schemes to allocate processors as complete subcubes.
- Develop initial algorithms for the scheduling and allocating processors to frontal tasks.

From the earlier results of the bounded parallelism PRAM simulations and the characteristization of the nCUBE 2 multiprocessor, an appropriate strategy for parallel implementation would be blocked (column-oriented) parallelism within frontal matrices and concurrency between independent frontal matrices. Such an approach is taken. Furthermore, it is necessary to make several other assumptions for revision of the model. These added assumptions are:

- Assembly will consist of both the communication of contributions between frontal matrices (done as point to point message passing) and the numerical assembly (addition) of contributions to the corresponding frontal matrix entry. Both components will initially be done sequentially within each frontal matrix.
- Processor allocations will be done in full subcubes (i.e., in processor sets that are a power of 2) but specifically which processors are assigned to a particular task will not be tracked. Hence, there is an implicit assumption that available processors form a subcube, that is, no fragmentation of the hypercube occurs that would hinder allocations.

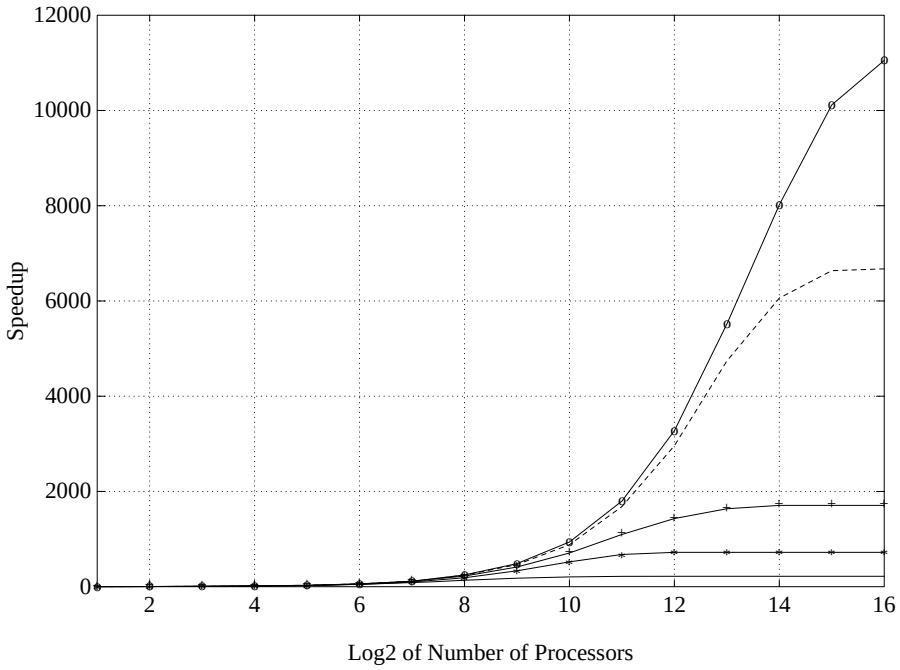

Figure 4-14. Model 4 Vertical Partitioning Speed-Ups

- As the latency of point to point message passing is dependent upon the path distance of the transmission and as specific processor allocations will not be tracked, a set distance of 10 links will be assumed for all communicating of contributions. Such a setting will insure an upper bound on this latency for all hypercubes of 1024 processors or less.

- Checking of numerical pivots will be accounted for by a column-based, partial pivoting scheme within the node weight function. However, the assumption is made that no anticipated pivots will be lost during the factorization process.

Node and Edge Weight Models

Time will be tracked in the distributed memory simulation model using the following node and edge weights for task and communication times respectively.

Node weights will be assigned initially based upon a sequential node execution model and then revised to a parallel time upon allocation of a specific set of processors to the node (task). The sequential node model is:

$$t_{exec} = t_a + t_m + t_p + t_u$$

where t_m, t_p, and t_u are the multiplier, pivoting, and updating times respectively as defined earlier in the dense matrix factorization model and t_a is the time for assemblies defined as:

$$t_a = 8 * A + \Sigma_{all\_in\_edges}[204 + 0.573(8 * A_e)]$$

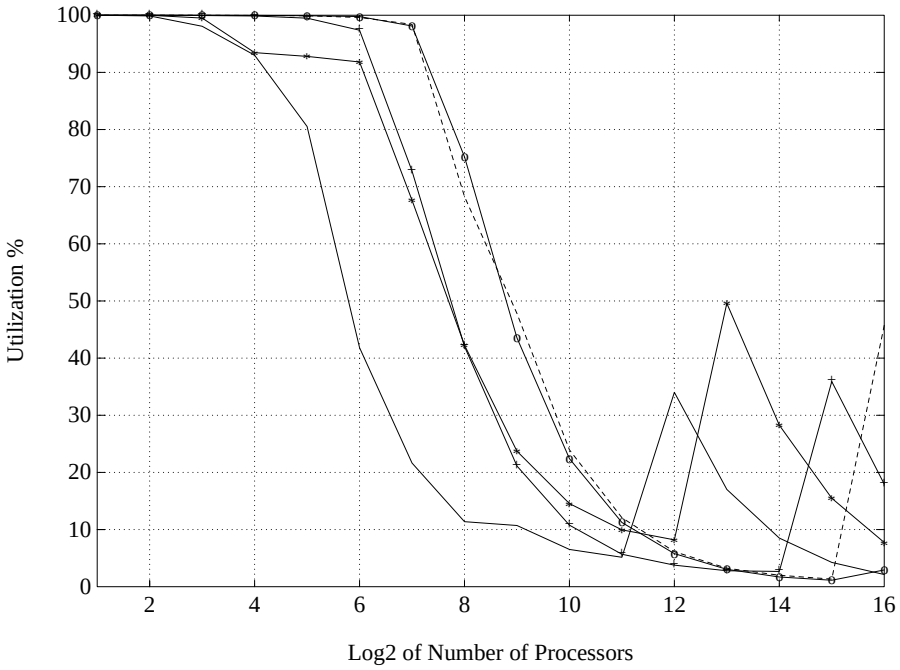

Figure 4-15. Model 4 Original Utilizations

where A is the total number of contributions and A_e is the number of contributions passed on a particular edge e. This formula is based on the point to point message passing latency experiments described earlier and an assumption of 8 bytes per entry to be assembled (4 bytes for index, 4 bytes for value). This second term (the summation) includes the edge weight definition.

The parallel node execution time model then becomes:

$$t_{exec} = t_a + t_m + t_p + \frac{t_u}{P}$$

where P is the number of processors assigned to the node. This parallel node execution time model will also be referred to in the next section as $T\_Para(ID, P)$; that is, the parallel time associated with task ID when using P processors.

Scheduling and Processor Allocation

For scheduling, a priority list scheme is used with priority assigned to tasks using node and edge weights with a critical path method. The node weights used are based on sequential execution time.

Processor allocation is quite challenging as allocating more processors to a task will cause it to execute faster (up to some limit) but will introduce more inefficiencies, as seen in the earlier dense matrix factorization experiments. Furthermore, such inefficient use of processors may preclude their more efficient use on other distinct nodes (tasks). Thus, for processor scheduling we define two quantities to be associated with each node: $P\_desired$ and P_most. $P\_desired$ is the preferred number of

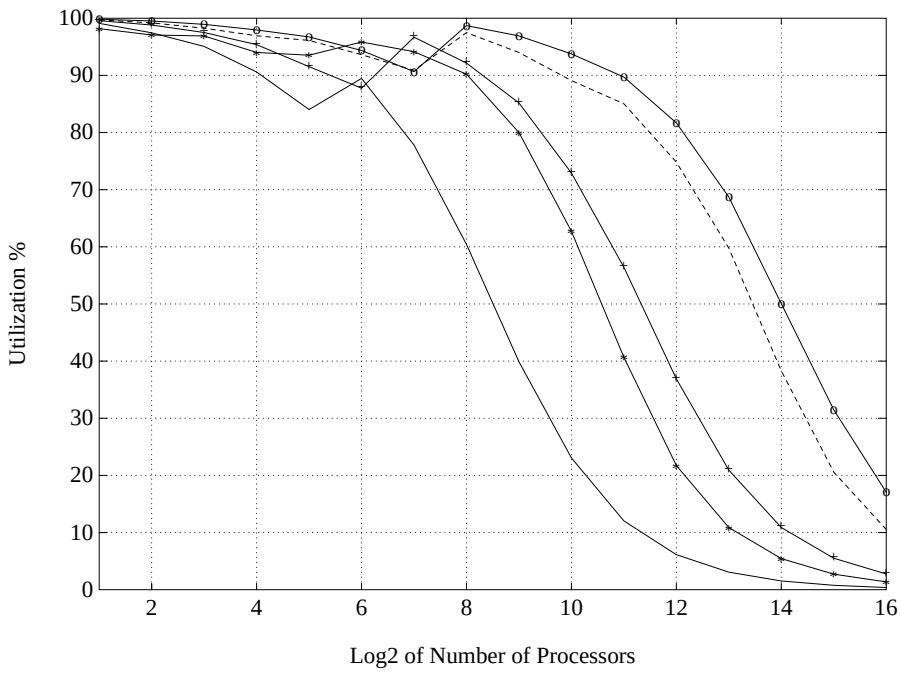

Figure 4-16. Model 4 Vertical Partitioning Utilizations

processors to assign to the node. P_most is the maximum number of processors to ever assign to a node. P_most is necessary due to the execution time increase seen when too many processors are assigned to a dense factorization task and due to the column-oriented nature of the dense factorization algorithm (that is, the number of columns provides an upper bound on the number of processors).

Determination of $P\_desired$ is essentially a question of efficiency and should be based both on the task characteristics and the number of available processors. That is, if there is a very large number of processors in the configuration, we can afford to be more inefficient in our allocation since the extra processors would likely go idle otherwise. Hence, a target efficiency is defined and $P\_desired$ is set via our analytical models to achieve an efficiency near the target. Furthermore, this target efficiency is set to different values based on the number of total number of processors in the system. Some initial experiments revealed that reasonable target efficiencies would be 0.8 for hypercube of dimension 3 or less (8 or fewer processors), 0.5 for hypercubes of dimensions 4 to 7 (16 to 128 processors), and 0.2 for hypercubes of dimension 8 or larger (256 or more processors).

Given these values for target efficiency, the value for $P\_desired$ is found by solving the following for P:

$$\frac{T_{sequential}}{P * T\_Para(ID, P)} = Target\_Efficiency.$$

Each processor executes the following:

do $k = 0, num\_of\_pivots - 1$
    if (column k is local) then
        do $i = k + 1, n - 1$
            compute multipliers $\lambda_{ik} = a_{ik}/a_{kk}$
        broadcast the column just computed
    else
        receive the column just computed
    endif
    for (all columns $j > k$ that are local) do
        do $i = k + 1, n - 1$
            $a_{ij} = a_{ij} - \lambda_{ik} a_{kj}$
        enddo
    endfor
enddo.

Figure 4-17. Distributed Memory Fan-Out Factorization

This turns out to be a nonlinear equation with $Plog_2(P)$ and P terms. Thus to simplify its solution $Plog_2(P)$ is replaced by P^2 and the conservative result is then rounded up to the nearest encompassing subcube. Table 4-10 below provides sample results from this calculation for the various target efficiencies:

The determination of P_most on the other hand simply involves the calculation of the value of P at which parallel execution time begins to increase. As parallel execution time is a function of row size, column size, number of pivots and number of processors, we can simply take the partial derivative of this equation with respect to the number of processors and solve for the value of P that zeros the derivative. The function for parallel time is:

$$T_{parallel}(r, c, k, P) = t_m(r, k) + t_p(r, k) + t_b(r, k, P) + \frac{t_u(r, c, k)}{P}$$

where t_m, t_p, t_b, and t_u are component functions for multipliers, pivoting, broadcasts, and updates (as defined earlier) and r, c, and k are row size, column size, and number of pivots. Thus, solving

$$\frac{\partial}{\partial P} T_{parallel} = 0$$

one gets

$$P = \frac{t_u(r, c, k) * \ln 2}{50.02 * k + (1.8 * k(r - \frac{k+1}{2}))}.$$

Once P has been determined, it is rounded down to the nearest power of 2. Then, $T\_Para(ID, P)$ and $T\_Para(ID, 2 * P)$ are evaluated and P_most assigned either P

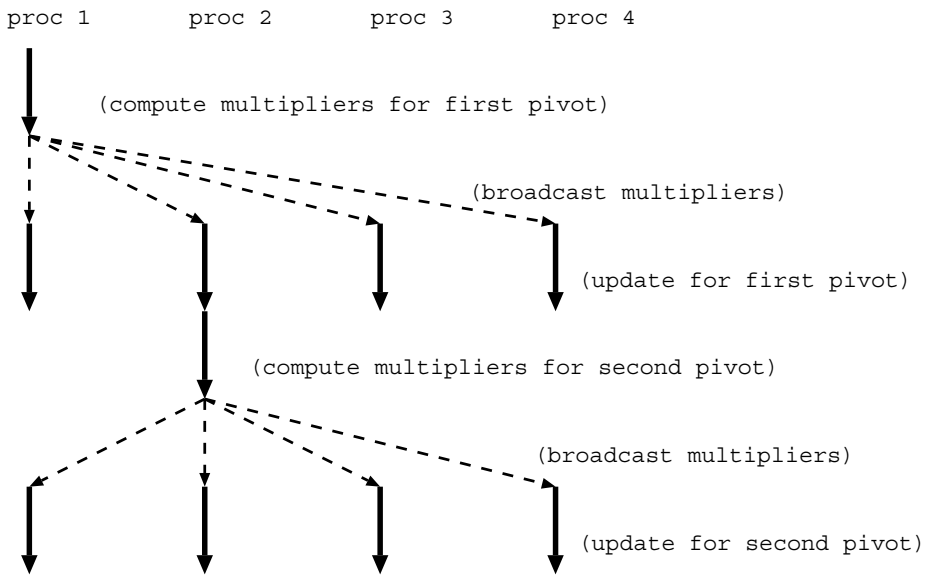

Figure 4-18. Time Line for Partial Dense Factor Routine

or $2 * P$ based on which produces the lesser execution time. As a column-oriented dense factorization algorithm is used, *P_most* is also restricted to be no larger than the smallest power of 2 that is greater than or equal to the number of columns.

As an example of what this calculation produces, all of the configurations in the *P_desired* example of Table 4-10 are limited to column size except for the 16 X 16 X 2 case, which is limited to 8 processors.

With *P_desired* and *P_most* now defined for each task node, the processor allocation and scheduling can be carried out with the decision sequence defined in Figure 4-19. This decision sequence is repeated until all of the system's processors have been allocated or until a task does not meet the allocation criteria. (In this latter, case it may be possible to allocate other tasks with lesser priority and achieve a better overall performance, but this extension produced significantly more run time overhead with only moderate improvements).

Test Matrices

With the updates just described made to the simulation program, traces of assembly DAGs from the latest version of the unsymmetric-pattern multifrontal method were used as input [35]. This latest version of the method, called *afstack*, produced assembly DAGS with significantly better parallelism than seen in the earlier DAGs used in the unbounded and bounded parallelism PRAM models. This better parallelism was reflected in DAGs with lower depth ratios (topological levels in the DAG divided by the number of nodes in the DAG), and fewer assemblies as a percentage of overall node weight.

Table 4-6. Empirical Factoring Speed-Ups

Rows X Cols X Pivots	1-Cube (P=2)	2-Cube (P=4)	3-Cube (P=8)	4-Cube (P=16)	5-Cube (P=32)
32 X 32 X 1	1.55	2.39	3.18	3.63	3.74
32 X 32 X 8	1.65	2.40	2.96	3.32	3.32
32 X 32 X 32	1.44	1.81	1.93	1.86	1.73
64 X 64 X 1	1.89	3.36	5.39	7.48	8.95
64 X 64 X 8	1.88	3.28	5.17	7.28	8.82
64 X 64 X 32	1.84	3.11	4.63	5.93	6.66
64 X 64 X 64	1.78	2.85	3.92	4.63	4.94
128 X 128 X 1	1.96	3.69	6.64	10.84	15.37
128 X 128 X 8	1.95	3.69	6.61	11.14	16.41
128 X 128 X 64	1.94	3.60	6.25	9.74	13.25
128 X 128 X 128	1.19	3.49	5.81	8.55	10.88

Table 4-7. Rectangular Matrix Speed-Ups

Rows X Cols X Pivots	1-Cube (P=2)	2-Cube (P=4)	3-Cube (P=8)	4-Cube (P=16)	5-Cube (P=32)
512 X 32 X 8	1.82	3.03	4.55	6.35	7.69
32 X 512 X 8	1.98	3.87	7.33	13.21	21.53

Specific characteristics of the matrices tested are provided in Table 4-11. The results for DAGs produced by the older version of the method are included in parentheses (where they were available).

Simulation Results

The simulation results are provided in Tables 4-12 and 4-13 below in terms of speed-ups achieved for hypercubes of dimensions 1 through 10 (processor sets of 2 to 1024). All maximum speed-ups were achieved with processor sets in this range.

Noticeable in these results is that matrices where a larger percentage of their node weight is attributed to assembly had a worse performance. This is consistent with the model assumption of purely sequential assemblies within the node tasks. As it is quite possible to perform at least some fraction of the assemblies concurrently within a node task, a revision to the model is justified.

The revision was completed by redefining assembly time so that numerical assemblies were done fully in parallel by the number of processors assigned to the frontal matrix task. Thus, t_a became:

$$t_a = \frac{8 * A}{P} + \Sigma_{all\_in\_edges}[204 + 0.573(8 * A_e)].$$

Table 4-8. Empirical Versus Analytical Times

Rows X Cols X Pivots	0-Cube (P=1)	1-Cube (P=2)	2-Cube (P=4)	3-Cube (P=8)	4-Cube (P=16)	5-Cube (P=32)
32 X 32 X 8 (e)	33.5	20.3	14.0	11.3	10.1	10.1
32 X 32 X 8 (m)	32.3	19.1	12.0	8.9	7.7	7.6
64 X 64 X 1 (e)	21.4	11.3	6.4	4.0	2.9	2.4
64 X 64 X 1 (m)	20.3	10.7	5.8	3.5	2.4	1.9
128 X 128 X 64 (e)	3165.8	1632.9	879.4	506.9	325.3	238.9
128 X 128 X 64 (m)	3078.1	1581.1	832.0	464.5	287.9	206.7

Table 4-9. Empirical Versus Analytical Speed-Ups

Rows X Cols X Pivots	1-Cube (P=2)	2-Cube (P=4)	3-Cube (P=8)	4-Cube (P=16)	5-Cube (P=32)
32 X 32 X 8 (e)	1.65	2.40	2.96	3.32	3.32
32 X 32 X 8 (m)	1.69	2.69	3.63	4.17	4.28
64 X 64 X 1 (e)	1.89	3.36	5.39	7.48	8.95
64 X 64 X 1 (m)	1.90	3.49	5.84	8.50	10.54
128 X 128 X 64 (e)	1.94	3.60	6.25	9.74	13.25
128 X 128 X 64 (m)	1.95	3.70	6.63	10.69	14.89

With parallel execution time thus redefined by the new t_a, corresponding revisions were needed and made to the formulas for P_desired and P_most. The results of this revision to the model are shown in the Tables 4-14 and 4-15.

4.4 Conclusions

While the results may not be outwardly spectacular, they are actually very promising when compared to a similar simulation study based on a distributed memory version of the classical multifrontal method [122]. In this study by Pozo, speed-ups for the BCSSTK24 matrix were 29.27 based on an iPSC/2 hypercube and 23.35 based

Table 4-10. Sample P_Desired Calculations

Rows X Cols X Pivots	Eff=0.8	Eff=0.5	Eff=0.2
16 X 16 X 2	P=1	P=4	P=8
32 X 32 X 8	P=2	P=8	P=16
64 X 64 X 8	P=4	P=16	P=32
128 X 128 X 64	P=8	P=16	P=32

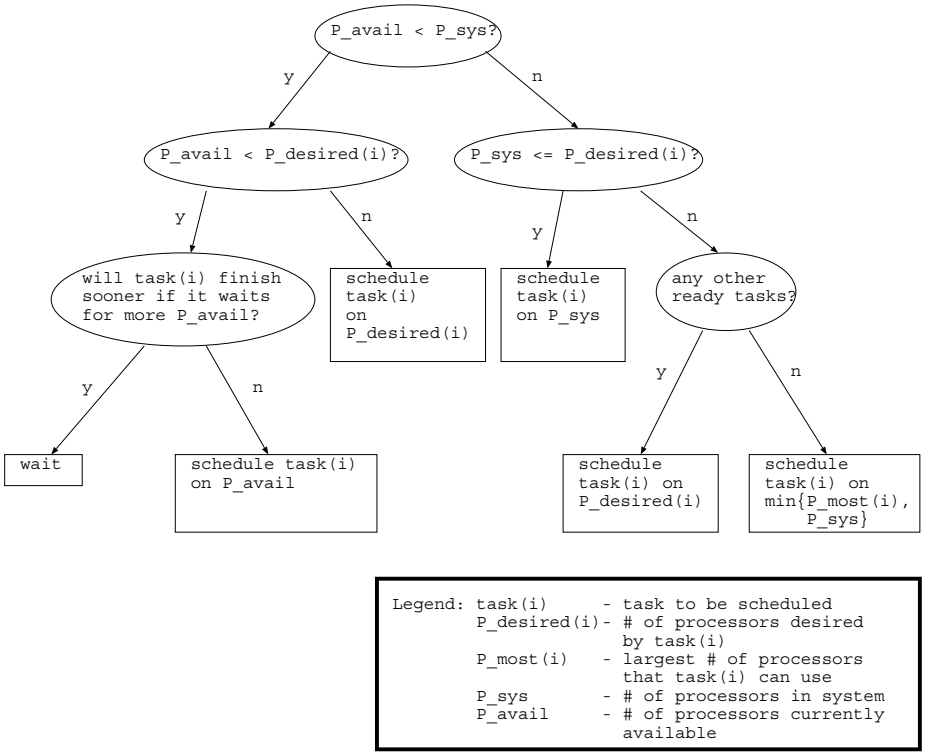

Figure 4-19. Scheduling Decision Diagram

on an iPSC/860 hypercube. These figures compare to 32.67 and 43.99 (sequential assembly and parallel assembly respectively) for the nCUBE 2 based simulations studies documented here. Insufficient specifics were provided in the referenced article to fully assess the relative differences in assumptions between my simulation model and theirs, but from the details that were provided, my assumptions appear to be more conservative. Thus, I have some confidence in the significance of these results.

It is important to keep in mind that these speed-up figures are relative to a sequential version of the same algorithm. Yet, as both the classical and unsymmetric-pattern multifrontal methods are competitive when run sequentially, the speed-ups as reported here are still significant.

Chapter Summary: The purpose of this chapter has been to investigate the theoretical and achievable parallelism of the unsymmetric-pattern multifrontal method for sparse LU factorization. This investigation was based on the assembly DAG produced by the method during and analyze-factorize operation. The initial analytical models showed that significant parallelism was available if the parallelism both between and within nodes of the assembly DAG was exploited. When medium grain parallelism was used within frontal matrices, speed-ups as high as 167 are possible on the relatively modest sized test problems. Furthermore, when fine grain parallelism was used within frontal matrices, the theoretical speed-ups increased to as much as 6,868 and 11,425 for two of the test matrices.

Table 4-11. Test Matrix Characteristics

MATRIX	ORDER	NONZEROS	DEPTH RATIO	% OF ASSEMBLIES
MAHINDASB	1258	7682	0.031 (0.29)	8.64% (20.0%)
GEMAT11	4929	33185	0.014 (0.02)	6.19% (10.4%)
GRE_1107	1107	5664	0.051 (0.08)	4.97% (17.6%)
LNS_3937	3937	25407	0.057 (0.18)	10.41% (32.9%)
SHERMAN5	3312	20793	0.005 (0.07)	2.21% (20.8%)
RDIST1	4134	94408	0.075	1.97%
BCSSTK24	3562	159910	0.085	2.08%

Table 4-12. Sequential Assembly Version Results - Part 1

MATRIX	P=2	P=4	P=8	P=16	P=32
MAHINDASB	1.30	1.75	2.89	3.32	3.39
GEMAT11	1.65	3.18	5.68	7.95	11.49
GRE_1107	1.77	3.23	5.07	7.68	10.21
LNS_3937	1.62	2.73	4.36	6.04	7.29
SHERMAN5	1.87	3.20	5.96	9.49	12.64
RDIST1	1.82	3.23	5.83	8.65	16.25
BCSSTK24	1.89	3.47	6.25	9.85	15.22

Whereas the initial analytical models were based on an assumption of an unbounded number of available processors, the simulation models that followed assumed a bounded number of processors. The results of these models showed that the theoretical parallelism seen in the analytical models was achievable on realistically sized processor sets. Specifically, the speed-ups of up to 167 seen when using medium grain parallelism with nodes and parallelism across nodes were achievable with a maximum of 512 processors. Moreover, when the parallelism within nodes was changed to fine grain, the corresponding speed-ups of up to 6,868 and 11,425 were achievable on no more than 65,536 processors.

The final objective of this chapter was then to revise the simulation models to represent an existing architecture and estimate how much of the theoretical parallelism was actually achievable. The earlier performance characterization of the nCUBE 2 multiprocessor was used to set the simulation parameters and the model

Table 4-13. Sequential Assembly Version Results - Part 2

MATRIX	P=64	P=128	P=256	P=512	P=1024
MAHINDASB	3.39	3.39	3.43	3.43	3.43
GEMAT11	12.48	12.60	14.01	14.02	14.02
GRE_1107	11.78	12.09	12.81	12.81	12.81
LNS_3937	7.93	8.11	8.87	8.88	8.88
SHERMAN5	18.62	20.22	22.85	22.98	22.98
RDIST1	22.29	22.50	26.08	26.08	26.16
BCSSTK24	26.57	32.67	38.87	39.29	39.29

Table 4-14. Parallel Assembly Version Results - Part 1

MATRIX	P=2	P=4	P=8	P=16	P=32
MAHINDASB	1.31	1.81	2.14	4.14	4.29
GEMAT11	1.68	3.21	6.05	8.49	12.37
GRE_1107	1.82	3.29	5.54	8.71	12.84
LNS_3937	1.75	3.10	5.13	7.83	10.16
SHERMAN5	1.91	3.34	6.07	11.00	17.95
RDIST1	1.87	3.40	5.96	9.38	16.16
BCSSTK24	1.93	3.67	6.72	11.28	18.01

revised to correspond to a distributed memory environment. The results of the distributed memory model showed that 20 to 25 percent of the theoretical parallelism was typically achievable. Furthermore, performing the assemblies in parallel produced significant benefits.

With the prospective of achieving significant parallel performance established by the models of this chapter, the next step is to actually implement a distributed memory, unsymmetric-pattern multifrontal method. Chapter 5 begins the implementation by developing and analyzing a number of highly tuned partial dense factorization kernels, which will account for most of the computational workload.

Table 4-15. Parallel Assembly Version Results - Part 2

MATRIX	P=64	P=128	P=256	P=512	P=1024
MAHINDASB	4.29	4.29	4.41	4.41	4.41
GEMAT11	15.17	16.47	17.89	17.96	17.96
GRE_1107	15.32	15.91	17.02	17.02	17.02
LNS_3937	15.35	16.83	18.37	18.45	18.45
SHERMAN5	24.95	29.55	32.39	32.64	32.65
RDIST1	23.24	27.20	32.02	32.02	32.09
BCSSTK24	31.88	43.99	53.49	58.63	58.66

CHAPTER 5
PARTIAL DENSE FACTORIZATION KERNELS

The primary computational function performed within an unsymmetric-pattern multifrontal method is the partial factorization of dense submatrices (frontal matrices). The term partial factorization refers to the application of only a limited number of pivot steps for each frontal matrix. The remainder of the matrix entries are updated by the pivot steps but not factored themselves and will be passed on and assembled into other frontal matrices where they may be further updated and eventually factored.

This chapter develops a number of routines to perform this partial dense factorization (PDF) function. A basic column-scattered, fan-out algorithm is first developed. This algorithm is improved upon by use of optimized vector functions from the Basic Linear Algebra Subroutines (BLAS 1) and by the introduction of pipelining. Numerical considerations are then incorporated into both the basic and pipelined algorithms with a major emphasis placed on finding alternative pivots in other columns when anticipated pivots are numerically unacceptable. A parallel prefix operation among the participating processors provides an efficient, consistent, and deterministic recovery mechanism. A special streamlined algorithm is developed for the single pivot case. The performance of all the developed algorithms is measured and analyzed. Specific performance effects of the various enhancements are reported. Finally, performance prediction formulas are derived from analytical models of the routines and compared against observed performance. These predictions will be useful in the next chapter when task scheduling allocation are addressed.

The strategy of a column-scattered allocation of frontal matrices is adopted for all of the partial dense factorization kernels to be discussed in this chapter. There are a number of reasons for this. First, the column-scattered allocation provides good load balancing. Also, it provides easy incorporation of partial pivoting strategies and allows the location of entries to be easily determined.

5.1 Basic Algorithms

The first two partial dense factorization kernels assume that the diagonal elements of the pivot block are numerically acceptable and the kernels use them as pivots without any numerical checking. The assumption of numerically acceptable pivots on the diagonal is consistent with matrices with properties such as diagonal dominance. The first such kernel is called P1 and uses a simple fan-out approach. The second is called P2 and improves performance by the use of pipelining.

5.1.1 P1 – Basic Fan-Out Algorithm

The first algorithm implemented is the basic fan-out column-scattered method introduced earlier in Section 2.6. This method has each participating processor loop through all of the frontal matrix's pivots. If the processor owns the current pivot, it will compute and broadcast the multipliers (column of L corresponding to that pivot). If the processor does not own the current pivot, it will wait to receive the multipliers from the owner processor. Following completion of the broadcast, all processors will update the locally held columns of the active submatrix. An outline of this algorithm, referred to as P1, is provided in Figure 5-1. A major drawback of this algorithm is that all of the processors that do not own the current pivot are essentially idle during the computation of multipliers and subsequent broadcast initiation. While in a larger context this idle time could be used for other processing (such as completing assemblies), a more desirable situation would be to develop a more efficient factorizing algorithm.

P1 ALGORITHM:

```
(Basic fan-out based on a column scattered allocation)

for ( pivots 1 thru number of pivots ) do
    if ( next pivot column is local to this processor )
        - compute multipliers
        - broadcast multipliers to subcube for frontal
    else
        - receive broadcast
    endif
    - update remaining active local columns using
        the current multipliers
endfor
```

Figure 5-1. P1 – Basic Fan-Out Algorithm

5.1.2 P2 – Pipelined Fan-Out Algorithm

The second algorithm uses the concept of pipelining to minimize idle time and overlap computation with communication. This algorithm was originally suggested by Geist and Heath [62]. In this pipelined version, once the processor that owns the next pivot receives the multipliers for the current pivot, it updates only the next pivot column and then computes and broadcasts the multipliers for that pivot. Only after the broadcast of these subsequent multipliers has been initiated, will the rest of the updates for the current pivot be done. In this way, the multipliers for the next pivot will be already computed and distributed by the time the other processors are ready for them. The effect is to overlap the updating due to the current pivot with

the computation of multipliers for the next pivot and thus preclude much of the idle time experienced with the basic fan-out algorithm. An outline of the basic pipelined algorithm (P2) is provided in Figure 5-2.

P2 ALGORITHM:

```
(Pipelined fan-out based on a column scattered allocation)

if ( first pivot column is local to this processor )
    - compute multipliers
    - broadcast multipliers
    - save a ptr to these next multipliers
endif
for ( all potential pivot columns ) do
    if ( current pivot column is not local to this processor )
        - receive broadcast from pivot owner
    else
        - adjust ptr to use saved next multipliers
    endif
    if ( next pivot column is local to this processor )
        - update next pivot column with received multipliers
        - compute multipliers
        - broadcast multipliers
        - save a ptr to these next multipliers
    endif
    - update remaining active columns of the frontal matrix
endfor
```

Figure 5-2. P2 – Pipelined Fan-Out Algorithm

5.2 Use of BLAS 1 Routines

As most of the computations required by a partial dense factorization algorithm entail repeated use of simple vector operations, performance improvements in these operations will significantly enhance the performance of the overall factorization routine. The Basic Linear Algebra Subroutines 1 (BLAS 1) provide these optimized vector operations [110, 111]. Furthermore, these routines are available for most high performance computers and have been specifically tailored for their host architectures. Thus, the BLAS provide both high performance and portability.

The original motivation of the BLAS 1 routines was to exploit the high performance available from vector processing units. While the nCUBE 2 processors are not equipped with vector processing units, it is possible to significantly improve performance of BLAS 1 routines by making efficient use of the nCUBE 2's instruction cache, many addressing modes, and CISC-style instructions. The assembly coded

BLAS 1 routines available on the nCUBE 2 do this and achieve a performance that is 3.3 to 7.8 times faster than comparable C code.

A number of BLAS 1 routines were integrated into the first two algorithms and will be used in the subsequent algorithms. Specifically, the *Saxpy* (*Daxpy* for double precision) routine does a vector-scalar product added to another vector ($\bar{y} \leftarrow \bar{y} + \alpha\bar{x}$) and is used to perform the outer product update (by columns) of the active submatrix for each pivot. The *Sscal* (*Dscal* for double precision) routine scales a vector by multiplying it by a scalar term ($\bar{x} \leftarrow \alpha\bar{x}$). The Sscal routine is used to calculate the multipliers (column of L) for a particular pivot. These routines improved of the algorithm's execution time by 2 to 3.67 times with the larger speed-ups occurring with the larger matrices.

In the next algorithms where threshold partial pivoting is incorporated, three other BLAS 1 routines will be useful. The *Isamax* (*Idamax* for double precision) routine finds and returns the index of the largest magnitude entry in a specified vector. The *Sswap* (*Dswap* for double precision) routine swaps the location of two vectors. Finally, the *Scopy* (*Dcopy* for double precision) makes a copy of a vector. All of these BLAS 1 routines allow non-unit stride (adjacent vector entries need not be adjacent in memory, but do need to be a constant distance apart). This enables easy access to matrix rows in a column-major storage scheme (as well as columns in a row-major scheme). Furthermore, with most of the nCUBE 2 BLAS 1 routines there is no performance difference between unit and non-unit stride operations.

5.3 Inclusion of Row and Column Pivoting

While the basic concept of a partial factorization algorithm is clear and easily understood, numerical considerations become more challenging. In particular, pivot choices are limited to the leading principal submatrix with order equal to the number of pivot steps to be applied. This submatrix is referred to as the *pivot block*. As the pivot block will usually not include all of the pivot columns' entries, there can be large entries in pivot columns that are not part of the pivot block. As a result no pivot block entries for that column may pass the threshold pivoting criteria and that pivot becomes lost. However, if another acceptable pivot block column is used then its resulting update may cause the earlier column to become acceptable.

As a result, numerical checks must determine the maximum magnitude value in a pivot column and attempt to find a pivot (from the pivot block) that can pass the corresponding pivot threshold. Upon a failure, other pivot columns should be checked accordingly. Only when no pivot column has an acceptable pivot should the search be terminated. Furthermore, once a successful pivot has been applied, pivot columns that previously failed may be rechecked as their values have been updated by the pivot application.

Since pivot columns are distributed across processors, checking of all pivot columns will require communication. As the pivot search operation has a relatively low computational cost (compared to communication costs), the strategy of checking all the local pivot columns prior to initiating communication is adopted. This strategy was integrated into both the basic and pipelined versions of the fan-out, column-scattered partial factorization algorithms.

5.3.1 P3 – Basic PDF with Pivot Recovery

Since the basic fan-out algorithm essentially leaves all but the current pivot owner processor idle during pivot selection and the computation of the corresponding column of L, a natural modification would be to have all processors search through their active pivot columns while the current pivot owner is doing its pivot selection. Thus, if the current pivot owner fails to find an acceptable pivot column, the other processors will at least have started searching for an alternative pivot. Upon notification of a pivot failure, the other processors will either continue their search or reply with their results if their search is already completed. Furthermore, if the processors that do not own the current pivot complete their search before any notification from the current owner, they may even proceed to the computing of the corresponding column of L (in a separate work area) if their search was successful.

Care must be taken however, to not introduce excessive additional costs in the situations where the current pivot owners typically find a good pivot. To protect against this, each non-current pivot owner will search a single column at a time and then check if the current owner has broadcasted its results. (The test for the received broadcast can be done via the *ntest* function on the nCUBE 2 system [111]). If the notification is of a successful pivot, the corresponding multipliers (column of L) is included in the notification and the alternative searches are terminated immediately.

The other challenge is to determine an appropriate new pivot owner in an efficient and deterministic fashion when the current owner fails to find a good pivot. The parallel prefix operation, *imin*, is an excellent tool for this purpose [111]. The *imin* function operates on a specified cube/subcube with each node providing a single integer value. The minimum value is determined using a parallel prefix computation and the result provided to either one or all of the participating nodes. For purposes of determining a new pivot owner, each processor node in the subcube for the frontal matrix will provide either the number of nodes in the current subcube (if this node fails to have a good pivot) or the forward difference from the ID of the current pivot owner processor node to the reporting node's ID. If there is an acceptable pivot available, the *imin* operation will determine the forward delta (difference in IDs) from the current pivot owner to the nearest processor with a good pivot (in an increasing node ID order). Alternatively, if there is no acceptable pivot in any active pivot column, this delta will be equal to the number of nodes in the subcube and all nodes will know to discontinue factorizing of this frontal matrix. If a new owner is found, it will compute the corresponding multipliers (if not already precomputed during earlier idle time) and then broadcast them to the rest of the cube/subcube. The other processors will know who the new owner is from the *imin* operation and simply wait for that node to provide its good multipliers.

The whole process is done with a single small broadcast notifying the subcube of the owner's failure to find a good pivot, an *imin* parallel prefix operation to determine the new owner, and the broadcast of the resulting good multipliers. Much of the search for alternative pivots is done during what would otherwise be idle time.

To avoid nondeterminism, implementation of this algorithm always begins the alternative pivot search (when not the current pivot owner) using a static ordering

of its active pivot columns and a fixed starting position. If this is not done, the column on which the search is terminated, due to receipt of a good pivot from the current owner, will be timing dependent and an initiation of the next search based on this previous termination point can produce different results (different selected pivot column orderings).

An outline of the P3 algorithm and the component subroutines are provided in Figures 5-3, 5-4, 5-5, and 5-6.

P3 ALGORITHM:

```
(Basic fan-out with lost pivot recovery across columns and)
(overlapped column pivot searching.)

for ( pivots 1 thru number of pivots ) do
    if ( this processor's turn for current pivot )
        call ATTEMPT_LOCAL_PIVOT
    else /* not this processor's turn for current pivot */
        call LOOK_FOR_ALT_PIVOTS
        if ( pivot owner does NOT have a good pivot column )
            call RESOLVE_NEW_PIVOT_OWNER
        endif
    endif
    - update the row and column permutations for the
        resulting pivot column
    - determine which processor will own next pivot
        (look forward from current pivot owner to next)
        (processor that has active pivot columns )
    - update active local frontal matrix columns using the
        current multipliers
endfor
```

Figure 5-3. P3 – Basic PDF with Pivot Recovery

In order to accurately track both row and column permutations to the frontal matrix being partially factorized, each participating processor maintains row and column permutation vectors. Broadcasted messages that include multipliers also include a header which specifies status and the row/column permutations for the corresponding pivot. In this way, all processors can maintain consistent records of the applied permutations. In order to avoid unnecessary memory to memory transfers for message preparation, multipliers (which occur consecutively in the column-major storage format) are prefixed with the header information and provided to the broadcast function in place. Allocation of the frontal matrix data area includes the necessary extra prefix memory. Data objects displaced by the header are saved before the header is created and restored after the broadcast. To avoid data alignment problems, the

Subroutine ATTEMPT_LOCAL_PIVOT:

```
while ( this processor has active pivot columns to check )
        and ( have not found a good pivot column ) do
    - check next local pivot column
endwhile
if ( this processor has a good pivot column )
    - compute multipliers and row/col permutations
    - broadcast results to the others
else
    - broadcast "no_pivot" status
    - determine nearest good pivot column holder via
        imin parallel prefix computation
    if ( no others have a good pivot col )
        - return ( number of good pivots )
    else
        - receive multipliers from new pivot owner
    endif
endif
```

Figure 5-4. P3 Subroutine: ATTEMPT_LOCAL_PIVOT

integer header information is converted into the floating point format for its storage as a prefix to the floating point message. This limits the values allowed for these integers to 24 bits but this is sufficient for the problem sizes currently targeted. These broadcast message mechanisms are also used by the P4 algorithm to be discussed next.

5.3.2 P4 – Pipelined Partial Factoring with Pivot Recovery

The problem with the P3 algorithm just described is that when pivots are easily found, its performance is only near that of the basic fan-out algorithm. A more desirable situation would be to integrate the pivot recovery mechanisms of the third algorithm (P3) with the pipelining concept of the basic pipelined algorithm (P2). An effective integration of these two concepts would hopefully preserve the high performance of pipelining when pivots are easily found without jeopardarizing efficient overlapped searching for alternative pivots. The P4 algorithm is such an integration.

With the P4 algorithm, the owner of the next pivot searches its own pivot columns prior to completing the updates for the current pivot. The results of the search will be broadcast to the other nodes and buffered until they complete their current updates. If a pivot is found, the multipliers are already computed and available in the broadcast message. If the search for the next pivot is unsuccessful, the initial next pivot owner completes the rest of its current pivot updates before participating in the resolution of the next pivot's recovery. This allows the other nodes time to complete their current

Subroutine LOOK_FOR_ALT_PIVOTS:

```
while ( this processor has active pivot columns to check )
        and ( have not found a good pivot column )
        and ( have not received a good pivot column ) do
    - check my next active pivot column
    if ( pivot owner has sent broadcast (ntest) )
        - receive broadcast and determine status
    endif
endwhile
if ( pivot own has not yet sent broadcast )
    and ( this processor has a good pivot column )
    - compute multipliers using my pivot column and
        hold them in a temporary buffer, also the
        corresponding row/col permutations
endif
if ( pivot owner has not yet sent broadcast )
    - wait for and receive broadcast
endif
```

Figure 5-5. P3 Subroutine: LOOK_FOR_ALT_PIVOTS

pivot updates and perform their local searches for a next pivot. As the amount of computations done by each processor is roughly equivalent, all nodes should be ready for the *imin* determination of the next pivot owner at nearly the same time. (This new pivot owner determination is done in the same way as it was in P3). A graphical representation of this process is seen in the time line of Figure 5-7. Here processor Proc1 is the initial next pivot owner and does its search of active pivot columns (updating each pivot column with the previous pivots) prior to updating the rest of the active columns with the updates for the current pivot. Meanwhile the other processors do the current update and then find the status of the next pivot and do their local pivot searches while processor Proc1 completes its current pivot updates.

While much of the recovery effort is overlapped as it was in the P3 algorithm, the recovery will result in a drain of the pipeline. The pipeline will be "reprimed" with the multipliers produced by the new owner of the next pivot (Proc2 in Figure 5-7). During both the initial priming of the pipeline and any subsequent "repriming", it is possible to take advantage of the idle time on the nonpivot owner nodes by having them search for alternative $k+2$ pivots. This would improve the worst case behavior of P4 to be closer to the worst case behavior of P3. However, the worst case behavior is anticipated to be infrequent and the additional code complexity deemed to not be worth the potential advantage.

A more specific statement of the P4 algorithm can be found in Figures 5-8, 5-9, 5-10, and 5-11. As alternative pivot searching in this algorithm is only done when

Subroutine RESOLVE_NEW_PIVOT_OWNER:

```
if ( this processor has a good pivot column )
    - send delta from pivot owner's node id to this
        node id in imin prefix computation
else
    - send number of processors for this frontal matrix
        in the imin prefix computation
endif
- receive delta to new pivot owner from imin computation
if ( delta == number of frontal processors )
    /* that is, no processor has a good pivot column */
        - return ( number of good pivots )
else
    if ( this processor is the new pivot owner )
        if ( multipliers not yet computed )
            - compute the multipliers and row/col permutes
            - broadcast the results
        else
            - broadcast multipliers from temporary buffer
                together with row/col permutations
            - copy multipliers to frontal matrix
        endif
    else
        - receive broadcast from the new pivot owner
    endif
endif
```

Figure 5-6. P3 Subroutine: RESOLVE_NEW_PIVOT_OWNER

required and in a deterministic manner, there is no timing dependent behavior. The algorithm is deterministic.

5.4 Single Pivot Version

Frequently in the current sequential implementation of the unsymmetric-pattern multifrontal method [34, 35], frontal matrices are encountered that require only a single pivot step. In such frontal matrices the pivot block is a single entry and no row or column permutations can be made. Furthermore, the looping overhead for all the pivots can be eliminated as can other related overhead. As a result, a streamlined partial dense factorization routine, called P5, was developed for exclusive use on single pivot frontal matrices. The logic is very similar to that of the P1 algorithm with the

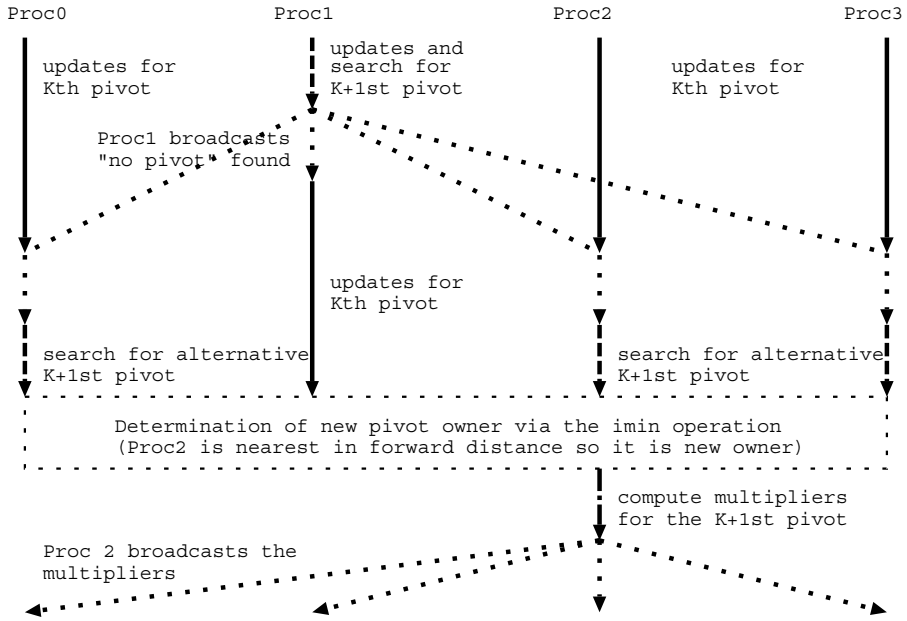

Figure 5-7. P4 Pivot Recovery Timeline

loop for all pivots removed and a simple check for numerical acceptability of the single pivot added.

5.5 Performance Achieved

In order to assess the performance achieved by each of the five algorithms introduced, user-defined events were inserted into the factorization algorithms and the *etool* facility used to determine the times at which the events occurred. From the results, detailed timings of the partial dense factorization algorithms were achieved. A driver program created square frontal matrices of orders 32, 64, 128, and 256 each run with 1, 8, 16, and 32 pivot steps. Subcubes of dimensions 0 through 5 were used and speed-ups determined based on comparison with the dimension 0 run times. Pivot rows and columns were generated using random numbers and were densely populated. In most runs, the pivot block was forced to be diagonally dominant to preclude the need for permutations. (However, some runs were made to force row permutations).

As the number of floating point operations for factorization of each of these combinations is well defined, achieved Mflops could also be determined. Comparison is also made of the basic and pipelined versions together with the impact of using the BLAS 1 routines. The consequences to performance of taking numerical considerations into account are also analyzed. Finally, brief discussions of the single pivot algorithm and overall double precision results are provided.

5.5.1 Floating Point Operations

The achieved Mflops (for the fastest versions of each kernel) are shown for single precision (32 bits) are shown in Table 5-1 and for double precision (64 bits) in Table 5-2. These results are based on a frontal matrix of 256 by 256 entries with 32 pivots

```
P4 ALGORITHM:

    (Pipelined fan-out with lost pivot recovery across columns)

    if ( this processor's turn for first pivot )
        call FIND_PIVOT_AND_COMPUTE_MULTIPLIERS
    endif
    for ( all potential pivots in the frontal matrix ) do
        if ( NOT this processor's turn for current pivot )
            - receive broadcast from pivot owner
            if ( status is broadcast was "no_pivot" )
                call NONPIVOT_OWNER_RECOVERY
            endif
        else if ( last pivot selection was successful )
            - make resulting multipliers current
        else
            call PIVOT_OWNER_RECOVERY
        endif
        if ( this processor's turn for next pivot )
            call FIND_PIVOT_AND_COMPUTE_MULTIPLIERS
        endif
        - update the row and col permutations
        - update remainder of active pivot cols
            ( some may have been updated during the pivot search)
        - update the cols that are not potential pivot cols
    endfor
```

Figure 5-8. P4 – Pipelined PDF with Pivot Recovery

applied (the largest problem size tested). For reference, the asymptotic flops rating of the nCUBE 2 processor and the BLAS 1 routines in use are shown in Figure 5-12. Realizing that the *Saxpy/Daxpy* operations dominate the computations, we see that in the single processor configurations the algorithms achieve better than 90 percent of peak BLAS performance and 62 to 70 percent of peak machine performance. Furthermore, on 32 processors, they achieve as much as 45 percent of peak BLAS performance and 32 percent of machine peak performance.

5.5.2 Use of BLAS 1 Routines

Use of the assembly coded BLAS 1 routines improved the sequential execution times of the basic fan-out algorithm (P1) by 25 to 50 percent as compared to the use of the fastest C coded version of the BLAS 1 routines developed. For example, the 256 by 256 frontal matrix with 32 pivots took 3396.98 msecs using the C coded BLAS and 1783.91 msecs using the assembly coded BLAS. Furthermore, the improvements

Subroutine FIND_PIVOTS_AND_COMPUTE_MULTIPLIERS:

```
while ( active pivot cols to check ) and
        ( NOT found a good pivot col ) do
    - check my next pivot col
endwhile
if ( this processor has a good pivot column )
    - compute multipliers and row/col permutations
    - broadcast results to the others
    - save multipliers and row/col permutes for my updates
else
    - broadcast "no_pivot" column status
endif
```

Figure 5-9. P4 Subroutine: FIND_PIVOTS_AND_COMPUTE_MULTIPLIERS

Table 5-1. PDF Mflops Achieved (Single Precision)

PDF ROUTINE	P=1	P=2	P=4	P=8	P=16	P=32
P1	2.07	4.01	7.48	13.09	20.56	28.09
P2	2.06	4.06	7.73	14.01	23.21	33.50
P3	2.06	3.96	7.30	12.57	19.88	25.77
P4	2.05	4.03	7.64	13.69	22.32	31.68

were as much as four times better than the original partial dense factorizing routine developed for the distributed memory simulation reported earlier. In this original routine, the 64 by 64 frontal matrix with 32 pivots took 399.79 msecs and only 102.33 msecs with the P1 routine using the assembly coded BLAS.

While use of the assembly coded BLAS has major benefits in terms of reducing execution time, there is an adverse effect on speed-ups due to parallelism. As the assembly coded BLAS effectively reduce the cost of computations, they lower the ratio of required computation time over communication time (frequently called the R/C ratio) [135]. The result is a higher relative cost for communication and reduced speed-ups. An example of this effect is seen with the 256 by 256 frontal matrix using 32 pivots where the speed-up obtained using the fastest C coded BLAS on a 5 dimensional cube was 17.38 but the assembly coded BLAS on the same processor set achieved only a 13.57 speed-up. Even more telling is the case of the 128 by 128 frontal matrix with 8 pivots using the original routine where the 5 dimensional cube speed-up achieved was 16.41 compared to 7.78 with the P1 routine using assembly coded BLAS. In terms of the R/C ratio, which was determined using *ctool* on a 3 dimensional cube, the original routine had an R/C ratio of about 8, the P1 algorithm

Subroutine NONPIVOT_OWNER_RECOVERY:

```
while ( more active pivot columns to check) and
        ( have NOT found a good pivot ) do
    - update next potential pivot column with current multipliers
    - check numerical acceptability of pivot
endwhile
if ( this processor has a good pivot column )
    -send forward distance from pivot owner's node id to this
        processor's node id in the imin operation
else
    -send subcube size in imin operation
endif
if ( no processor has a good pivot )
    - return( number of successful pivots )
else if ( this processor has the new pivot )
    - compute and broadcast multipliers and permutations
else
    - receive multipliers and permutations from new pivot owner
endif
```

Figure 5-10. P4 Subroutine: NONPIVOT_OWNER_RECOVERY

Table 5-2. PDF Mflops Achieved (Double Precision)

PDF ROUTINE	P=1	P=2	P=4	P=8	P=16	P=32
P1	1.71	3.28	5.97	10.06	15.05	19.46
P2	1.73	3.39	6.35	11.19	17.71	24.24
P3	1.67	3.18	5.75	9.58	14.13	18.97
P4	1.67	3.26	6.10	10.67	16.80	22.90

using the fastest C coded BLAS had a ratio of 4, and the P1 algorithm using assembly coded BLAS had a ratio of less than 2. Hence, while the assembly coded BLAS do significantly improve performance, they have adverse effect on the exploitation of parallelism.

For the rest of the results reported in this chapter, the assembly coded version of the BLAS 1 routines will be used.

Subroutine PIVOT_OWNER_RECOVERY:

```
-send size of subcube in imin operation
if ( imin returns subcube size )
/* no replacement pivot is available */
    -return( number of successful pivots )
else
    -receive multipliers and permutations from new pivot owner
endif
```

Figure 5-11. P4 Subroutine: PIVOT_OWNER_RECOVERY

nCUBE 2 PEAK PERFORMANCE

Single precision: 3.28 Mflops
Double precision: 2.40 Mflops

nCUBE 2 BLAS I PERFORMANCE

Saxpy: 2.31 Mflops
Sscal: 2.88 Mflops
Daxpy: 1.83 Mflops
Dscal: 1.80 Mflops

Figure 5-12. nCUBE 2 Peak and BLAS Performance

5.5.3 Basic versus Pipelined Performance

There are two performance issues involved with the introduction of pipelining. The first is the sequential overhead required to facilitate the pipelining and the second is the actual benefit from pipelining.

The sequential overhead added by pipelining was consistently less than two percent additional execution time. The worst relative increase came with the 32 by 32 frontal matrix using 32 pivots. In this case, the execution time on a single node went from 22.23 msecs without pipelining to 22.78 msecs with pipelining. The worst absolute increase came with the 256 by 256 frontal matrix using 32 pivots where the inclusion of pipelining caused the execution time to go from 1783.91 msecs to 1791.88 msecs.

The sequential overheads were more than offset by significant improvements in parallel performance. Furthermore, this improved parallel performance became better with larger frontal matrices, more pivots, and also with larger cube sizes (there is

still a point at which performance degrades with larger cube sizes as broadcast costs grow linearly with cube size). The best results came with the 256 by 256 frontal using 32 pivots on a 5 dimensional cube (largest cube tested). In this case the use of pipelining reduced the parallel execution time from 131.42 msecs to 110.20 msecs, a reduction of 16 percent. The corresponding speed-up improvements in this case went from a speed-up of 13.57 without pipelining to 16.26 with pipelining.

In addition, analysis based on comparing the analytical performance functions of the next section with the observed timings and separately using the *ctool* communication profiling program on the nCUBE 2 showed that 35 percent of the broadcast communication costs were overlapped with computations in addition to the overlapping of multiplier computation with the updating of previous pivots.

5.5.4 Inclusion of Numerical Considerations

When numerical checking and pivot recovery were added to the basic fan-out (resulting in the P3 algorithm) and the pipelined fan-out (resulting in the P4 algorithm) routines, additional execution overheads were introduced. In the uniprocessor tests, the additional execution time overheads were consistently less than two percent. The increases in parallel execution times were more pronounced. Based on the tests with a 256 by 256 frontal matrix and 32 pivots using a 5 dimensional cube, the basic fan-out algorithm parallel time went from 131.42 msecs (using the P1 algorithm) to 143.26 msecs (using the P3 algorithm). This is an increase of 9 percent. This large increase is due to the fact that the numerical testing of the P3 algorithm comes in the code section that is essentially sequential (not overlapped with computations on processors not owning the current pivot). The corresponding results on the pipelined algorithms for the same configuration were less severe with parallel time going from 110.20 msecs using the P2 algorithms to 115.40 msecs using the P4 algorithm, which is an increase of only 4.8 percent. With the pipelined methods, much of the additional work is effectively overlapped with other computations due to the pipelining.

Corresponding to the additional parallel execution time overhead is a decrease in the achieved speed-ups when numerical considerations are added. Specifically, for the configuration defined above, the speed-up achieved with P1 was 13.57 which dropped to 12.53 with the P3 algorithm. The speed-up for P2 was 16.26 and dropped to 15.39 with P4. In all of these tests the pivot blocks were forced to be diagonally dominant so no actual permutations were necessary.

A second question when introducing numerical considerations is the cost of actually doing the required permutations. Only row permutations were addressed in these tests with frontal matrices constructed such that row permutations were required for all but the last pivot. The worst case relative overhead occurred with the 32 by 32 frontal matrix using 32 pivots where the time without permutations was 24.13 msecs and with permutations was 25.04 msecs. More typically, the execution time increase (for both single and multiple processor runs) was less than 2 percent.

5.5.5 Single Pivot Performance

As mentioned earlier, the P5 algorithm was developed to handle the case of a single pivot. Tests comparing the performance of this algorithm on single pivot

frontal matrices against the P4 algorithm revealed up to an 8 percent reduction on single processor execution times and up to a 27 percent reduction in parallel execution times. These relative benefits increased with cube size and decreased with frontal matrix size. While the best relative reductions just reported occurred with the smallest frontal matrices (32 by 32), the largest frontal matrices tests (256 by 256) saw single processor execution time reductions of 1.4 percent and 5 dimensional cube parallel time reductions of 9.5 percent. Hence the inclusion of the P5 algorithm for single pivot frontal matrices would be worthwhile especially when using multiple processors.

5.5.6 Double Precision Results

All of the partial dense factorization routines were written with conditionally compiled single or double precision floating point operations. When running in the double precision configuration, uniprocessor runs saw execution times increase by 20 to 25 percent which is in accordance with the additional time requirements of double precision operations as reported in the nCUBE 2 literature [112, 110, 111]. However, parallel execution times increase more dramatically, by as much as 40 percent. The worst relative increase occurred using the P3 algorithm on the 256 by 256 frontal matrix using 32 pivots where the single precision parallel time on a 5 dimensional cube was 143.26 msecs and the corresponding double precision time was 204.32 msecs (almost a 43 percent increase). The reason for this larger increase is that the broadcasted message size doubles with double precision. Again with the transition to double precision we find the relative cost of communication compared to computation increasing. The resulting loss of exploitable parallelism is found in the speed-up numbers for the case above which went from 12.53 for the single precision case to 10.82 for the double precision case.

5.6 Analytical Performance Predictions

In the next chapter, the partial dense factorization routines will be considered as tasks within a directed acyclic graph structure that describes the precedence relations between these tasks. An accurate prediction of execution times of these tasks on various processor sets will help to facilitate effective scheduling. In this section, the analytical formulas used to make these execution time predictions are developed and compared against the observed performance.

As the partial dense factorization routines are composed of well defined subfunctions, each subfunction will be modeled separately and then the subfunction models combined into a composite model of the entire partial dense factorization process.

In these models, the terms defined below will be commonly used:

- M - number of frontal matrix rows.
- N - number of frontal matrix columns.
- K - number of frontal matrix pivots.
- E - number of original entries to assemble into the frontal matrix.

- P - number of processors assigned to the frontal matrix.

All times expressed by these models will be in microseconds.

5.6.1 Component Times

There are seven primary subfunctions for the partial factorization routines. These are:

- t_f - fixed overhead plus loop handling time.
- t_c - time to allocate and clear the frontal matrix data area.
- t_a - time to assembly original matrix entries into the data area.
- t_p - time to do the check required for threshold pivoting.
- t_m - time to compute the multipliers for each pivot.
- t_b - broadcast time for multipliers.
- t_u - time to update the active submatrix for each of the pivots.

Fixed Overhead Plus Loop Handling Time (t_f): This time includes the time required for initialization plus the looping construct used to go through all of the pivots for a given frontal matrix. The model is

$$t_f = \alpha_f + \beta_f \cdot K$$

where α_f is 100 for the basic algorithms (P1, P3, and P5) and 200 for the pipelined algorithms (P2 and P4). The β_f constant is 10 for P1 and P3, 0 for P5, and 20 for P2 and P4.

Allocate and Clear Time (t_c): The *calloc* function of the C language is used to acquire the frontal matrix data area from the heap and also clears the provided memory. The time to complete this function is

$$t_c = 170 + 0.175(M \cdot N).$$

This formula is based on the experimentally derived figures presented in the earlier nCUBE performance evaluation of Chapter 4.

Original Entry Assembly Time (t_a): The original entries of the matrix that fall within the particular frontal matrix are assembled into the data area using a C coded scatter operation. The formula for predicting this time is

$$t_a = 10 + 2 \cdot E.$$

This formula can also be used for the assembly of contributions to the frontal matrix from other frontal matrices.

Pivot Determination Time (t_p): Pivot determination requires identifying the largest magnitude values in the pivot block and pivot column and checking them against the threshold pivoting parameter. The *Isamax/Idamax* BLAS 1 functions are used for finding these values and C code for the necessary comparisons. The corresponding formula is

$$t_p = \Sigma_{i=1}^{K} [\alpha_p + \beta_p(M - i + 1)]$$

where α_p is 5.6+10 for single precision and 4.8+10 for double precision (5.6 and 4.8 come from the *Isamax* and *Idamax* timings and 10 from the necessary C code) and β_p is 1.2 for single precision and 1.5 for double precision (these come from the *Isamax* and *Idamax* timings).

Multiplier Computation Time (t_m): The multiplier computation time is based on the *Sscal/Dscal* BLAS 1 operation and the model is

$$t_p = \Sigma_{i=1}^{K} [\alpha_m + \beta_m(M - i)]$$

where α_m is 8.4+8 (single precision) or 6.8+8 (double precision) and β_m is 0.348 (single precision) and 0.554 (double precision). The 8 in the α_m term accounts for the required C code and all the other numbers are taken from the BLAS 1 timings of the *Sscal* and *Dscal* operations.

Multiplier Broadcast Time (t_b): The multiplier broadcast time is taken from the earlier timing experiments on the nCUBE 2 broadcast function. The model is

$$t_b = \Sigma_{i=1}^{K} [195.22 + 50.02\ d + 0.122\ b\ (M - i) + 0.45\ b\ d\ (M - i)]$$

where b is the number of bytes required for a data value (4 for single precision and 8 for double precision) and d is the dimension of the subcube. In the case of a single processor, there is no broadcast so this time is set to zero in that situation.

Active Submatrix Update Time (t_u): The updating of the active submatrix for each pivot involves an outer product operation that is done as a series of *Saxpy/Daxpy* operations on each column. The derived model is

$$t_u = \Sigma_{i=1}^{K} \Sigma_{j=1}^{N-i} [\alpha_u + \beta_u(M - i)]$$

where α_u is 13+5 for both single and double precision (5 accounts for the necessary C code and 13 for the *Saxpy/Daxpy* start up) and β_u is 0.866 for single precision and 1.09 for double precision.

With these well defined component times we can now create analytical models of the five partial dense factorization routines.

5.6.2 Aggregate Times

Basic Fan-Out Algorithm (P1): The basic fan-out algorithm does not utilize partial pivoting so the t_p time is not included. The corresponding model is

$$t_{exec} = t_f + t_m + t_b + \frac{t_c + t_a + t_u}{P}$$

where P is the number of processors in the utilized subcube.

As an example of the accuracy of this model, the predicted and observed times (in milliseconds) for a 128 by 128 frontal matrix with 8 pivots are shown Table 5-3 for a variety of processor set sizes.

Table 5-3. P1 Predicted Versus Observed Performance

	P=1	P=2	P=4	P=8	P=16	P=32
Predicted	131.18	70.23	39.82	25.71	19.74	17.85
Observed	132.62	71.04	41.12	26.64	19.72	17.04

Not all predictions were this accurate, in particular the smallest frontal matrices on many processors were underestimated by as much as a third and the largest frontal matrix times on many processors were overestimated by as much as ten percent. But in general the accuracy was quite respectable. Furthermore, the use of many processors on a small frontal matrix is not likely to be scheduled.

Pipelined Fan-Out Algorithm (P2): Like the basic fan-out algorithm, the pipelined fan-out algorithm (P2) does not utilize partial pivoting so the t_p time is not included. The corresponding model is:

$$\begin{aligned} t_{exec} &= t_f + t_m^{(1)} + \left[\left(\frac{1}{K}\right) + 0.65 \cdot \left(\frac{K-1}{K}\right)\right] t_b \\ &\quad + \frac{t_c + t_a + t_m^{(K-1)} + t_u}{P} \end{aligned}$$

where P is the number of processors in the utilized subcube. The term $t_m^{(1)}$ accounts for unoverlapped computing of the first multipliers and the other $t_m^{(K-1)}$ term represents the computation of the rest of the multipliers. The complicated coefficient on the t_b term accounts for the fact that 35 percent of the multiplier broadcasts (after the first one) are overlapped with computations.

As an example of the accuracy of this model, the predicted and observed times (in milliseconds) for a 128 by 128 frontal matrix with 32 pivots are shown in Table 5-4 for a variety of processor set sizes.

Again larger discrepancies were found with the smaller frontals on large processor sets.

Table 5-4. P2 Predicted Versus Observed Performance

	P=1	P=2	P=4	P=8	P=16	P=32
Predicted	431.12	226.68	124.47	76.02	54.45	46.31
Observed	438.51	229.55	129.24	81.40	59.92	51.83

Basic Fan-Out With Pivoting Algorithm (P3): The basic fan-out with pivoting algorithm (P3) is very similar to P1 except that the t_p is also included. The corresponding model is

$$t_{exec} = t_f + t_p + t_m + t_b + \frac{t_c + t_a + t_u}{P}$$

where P is the number of processors in the utilized subcube.

As an example of the accuracy of this model, the predicted and observed times (in milliseconds) for a 64 by 64 frontal matrix with 8 pivots are shown in Table 5-5 for a variety of processor set sizes.

Table 5-5. P3 Predicted Versus Observed Performance

	P=1	P=2	P=4	P=8	P=16	P=32
Predicted	37.14	22.30	14.61	11.40	10.42	10.56
Observed	36.92	22.68	15.82	13.00	12.00	11.99

Pipelined Fan-Out With Pivoting Algorithm (P4): The pipelined fan-out with pivoting algorithm (P4) essentially extends the P2 algorithm with threshold partial pivoting (accounted for by t_p). This added t_p time is also spread across processors and overlapped with other computations. Thus this time is only counted for sequentially for the first pivot (i.e., $t_p^{(1)}$) and the rest is considered to be done is parallel. The corresponding model is:

$$\begin{aligned} t_{exec} &= t_f + t_m^{(1)} + t_p^{(1)} + \left[\left(\frac{1}{K}\right) + 0.65 \cdot \left(\frac{K-1}{K}\right)\right] t_b \\ &\quad + \frac{t_c + t_a + t_m^{(K-1)} + t_p^{(K-1)} + t_u}{P} \end{aligned}$$

where t_p is as described above and the rest of the terms are as described in the P2 formula.

As an example of the accuracy of this model, the predicted and observed times (in milliseconds) for a 256 by 256 frontal matrix with 16 pivots are shown in Table 5-6 for a variety of processor set sizes.

Single Pivot Algorithm (P5): The single pivot algorithm (P5) has much of the flavor of P3 in that there is a partial pivoting subfunction but this function is limited to simply the check function as no row permutations are possible in the single

Table 5-6. P4 Predicted Versus Observed Performance

	P=1	P=2	P=4	P=8	P=16	P=32
Predicted	955.11	486.84	253.67	139.75	85.46	60.97
Observed	950.50	482.90	254.56	141.47	86.49	60.09

entry pivot block. Also, much of the fixed overhead is reduced. Accordingly the C code portion of α_p is reduced from 10 to 8 and t_f becomes simply equal to 8. The corresponding model is

$$t_{exec} = t_f + t_p + t_m + t_b + \frac{t_c + t_a + t_u}{P}.$$

As an example of the accuracy of this model, the predicted and observed times (in milliseconds) for a 64 by 64 frontal matrix with 1 pivot are shown in Table 5-7 for a variety of processor set sizes.

Table 5-7. P5 Predicted Versus Observed Performance

	P=1	P=2	P=4	P=8	P=16	P=32
Predicted	5.96	3.57	2.35	1.82	1.64	1.63
Observed	5.76	3.73	2.52	2.07	1.84	1.83

While not perfect, these prediction formulas model the performance of the partial dense factorization routines reasonably well. The worst discrepancies were with small frontal matrices on large processor sets which is a combination that is not likely to be scheduled.

This chapter has focused on the development of a range of partial dense factorization routines. Incorporation of the assembly coded BLAS 1 routines and pipelining have provided significant performance enhancements. A streamlined single pivot routine also provided performance benefits. Inclusion of threshold partial pivoting (using row permutations) and column pivoting for alternative pivot selection provides a robust capability for dealing with numerical considerations. Use of the parallel prefix *imin* operation also provided efficient recovery when the current pivot owner could find an acceptable pivot.

This suite of partial dense factorization routines will provide the ground work for the construction of a series of distributed memory, unsymmetric-pattern multifrontal method implementations that will deal with the increasingly difficult issues of scheduling, task/data allocation, and lost pivot recovery.

CHAPTER 6
FIXED PIVOT ORDERING IMPLEMENTATION

The initial implementation of the distributed memory, unsymmetric-pattern multifrontal LU refactorization software focuses on sequences of identically structured sparse matrices where the pivot ordering selected for the first matrix can be maintained for all matrices in the sequence. Such a facility has practical application only if the matrices of the sequence have special properties (such as diagonal dominance) that insure the numerical acceptability of the originally selected pivots. Within the context of this research effort, however, this initial restricted implementation provides a test bed upon which issues such as scheduling, allocation, and assignment can be explored. The implementation also provides the first empirical evidence that the parallel speed-ups predicted by the previously presented performance models can actually be achieved.

Throughout this chapter, *scheduling* refers to the process of ordering the execution of the factorization tasks associated with individual frontal matrices. The term, *allocation*, will refer to the determination of how many processors (the size of the subcube) will participate in the factorization for each frontal matrix. Finally, *assignment* will refer to the selection of a specific subcube of the allocated size and the mapping of columns within a frontal matrix to the specific processing nodes of that subcube.

The discussion in this chapter is divided into three sections. In the first section, the focus is on the preprocessing that takes place within the host processor. This includes the process of acquiring and refining the specification of the assembly DAG from the UMFPACK software [32], the two developed scheduling methods, the subcube allocation process (which is integrated into the scheduling), and the various techniques developed for the assignment problem. The second section describes the parallel processing routines that accept the statically produced schedule and frontal matrix descriptions and then perform the parallel factorization. Particular attention is placed on the handling of contributions passed between frontal matrices. A description of the distributed triangular solves are also provided in this section. Finally, the third section identifies the key performance issues, describes how these issues were addressed, and reports the performance evaluation results.

6.1 Host Preprocessing

The initial implementation of the parallel, distributed memory, multifrontal LU refactorization package is composed of four principal, user-callable routines. The routines are:

- SCHEDULE_REFACTOR: This routine accepts the assembly DAG description produced by UMFPACK and a set of original matrix values in triplet format, together with various scheduling and assignment parameters. The assembly DAG is analyzed and a static schedule, subcube allocation, and data assignment are developed. A hypercube of the appropriate dimension is then allocated and the parallel refactorization (PRF) code executed on each hypercube node. The frontal matrix descriptions and original matrix entries are then forwarded, via message passing, to the hypercube nodes, which then complete the refactorization and retain their portions of the LU factors. The SCHEDULE_REFACTOR code then accepts the results status from the hypercube nodes and returns that status to the invoking process. This SCHEDULE_REFACTOR routine must be called prior to any of the other routines in the package.

- SUBSEQUENT_REFACTOR: Once the assembly DAG has been analyzed, scheduled, allocated, and assigned, subsequent refactorizations can be done using these structures that are already distributed to the hypercube's nodes. Only the new matrix values are required. The SUBSEQUENT_REFACTOR routine accepts these new values and sends them, together with a command to perform the refactorization, to the parallel factorization (PRF) code. The SUBSEQUENT_REFACTOR code will also wait for the results of the factorization and pass this status back to the calling routine.

- TRIANGULAR_SOLVE: After a refactorization has been done, the LU factors are retained in a distributed format across the hypercube's nodes. The TRIANGULAR_SOLVE accepts a right hand side (the $\bar{b}$ vector in the equation $A\bar{x} = \bar{b}$) and sends this vector (actually a permutation of it as will be detailed later) to the parallel block triangular solve code. The block triangular solve code on the hypercube processing nodes will perform the required forward and back substitutions and return the solution vector, $\bar{x}$. For purposes of this prototype implementation, the triangular solve function is provided primarily to verify results of the refactorizations. To this end, it will first generate a solution vector from either (a) all ones, (b) random numbers, or (c) a file of values. A right hand side is produced by multiplying the coefficient matrix, A, times this known solution. The solution computed by the block triangular solve code can then be compared to the known solution and the relative error determined (using the infinity norm).

- TERMINATE_REFACTORS: This routine simply commands the hypercube's nodes to deallocate their dynamically allocated data objects and terminate processing. The hypercube itself is also deallocated as part of this process.

By far the most extensive host processing is contained in the SCHEDULE_REFACTOR routine and this routine will be the focus of the rest of the host processing. This discussion will include a description of the assembly DAG input format, the two scheduling/subcube allocation methods developed, details on the various data

allocation options, and the message passing required to launch an initial parallel refactorization.

6.1.1 Assembly DAG Acquisition

The description of the assembly DAG produced by UMFPACK is held in an integer array format [32]. For convenience, UMFPACK dumps this array to a file with one entry per line and the SCHEDULE_REFACTOR routine simply reads this file to obtain the assembly DAG, which also includes other pertinent data such as the permutation matrices and descriptions of the LU factors. The format of this integer array produced by UMFPACK is as follows:

- Seven entries that include:

  1. Length of the assembly DAG descriptor array.
  2. Order of the matrix (N).
  3. Number of diagonal blocks in the block upper triangular form (BLOCKS).
  4. Number of nonzero entries not included in a diagonal block (NZOFF).
  5. Number of nonzero entries in the diagonal blocks (NZDIA).
  6. Number of frontal matrices (NFRONT).
  7. Array address of a list of array pointers to the start of each frontal matrix description within the array (LUPP).

- N entries which define the permutation matrix P.

- N entries which define the permutation matrix Q. Together P and Q define the row and column permutations respectively that were used by UMFPACK to obtain the block upper triangular form of the matrix and to establish an acceptable pivot ordering. The parallel refactorization will deal strictly with the matrix that results from applying these permutations to A, that is, the matrix A' where $A' = PAQ$.

- BLOCKS+1 entries that describe the diagonal blocks in terms of a sequential numbering.

- BLOCKS+1 entries that describe the diagonal blocks in terms of their leading diagonal entries.

- N+1+NZOFF entries that further describe the block upper triangular form. These entries are only present if there is more than one diagonal block and are not used by either the host or parallel software.

- Frontal matrix descriptions, in lead diagonal entry order. Frontal matrices constitute the nodes of the assembly DAG and their descriptions include a specification of the assembly DAG edges and thus completely define the assembly DAG. The basic format of each frontal matrix description is as follows:

1. Number of pivots in the frontal matrix.
2. Row degree (UDEG). Specifically this is the number of columns in the frontal matrix other than those in the pivot block.
3. Column degree (LDEG). This is the number of rows in the frontal matrix other than those in the pivot block.
4. Number of children (CHILDS). This is the number of incoming contribution edges to this frontal matrix from preceding frontal matrices.
5. LDEG entries that define the column pattern (row indices) of the frontal matrix rows that are not in the pivot block. The pivot block row indices are implicitly provided via a sequential numbering starting with the diagonal position of the first pivot in the frontal matrix's pivot block.
6. UDEG entries that define the row pattern (column indices) of the frontal matrix columns that are not in the pivot block. The pivot block column indices are also implicitly the sequential set of numbers starting with the index of the first pivot.
7. CHILDS entries that are the frontal matrix IDs (in a consecutive ordering of the frontal matrices) for the frontal matrices that provide contributions to the currently described frontal matrix. These entries specify the incoming edges to this node of the assembly DAG and have the type of the edge also encoded.

- NFRONT entries that are the array addresses of each frontal matrix description.

During the process of acquiring the assembly DAG, frontal matrix descriptions are expanded to include full row and column patterns, the children frontal matrix IDs are converted to a scheme that identifies frontal matrices by the position of their lead pivot in the given pivot ordering, space is allocated to maintain the assignments of columns to processors as well as other necessary descriptive information, and a topological sort of the assembly DAG is done. The topological sort is simplified by the fact that child frontal matrices are always defined prior to their parents in the descriptions provided by UMFPACK.

After the assembly DAG has been acquired, a detailed analysis of its nodes and edges is required prior to commencing the scheduling and allocation functions. Specifically, each frontal matrix designated by an assembly DAG node is assigned a weight based on how much computation effort is required for its partial factorization on a single processor. Additionally, the performance prediction formulas derived in the preceding chapter are used to predict factorization times based on allocating different size subcubes to the frontal matrix's factorization. This process starts with the smallest subcubes and computes and saves the predictions for each subcube size until either the largest possible subcube size is reached or the predicted factorization times begin to increase with larger subcube allocations. The dimension of the largest possible subcube that would not increase factorization time is defined as DIM_MOST in the frontal matrix description and no larger subcubes will ever be allocated to the frontal matrix.

Next the assembly DAG node (frontal matrix) descriptions are augmented with list of parent nodes (frontal matrices to which they will pass their contribution block). This essentially creates adjacency lists for both the incoming and outgoing edges for each node.

These parent-oriented edge lists together with the node weights (based on the single processor factorization times) are used to perform a critical path analysis of the assembly DAG. This requires a reverse ordered topological sort to first be done. Critical path priorities are assigned to each node (frontal matrix) based on the heaviest weighted path from that node to an exit node. (An exit node is defined as a node with no successor/out-going edges).

The final prerequisite task to be completed prior to scheduling, allocation, and assignment is to refine the edge descriptions. This involves the determination of the row and column patterns of each contribution passed between frontal matrices. Careful accounting is required to insure that each entry in the contribution block of the source frontal matrix is passed to one and only one destination frontal matrix. An example of this process is seen in the example of Figure 6-1. In this example, the frontal matrix has four direct parents: A, B, C, and D. The frontal matrix A, as an L parent, takes the entire first row. Next the U parent B takes what remains of the first column. The L parent C can then take the second row minus the entry from the first column, as it was taken by B. As D is an LU parent, it can accommodate the remaining contributions. At the conclusion of this process, we find that the contribution to B has a column length of 3, the contribution to D as a column length of 2, and the contributions to A and C only have column lengths of 1 each.

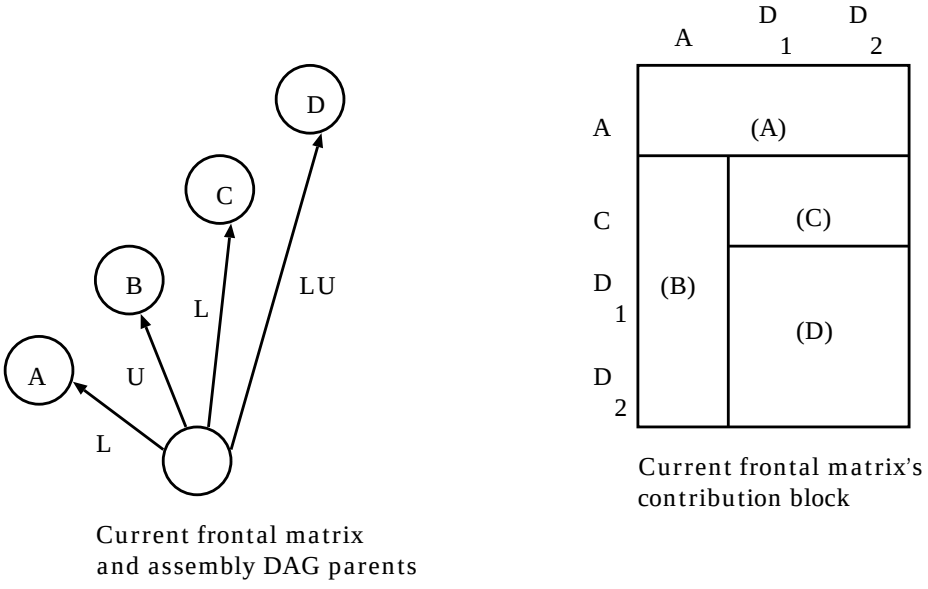

Figure 6-1. Edge Refinement Example

These refined edge descriptions will be used during subsequent process of assigning frontal matrix columns to processors so as to reduce message passing requirements. The edge descriptions will also be passed as part of the frontal matrix descriptions to

the parallel processing nodes and used in the parallel factorization process to expedite the forwarding and assembly of contributions. The child edges for each frontal matrix are also sorted by decreasing column degree (number of rows in the contribution of the edge). In this way, the subcube assignment process can attempt to eliminate (or at least localize within the given set of processors) the highest cost messages.

With the assembly DAG acquired and its definition refined per the processing just discussed, the scheduling, allocation, and assignment process can commence in earnest.

6.1.2 Frontal Matrix Scheduling

The scheduling of frontal matrix factorization tasks is made especially challenging as these tasks are themselves parallel tasks that may be executed on more than one processor. Thus, in addition to scheduling, we need to determine how large of a subcube to allocate to each frontal matrix task. The specific subcube assignments and frontal matrix column assignments will also impact message passing requirements and thus execution time.

The approach taken is to first estimate appropriate subcube allocations for a set of frontal matrix tasks. The scheduling of these tasks is then done based on these allocations with modifications possible based on current subcube availability. As tasks are scheduled, their specific subcube is assigned based on availability and to maximize the overlap with child frontal matrices (predecessors that provide contributions) in an effort to reduce message passing requirements. Once a subcube assignment is made, the specific columns of the frontal matrix are assigned to processors within the subcube based on a number of alternative criteria.

Hence the functions of scheduling, allocation, and assignment are interleaved into a composite process with the results of each of these functions affecting the processing of the other functions. However, it is possible to isolate the specific scheduling and subcube allocation/assignment mechanisms from the data assignment function for ease of description. This section does this and first focuses on the scheduling and subcube allocation/assignment issues. The specific data assignment options will then be presented separately.

Subcube Allocation

Before a frontal matrix task can be scheduled, its subcube allocation (how many processors it will be provided) must be defined. There are a number of considerations in this allocation process.

- Larger subcubes will allow faster execution, but given a specific size frontal matrix, there is a limit on the subcube size that can be employed without extending execution time. Furthermore, the principle of diminishing marginal returns applies as the partial factorization process's efficiency decreases with larger subcubes.

- Inter-task parallelism (concurrent execution of independent frontal matrix tasks) is more efficient than intra-task parallelism where multiple processors are used

on a single frontal matrix task. Hence when more tasks are available for execution, it would make sense to lower the number of processors assigned to each task and execute more tasks concurrently.

- Larger tasks will be able to make more efficient use of larger subcubes indicating that some type of proportional allocation is desirable.
- Tasks with the highest critical path priorities should be provided with the subcubes closer to their maximal requirements so they may finish sooner and reduce the execution times of the heaviest weighted paths through the assembly DAG.

The following approach to subcube allocation addresses these considerations by using a proportional allocation that is based on a task's weight, the number of tasks currently available for execution, the total weights of available tasks (or a subset of available tasks), and the number of processors in the system. Prior to specifically defining the allocation mechanism, a few definitions are required.

- P_{sys} is the number of processors in the hypercube.
- Q_{ready} is a priority queue of tasks available for execution (all predecessors have completed) which is ordered by the critical path priorities.
- $Q_{eligible}$ is a subset of the first $\min\{|Q_{ready}|, P_{sys}\}$ tasks from Q_{ready}.
- T_i is the ith frontal matrix task.
- $P_{time}(T_i, d)$ is the parallel execution time of task T_i running on a subcube of dimension d.
- *total_work_for_block* is the total amount of work associated with the tasks in $Q_{eligible}$ if each task were given a single processor (0 dimensional subcube). That is,
$$total\_work\_for\_block = \Sigma_{all\ T_i \in Q_{eligible}} P_{time}(T_i, 0).$$
- $dim\_most(T_i)$ is the dimension of the largest subcube that could be allocated to task T_i without extending its execution speed or exceeding P_{sys}.

Each task T_i in $Q_{eligible}$ is thus initially allocated a subcube of dimension $dim\_alloc(T_i)$ defined as

$$dim\_alloc(T_i) = \min\{dim\_most(T_i), \max\{0, \lfloor \log_2(P_{sys} * \frac{P_{time}(T_i, 0)}{total\_work\_for\_block}) \rfloor\}\}.$$

The use of the floor function within this formula tends to under allocate subcubes to tasks. To provide an alternative allocation that avoids under allocation, a second subcube dimension is computed as

$$alt\_dim\_alloc(T_i) = \min\{dim\_most(T_i), dim\_alloc(T_i) + 1\}.$$

When there are large numbers of tasks available, the *total_work_for_block* will be larger and smaller allocations will be made for each task. With heavier tasks, the ratio of $P_{time}(T_i, 0)$ to *total_work_for_block* will be larger and higher dimension subcubes will be allocated. Finally, when many processors are available (P_{sys}), larger allocations are made. Yet in all the cases protections are in place to provide at least one processor per task and to not provide a larger allocation than a task can utilize.

These two alternative subcube allocations can then be used by the two scheduling methods to be described shortly. Prior to this discussion of the scheduling methods however, it is appropriate to discuss how specific subcubes will be assigned as this is an integral and key component of both the scheduling methods developed.

Subcube Selection and Assignment

In both the S1/S2 and S3 scheduling/allocation methods to be defined and discussed shortly, there is a need to select and assign subcubes to frontal matrix tasks from a number of available subcubes of the required size. An appropriate criteria upon which to base this selection is the maximum amount of contribution message passing that can be eliminated by a particular subcube's selection. However, load balancing concerns also require that the columns of the frontal matrix be evenly distributed across the subcube's processors.

In order to facilitate use of this selection criteria, each available subcube of the required dimension is analyzed to determine how much of the message passing from its child frontal matrices can be eliminated if the particular subcube is selected. The subcube with the greatest potential for message passing to be eliminated is then selected and assigned to the frontal matrix task.

As a message's communication cost grows linearly with the message's length, greater advantage is obtained by eliminating the longer messages (although specific scenarios can be developed where this criteria is not optimal). With this in mind, the evaluation of the child edges and the corresponding source frontal matrices is done in decreasing order of the message sizes. (This is my one of the reasons why the child edges were sorted by decreasing column degrees (number of rows) during the edge refinement process).

The other concern is that load balancing be maintained. To this end, each processor is only allowed to be assigned up to a specific share of the frontal matrix's N columns. This share is defined as

$$share(T_i) = \lceil \frac{N}{2^{dim\_alloc(T_i)}} \rceil .$$

With these concepts in mind, the evaluation procedure defined in Figure 6-2 is used on each candidate subcube for a particular frontal matrix. The subcube that eliminates the most message passing (as predicted by this procedure) will be assigned to the frontal matrix task. Furthermore, as this procedure makes some tentative column to processor assignments for each evaluation, it can be used with a special switch passed that causes the tentative column assignments to be made permanent. The logic for this permanent assignment function is not shown but would be invoked

by a call to the procedure with the finally selected subcube specified together with the special switch enabled.

OVERLAP DETERMINATION ALGORITHM:

```
for each child_edge by decreasing size do
    - load the column-to-processor assignments for the
    source frontal matrix task into a look up table
    for each column of the current edge do
        - look up the column's processor assignment in the
        source frontal matrix
        if (the column's processor assignment
                is within this subcube) then
            if (corresponding column in destination frontal matrix
                    is or can be assigned to same processor without
                    exceeding that processor's share of columns) then
                - include this column's message cost in that to be
                eliminated by this subcube's selection
                - assume this column-to-processor assignment in the
                destination frontal matrix for the rest of this
                routine's current execution
            endif
        endif
    endfor
    - clear the source frontal matrix's column to processor
    assignments from the look up table
endfor
```

Figure 6-2. Overlap Determination Algorithm

S1/S2: Assignment Flexibility

The concept of determining the sizes of subcubes allocated to frontal matrix tasks in blocks of tasks suggests the idea of also scheduling these blocks of tasks together. Within each scheduling block, all the parallel processors could be initially assumed to be available, which would provide a great deal of flexibility in selecting specific subcubes for tasks. This flexibility could be exploited using the subcube selection criteria just described. This would minimize the amount of required message communication by assigning the source and destination frontal matrix columns to the same processors whenever possible. The data passing then becomes local within the processor and the setup and transmission costs of message passing are avoided.

The S1/S2 scheduling/allocation method was designed to take advantage of this concept. Furthermore, there are actually two alternative schedulings determined for

each scheduling block and the most efficient of the two (the most work done per unit time) is chosen. The first of these two alternatives is the S1 method, which uses the smaller subcube allocations as earlier discussed and specified as $dim\_alloc(T_i)$. Use of these smaller subcubes insures that all the tasks in $Q_{eligible}$ can be scheduled within the scheduling block. This method emphasizes inter-task parallelism, however, it can under allocate subcubes to frontal matrices when only a small number of frontal tasks are ready for execution. In order to counter this deficiency, the S2 method uses the larger subcube allocations per task, $alt\_dim\_alloc(T_i)$. However, it does include provisions for smaller subcube allocations per task if doing so would improve efficiency.

Specifically, the S1 method takes each task T_i in $Q_{eligible}$ in order of largest allocated subcubes first. As each task is scheduled, its execution time, which is partially a function of the dimension of its subcube allocation, is used to see if the execution time for the block of tasks needs to be updated. Tasks using the same subcube dimensions are allocated to distinct subcubes unless either they can be done on the same subcube as an earlier task (with the same required subcube dimension) without extending the execution time for the block of tasks or if there are no subcubes that have not already been assigned tasks. In this latter case, the current task is assigned to the same subcube as an earlier task requiring the same dimension subcube and such that the total execution time of the set of identically assigned tasks is minimal. The specific assignments made for each subcube are determined using the subcube selection criteria and procedure defined in the previous section.

Once all the tasks in $Q_{eligible}$ have been scheduled and assigned subcubes via the S1 method, there may be idle processors. If this is the case, additional tasks (if available) are taken from Q_{ready}. Subcube allocations for these tasks are determined using the tasks' sequential workloads and the total sequential workload of the tasks originally in $Q_{eligible}$ for this block. Additional tasks are taken from Q_{ready} until either Q_{ready} is exhausted or the next task cannot be scheduled without extending the execution time of the current scheduling block. The total of the sequential work loads for all the tasks scheduled by S1 in this block is then divided by the execution time of the scheduling block to obtain a measure of the efficiency of the S1 schedule for the block; that is, how much work was done per unit time.

A second alternative scheduling for the scheduling block is developed using the S2 method. In this method, tasks are taken from $Q_{eligible}$ in critical path priority order and an attempt is made to assign the task to the subcube of dimension $alt\_dim\_alloc(T)$ that minimizes required message passing. If no subcube of the required size is available, successively smaller sized subcubes are tried. If use of the smaller sized subcube will not extend the execution time for the block of tasks or if it will improve the efficiency beyond that achieved by S1, the subcube assignment is made. Otherwise, the task is left unscheduled (and will be returned to Q_{ready} if the S2 schedule is chosen for the block). The efficiency of the S2 schedule for the block is determined by summing the sequential workloads for each task that was actually scheduled and dividing the sum by the execution time for the block. Of the schedules for the block developed by S1 and S2, the one with the greatest efficiency is used.

S3: Execution Time Minimization

While the S1/S2 scheduling/allocation method provides a great deal of flexibility in assigning subcubes, its implied barrier synchronization, while not actually enforced, can produce significant idle times at the end of blocks while subsequent frontal matrix tasks wait for their full complement of processors to become available. This situation is even more likely with the task graph defined by the assembly DAG as later tasks are typically larger and will use larger subcubes. Many earlier tasks will have been using these processors and the large tasks must wait for all of the predecessor tasks to complete, regardless of whether or not there is a data dependency between the tasks. Furthermore, UMFPACK's aggressive edge reduction in building the assembly DAG produces relatively small communication requirements so there is less advantage to be gained by reducing communication.

In response to these issues, a second scheduling/allocation method was developed to emphasize the efficient use of available subcubes. The S3 method attempts to allocate subcubes to frontal matrix tasks as soon as the subcubes become available. Selection between alternative available subcubes is done in a manner that maximizes the potential for reducing communication. Care is taken to avoid unnecessary fragmentation of the hypercube.

The key to the S3 method is the efficient management of subcube allocations and the ability to easily coalesce adjacent available subcubes into larger subcubes. An excellent method for doing this is the binary buddy system [95]. One of the interesting (and frequently useful) properties of the hypercube topology is that it can be split along any dimension into two subcubes of the next smaller dimension. This provides a great deal of flexibility in subcube allocations and extensions to the binary buddy system have been proposed to take advantage of this property [20, 2]. However, the added flexibility comes at a significant cost in terms of complexity of the allocation mechanisms. Furthermore, splitting the hypercube along multiple alternative dimensions can create complicated fragmentation scenarios that can limit availability of larger subcubes. For this application, the initial hypercube is split into successively smaller subcubes in a fixed ordering of the dimensions.

Specifically, the dimension represented by the most significant bits of node IDs are used first. For example, consider an initial hypercube of dimension four (16 processing nodes). Each node ID can be defined by a unique 4 bit string (i.e., $****$, where $*$ represents either a 0 or 1). If a subcube of dimension two (4 processing nodes) is required, the initial cube could be split into two subcubes of dimension three (i.e., the subcubes $0***$ and $1***$). One of these two 3-cubes could then be split further into two subcubes of dimension two and one of them allocated. Assume the $0***$ 3-cube is chosen, then it can be split into the $00**$ and $01**$ 2-cubes. In this manner, any subcube of dimension d can be uniquely defined by the most significant $n - d$ bits of its node IDs, where n is the dimension of the initial hypercube. That is, all of the nodes in the d-cube will have the same $n - d$ most significant bits with the d least significant bits generating each node's unique ID within the d-cube.

Furthermore, two subcubes of dimension $d - 1$ can be coalesced into a subcube of dimension d if they agree in their $n - d$ most significant bits. For example, the

2-cubes: 10^{**} and 11^{**}, can be coalesced into the 3-cube: 1^{***}, but the 2-cubes: 01^{**} and 11^{**}, could NOT be coalesced (actually they could be coalesced into the 3-cube $*1^{**}$ but that would violate our simplifying dimension ordering convention).

The binary buddy system is based on these principles. When managing a hypercube of dimension n, the binary buddy system would use $n+1$ lists with the ith list containing descriptions of the available subcubes of dimension i. Initially only the nth list is nonempty and it contains a single entry that describes the entire hypercube. When an allocation request is received for a subcube of dimension i, the ith list is checked. If it is nonempty, a subcube from this list is selected and assigned. (In this application, the selection is done in a manner that maximizes the overlap potential with predecessor tasks with which there is a data dependency). However, if the ith list is empty, the $(i+1)$st list is checked, and then the $(i+2)$nd list until either a nonempty list if found or the nth list is reached and found to be empty. In this latter case, no acceptable subcube is available and the allocation request is postponed or rejected. If a list with higher dimension subcubes is found to be nonempty, ALL of the available subcubes on this list are fragmented into subcubes of dimension i and the i-cube that maximizes potential overlapping is selected. The nonselected i-cubes are then coalesced as much as possible. This is a departure from the classical binary buddy system where only one of the larger dimension subcubes would be fragmented at each intermediate dimension until the requested dimension is reached and an appropriate subcube allocated. The classical approach precludes the need to coalesce subcubes after the allocation, but provides less assignment flexibility than this adaptation.

With this binary buddy system in place to manage subcube allocations, the S3 scheduling method can be easily defined. First, the subcube dimension selected for each frontal matrix task T_i is defined to be the previously determined value: $dim\_alloc(T_i)$. (The $alt\_dim\_alloc(T_i)$ is not used by S3). As the subcube allocations for the set of tasks in $Q_{eligible}$ are made, their outgoing edges are considered satisfied and any freed subsequent tasks are added to Q_{ready}. This continues until subcube dimensions have been determined for all the tasks. Then Q_{ready} is rebuilt with the initially ready tasks and the actual scheduling is initiated. Tasks are taken from Q_{ready} based on their critical path priorities and their subcube assignments made using the binary buddy system just described. As each task is assigned its subcube, its completion time is determined based on a simulated current time and the task's predicted execution time, which was determined earlier and is a function of the dimension of the subcube it has been allocated. An ordered queue of these completion events is maintained. When a task's subcube assignment cannot be made or Q_{ready} has been exhausted, the next completion event is taken from the ordered queue. The simulation time is advanced accordingly and the corresponding subcube released back to the binary buddy system where it is coalesced into larger subcubes to the maximal extent possible. The outgoing edges of the completed task are considered satisfied and any released tasks are added to Q_{ready}. Additional allocation are then attempted as before and this process continues until all tasks have been scheduled and assigned subcubes.

The S3 scheduling/allocation method emphasizes the scheduling of tasks as soon as appropriate subcubes are available and thus should effectively reduce execution time. Furthermore, the modifications made to the classical binary buddy system allocation mechanism provide greater assignment flexibility that is used to maximize the potential for communication reductions.

Near-Edge-Only Consideration

The *near-edge-only consideration* option was developed in response to an observation that eliminating contribution message passing is much more important when the source task has only recently completed. In this case, the receiving task could be delayed while waiting for the message transmission to take place. However, if the source task had completed a long time ago, its contribution message would have had sufficient time to be already delivered to the communication buffer of the receiving task's processor.

Near edges can thus be defined as edges from child frontal matrix tasks (preceding tasks with which there is a data dependency) such that the child task's completion time together with the maximum message transmission time are greater than or equal to the start time of the destination task. In other words, the message satisfying the data dependency represented by the edge may not be received by the time the receiving task is ready for it.

Thus, when the near edge only consideration option is enabled, the determination of overlapping assignment potential can be limited to only child frontal matrix tasks with such near edges. This increases the chances that a near edge will be eliminated by overlapping and thus reduce overall execution time. Furthermore, as this option reduces the scope of the overlap potential determination, the amount of time required for the scheduling process is also likely to be reduced.

6.1.3 Data Assignments

Once a subcube has been selected and assigned to a particular frontal matrix task, the frontal matrix itself must be distributed to the processors of this subcube. This data assignment needs to be done as soon as the subcube assignment is made so that subcube selections for subsequent frontal matrix tasks can determine how their columns will overlap with those of previously scheduled frontal matrices. Furthermore, data assignments will be done by frontal matrix columns in adherence with the conventions and reasoning presented in the previous chapter on the partial dense factorization kernels.

In the development of the partial dense factorizing kernels, a scattered strategy was used. The scattered method places adjacent columns of the frontal matrix on consecutive processors. Alternatively, a blocking method would place adjacent columns of the matrix onto the same processor until that processor has obtained its share of the data.

However, the partial dense factorizing kernels did not address the passing of contribution blocks between frontal matrices. As each column of each contribution edge is potentially a distinct message, there is the possibility of a great deal of message traffic for contributions. Furthermore, in the current state of the art distributed memory

multiprocessors, such message communication is expensive compared to computation time, especially as the assembly of contributions only requires a single addition operation per contribution entry.

Hence, data assignment schemes that can reduce message communication requirements may have beneficial effects on overall execution time. To this end, a number of special data assignment mechanisms were developed and can be optionally selected as part of the user's call to the SCHEDULE_REFACTOR routine. These mechanisms include overlapping, clustering per children, and clustering per parents. Any columns left unassigned by the user's selection of these mechanism would be assigned using the default assignment scheme. The default data assignment scheme is also user-selectable with the alternatives being either scattered or blocked.

Overlapping

The idea of overlapping attempts to place the columns of the current frontal matrix on the same processor to which they were assigned in a child frontal matrix that makes a contribution to this column. Maximum overlapping is the criteria used to make the selection between alternative subcubes. In fact, the procedure that makes this evaluation also makes the overlapping data assignments. A flag parameter passed to this routine directs the tentative column-to-processor assignments that were made to evaluate the potential overlapping to be retained as permanent assignments.

The benefit of overlapping assignments is that no message passing is required for the contributions made between the overlapped columns of the two frontal matrices.

Clustering Per Children

Another way to reduce message communication requirements for contributions is to combine multiple columns of a specific contribution into a single message. This eliminates the additional message setup costs that would of been required if each column of the contribution was passed with a distinct message. The term: *clustering* is used to refer this collection of multiple contribution columns into a single message.

In order for clustering to be accomplished, multiple columns of a contribution must all reside on a single processor in the source frontal matrix and all reside on a single processor in the destination frontal matrix. When data assignments are made, the source column to processor assignments have already been established. The maximum clustering potential for each child edge is evaluated. The clustering potential is defined as the greatest number of contribution columns that reside on a single processor in the source frontal matrix. These clusterings are used for data assignment in decreasing order of their potentials. Furthermore, as each new set of assignments is made, the clustering potential across all the edges can change and must be updated.

When attempting to assign a clustered set of columns in the destination frontal matrix, a best fit criteria is used based on the current state of partial assignments. If the entire set of clustered columns cannot be assigned to a single processor without exceeding the processor's share of columns, the processor that can hold the largest number of the clustered columns is selected. In locating clustered columns, care is taken to not include columns already assigned in the destination frontal matrix.

Clustering Per Parents

When using clustering per children, data assignments are made based on the clustering already available in the predecessor frontal matrices. Furthermore, the effectiveness of clustering per children is very much affected by how well the columns of a contribution edge are already clustered in the source frontal matrix. Thus, it would make sense to try and make data assignments that would maximize the opportunity for clustering when data assignments in subsequent parent frontal matrices are made. That is, unassigned columns that are all passed (either partially or in their entirely) to the same parent frontal matrix should be assigned to the same processor. Not only should this improve the possibility of effective future clustering per children but it also should improve the possibility that all these columns can be overlapped on the same processor and thus eliminate the associated message passing in its entirety.

When doing clustering per parents data assignments, the edges with the largest column degree (number of rows) are considered first (again, taking advantage of the edge sorting that was done during the edge refinement process). These assignments of columns to processors are also done using a best fit strategy. However, if some of the columns of the cluster have already been assigned to the same processor, that processor is used to hold as much of the rest of the cluster as possible. Again, however, no processor is assigned more than its share of columns.

6.1.4 Launching the Parallel Factorization

Once scheduling/allocation/assignment process is complete, we have schedules for each parallel processor that take the form of a list of frontal matrix IDs per processor and frontal matrix column to processor assignments that have been stored within the frontal matrix descriptions. This information together with the rest of the frontal matrix description will be sent to the parallel processors as part of the *launching* process.

The first step in the launching process is to allocate a subcube of the appropriate dimension (as specified by the user in the call to SCHEDULE_REFACTOR). This is done using the nCUBE 2 *nopenx* routine [111]. Once the subcube has been allocated, the nCUBE 2 *rexecl* routine is used to remotely executed the parallel refactorization and triangular solve code (designated PRF) on the hypercube processors.

With the parallel code running on the subcube's processors, the next step is for the host to send them the INITIAL_REFACTOR command. This command informs the parallel processors to accept a new factorization structure and then perform the associated factorization. The command is augmented with four global scalar values that are the order of the matrix, the number of frontal matrices, the number of diagonal blocks, and the pivoting threshold. Also, passed with this command are the permutation matrices P and Q. With this information, the parallel code can begin to allocate some of its dynamic data structures while the rest of the launch messages are built and sent out.

The next of these launch messages contains the schedule, which is unique to each parallel processing node. The schedule consists of a list of the frontal matrix IDs that the node will participate in factorizing.

Once the node has received its schedule, it knows both the total number and the specific IDs of the frontal matrices that it will factor. Hence, the next step is to send each node a description of each frontal matrix that it will work on. There is a distinct message for each frontal matrix and it is sent only to the nodes that will participate in its factorization. The frontal matrix descriptions include information such the numbers of rows, columns, and pivots; a dimension and mask defining the frontal matrix's assigned subcube; the number of both children and parent edges incident to the frontal matrix in the assembly DAG; the IDs of the rows and columns that constitute the frontal matrix; the column to processor assignments; and the detailed descriptions of each edge. The edge descriptions include the lengths of the edge's contribution row and column patterns, the actual contribution row and column patterns, and the contribution column locations (processing node IDs) within the frontal matrix that is connected by the edge.

Finally, a file containing the original matrix values is read by the host program. Each entry, which is specified in (row, column, value) triplet format, is looked up in the frontal matrix patterns to determine which frontal matrix and which processor will hold it. Entries that do not reside in a diagonal block are marked with a special tag and distributed to the parallel processing nodes in a row scattered format. While not used in the factorization process, these entries are required for the subsequent triangular solves and the row scattered format will provide some parallelism. Furthermore, the row and column indices are also permuted by the P and Q permutations, respectively. Lists are built of the entries to be sent to each parallel processing node. Once these lists are completed, they are used to build and send distinct messages for each processing node.

This completes the launch process. The remainder of the SCHEDULE_REFACTOR routine simply waits for a status message reply from the parallel processing nodes and passes the received success or failure status to the user process.

6.2 Parallel Factorizations

While the host processor performs the scheduling, allocation, assignment, and launching of the factorization(s), the actual factorization process is performed on a set of parallel processors connected via a hypercube topology interconnection network. The actual calling sequence is similar to that employed for remote procedure calls [115]. A host-based user processor calls one of the four host resident routines that provide the interface to the parallel factorization code. Once these routines send the appropriate messages to the parallel processors, they wait for a response and return the received results via the standard function return value.

The primary difference between this implementation and a classical remote procedure call is that the host resident interface routines, (SCHEDULE_REFACTOR, SUBSEQUENT_REFACTOR, TRIANGULAR_SOLVE, and TERMINATE_REFACTORS), do more than simply packing parameters into messages, sending them, and waiting for and returning the results. As was just discussed, the SCHEDULE_REFACTOR routine does a significant amount of processing to obtain and refine the assembly DAG description and to develop a static schedule and set of assignments as well as to launch the parallel processes and send them the necessary

data. The SUBSEQUENT_REFACTOR routine must also read a file of new matrix entry values, determine their destination processor and frontal matrix (or list of off diagonal block entries), and send each processor its set of values. Likewise, the TRIANGULAR_SOLVE routine, which will be discussed shortly requires some additional processing.

Another difference is that the remote procedures invoked by calls to these interface routines reside and execute on multiple processors. Within each of these parallel processors, the same program is running but each will have their own portion of the data. This programming model of a single program resident on each processor but with different data is referred to as a *Single Program, Multiple Data (SPMD)* model [110]. Yet each processor is free to execute instructions independent of the other processors and any required synchronization is handled via explicit calls to message passing directives. This concept of independent instruction execution, when combined with the distribution of different data to each processor, is defined by another model that governs this implementation, the *Multiple Instruction, Multiple Data (MIMD)* model [1].

With this conceptual framework established, the rest of this section will describe the implementation of the primary functions of the parallel refactorization code. This discussion will begin with a description of the primary data objects used by the parallel refactorization code followed by an overview description of the parallel refactorization algorithm. The primary subfunctions required in the refactorization algorithm are then detailed to complete the discussion.

6.2.1 Schedules and Frontal Matrix Descriptions

There are two primary types data objects utilized by the parallel refactorization code (PRF). The first is the schedule, which is essentially an ordered list of the frontal matrices that each processor will factor. The second object type is the frontal matrix description. These descriptions detail not only the frontal matrix but also the subcube that will cooperatively perform the factorization and the edges to other frontal matrices that will either provide or receive contributions to/from the frontal matrix.

In the implementation and distribution of these data objects there are two primary concerns. The first is to provide each parallel processor with only the specific data object instances that it requires. This allows large problems to be broken up into smaller pieces and the pieces distributed across processors. If done effectively, each processor will only require a limited amount of memory to accomplish its portion of solving the overall problem.

The second concern is flexibility. In this baseline implementation, the schedules, allocations, assignments, and interfaces are all known a priori and need not change during the execution. However, in the next chapter, mechanisms for recovering from lost pivots will be discussed. These may require alterations to schedules, assignments, and interfaces to be made dynamically during the factorization process. Each processing node must have enough information to be able to assist in determining what alteration is required. The data object implementation must also allow the data objects to be modified in accordance with any required alterations.

With these concerns in mind, the implementation of the schedule data object is simply an integer vector of frontal matrix IDs. Each processor will have an instance of this object that contains only the frontal matrix IDs that it will factor. Whether the particular processing node will factorize the entire frontal matrix or participate with other processors in an encompassing subcube is not reflected in the schedule. This information is held in the frontal matrix descriptions.

The frontal matrix descriptions control all processing required for the frontal matrix. Each processing node will only hold the frontal matrix descriptions that it will work on. Furthermore, the processing node will only allocate space to hold its portion of the frontal matrix and to retain its portion of the LU factors resulting from factorization. In this way, the memory requirements per processing node are much less than the memory requirements for the entire problem. If this is done effectively, the algorithm is said to *scale* its data requirements well. That is, the algorithm has good *data scalability* properties. As one of the important features of a distributed memory multiprocessor is its ability to provide large amounts of memory, the data scalability of the PRF code will be a key performance measure used in assessing the code's ability to effectively utilize the additional memory to accommodate larger problem sizes.

The other major considerations are for each processor to have sufficient information to facilitate lost pivot recovery and for frontal matrix descriptions and their components to be modifiable in both size and content to accommodate any necessary alterations. For this reason each frontal matrix description includes information describing all of its interfaces to other frontal matrices (even if the particular processing node does not participate in the interface) as well as descriptions of how the entire frontal matrix is assigned. Furthermore, components of the frontal matrix description that may need to be altered in size are distinctly allocated from dynamic memory (heap). In order to access frontal matrix descriptions quickly, each processing node uses a vector of pointers to its frontal matrix descriptions. The vector index (equal to the frontal matrix ID) is used to identify the pointer to a particular frontal matrix description. If the processing node does not work on a particular frontal matrix, the vector entry corresponding to that frontal matrix will be NULL. This mechanism allows quick indexed access to frontal matrix descriptions as well as room for adding additional descriptions if necessary.

The actual frontal matrix description is a record (implemented using the C language *struct* directive) with component fields defined as follows:

- A: Pointer to the frontal matrix storage dynamically allocated to hold this processor's portion of the frontal matrix's values.
- *fid*: This frontal matrix's ID (diagonal index of its first pivot).
- m: Number of rows in the frontal matrix.
- n: Number of columns in the frontal matrix.
- g: Number of pivot steps to be applied to this frontal matrix.

- *dim*: Dimension of the subcube to be used to factor the frontal matrix.
- *mask*: Bit mask to identify the bits that vary in the processing node IDs within the subcube that will factor the frontal matrix.
- *num_childs*: Number of frontal matrices providing contributions to this frontal matrix (number of child edges).
- *num_parents*: Number of frontal matrices that will receive contributions from this frontal matrix (number of parent edges).
- *my_node*: Processing node ID of this processor relative to the subcube assigned to the frontal matrix (determined by ANDing the node's global ID with the mask defined for the frontal matrix's subcube).
- *my_g_cnt*: Number of the anticipated pivots that are held in this processing node.
- *my_g_actual*: Number of the successful pivots that are held in this processing node.
- *my_cols*: Number of this frontal matrix's columns that are held by this processing node.
- *stride*: Number of entries allocated for each column in the column major storage organization for the frontal matrix.
- *cube_assigns*: Pointer to an array of length n (indexed by local column IDs) that defines the processing node ID to which each column of the frontal matrix is assigned. (Local row and column IDs are the sequentially assigned, zero-based indices into the frontal matrix).
- *row_ids*: Pointer to an array of length m (indexed by local row IDs) that defines the column pattern of the frontal matrix.
- *col_ids*: Pointer to an array of length n (indexed by local column IDs) that defines the row pattern of the frontal matrix.
- *row_permutes*: Pointer to an array of length m (indexed by local row IDs) that defines row permutations made within the frontal matrix.
- *col_permutes*: Pointer to an array of length n (indexed by local column IDs) that defines column permutations made within the frontal matrix.
- *my_matrix_cols*: Pointer to an array of length *my_cols* that defines the frontal matrix columns held within this processing node.
- *my_pivot_cols*: Pointer to an array of length *my_g_cnt* that defines the local column IDs of the pivot columns held by this processing node.

- *local2frontal*: Pointer to an array of length n that maps the columns held by this processing node to their positions within the frontal matrix.
- *subcube2cube*: Pointer to an array that maps subcube-relative node IDs to their node IDs within the hypercube allocated to the entire factorization.
- *turns*: Pointer to an array that contains the number of yet-to-be processed pivot columns held by each participating processing node.
- *orig_value_list*: Pointer to a list of the original values that fall within this frontal matrix. (Each list entry contains the local row and column IDs of the entry and its value).
- *childs*: Pointer to an array of the frontal matrix IDs that provide contributions to this frontal matrix.
- *parents*: Pointer to an array of the frontal matrix IDs that receive contributions from this frontal matrix.
- *child_edges*: Pointer to a list of edge description records that describe each child edge. (Ordering of this array is the same as that of the *childs* array).
- *parent_edges*: Pointer to a list of edge description records that describe each parent edge. (Ordering of this array is the same as that of the *parents* array).
- *num_local_childs*: Number of child edges that provide contributions to columns held by this processing node.
- *local_childs_list*: Pointer to an array that contains offsets into the full set of child edges descriptions for each local child edge.
- *local_childs*: Pointer to an array that contains the IDs for each frontal matrix that provides a contribution to a column held by this processing node.
- *num_local_parents*: Number of parent edges across which contributions from columns held by this processing node will be sent.
- *local_parent_list*: Pointer to an array that contains offsets into the full set of parent edge descriptions for each local parent edge.
- *expected_contributions*: Number of contributions columns expected to be received by this processing node for the frontal matrix.
- *received_contributions*: Number of contribution columns that have been received by this processing node.
- *outstanding_local_contribs*: Number of local contributions sent from the frontal matrix columns on this processing node to other frontal matrices' columns also assigned to this processing node.

- *local_contrib_list*: List of local contribution descriptions for contributions from columns of child frontal matrices that are also assigned to this processing node. (Each local contribution description will hold the source frontal matrix ID and a point to the source frontal matrix's edge description for this contribution).

- *pre_read_msg_list*: List of pointers to messages in the communication buffer that contain contributions to this frontal matrix but were obtained prematurely via an *nreadp* operation issued during processing of an earlier frontal matrix.

- L: Pointer to the storage area used to hold portions of the LU factors that correspond to the frontal matrix pivot columns held by this processing node. In addition to holding the appropriate columns of L, this area also holds the corresponding columns of U that fall within the frontal matrix's pivot block.

- U: Pointer to the storage area used to hold the remaining partial columns of U that are contained within the frontal matrix columns held by this processor but that lie outside of the pivot block.

Two of the fields within the frontal matrix descriptions are pointers to edge descriptions. These edge descriptions are of sufficient complexity to require some further discussion. Specifically, there is an edge description for each child and parent edge incident to the assembly DAG node that corresponds to the frontal matrix. Each edge description contains:

- *type_of_edge*: Defines the edge as corresponding to either a L, U, or LU relationship.

- *row_pattern_len*: Number of columns in the contribution passed on the edge.

- *col_pattern_len*: Number of rows in the contribution passed on the edge.

- *row_pattern*: Pointer to an array with the column IDs of the columns that are included in the contribution of the edge.

- *col_pattern*: Pointer to an array with the offsets to the frontal matrix rows corresponding to the rows of the contribution of the edge.

- *col_location*: Pointer to an array with the processing node assignments for each contribution column in the other frontal matrix.

- *local_row_pattern_len*: Number of the edge's contribution columns that are held by this processing node.

- *local_row_pattern*: Pointer to an array that holds offsets to the edge's contribution columns that are held by this processing node. (These last two components are provided to expedite the assembly of contributions during the factorization process).

While the schedule and frontal matrix description data objects are not the only data objects in use by the parallel refactorization (PRF) code, they are the primary ones. Furthermore, they are limited to only the frontal matrices to be operated on by the particular processing node and include storage for only the appropriate portion of the frontal matrix and the corresponding parts of the LU factors. Hence, they should provide a good degree of data scalability for the PRF code.

Both the schedule and frontal matrix description data objects are built by routines that receive and interpret the corresponding messages sent from the host processor. This process is fairly straight forward and discussion of the details of these routines is not necessary. The most significant processing within the parallel refactorization code is contained within the refactorization routine that uses the data objects just described to perform the refactorizations. It is this portion of the code that will be discussed in detail and will be the focus of the performance evaluation.

6.2.2 Refactorization Process

The parallel refactorization routine description begins with an overview of the refactorization process with detailed discussion of the major subfunctions provided in the subsequent subsections. This discussion of subfunctions will include an assessment of the subfunction's potential parallelism.

With the schedule, frontal matrix descriptions, and other necessary data objects in place, the refactorization routine can commence on each of the parallel processors. A summary of the parallel refactorization routine is provided in Figure 6-3.

The inner *for* loop of this algorithm executes on each of the parallel processors and the number of iterations is governed by the number of frontal matrices assigned to the processing node. The outer *for* loop specifies the parallel execution on each processing node.

Frontal Matrix Storage Allocation

The first step in the body of this loop is to allocate and clear the storage needed for this processing node's portion of the frontal matrix. This is done via the *calloc* routine of the C programming language. This storage area will persist throughout the factorization of the frontal matrix and until all of its contribution block has been forwarded to other frontal matrices via message passing to other processing nodes or direct assembly into other frontal matrices on this processing node. The actual allocation and clearing done by the *calloc* routine consists of a fixed overhead for the allocation and an additional time required for the clearing that is proportional to the amount of storage to be cleared. Obviously, the fixed overhead of the allocation will be unaffected by allocating additional processors to the frontal matrix, but the clearing process time should scale nicely with additional processors since the storage requirements will be near evenly distributed across the processors.

Assembly of Original Entries

The assembly of original matrix entries is a simple process of traversing the list of those entries, using the row and column indices to compute the entries offset into the frontal matrix storage area, and then adding in the value. As more processors

```
PARALLEL REFACTORIZATION ALGORITHM

for each node do
    for each frontal matrix in node's execution list do
        - allocate/clear frontal matrix storage
        - assemble original entries
        - assemble contributions from other frontal matrices
        if (only one pivot row) then
            if (only one row) then
                - handle prefactored frontal matrix
            else
                - single pivot partial factorization
            endif
        else
            - partial dense factorization
        endif
        - forward contributions
        - retain LU factors
    endfor
endfor
```

Figure 6-3. Parallel Refactorization Algorithm

are allocated to the frontal matrix, the original values will hopefully be fairly evenly distributed across the columns and evenly divided across the processing nodes. However, when a blocked allocation of columns to processors is in use, the pivot columns of the frontal matrix, which typically make up more than half of the target for original values, would be assigned on a relatively small set of processors. Thus, this subset of processors would have a disproportional amount of original entry assemblies to do.

Assembly of Contributions

The assembly of contributions from other matrices is a more difficult task and constitutes a much larger portion of the total required assemblies. Furthermore, implied within the contribution assembly process is also checking that the required precedence constraints have been satisfied. These precedence constraints are specifically the data dependencies represented by the edges in the assembly DAG. They are resolved in one of two ways. If a contribution is coming from a portion of a frontal matrix that was assigned to the same processing node, the node's schedule will implicitly guarantee that the contribution is available. If the contribution is coming from another processing node, it will arrive as a message with the type of the message identifying the destination frontal matrix ID. The contribution assembly routine will do a blocking message receive operation to insure that all contributions have been received prior to

proceeding with the factorization process. All local contributions are assembled first as they are guaranteed to be available due to the implied synchronization imposed by the node's schedule. Then the contribution messages are read. Furthermore for reasons of efficiency, the *nreadp* operation is used to read these messages. This operation simply returns a pointer to the message in the communication buffer and precludes the normal copying of the message from the communication buffer to a user buffer.

A key element of this mechanism is the ability to selectively read messages for a specific frontal matrix. The nCUBE 2 allows user message types to be in the range 0 to 32767 and also for only messages of a specified type to be read [110]. Therefore, message types in the range of 0 to 29999 are reserved for the passing of contributions. A contribution message to be sent to the frontal matrix who's ID is i would be given a message type of i mod 30000 and the receiving frontal matrix would use this same message type on its contribution message read operations. However, when the order of the matrix is greater than 30000, the message type may not be unique for each destination frontal matrix. Thus, the true destination frontal matrix ID is contained within the body of each contribution message and this value is checked following the message read. As the contribution messages are also sent to specific destination processors, a read message that is not for the current frontal matrix will be for a subsequent frontal matrix on the same processing node. The pointer to the message is then inserted on the *pre_read_msg_list* for the true destination frontal matrix. As part of the assembly process, this *pre_read_msg_list* will be checked before any message reads are posted.

The format used for these contribution message is of variable length and defined with the regular expression

$$dest\_fid\ src\_fid\ num\_cols(col\_id\ col\_len\ (contrib)^{col\_len})^{num\_cols}$$

where $dest\_fid$ and src_fid are the destination and source frontal matrix IDs respectively, num_cols is the number of contribution columns included in the message, and for each such column there is a column ID (col_id), a column length (col_len), and the series of col_len contribution values shown as $(contrib)^{col\_len}$.

The specific processing details of the contribution assembly routine are shown in Figure 6-4. The edge refinements performed during the host's preprocessing, together with the additional refinements to distinctly describe local child edges, allow the assembly process to be sped up significantly as these refinements include local row and column offsets into the frontal matrix storage area.

Within the ASSEMBLE_CONTRIBUTIONS routine, incoming messages are not read in a prescribed order. Since floating point addition is not associative, this can lead to nondeterministic results. Deterministic behavior can be enforced by prescribing the order in which incoming contribution messages are read, but this would likely introduce additional idle time. As the effects of this nondeterminism are anticipated to be minimal, the approach seems reasonable.

Once the assembly of all the contributions is completed, we are assured that all prerequisite data dependencies have been satisfied and may commence factorization of the frontal matrix.

While contribution edges initially result from contribution entries that reside within the pivot rows or columns of the frontal matrix, the aggressive edge reduction done by UMFPACK [34, 35] causes contributions to be possible for any entry within the frontal matrix. Thus, contributions can be spread across all frontal matrix columns with beneficial effects on parallelism. However, the distribution of these contributions is highly problem specific, so no hard and fast parallelism results can be predicted.

Partial Dense Factorizations

Factorization of the frontal matrices is done using the partial dense factorization kernels developed in the preceding chapter. Special cases include when only a single pivot is involved. If this is the case, but there is only a single row in the frontal matrix, there is essentially no work to be done except for updating some of the results fields in the frontal matrix description. In the case of a single pivot with more than one row, the specially tuned P5 single pivot factorization routine is employed. In any other case, a more general routine is required. As was discussed in the preceding chapter, the P1, P2, P3, and P4 partial dense factorizing routines all accomplish this function. However, P3 and P4 also check the the pivots against the threshold pivoting criteria and attempt lost pivot recovery within the frontal matrix. The P2 and P4 routines also employ the concept of pipelining to improve performance. Any of these four routines may be selected as the partial dense factorization routine through the setting of compile time switches.

The potential parallelism within these routines was discussed in Chapter 5 and need not be addressed here.

Contribution Forwarding

Once the frontal matrix has been factored, its contribution block must be forwarded to the appropriate frontal matrices. The results of the edge refinement done as part of the host's preprocessing are of benefit here, as are the further refinements of these descriptions to detail the specific components of the edges that involve contributions local to the particular processing node. A particular processing node need only concern itself with these local parent edges.

For reasons of efficiency and performance, all contributions passed from one processing node for a particular frontal matrix to a distinct processing node/frontal matrix combination should be combined into a single message. The processing of each edge requires four distinct phases for each local parent edge. In the first phase, the local row pattern from the edge description is traversed to determine how much storage is required for each destination processor on that edge. This message size requirement is then used by the second phase to allocate message buffers in the communication buffer area. The third phase does another sweep of the local row pattern and places the appropriate information into the message buffers. The format of these messages is exactly that described earlier in the contribution assembly routine discussion. Finally, the fourth phase sends out the messages. Since a blocking receive is

used to read these messages by the destination frontal matrix, an asynchronous message send is sufficient and appropriate here. Thus, this routine can proceed without delay.

During each of these phases, a check is made to determine if the column's destination is the current processor. If this is the case, the first three phases simply skip over any further processing of the column. In the fourth phase, the source frontal matrix ID and a pointer to the corresponding edge description are enqueued on the local contribution list of the destination frontal matrix description. Then a counter is also incremented in the source frontal matrix description to indicate that a local contribution is pending and must be assembled before the storage for this frontal matrix can be released.

An algorithmic description of the process of forwarding contributions is provided in Figure 6-5.

As each entry in the contribution block of a frontal matrix must be passed once and only once to a subsequent parent frontal matrix, a good distribution of the contribution block columns across processors will aid in the parallelism achieved in this process. Furthermore, as the number of frontal matrix entries is typically dominated by its contribution block, there should typically be a very strong possibility that the columns of the contribution block will be well distributed with any of the data assignment schemes. However, as more processors become involved, there is a lessened chance at overlapping columns across frontal matrices. Thus, more message passing is involved and it requires greater overhead. This limits the gains available via parallelization.

Retaining the Frontal LU Factors

The final step in the loop iteration for a particular frontal matrix is to retain the LU factors associated with the locally held portion of the frontal matrix. This storage was already allocated during the creation of the frontal matrix description, so the process only really entails a copy operation. The *Scopy/Dcopy* operations from the Basic Linear Algebra Subroutines are used here for efficiency and performance. Besides the copying function, the number of outstanding local contributions is also checked. If there are no outgoing local contributions for this processing node, its frontal matrix storage area may be released since all of its contributions will have already been sent out as messages.

If the columns of the LU factors are nicely distributed across the subcube processors assigned to this frontal matrix, this procedure should parallelize very well as the overhead associated with the copy operations in minimal. However, data assignment schemes such as blocking can adversely affect this distribution.

```
ASSEMBLE CONTRIBUTIONS

for each local contribution do
    - use src_fid to look up destination edge description
    for each local col in source edge description do
        - look up offset in local destination col look up table
        if (both src and dest cols are local to this node) then
            - determine address of src and dest cols
            for each edge entry do
                - assemble in contribution
            endfor
            - increment received contribution count
        endif
    endfor
    if (that was the last contribution from src frontal matrix) then
        - release the storage for src frontal matrix
    endif
endfor
while (more contributions are expected) do
    - get next msg from preread msg queue or received msg buffer
    - extract src frontal ID and num_cols
    - look up col pattern for this edge
    repeat num_cols times
        - extract col_id and col_len
        - look up col_id's local offset in dest frontal
        - determine address of dest col
        for each of the col_len contributions do
            - assemble next contribution entry per pattern
        endfor
        - increment received contribution count
    endfor
endwhile
```

Figure 6-4. Assemble Contributions Algorithm

```
FORWARD CONTRIBUTIONS

for each local parent edge do
    for each local column of this edge do
        - determine destination node for this column
        - increment message buffer size needed for destination node
    endfor
    for each destination node for this edge do
        if (destination node is remote) then
            - allocate message buffer in communication area
        endif
    endfor
    for each local column of this edge do
        if (destination node is remote) then
            - put header into next slot in destination
            node's message buffer
            for each contribution in edge column pattern do
                - deposit contribution into the message buffer
            endfor
        endif
    endfor
    for each destination node for this edge do
        if (destination node is remote) then
            - send the message for this destination node
        else
            - enqueue local contribution to destination frontal matrix
        endif
    endfor
endfor
```

Figure 6-5. Contribution Forwarding

Synchronization Issues

As was discussed in the details of the contribution assembly process, the data dependencies represented by the edges of the assembly DAG are satisfied via the use of special message types for contribution messages and the use of a blocking receive operation to read these messages. This is not, however, the only synchronization issues that need to be addressed. Specifically, the partial dense factorization (PDF) kernels for different frontal matrices all use the same message type for broadcasting their multipliers. It is critical that the (PDF) messages for different frontal matrices do not get confused with one another.

Furthermore, the algorithm just outlined does not impose any barrier synchronizations on the subcubes for specific frontal matrices prior to the factorization of the frontal matrices. Correct processing must be guaranteed even if only some of the processors within a frontal matrix's subcube have started factorization while the others are still working on an earlier factorization. These two concerns are related and can be easily resolved once some prerequisite facts have been established. Theorems 6.1 through 6.3 establish the prerequisite facts. Theorem 6.4 resolves the lack of barrier synchronization question and Theorem 6.5 then follows directly and resolves the (PDF) broadcast message confusion issue.

Theorem 6.1 (Sequential Delivery of Broadcast Messages) If, in the nCUBE 2, two messages are broadcasted from a single node to an encompassing subcube(s), any node common to the receiving subcubes will receive the messages in the order they were sent.

PROOF: The broadcast mechanism of the nCUBE 2 communication hardware is built on the concept of *wormhole routine* [109]. A mask for each broadcast defines the "don't care" bit positions with ones and the "must match" bit positions with zeroes. When a broadcast is initiated from a particular node, it is first sent to each of the node's outbound channels that correspond to a single "don't care" bit setting. All these outbound channels must be available for communication to progress on any one of them. The process is repeated at the receiving nodes for the outbound channel numbers higher than that on which the message was received until all destination nodes have been reached. Furthermore, all acquired channels remain allocated to the broadcast until the entire message has been passed. Hence, there is a fixed path from any broadcast source to each of its destinations. The first broadcast message sent will be the first to acquire the paths to each destination and thus be assured of delivery prior to any subsequent broadcast from the same source.□

Theorem 6.2 (Well-Defined PDF Broadcasts) The broadcast message source nodes for each broadcast in a partial dense factorization (PDF) routine are well and consistently defined in all nodes of the participating subcube.

PROOF: In the P1 and P2 kernels, the pivot ordering is fixed to that which was originally assigned in the problem instance. The sequence of broadcast sources is thus uniquely and consistently defined by the processor assignments of each pivot

column. In the P3 and P4 kernels, the pivot ordering may change due to the numerical inadequacy of anticipated pivots as determined using the threshold pivoting criteria. In both cases, the pivot column chosen to replace a lost pivot is selected based on a minimal forward distance criteria using a parallel prefix minimum operation. This mechanism insures that the order of broadcast sources for both the lost and alternative pivots are well defined and consistent in each participating processor. Finally, the P5 kernel assumes a single pivot, with the source of the resulting broadcast uniquely determined by the processor assignment of that pivot's column.□

Corollary 6.3 (Complete Subcube Participation in PDF Broadcasts) All nodes in the subcube performing one of the partial dense factorization (PDF) kernels will participate as a sender or receiver in each required broadcast.

PROOF: As the sequence of broadcasts for each of the five PDF kernels is well defined and consistent for each participating node in the subcube from Theorem 6.2 and as each of the kernels is written to participate in each such broadcast, the PDF kernel running on each node in the subcube will correctly participate in each broadcast and not exit until the full sequence of broadcasts has completed.□

Theorem 6.4 (Frontal Matrix Subcube Synchronization) Explicit barrier synchronization is not required for the subcube nodes assigned to specific frontal matrices.

PROOF: The first frontal matrices factored by each node are well defined by the schedule and the specific source blocking broadcast receives of the PDF kernels assure that each node waits for the correct first broadcast message to read. Any other broadcast messages received will be buffered sequentially by source per Theorem 6.1. Thus, synchronization of the first PDF kernels on each processor are achieved. Once the first kernels are started, the sequences of broadcasts are well defined per Theorem 6.2 and no participating processor will proceed to a subsequent frontal matrix's PDF kernel until it has participated in the completion of each broadcast for the first PDF kernel per Corollary 6.3.

At the conclusion of each PDF kernel, the nodes will proceed to the next frontal matrix, which is well defined by its schedule. Within the PDF kernel for this subsequent frontal matrix, the broadcast sequence is again well defined. If any of these broadcasts had been already received in the communication buffer, they will be read in the same order that they were produced and received per Theorem 6.1. As this production order is well defined with modifications only possible if all affected nodes participate in the *imin* alternative pivot selection process, the correct order of broadcasts will be maintained.

If any node in the subcube for a particular frontal has not yet begun execution of the PDF kernel for that frontal matrix, any received broadcasts will be saved in the proper order within the communication buffer. If the delayed node is the source of the next broadcast for the subsequent frontal matrix's PDF kernel, the other nodes in the subcube will block on the receive for that broadcast and synchronization will be achieved. Likewise, if the pivot ordering for the subsequent frontal matrix is

to change, the *imin* alternative pivot selection mechanism will block the remaining nodes until all the nodes in the subcube execute the *imin* directive. This also causes synchronization to be achieved.

Hence, as the sequence of broadcasts and *imin*s is always well defined, each processing node may proceed asynchronously until it cannot longer proceed due to either an unresolved data dependency due to an unreceived PDF kernel broadcast or an unresolved control dependency due to an alternative pivot selection *imin* operation. Thus, there is no need for explicit barrier synchronization prior to each PDF kernel.□

Theorem 6.5 (PDF Broadcast Messages Not Confused) The broadcast messages for the partial dense factorizations (PDFs) for distinct frontal matrices will not be confused with one another.

PROOF: This theorem follows directly from the proof of Theorem 6.4.□

6.2.3 Distributed Triangular Solves

One of the key features of the parallel refactorization (PRF) code is that by distributing the frontal matrices and the resulting LU factors across the various processors, very large problems may be solved that would not be possible given the memory constraints of a single processor. Furthermore, the LU factorization itself is of little use if subsequent solves can not be done and such solves can also be effectively used to verify the correctness of the factorization. Hence, there is the need for a distributed solve algorithm that can operate on the LU factors in their distributed storage format.

In response to this need, a distributed triangular solve algorithm was developed. It is referred to as distributed since it focuses on doing the triangular solves with the LU factors left in their distributed storage format and provides for only minimal parallelism. In fact, execution times for this solve should increase in a near linear manner with respect to dimension of the hypercube in use.

Development of a more parallel triangular solve algorithm that still takes advantage of the distributed LU factors' storage is a necessary and worthwhile project but is outside of the scope of this effort. However, some comments on such an algorithm are provided at the end of this section.

The Linear Algebra Involved

The overall objective behind most LU factorizations is to solve a system of linear equations in the form

$$A\bar{x} = \bar{b}.$$

This is done by factorizing the matrix A such that $PAQ = LU$ with L and U lower and upper triangular matrices, respectively, and P and Q row and column permutations, respectively [48]. The row and column permutations are typically done to improve numerical stability and retain sparsity, but may also be used to put the matrix into a block upper triangular form [48]. The implications of such a form will be discussed shortly.

The permuted form of the problem becomes

$$LU\bar{z} = P\bar{b} \text{ where } LU = PAQ \text{ and } \bar{z} = Q\bar{x}.$$

Furthermore, the permutations P and Q may be each created via two component permutations such that $P = P_1P_2$ and $Q = Q_2Q_1$ where the permutations P_1 and Q_1 were used to put A into block upper triangular form and to select an initial pivot order and permutations P_2 and Q_2 made during subsequent refactorization to facilitate lost pivot recovery. For the remainder of this subsection, block upper triangular form is assumed to not be in use. This simplifies the discussion. In the next subsection, the additional implications of this special matrix form are discussed.

Thus, the form of the problem now becomes

$$LU\bar{z} = P_1P_2\bar{b} \text{ where } LU = P_1P_2AQ_2Q_1 \text{ and } \bar{z} = Q_2Q_1\bar{x}.$$

Within the context of this implementation the permutations P_1 and Q_1 could be retained on the host (in permutation vector form) and the permutations P_2 and Q_2 that result from the refactorization maintained on the parallel processing nodes. The solution process thus becomes:

1. On the host, compute $\bar{b}_1 \leftarrow P_1\bar{b}$ and send results to the parallel processing nodes.
2. On each parallel processing node, compute $\bar{b}_2 \leftarrow P_2\bar{b}_1$.
3. In a distributed manner, solve for $\bar{y}$ in $L\bar{y} = \bar{b}_2$.
4. Also in a distributed manner, solve for $\bar{z}$ in $U\bar{z} = \bar{y}$.
5. Compute $\bar{w} \leftarrow Q_2\bar{z}$ on the parallel processing nodes and return the results to the host.
6. On the host, compute the final results as $\bar{x} \leftarrow Q_1\bar{w}$.

Specifically, the permutations P_2 and Q_2 could be built from the pivot block row and column permutations within each frontal matrix using vector maximum (by component) parallel prefix operation such as the nCUBE 2 provides with the *nimaxn* routine [111].

Implication of Block Upper Triangular Form

As mentioned earlier, the permutations P and Q may include putting the matrix into a block upper triangular form. Such a form is useful because it:

- restricts the factorization to the diagonal blocks,
- precludes the need to factorize the entries outside of the diagonal blocks,
- limits fill-in to the diagonal blocks,

- and provides a source of parallelism in the factorization process.

An abstract sample of a system of equations with the matrix in block upper triangular form is seen in Figure 6-6. Here the blocks designated B_1 to B_3 are the diagonal blocks that will be factored. Each one of these blocks results in a distinct connected component in the assembly DAG, which provides a valuable source of large grain parallelism. The off diagonal blocks are designated C_1 to C_3 and need not be factored. Only the blocks B_1 through B_3 need to be individually factored into L_1U_1 through L_3U_3 respectively. The subsequent triangular solves would proceed by blocks starting with L_3U_3 where the right hand side vector would be $\bar{b}_3$ and the component of the solution computed would be $\bar{x}_3$. The block C_3 would then be multiplied by $\bar{x}_3$ and the results subtracted from $\bar{b}_2$ to obtain $\bar{b}_2^{(1)} \leftarrow \bar{b}_2 - C_3\bar{x}_3$. Then, $\bar{x}_2$ could be solved for using L_2U_2 and $\bar{b}_2^{(1)}$. The next step would be to update $\bar{b}_1$ as

$$\bar{b}_1^{(1)} \leftarrow \bar{b}_1 - C_1\bar{x}_2 - C_2\bar{x}_3.$$

Finally, $\bar{x}_1$ could be solved for using L_1U_1 and $\bar{b}_1^{(1)}$. This process is referred to as a *block solve* and is the type of solve implemented. Notice that this degenerates into the traditional triangular solves when there is only a single diagonal block for the matrix.

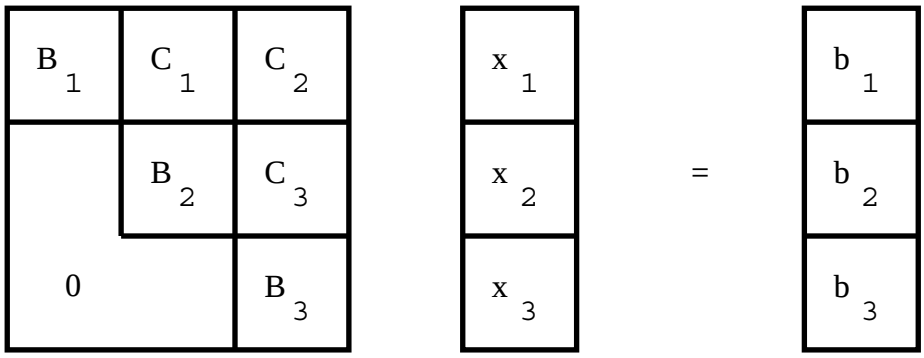

Figure 6-6. Block Upper Triangular Matrix Form Example

With the concept of a block solve thus established, the only details left to be defined are the specific distributed algorithms used to (a) update the right hand side for the next triangular solve within a block, (b) do the forward substitution, and (c) do the back substitution. The next three subsections provide these details.

Blocked Right Hand Side Updates

The updates to the portions of the right hand side (constraint vector) are done using unfactored entries in the off diagonal blocks. These entries were scattered by rows across the processors with one list provided for each row. Within the update for a particular block each processing node in the cube will traverse its lists for rows that fall within the block and multiply the entries by the appropriate entries in the solution vector (which have already been determined). The updates are then accumulated

using a vector maximum (by entry) parallel prefix operation (available as *ndmaxn* in the nCUBE 2 parallelization library [111]) on the positive valued updates and a vector minimum operation (*ndminm*) on the negative valued updates. (Each processor would put a zero into each input vector component that it does not determine). Both sets of accumulated updates are then applied by each processing node to its copy of the right hand side. Care is taken during this process to account for any column permutations that may exist and be represented in the Q_2 permutation matrix.

These block updates to the right hand side vector are one of only two places where parallelism is exploited in this algorithm. Specifically, each processing node can compute the updates for its rows of the block in parallel.

Forward Substitutions

The forward substitution proceeds down the columns of L_i for the ith diagonal block and is based on a standard column-oriented forward substitution [78]. Each processing node checks if it contains the next column of L_i. If it does, it computes the next component of the substitution's solution and then uses a *SAXPY* operation with this component and the rest of this column to update the right hand side. The updated portion of the right hand side is then broadcast to the rest of the processing nodes. The other processing nodes will determine that they do not own this column of L_i and will simply wait for the broadcast.

Back Substitutions

Like the forward substitutions, the back substitutions are done in a column-oriented manner (which stems directly from the column-oriented data allocation scheme in use). These back substitutions, however, proceed in two parts and operate on the portions of each U_i that are held in a particular frontal matrix. The first part of this processing uses a series of *SAXPY* operations to multiply the frontal matrix's columns of U_i that lie outside of the pivot block by the appropriate component of the solution vector (these components have already been determined via the backwards orientation of the process). As these off pivot block columns of U_i may be scattered across processors, there is some exploitable parallelism in this process. The updates determined are combined using a vector sum (by components) parallel prefix operation (provided in the nCUBE 2 Parallelization Library as *ndsumn* [111]) and then applied to update the right hand side associated with triangular system. All of this processing is confined to the subcube that has been assigned to the particular frontal matrix.

The second part of the processing within a frontal matrix's portion of U_i deals with the part of U that falls within the pivot block of the frontal matrix. This is a column-oriented back substitution implemented in much the same way as the forward substitution except that the broadcasts are limited to the subcube assigned to the particular frontal matrix. Upon completion of a frontal matrix, the root (node 0) of the subcube does a broadcast of the results to the other processing nodes.

The process continues in a reversed sequential ordering of the frontal matrices until the entire back substitution has been completed.

Suggestions for Improved Parallelism

The distributed triangular solve algorithm just outlined effectively takes advantage of the distributed storage of the LU factors. However, the many broadcast and parallel prefix operations together with the sequential orientation of the actual forward and back substitutions causes parallel execution time to be unacceptable for real applications. Furthermore, there are significant earlier efforts to implement parallel triangular solve algorithms that have been achieved better performance [56, 99, 85]. Therefore, I conclude the discussion of the distributed triangular solve algorithm with some thoughts on improving parallel execution time.

I believe the most significant performance gains to be achieved by exploiting the natural parallelism revealed by the multifrontal method. Specifically, the data dependencies within a parallel solve operation are exactly those reflected by the "true" L and LU edges for the forward substitution and the "true" U and LU edges for the back substitution. In order to see this consider the data dependency chart for a lower triangular solve seen in Figure 6-7 [59]. At each diagonal entry, a component of the solution is determined and can then be multiplied by the rest of that column with the resulting set of updates subtracted from the right hand side in a column-oriented approach. Each particular update term however, need only be applied just before the solution component corresponding to its row is computed. Hence the update need only be passed to the task that computes that component of the solution. In the case of a dense lower triangular matrix, this involves a great deal of message traffic.

However, in the case of a sparse lower triangular matrix, the message traffic requirements can be significantly less. Furthermore, zeroes in the strictly lower triangular portion will create independence between operations in much the same way as is seen in the multifrontal factorization method. Specifically, if the l_{ij} (where $i > j$) entry of the lower triangular matrix L is zero, then the ith component of the solution can be computed independently of the jth component (provided no other set of dependencies produces a dependency from i to j due to transitivity).

If l_{ij} where nonzero, then a data dependency would exist. But this is exactly the criteria for an L relation between i and j in the assembly DAG. Hence, the "true" L (and LU) edges of the assembly DAG also represent the data dependencies of the lower triangular solve (forward substitution).

As an example, consider Figure 6-8. In this figure, the data dependency chart for a sparse lower triangular matrix is illustrated. If tasks are defined by columns of the matrix and identified by the column index, we find that the task graph for lower triangular solve becomes exactly that shown in Figure 6-9. Furthermore, the edges of this task graph are exactly the "true" L (and LU) edges of the assembly DAG for the multifrontal LU factorization. Going a step further, a blocked allocation of matrix columns could allow columns like 3 and 4 in Figure 6-8 to be on the same processing node. This would allow tasks 3 and 4 to be combined and message passing requirements further reduced.

A similar relationship exists for the upper triangular solve (back substitution) except in this case the U (and LU) edges define the dependencies and do so in a

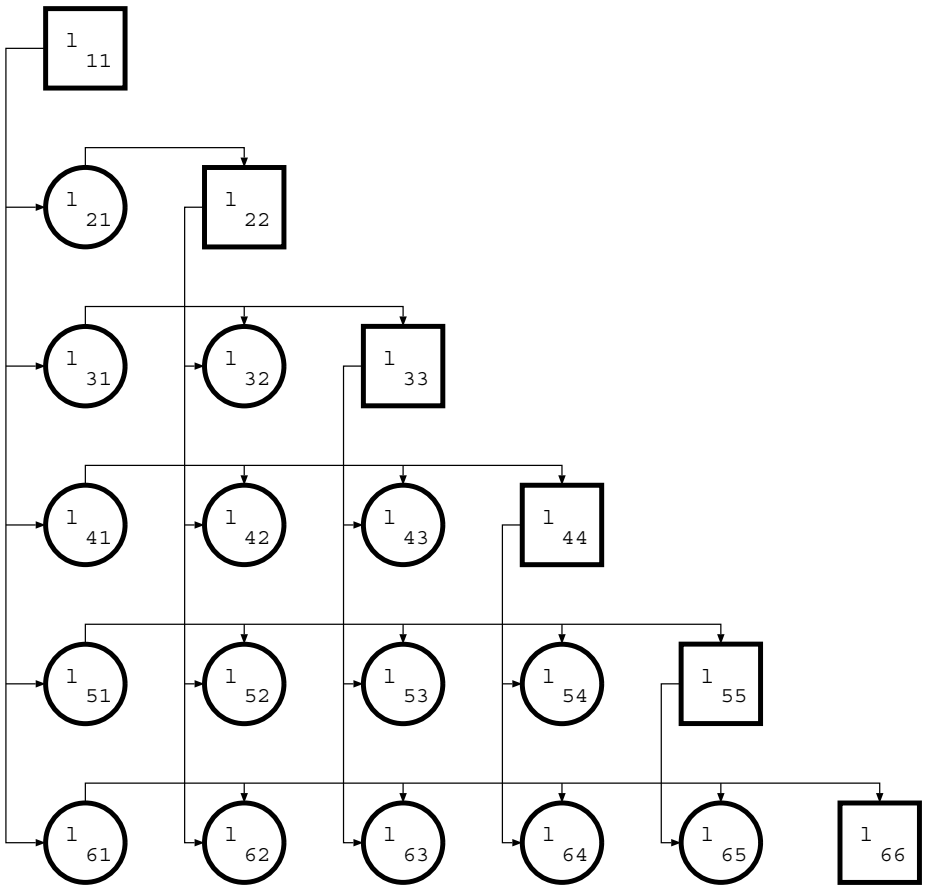

Figure 6-7. Dependency Chart For A Lower Triangular Solve

reversed fashion. That is, a U relation from pivot i to pivot j imposed by a nonzero a_{ij} entry ($i < j$) creates a data dependency from j to i in the back substitution.

The conclusion is that the "true" data dependencies of the multifrontal assembly DAG also define the precedence constraints imposed on the triangular solves. Scheduling of these solve tasks and the corresponding message passing requirements can be statically determined and distributed to the parallel processing node such that the triangular solves' parallel performance can be improved.

There are, however, several additional complexities that must be resolved in the implementation of this multifrontal approach for triangular solves. First among these is that the assembly DAG produced by UMFPACK and used by the parallel refactorization (PRF) code does not explicitly contain all of the "true" dependencies. This is a result of the aggressive edge reduction done by UMFPACK [34, 35, 32]. Whether or not this reduced edge set is sufficient to reflect the triangular solve dependencies and the implications of using this edge set both require further investigation.

Furthermore, neither the columns nor the rows of the upper triangular factor U are necessarily stored on a single parallel processing node by the PRF code. The implications of this fact could include both additional message passing requirements

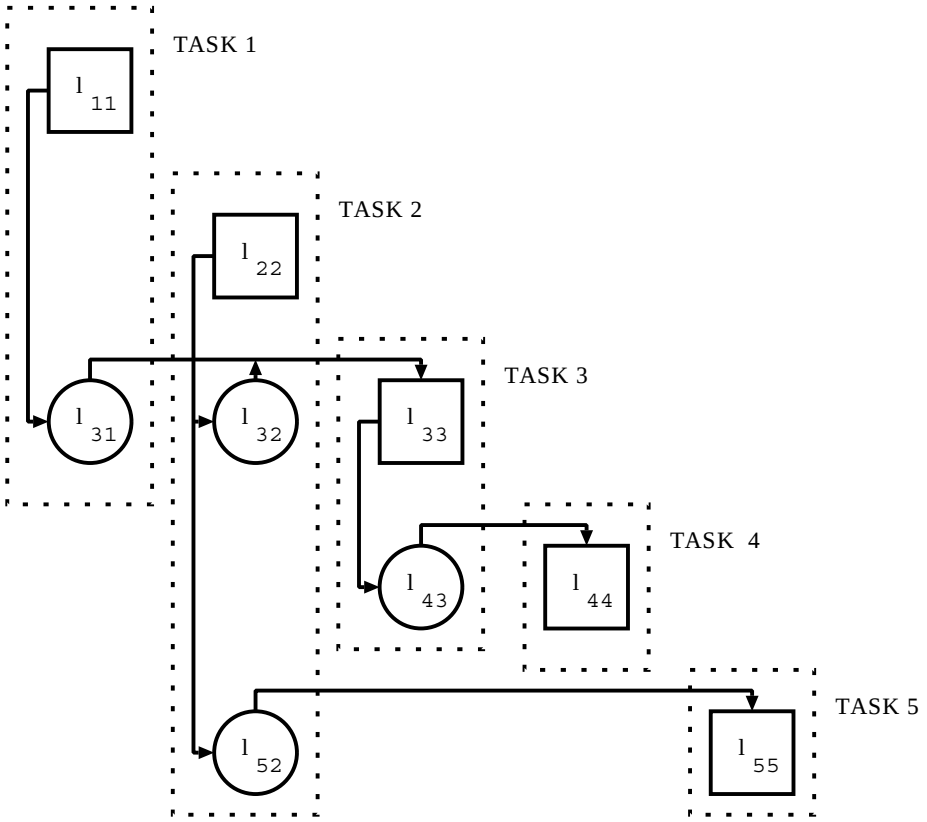

Figure 6-8. Dependency Chart For A Sparse Lower Triangular Solve

but also enhanced parallelism. Development of an upper triangular solver would require careful analysis of these implications.

Additional complications would be introduced with the inclusion of lost pivot recovery during the refactorization process. These recovery mechanisms could alter the pivot ordering and even the patterns and numbers of frontal matrices dynamically during the parallel refactorization. Corresponding changes to the already distributed schedules and to message passing requirements for the triangular solves would need to be made.

These additional complications together with the low ratio of computation to communication for a distributed triangular solve diminish the prospects that significant speed-ups would be achievable. However, even if no speed-up or even minor increases in execution time are the best performance achievable, such a parallel triangular solve algorithm would be very beneficial when combined with the parallel refactorization code to form a complete parallel software package that would both factorize and solve sequences of identically structured systems of linear equations.

6.3 Performance Issues

The focus of this chapter has been on the initial implementation of a distributed memory, parallel multifrontal LU refactorization capability. A significant portion of this initial implementation involved the development of a number of scheduling,

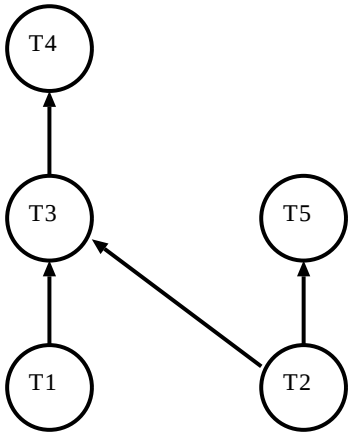

Figure 6-9. Task Graph For A Sparse Lower Triangular Solve

allocation, and assignment options. Before moving on to the investigation of lost pivot recovery mechanisms, a detailed performance evaluation of these options would allow efforts to be directed to only the most promising configurations. Furthermore, a performance evaluation could also be used in validating the parallel performance predictions of the simulation models developed earlier.

First, the set of test cases used for the evaluation are defined and their selection justified. The next step is to evaluate the alternative scheduling and subcube allocation methods, S1/S2 and S3. The corresponding performance impacts on near edge only consideration, communication reducing assignments, and the default assignment strategies are also investigated through their impacts on the two scheduling/allocation methods. As we shall see, S3 is clearly superior and thus the subsequent detailed evaluation of overlapping assignments, clustering per children, and per parents is limited to the S3 method. As these evaluations involve the testing of many program and processor set configurations, a small set of matrices are used.

The next evaluations are aimed at assessing the parallelism, competitiveness, and scalability of the baseline parallel refactorization code. Parallelism is evaluated by comparing the execution time of the parallel refactorization code running on a single processor to its execution on larger dimension hypercubes. Competitiveness is addressed by running the Harwell MA28B general sparse matrix refactorization code [50] on a single nCUBE processor and comparing its execution time to that of the parallel refactorization code on both a single processing node and a 6 dimensional hypercube. Dynamic memory utilization is traced for the runs on various hypercube sizes. The corresponding reductions in memory utilization seen as larger hypercubes are employed illustrates the scalability of the method. Finally, an execution profiling is done to investigate the parallelism achieved in each of the subfunctions.

All of the performance evaluation runs are executed using the double precision (64 bit) representation of real data items. The need for this higher level of precision was made evident during the testing of the PRF code. The size and condition of several of the matrices used for both testing and performance evaluation resulted in the loss of up to seven significant decimal digits. The single precision representation of real

numbers in the nCUBE 2 provides only a 23 bit mantissa (about 7 significant digits [109]) and thus its use would leave no significant digits in these worst cases. The double precision representation provides a 52 bit mantissa with about 15 significant digits [109] which provided much more satisfactory results. Furthermore, the relative errors observed in the PRF results were consistent with those observed when using either UMFPACK or MA28 on the same matrices.

6.3.1 Performance Measures

The primary performance measure was execution time. This was typically measured using the *toggle* and *event* features of the nCUBE 2 *ETOOL* event-driven profiler [110]. However, when comparisons are made with MA28B, the nCUBE *n_time* time routine was used because the *ETOOL* routines were not available within FORTRAN for timing MA28B. Both timing measures were accurate to 1 microsecond, although the *n_time* timings also included some system processing and were typically a few percent larger than comparable *ETOOL* timings. As a result, care was taken to only compare times taken from the same source.

In the case of both timing mechanisms, the timers were started after all the processing nodes in the hypercube had received their schedules, frontal matrix descriptions, and original matrix values. The original matrix values were held in an unassembled format and their assembly into the actual frontal matrix storage area was considered within the timed portion of the code. A barrier synchronization was performed prior to starting the timers. This synchronization was done using the nCUBE 2 *sync* routine that also does a synchronization of the clocks in each of the processing nodes.

A second global barrier synchronization was done at the end of each parallel factorization to insure that the entire factorization had completed. The timers are stopped only after this synchronization has completed. This synchronization is done using the nCUBE 2 *imin* operation, which incurs overhead costs ranging from 295 microseconds for a 1-dimensional hypercube (2 processors) to 1285 microseconds for a 6-dimensional hypercube (64 processors). These times were relatively insignificant compared to the overall factorization times and thus no attempt was made to reduce overall execution times to account for them.

Thus, the scope of the timed parallel factorizations starts with all the initial data distributed to the processing nodes in an unassembled format and concludes only after all processing nodes having completed their portions of the factorization and retained their appropriate portions of the LU factors in their local memories. This is a significant assumption in that some applications could require very substantial redistribution of the data. However, other applications may require little or no data redistribution. Thus, the data distribution issue is kept separate from the factorization's performance.

Speed-ups were calculated by taking the single processor execution times and dividing them by the appropriate parallel execution times.

Another performance measure was the amount of required communication. This was measured as the percentage of total communication required in a worst case scenario. The communication reductions were measured by first calculating the total

required communication based on a distinct message for each column of each contribution sent across an edge in the assembly DAG as this would be a worst case scenario. Communication costs for a message are given in terms of a message setup cost (α) and a transmission cost (β) times the length of the message (l). Based on the results of the performance evaluation of the nCUBE 2 communication facilities reported in an earlier chapter, relative values of 200 and 6 were assigned to α and β respectively based on the assumption of an eight byte data objects with a 30 percent additional data overhead for the indexing information.

After calculating the total communication cost that could be required, the actual communication costs were calculated based on the number and sizes of messages that were actually needed. In doing so, a column that was assigned to the same processor in both the source and destination frontal matrices requires no message passing and incurs no communication cost. When multiple columns of a contribution edge are all assigned to one processor in the source frontal matrix and all assigned to a single but different processor in the destination frontal matrix (that is, they were effectively "clustered"), they may be passed together as a single message thus eliminating the setup costs (i.e., α) for all but one of the columns. From these calculations, the amount of communication reduction can be computed as a percentage of the total possible communication costs.

The final performance measure is the amount of dynamically allocated memory utilized. This was done using alternative heap allocation/deallocation routines (malloc, calloc, and free). These routines trace the amount of heap memory currently allocated and the maximum amount ever allocated throughout the entire run. They also report the amount of heap memory allocated prior to the start of the parallel allocation. Test runs with these routines selected and not selected showed them to have minimal impact on performance.

6.3.2 Test Case Selection

As the number of scheduling, allocation, and assignment options combined with the various hypercube sizes to be tested produce a large number of configurations to be evaluated, a small but representative set of test matrices would be appropriate to enable the evaluations to be completed in a timely and efficient manner. To this end, three test matrices were chosen. RDIST1 and EXTR1 come from chemical engineering applications [144, 143] and GEMAT11 from a power system application [50]. RDIST2 and RDIST3A are also from chemical engineering applications [144, 143] and will only be used as additional test matrices for the parallelism. All of these matrices are the lead matrices in sequences of identically structured, unsymmetric pattern sparse matrices. Some of the characteristics of these matrices are provided in Table 6-1. The asymmetry statistic is defined as the number of unmatched off diagonal terms divided by the total number of off diagonal terms. Hence, a perfectly symmetric pattern matrix has an asymmetry of 0 and a perfectly asymmetric pattern matrix has an asymmetry of 1.

Table 6-1. Test Matrix Characteristics

MATRIX	ORDER	NONZEROS	ASYMMETRY
RDIST1	4134	94408	0.941
EXTR1	2837	11407	0.996
GEMAT11	4929	33108	0.999
RDIST2	3198	56834	0.954
RDIST3A	2398	61896	0.860

6.3.3 Scheduling/Allocation Issues

Earlier in this chapter, the two primary scheduling and subcube allocation methods were defined and discussed. The S1/S2 method did scheduling and allocations by blocks of frontal matrix tasks. At the beginning of each block, all processors were assumed to be available (an implied barrier synchronization was assumed). This additional processor availability would allow greater subcube assignment flexibility to aid in reducing communication. The scheduling of this method proceeds in two parts, S1 attempts to exploit inter-task parallelism by allocating smaller subcubes to frontal matrix tasks and allowing more tasks to be scheduled within a scheduling block. The purpose of S2 was to preclude under allocations by allocating tasks larger subcubes. Within each scheduling block, the S1 and S2 schedules are compared and the most efficient of the two is chosen. Importantly, the barrier synchronization between blocks assumed with this method is not enforced at run time.

While the S1/S2 method provides significant flexibility in subcube allocation and assignment, its blocked scheduling orientation can result in excessive idle times as execution times of tasks within the block can vary significantly. In response to this concern, the S3 method was developed to make maximal use of available processors/subcubes. Subcubes are scheduled in strict critical path priority with subcube allocations managed by an augmented binary buddy management system [95].

As the performance of both the S1/S2 and S3 methods will be directly affected by the set of edges considered in the task graph (all or near edges only), the use of communication reducing column assignment features (overlapping, clustering per children, and clustering per parents), and the default assignment method, a number of program configurations for both S1/S2 and S3 needed to be considered. Specifically, eight configurations of S1/S2 and eight of S3 were tested. These configurations are outlined in Table 6-2 where the assignment features include overlapped assignments, clustering per children, and clustering per parents.

Each configuration was run on hypercubes of dimensions zero to six (1 to 64 processors) using each of the three test matrices, which required a total of 336 runs. Timing of the runs was done using the *toggle* and *event* facilities of the nCUBE *ETOOL* event-driven profiler [110].

Analysis of the various runs revealed a number of significant trends. First and foremost, the S3 scheduling/allocation method significantly out-performed the S1/S2

Table 6-2. Test Configurations

EDGE CONSIDERATION	ASSIGNMENT FEATURES	DEFAULT ASSIGNMENTS
All	All	Blocked
All	All	Scattered
All	None	Blocked
All	None	Scattered
Near only	All	Blocked
Near only	All	Scattered
Near only	None	Blocked
Near only	None	Scattered

method especially with the larger hypercubes. Without exception, the S3 execution times were less than the S1/S2 execution times for otherwise like configurations. Furthermore, the magnitudes of these differences were especially significant for the larger hypercubes with the S3 execution times being from 15 to 32 percent less than those for the S1/S2 configurations running on 6-dimensional hypercubes (64 processors). For example, the GEMAT11 matrix run on a 6-dimensional hypercube with all edges considered, no assignment features and blocked default assignments produced an execution time of 291.0 milliseconds using S1/S2 and only 197.3 milliseconds using S3. The RDIST1 matrix on the same configuration produced execution times of 2273.3 milliseconds using S1/S2 and 1862.0 milliseconds using S3. The obvious conclusion from this is that further efforts should concentrate on only the S3 method.

Beyond the differences in execution times between S1/S2 and S3, the goal of reducing communication with an S1/S2 produced schedule was not consistently achieved. In fact, S1/S2 typically had 1 to 3% less communication than S3. Further investigation revealed that S1/S2 did consistently produce better overlap but S3 more than made up for the difference with better clustering (the contributions of several columns on the same processor could be be combined into a single message as they all had the same destination). The nCUBE 2 has a relatively low message set up cost (α). Higher set up costs (relative to transmission costs) make clustering even more attractive. The S3 method seems to be superior given the current performance characteristics of distributed memory multiprocessors.

Other interesting communication reduction results include the observation that the communication reducing assignment features (overlapped assignments, clustering per children, and clustering per parents) produced communication reductions that were as much 11.5 percent better than like configurations without these features. Within this observation, there was a strong correlation between larger hypercubes and larger differences. A degradation of up to 10 percent was also seen when a scattered default assignment mechanism was used with larger hypercubes. The combined effect of disabling the communication reducing assignment features was to have up to a 10

percent degradation when using a blocked default assignment method and up to a 22.5 percent degradation when using a scattered default assignment. This advantage of blocking over scattering is likely the result of the manner in which frontal matrices are defined in the UMFPACK software [34, 35]. A specific illustration of these results in seen Table 6-3, which is an excerpt of the results for the RDIST1 matrix. This data comes from the S3 method with all edges considered and indicate the percentage of worst case required communications that have been eliminated.

Table 6-3. Percentage of Communication Reduction

CONFIGURATION	1-CUBE	4-CUBE	6-CUBE
All features, blocked	85.71	64.37	57.47
All features, scattered	85.71	64.37	57.47
No features, blocked	85.71	55.64	46.15
No features, scattered	85.71	51.65	36.00

Also, evident from these results is that little, if any, difference results when a blocked versus scattered default is used with all of the communication reducing features enabled. There is also a general trend where the typically higher reductions were achieved for smaller hypercubes. Indeed on a single processing node all message passing communication of contributions are eliminated.

Despite the genuine ability for the communication reducing assignment features to achieve their goal, their use did not typically produce the best execution times. For hypercubes of dimensions of one or two, the S3 method with no assignment features, all edges considered, and a scattered column allocation typically produced the best times. For hypercubes of dimensions five or six, the S3 method with no assignment features, all edges considered, and a blocked default allocation was the best configuration. The results for hypercubes of dimension three and four were dependent upon the particular test problem with blocked assignments better for GEMAT11 and EXTR1 and scattered assignments better for RDIST1. When the communication reducing assignment features were enabled the execution times went up but typically by no more than 5 percent. Furthermore, with these features enabled, the default assignment method (blocked or scattered) made little or no difference. This is further justified by the observation that the communication reducing features generally made 80 percent or more of the assignments when enabled leaving few assignments to be done by default.

Hence I am drawn to the conclusion that while the communication reducing assignment features are effective in reducing communication and did produce the best execution times for several of the smaller hypercube configurations, the performance benefits within the partial dense factorizing kernels of a strictly scattered assignments (for smaller hypercubes) and strictly blocked assignments (for larger hypercubes) have a greater impact on overall performance. However, matrices that have

larger total edge weights and/or implementation platforms with higher communication costs could make the use of these communication reducing assignment features produce significant benefits.

Finally, use of the near edge only feature for scheduling and allocation produced mixed results, which were dependent upon both the matrix and the size of the hypercube. For example, with the EXTR1 matrix and hypercubes of dimensions 2, 4, 5, and 6, the best execution times were achieved by considering only near edges. However, with the RDIST1 matrix, near edge only consideration produced the best times only on hypercubes of dimensions 2 and 5 and, with the GEMAT11 matrix, only hypercubes of dimensions 2 and 6 produced the best times when using near edges only. Yet, with all combinations the execution time differences between all or near edge only consideration of otherwise like configurations were less than 5 percent. Thus, considering near edges only did improve execution times in a significant number of configurations. While its benefits were never significant (a 3.2 percent improvement was the best observed), its costs were never great either. Hence, it is reasonable to be kept as an user-selectable option to try when optimizing performance for a particular problem. Furthermore, in more communication bound problem instances, its benefits could be more significant.

The conclusions that I draw from the analysis of this set of runs are that:

- S3 is the far superior scheduling/allocation method and S1/S2 need not be further considered.
- Typically, the best program configurations use all edge consideration, no communication assignment features, and scattered assignments for smaller hypercubes and blocked assignments for larger hypercubes.
- The communication reducing assignment features are effective in their goal of reducing communication but the assignments they produce are not optimal for the partial dense factorization kernels. However, they should however be kept around for problem cases and/or host platforms that incur higher communication costs.
- Near edge only consideration can benefit specific problem/hypercube size combinations. Therefore, this feature should be maintained as an optional feature.

6.3.4 Assignment Issues

In order to further investigate the impact of the communication reducing assignment features, additional configurations were run using the S3 scheduling/allocation method: all edge consideration, near edge only consideration, blocked, and scattered default assignments. The communication reducing assignment feature combinations included (a) just overlapped assignments, (b) overlapped assignments with clustering per children and (c) overlapped assignments with clustering per parents. These various combinations were run against all three test matrices and on hypercubes of dimensions 0 through 6. In total this required an additional 252 runs.

Analysis of the results of these runs show that the inclusion of each feature reduces communication but adversely affects the execution time. However, all of the configurations that use one or more of these assignment features are better than the default configuration that was inappropriate for the particular hypercube size (i.e., blocked for small hypercubes and scattered for large hypercubes). Furthermore, adding clustering per children to overlapped assignments is more beneficial than adding just clustering per parents.

The conclusion is that the communication reducing features eliminate the most communication. However, the best execution times are typically produced by strictly using a scattered or blocked assignment of columns depending on the test problem and the size of the hypercube in use.

6.3.5 Parallelism within the Method

In order to assess the parallelism exploitable by this initial implementation of the distributed memory, multifrontal LU refactorization algorithm, the execution times for the various sizes of hypercubes were compared to the execution times achieved on a single processing node (0-dimensional hypercube). The program configuration used included the S3 scheduling/allocation method, all edge consideration, no communication reducing assignment features, and blocked column assignments. While not optimal for the smaller sized hypercubes, this configuration was the best for the larger hypercubes (where the greatest speed-ups are attained) and was appropriate for the single node case as all configurations have like performance on a single processor.

In order to provide a better picture of the achievable parallelism, the RDIST2 and RDIST3A matrices were also included in this set of runs. The achieved speed-ups are presented in Figure 6-10.

These results are quite encouraging as each of these test cases is of a moderate size, yet still achieves reasonable speed-ups on the largest hypercubes tested. Furthermore, the RDIST1 and GEMAT11 results are consistent with the earlier distributed memory simulation results, which predicted a speed-up of 23.24 for RDIST1 on the 6-cube and 15.17 for GEMAT11 on the 6-cube. In addition these simulations did not take into account the use of the Basic Linear Algebra Subroutines (BLAS) that cut computation times in half and hence reduce the ratio of computation time to communication time which limits speed-up. Since the parallel factorization code uses the BLAS, these speed-ups are especially encouraging.

6.3.6 Competitiveness of the Method

While the parallelism results for the distributed memory, multifrontal LU refactorization code are encouraging, the competitiveness of the code must be assessed based on comparison to other existing and widely accepted routines that perform similar functions. Ideally, this comparison would be made against a single best method that is optimal for all problem cases. In the case of sparse matrix factorization, however, this is not possible since the effectiveness of a particular method is strongly tied to the particular pattern of nonzeros and other characteristics of the matrix such as symmetry and the number of independent blocks after permutation to block upper

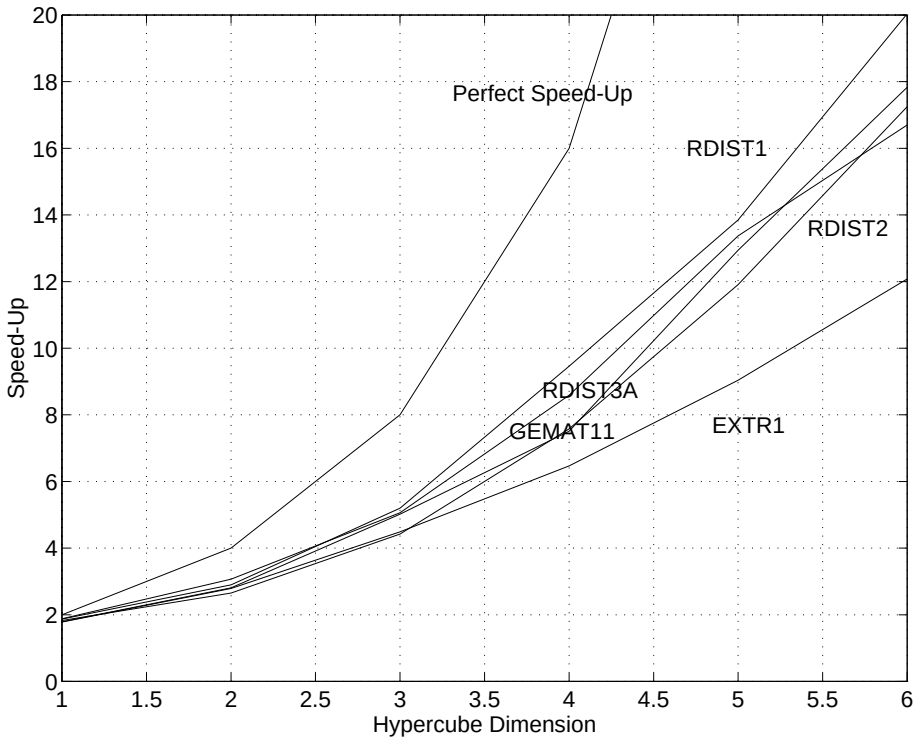

Figure 6-10. Parallel Refactorization Speed-Ups

triangular form. Thus, no one method is best for all cases. None the less, Harwell's current MA28 package, including the MA28B refactorization routine, is a very commonly accepted standard for general, unsymmetric pattern sparse matrices [53]. This package is currently the most accepted standard routine for this function.

In order to compare the parallel refactorization code to MA28B, an MA28A/C (initial factor/solve) driver was modified to also call and time the MA28B routine. As both the driver and the MA28 package are written in FORTRAN and the *ETOOL* FORTRAN routines were not available on the nCUBE 2 at the University of Florida, the alternative timing routine *n_time*, which is callable from both FORTRAN and C, was used. The timings of both MA28B and the parallel refactorization (PRF) code used in this section were both obtained using the *n_time* routine.

Furthermore, due to the memory limitations of the nCUBE 2 processing nodes, MA28 could not factor the RDIST matrices. Thus competitiveness was only measured using the GEMAT11 and EXTR1 matrices. The timings (in milliseconds) for both MA28B and the parallel refactorization (PRF) code on a single node and a 6-cube are shown in Table 6-4 together with the achieved speed-up calculated as the PRF 6-cube time divided by the MA28B single node time.

As can be seen from these results, the comparison of the PRF code to MA28B is problem dependent. However, the EXTR1 matrix still gets a reasonable speed-up over MA28B and the GEMAT11 matrix is factorized almost 19 percent faster with the PRF code on a single processor.

Table 6-4. Competitiveness of Parallel Refactorization Code

MATRIX	PRF (0-CUBE)	MA28B	PRF (6-CUBE)	SPEED-UP
EXTR1	2607	1609	237	6.79
GEMAT11	2521	3103	183	16.96

6.3.7 Execution Time Profiling

In order to understand the significance of each major function of the parallel refactorization code in terms of its impact on execution time and the parallelism of the method, an execution time profile of the PRF code was done using the nCUBE 2 *ETOOL* event driven profiler *toggle* facility. Instrumenting the PRF code with calls to the *eprof_toggle* routine and dumping the results to a file for post execution analysis by *ETOOL* allowed the number of calls to each function, as well as the total, average, and standard deviation of each function's execution time to be obtained for each contributing processor. Profiling runs were done for the RDIST1, EXTR1, and GEMAT11 matrices on hypercubes of dimensions 0 to 6 using a program configuration of the S3 scheduling/allocation method, all edge consideration, no communication reducing assignment features, and a blocked assignment of columns to processors.

The total execution times (in seconds) for each major function for the 0-cube and 6-cube runs are shown in Table 6-5. The times for the 6-cube runs are the maximums for each individual function across all participating nodes. Also provided are the speed-ups resulting from dividing the functions' 6-cube times by the functions' 0-cube times. Within the "assemble contributions" function there are two times. The first time is the entire time spent in that function and the second time is the amount of idle time spent in the function while waiting for required contributions from other processors that have not arrived. This second time was obtained by putting calls to *eprof_toggle* on either side of the *nreadp* function that reads contribution messages. The overhead of the *nreadp* function itself is assumed negligible as the function simply checks for a received message with the required message type and source node ID and returns a pointer to that message. The message itself is left in the communication buffer. In addition the inclusion of the instrumentation in the code had minimal impacts on overall execution time. The RDIST1 times were extended by less than half a percent, the GEMAT11 times by less than 3 percent, and the EXTR1 had the worst adverse affects of 5 percent on the 0-cube and 9.1 percent on the 6-cube.

Analysis of these profiling results reveals that the partial dense factorizing function dominates the execution time and that only the assembly of contributions contributes any significant additional effect on overall execution time. Furthermore, all of the other functions have better parallelism than the partial dense factorizing kernels. Finally, one can also see that both the relative times in each function and the parallelism of each function are highly dependent upon the specific test problem and on how well it can be distributed across the processors.

Table 6-5. Sequential and Parallel Execution Profiles

MATRIX	FUNCTION	0-CUBE TIME (sec)	6-CUBE TIME (sec)	SPEED-UP
RDIST1				
	ALLOCATE	0.416	0.018	23.10
	ORIG VALUES	0.761	0.028	27.20
	CONTRIBUTIONS	4.976	0.485	10.26
	IDLE (NREADP)	4.976	0.116	42.90
	PARTIAL FACTOR	30.661	1.798	17.05
	FORWARD CONTRIB	0.414	0.066	6.27
	RETAIN LU	0.459	0.013	35.31
GEMAT11				
	ALLOCATE	0.237	0.005	47.40
	ORIG VALUES	0.286	0.006	47.67
	CONTRIBUTIONS	0.723	0.027	26.78
	IDLE (NREADP)	0.723	0.016	45.19
	PARTIAL FACTOR	1.684	0.165	10.21
	FORWARD CONTRIB	0.272	0.015	18.13
	RETAIN LU	0.229	0.005	45.80
EXTR1				
	ALLOCATE	0.317	0.023	13.78
	ORIG VALUES	0.125	0.004	31.25
	CONTRIBUTIONS	0.993	0.115	8.63
	IDLE (NREADP)	0.993	0.024	39.72
	PARTIAL FACTOR	1.031	0.184	5.60
	FORWARD CONTRIB	0.554	0.040	13.85
	RETAIN LU	0.222	0.013	17.08

6.3.8 Memory Utilization and Scalability

A typical objective of parallel algorithms for distributed memory environments is the effective utilization of the additional memory capacity and bandwidth that such an environment can provide. An algorithm that allows its data to be evenly distributed with little additional overhead required is said to be scalable in a data sense. That is, larger problem sizes can easily be handled by using near proportionally larger hypercubes.

Reported in Figure 6-11 are the maximum amounts of dynamic memory used on any one processing node for the factorizations of our three test matrices. Also shown are the theoretical memory requirements if perfect scalability were achieved. Hypercubes of dimensions 0 through 6 were tested. All the reported numbers are in terms of megabytes of heap allocated and include the memory overhead of 4 bytes per

allocated data object. These allocations include the prefactorizing storage of permutations matrices, schedules, frontal matrix descriptions, and lists of original matrix values together with the dynamic allocations and deallocations of frontal matrices during the factorization process and the storage required for the LU factors themselves. Not included in these overall memory utilization numbers are the memory allocations for contribution messages that are taken from a separately managed communication buffer area. Typically, the memory requirements for the communication buffer were only 10 to 20 percent of those for the program heap.

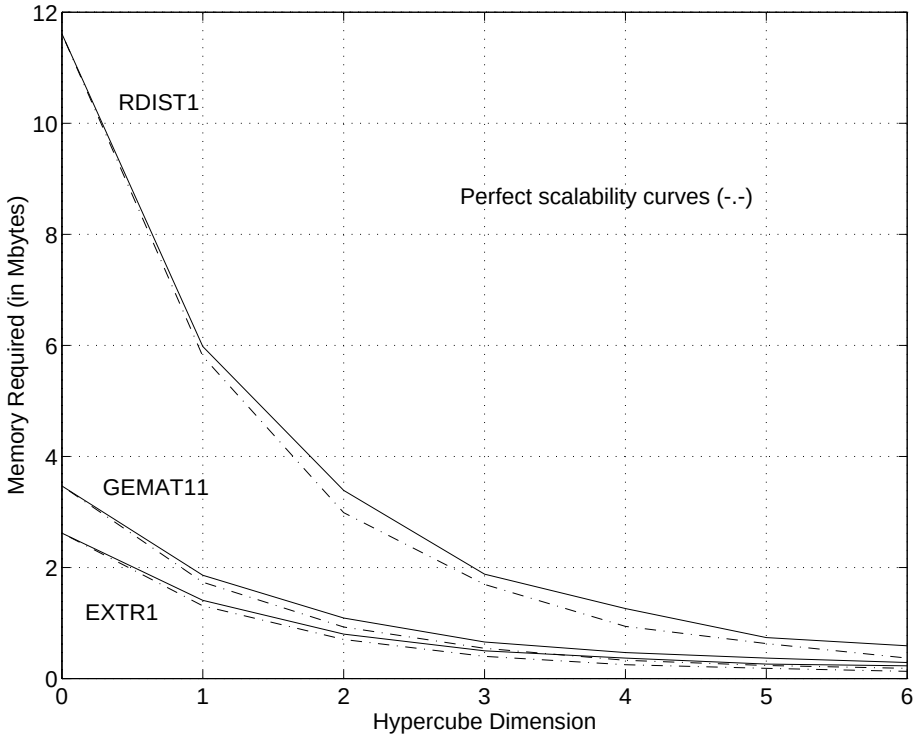

Figure 6-11. Memory Requirements and Scalability

Looking at these memory utilization numbers we see that the PRF refactorization method has very reasonable data scalability. While the memory requirements per processor did not reduce in strict proportion to the number of available processors, the reductions were significant indeed.

6.4 Summary and Conclusions

The focus of this chapter has been on the description and evaluation of the initial implementation of a distributed memory, multifrontal LU refactorization capability. This initial implementation was restricted to maintaining the pivot order established by the prerequisite analysis of the UMFPACK software [32, 34, 35].

A major emphasis of this initial implementation was to investigate the use of various scheduling, allocation, and assignment mechanisms. Also mechanisms were implemented as preprocessing to be performed on a sequential host processor. The

static schedules, allocations, and assignments produced would then be used during the subsequent parallel refactorization.

Two scheduling/allocation mechanisms were developed. The S1/S2 method emphasized reducing communication via assignment flexibility provided by implied barrier synchronizations for each block of tasks scheduled. The S3 method was oriented to reduce execution times by scheduling frontal matrix tasks has soon as subcubes of the appropriate size became available. Communication reduction was also addressed in S3 by extending the capabilities of the binary buddy subcube allocation system to choose the subcube that eliminates the most communication when multiple candidate subcubes are available.

The size of a subcube allocated to a particular frontal matrix was established using a proportional allocation scheme that considers such criteria as the task's priority and workload, the number and cumulative workloads of currently available tasks, and the number of processing nodes.

Specific assignments of subcubes to frontal matrix tasks was done to maximize the overlap of contribution columns onto processors, which eliminates the corresponding communication costs. Within subcubes, the columns of the frontal matrices are assigned to processing nodes in a variety of manners. Communication reducing assignment mechanisms include overlapping, clustering per children, and clustering per parents. Default assignments can be done in either a column scattered or blocked fashion.

Once the static schedules, allocations, and assignments have been established, the required information is passed to the parallel processing nodes via a launching process. The parallel refactorization then begins with the frontal matrix descriptions acting as task control blocks. Each frontal matrix is factored in the order specified by the schedules for each processing node. Data dependencies for contributions are resolved via either message passing between processors or scheduled data movements within processors. Each processing node dynamically acquires the frontal matrix storage it requires and retains its portion of the LU factors. Frontal matrix descriptions and processing node schedules are retained by the processing nodes and used for subsequent refactorizations that use different sets of matrix entry values.

A distributed triangular solve capability was also developed to complete the package and facilitate verification of the factorization results. This triangular solve capability was designed to make use of the distributed storage of the LU factors but did not emphasize parallelism or execution time. Several suggestions for further refinements to this capability were provided.

With the initial implementation completed, a performance evaluation was conducted to investigate the effectiveness of the various scheduling, allocation, and assignment options provided and to empirically assess the parallelism and competitiveness of a distributed memory, multifrontal LU factorization capability.

Based on this performance evaluation, the S3 scheduling/allocation method is clearly superior. Furthermore, while the communication reducing assignment features were successful in reducing required message passing overheads, they did not produce the best execution times. This was due to the fact that the partial dense

factorization (PDF) kernels' performance was strongly influenced by the data assignments. Strictly scattered column assignments for smaller hypercubes and strictly blocked assignments for larger hypercubes produced performance advantages that outweighed the advantages of the reduced communication gained by the more irregularly structured assignments.

The performance evaluation also revealed that the parallelism of the multifrontal approach anticipated by the simulation models is, in fact, achievable. Furthermore, the comparison of the initial implementation of the parallel refactorization (PRF) code to the refactorization code of the widely accepted MA28 package confirmed the competitiveness of the multifrontal approach.

While the results seen in this chapter are very promising, additional capabilities are required. Specifically, the issue of lost pivot recovery is yet to be addressed. During the factorization of a sequence of identically structured sparse matrices, there is the possibility for values of the anticipated pivots to become no longer numerically acceptable. While one recourse would be to discontinue the current factorization and initiate a new analysis and pivot order determination, less costly alternatives are possible. The next chapter explores such alternatives.

CHAPTER 7
LOST PIVOT RECOVERY

When factoring a sequence of matrices with identical patterns, the pivot ordering determined by an analyze-factorize operation on the first matrix can frequently be reused so that only factorization is required for the subsequent matrices in the sequence. However, as numerical values for the nonzero entries change across matrices in the sequence, the anticipated pivots may no longer be numerically acceptable. One recourse would be to perform the analyze-factorize operation on the current matrix to find a new acceptable pivot ordering. This however is costly and if only a few of the anticipated pivots have failed, it may be possible to make only minor alterations to the pivot ordering and complete the factorization without serious adverse performance effects.

This chapter develops and implements a lost pivot recovery capability for the unsymmetric-pattern multifrontal method. The capability is robust in that it will complete the factorization regardless of the number of failed pivots (subject to memory constraints and the existence of valid pivots (nonsingularity)). Concepts of avoidance and local recoveries (within frontal matrices) are employed to minimize the performance impacts. Significantly, the capability requires only minimal, if any, modifications to the assembly DAG and any necessary modifications can be determined a priori so the lost pivot recovery capability can be integrated into a statically scheduled, distributed memory implementation.

The discussion of this chapter starts with a development of the theory required for lost pivot recovery. Included in this theoretical development are the definition of a lost pivot avoidance strategy and a minimal impact local recovery that has no impact on the assembly DAG or the size of the LU factors. However, avoidance and local recoveries are insufficient to handle all possible failures and the bulk of the discussion focuses on recovering lost pivots across frontal matrices. With the necessary theory established, the second section details how the lost pivot recovery capability was actually implemented by extending the parallel refactorization capability developed in Chapter 6. This implementation is especially significant as it is within a distributed memory environment. The final section presents a performance evaluation of the lost pivot recovery capability. Both sequential and parallel performance are investigated as is the scalability of the method as determined by memory requirements.

7.1 Theoretical Development

The problem of anticipated pivots that become numerically unacceptable can be addressed at a number of levels within a multifrontal context. Perhaps the easiest approach is to avoid lost pivots by relaxing the pivot threshold parameter. For example, the analyze-factorize method used on the first matrix in the sequence could

use a pivot threshold factor of $\mu = 0.1$ and this parameter could be relaxed to $\mu = 0.001$ for later matrices in the sequence. Thus, more variation will be allowed in the matrix values before anticipated pivots are deemed to be numerically unacceptable. *Lost pivot avoidance* is a useful strategy but will not completely solve the problem. Changing the pivot threshold will also allow for greater growth in the error bounds as the bound on any particular entry's growth (ρ) when using pivot thresholding [48] is

$$\rho \leq (1 + \mu^{-1})^{n-1} \max_{i,j} |a_{ij}|.$$

Another recourse available is to find a new pivot within the pivot block of the current frontal matrix (assuming the frontal matrix has more than one potential pivot). As all the pivot rows and pivot columns of frontal matrices have an identical pattern, unsymmetric pivoting within the pivot block can occur in hopes of finding a suitable next pivot. The P3 and P4 partial dense factoring algorithms, discussed in Chapter 5 were designed to implement recoveries. These recoveries within pivot blocks are favorable since they do not change the frontal matrix's dimensions, structure of the assembly DAG, or the structure of the LU factors. Furthermore, pivoting within pivot blocks can be combined with the avoidance strategy discussed earlier to improve the chances of finding an acceptable pivot within the pivot block.

Unfortunately, acceptable pivots may not always be available within the pivot block. In this case, the failed pivots will have to be shifted to a later frontal matrix. This will change the weights of the frontal matrix nodes in the assembly DAG and could also change the structure as defined by the edges of the assembly DAG. Changing the edges of the assembly DAG is especially problematic within the setting of a distributed memory implementation as implementations are typically statically scheduled and changes in the assembly DAG edge set can invalidate the schedule.

The classical multifrontal method avoids changes in the edge set of the assembly DAG because the assumption of symmetry results in an assembly DAG that is a tree in which each frontal matrix can be recovered by its single direct LU parent [7, 4]. Thus the size and required computations for frontal matrices can change (and hence the structure of the LU factors), but the edge set of the assembly DAG (tree) remains constant.

Within the context of an unsymmetric multifrontal approach, lost pivot recovery becomes more difficult. An LU parent may not even exist for a particular frontal matrix and even if it does, there may be intervening L and/or U parents, which will be affected by the recovery. Furthermore, results to be established in this section will show that the assembly DAG edge set can be insufficient to facilitate lost pivot recoveries in the general case. Hence, the assembly DAG edge set will need to be augmented, prior to scheduling, with additional edges to accommodate any necessary lost pivot recoveries.

This section will establish the necessary theoretical foundation for an unsymmetric, multifrontal lost pivot recovery capability that will accommodate any number of failed pivots using an augmented assembly DAG that can be determined prior to scheduling and starting the refactorization process. Furthermore, a permutation to block upper triangular form will be shown to allow the original assembly DAG to

fully support all necessary lost pivot recovery and will guarantee the existence of an LU ancestor for all frontal matrices (except of course for those at the end of the diagonal blocks).

The basic strategy is to designate a recovery frontal matrix for each frontal matrix. The first LU ancestor (defined as the nearest frontal matrix to which there is both a path consisting strictly of L edges and a path consisting strictly of U edges) is used for the recovery matrix whenever such a frontal matrix is available. When an LU ancestor is not available, the last frontal matrix as (defined by the pivot ordering) is used. The rows and columns of any lost pivots in the frontal matrix will be symmetrically permuted such that they expand the pivot block of the recovery frontal matrix. Frontal matrices between the failed frontal matrix and its recovery matrix can be affected by the recovery of these failed pivots. But, once the recovery matrix is reached, all direct effects of the recovery process, such as those to intermediate L and/or U ancestors of the failed frontal matrix, will have been accounted for and the recovery process for the failed pivots concluded. (An L ancestor is similar to an L parent except that an L ancestor has an incoming path of L edges, whereas an L parent has just one L edge. A U ancestor has similar meaning except U edges and paths of U edges are used). If these pivots fail again in their new frontal matrix, they can be considered as failures in the new frontal matrix and handled accordingly.

When L and/or U ancestors occur between a failed frontal matrix and its recovery frontal matrix, they will be directly affected by the recovery. Specifically, an L ancestor will need to be augmented with the failed pivot columns and a U ancestor will need to be augmented with the failed pivot rows. This results directly from the symmetric permutations of the failed pivot rows and columns. New fill-in in the failed pivot columns and rows, respectively, is possible within the confines of these augmentations and will need to be passed to the frontal matrix that holds the particular entry in its expanded L or U factor.

A simple example of these basic recovery concepts is provided in Figures 7-1, 7-2, and 7-3. In Figure 7-1, there is an example of a sparse matrix with the rows and columns labeled according to the frontal matrices that own the corresponding pivot entries. The frontal matrix A has two potential pivots. The first pivot, a_{A_g,A_g} is acceptable. The second pivot, a_{A_l,A_l} fails to be numerically acceptable. Figure 7-2 shows the corresponding assembly DAG. In this assembly DAG, the frontal matrix D is designated to be the recovery matrix for the failed frontal matrix A since there is a path of L edges from A to B to D and a path of U edges from A to C to D. Hence, the failed pivot row and column of frontal matrix A will be symmetrically permuted to expand the pivot block of frontal matrix D as shown in the second matrix of Figure 7-1. As frontal matrices B and C are intermediate L and U parents, respectively, they are affected by the recovery of A. Figure 7-3 shows the individual frontal matrices as they are augmented for the recovery of A. As an L parent, frontal matrix B is augmented with the lost pivot column A_l and new fill-in occurs in the a_{D,A_l} entry (shown by a "*" in Figure 7-1). Likewise, the U parent C is augmented with the lost pivot row A_l and new fill-in results in the $a_{A_l,D}$ entry. The recovery matrix D has its pivot block expanded from the single $a_{D,D}$ entry to the 2 by 2 submatrix consisting of

the entries: $a_{D,D}$, a_{D,A_l}, $a_{A_l,D}$, and a_{A_l,A_l}. As unsymmetric permutations are allowed in the expanded pivot block of D, fewer entries are left in the pivot columns, and some the pertinent entries have been updated per earlier pivots, there is a greater likelihood of that the lost pivot can be recovered within its new location.

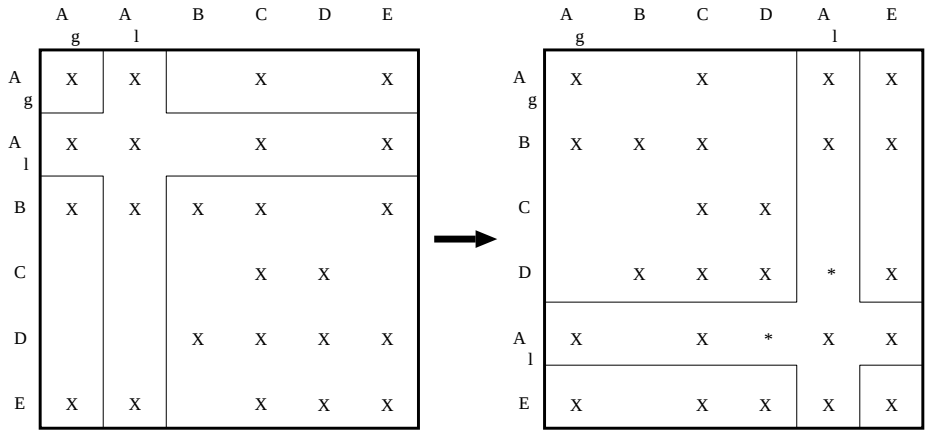

Figure 7-1. Sparse Matrix With Lost Pivot

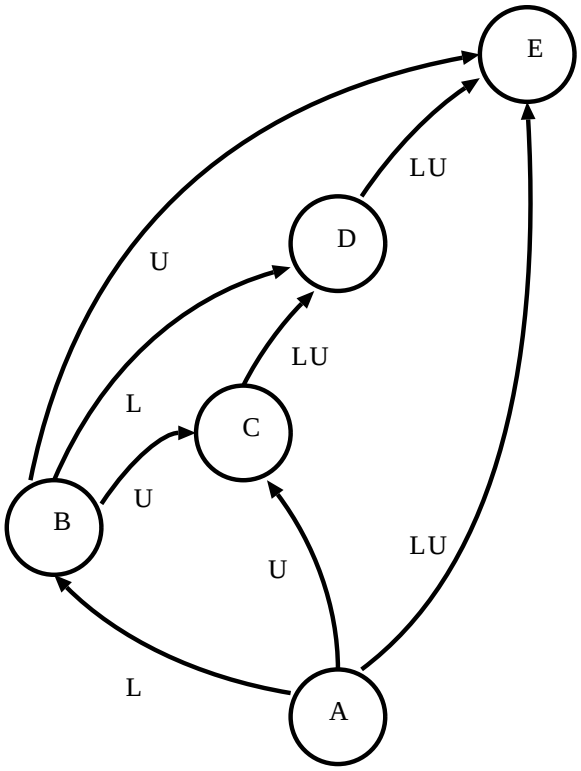

Figure 7-2. Sample Assembly DAG

Further complications arise when the possibility of multiple failures are considered. For example, when a frontal matrix is expanded by the lost pivot rows of

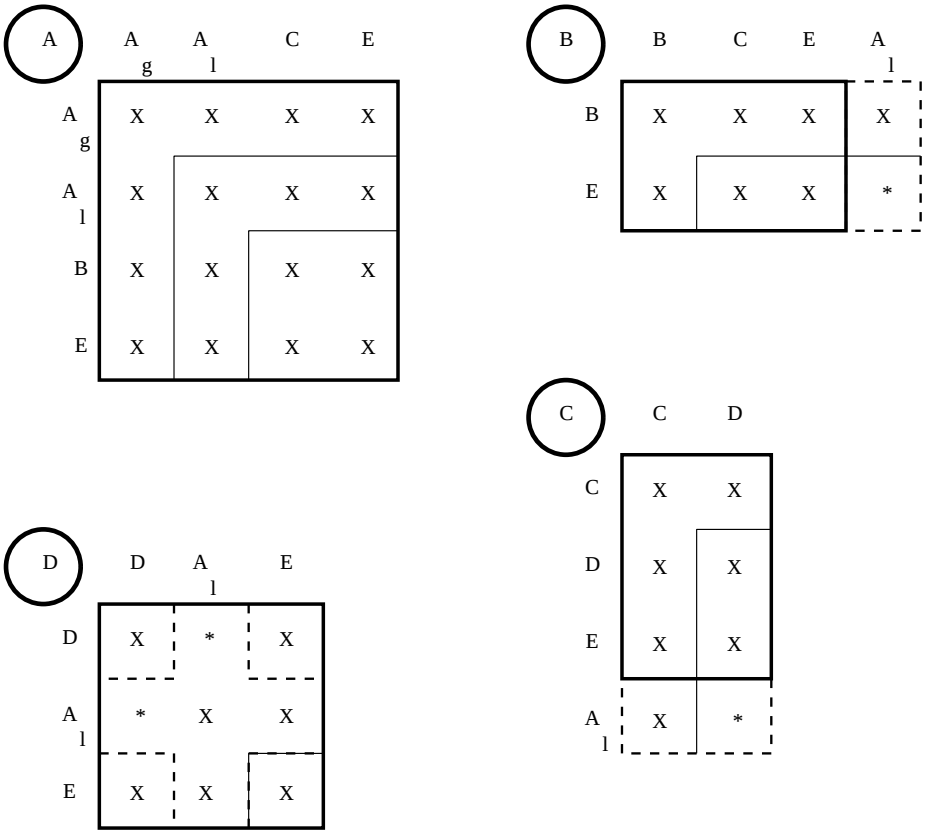

Figure 7-3. Frontal Matrices Augmented by Lost Pivot Recovery

one failed frontal matrix and the lost pivot columns of another frontal matrix, how should the contributions in the intersection of these new rows and columns be handled? Also, recovery matrices will have the definition of their L and U factors altered, which changes how they are affected by other failures. The theory to be developed will show that a well defined ordering on the resolution of recovery effects will insure that multiple failures are handled properly.

As mentioned earlier, unless block upper triangular form can be assumed, the assembly edge set may not be sufficient to support lost pivot recovery. Thus, the edge set used for lost pivot recovery must be derived from the *true* edge set, that is, the set of edges that defines all the relationships between frontal matrices. Actually, a transitive reduction of this true edge set is sufficient and some results will be presented to aid in determining this transitive reduction. The true edge set will however need to be augmented to insure that every frontal matrix has a path to the last frontal matrix, which is necessary to handle the case of no LU ancestor in which the last frontal matrix is designated to be default recovery matrix.

The remainder of this section details and formalizes the lost pivot recovery mechanisms just outlined together with proving their sufficiency and correctness. First, the necessary definitions and notations are established, followed by some foundational theorems. Then the effects of a single frontal matrix's failure are detailed with a description of how the absence of an LU ancestor has significant ramifications on

the assembly DAG's structure. The theory is then extended to address multiple and repeated failures. The communication path requirements are then detailed. Finally, the benefits of block upper triangular form are formalized.

7.1.1 Definitions and Notation

In order to formalize and prove the correctness of the lost pivot recovery mechanisms outlined above, some key attributes of frontal matrices and assembly DAGs must be defined and adequate notation introduced.

The ordered set, $\mathcal{F}$, is defined as the set of all frontal matrices in a multifrontal formulation of a particular sparse matrix factorization. Each element of this set represents a distinct frontal matrix and an equivalence relation is established between these frontal matrices and the original position of their first pivots in the pivot ordering (assuming the necessary permutations have been applied to put the matrix into the pivot ordering used by the initial analyze-factorize routine). Hence the ordering of the pivot entries provides the ordering for the set $\mathcal{F}$ and a frontal matrix's ID is defined to be the pivot column (or row) index of the first pivot within the frontal matrix.

When block upper triangular form is in use, the notation $\mathcal{F}_i$ will represent the subset of $\mathcal{F}$ that contains all of the frontal matrices in the ith diagonal block. Within the context of the assembly DAG, $\mathcal{F}_i$ will define the node set of a sub-DAG that is not connected to any other nodes in the assembly DAG (nodes not in $\mathcal{F}_i$).

Frontal matrices will typically be represented by capital letters. A special notation is reserved for frontal matrices that are recovery frontal matrices. Specifically, for $A \in \mathcal{F}$, the notation R_A (where $R_A \in \mathcal{F}$) represents the recovery frontal matrix of A. As mentioned earlier, R_A will be the first LU ancestor of A, if such an LU ancestor exists. Otherwise, the recovery matrix will be defined as the last frontal matrix in the ordered set $\mathcal{F}$.

As the set $\mathcal{F}$ is an ordered set, the less than ($<$), equal to ($=$), and greater than ($>$) relations are well defined and trichotomy insures that for any two frontal matrices A and $B \in \mathcal{F}$ only one of these relations applies and that if $A = B$, then A and B represent the same frontal matrix.

Within a particular frontal matrix, the specific components (submatrices) will frequently need to be referenced. This will be done using the same notation found in Davis and Duff's original work [34, 35]. Specifically, $\mathcal{L}_A$ refers to the set of row indices that define the rows from the overall matrix that make up the frontal matrix $A \in \mathcal{F}$. Likewise, $\mathcal{U}_A$ refers to the set of column indices that define the columns of the entire matrix. (Again, the indices of the overall matrix assume the necessary permutations have been applied to establish the initial pivot ordering as determined by the analyze-factorize algorithm). The subset $\mathcal{L}'_A \subseteq \mathcal{L}_A$ is the set of row indices that define the pivot rows of the frontal matrix A. Prior to any lost pivot recovery, $\mathcal{L}'_A$ will be equal to $\{A, A+1, \ldots, A+(k_A-1)\}$ where A is the row and column index of the first pivot of frontal matrix A and k_A is the number of potential pivots in frontal matrix A. (Here A is used as both a row/column index and a frontal matrix identifier to illustrate the correspondance between the lead pivot's row/column index and the frontal matrix identifier). The subset $\mathcal{L}''_A \subseteq \mathcal{L}_A$ contains all the row indices

of the nonpivotal rows of A. These nonpivotal rows of define the rows of the frontal matrix's contribution block and hence define the L edges from the frontal matrix A to L parent frontal matrices. In a similar fashion, $\mathcal{U}'_A \subseteq \mathcal{U}_A$ defines the indices of the pivot columns of A and will initially be equal to $\mathcal{L}'_A$. $\mathcal{U}''_A \subseteq \mathcal{U}_A$ defines the indices of the nonpivotal columns of A and hence also defines the columns of the contribution block and the U edges to U parents of A. The $\mathcal{L}$ and $\mathcal{U}$ notation will also be useful beyond the scope of a particular frontal matrix to define subsets of row and column indices of the rows and columns of the overall matrix.

Furthermore, sometimes subsets of the $\mathcal{L}''$ and $\mathcal{U}''$ patterns will need to be referenced. Specifically, the notation $\mathcal{L}''_A(B)$ will refer to the subset of frontal matrix A's pattern of nonpivotal rows that are pivotal rows for frontal matrices B and greater. Likewise, $\mathcal{U}''_A(B)$ refers to the nonpivotal columns of frontal matrix A that are pivot columns for frontal matrices B and greater.

When discussing the failed pivots of a frontal matrix, the notation $\mathcal{L}_A^{lost}$ and $\mathcal{U}_A^{lost}$ will be used to define the row and columns indices, respectively, of the lost pivots within frontal matrix A. The notation, $\mathcal{L}_A^{good}$ and $\mathcal{U}_A^{good}$ will be used to represent the row and column indices of the acceptable pivots within the frontal matrix A. These patterns accommodate any unsymmetric permutations that may have taken place within the frontal matrix's pivot block. Using this notation, the following relationships hold

$$\mathcal{L}'_A = \mathcal{L}_A^{good} \cup \mathcal{L}_A^{lost} \text{ and } \mathcal{L}_A^{good} \cap \mathcal{L}_A^{lost} = \emptyset$$
$$\mathcal{U}'_A = \mathcal{U}_A^{good} \cup \mathcal{U}_A^{lost} \text{ and } \mathcal{U}_A^{good} \cap \mathcal{U}_A^{lost} = \emptyset.$$

In the larger context of the entire matrix, the notation: $\mathcal{L}(\mathcal{F}_i)$ refers to the matrix rows that make up the ith diagonal block and $\mathcal{U}(\mathcal{F}_i)$ refers to the matrix columns that make up the ith diagonal block.

Relations between frontal matrices reflect the need to pass the contributions of one frontal matrix to another frontal matrix where the contributions will be applied to entries in the destination's pivot row and/or pivot columns. These are called the *true* relations and can be represented as edges in a graph representation of the matrix. (Other types of relations/edges are also useful and will be defined as needed). There are two types of these true relations/edges. An L edge from frontal matrix A to B occurs when $\mathcal{L}''_A \cap \mathcal{L}'_B \neq \emptyset$. That is, A produces contributions that need to be assembled into B's pivot rows. Such an L edge is represented by the notation

$$A \xrightarrow{L} B.$$

In such a relation, A is said to be the L child of B and B the L parent of A. Similarly, a U edge from A to B exists if $\mathcal{U}''_A \cap \mathcal{U}'_B \neq \emptyset$ and represents the fact that A produces contributions that need to be assembled into the pivot columns of B. Such an edge is represented by the notation

$$A \xrightarrow{U} B$$

and A is called the U child of B and B is the U parent of A. When L and U edges both exist from A to B, an LU edge is said to exist and is shown as

$$A \overset{LU}{\rightarrow} B$$

with A the LU child of B and B the LU parent of A. In Figure 7-2, an L edge exists from frontal matrix A to B, a U edge exists from A to C, and an LU edge exists from C to D.

Notice that due to the ordering scheme imposed on $\mathcal{F}$ and the fact that contributions are always passed to frontal matrices that appear later in the pivot ordering, all edges from a frontal matrix A to B will be such that $A < B$.

An L path exists between frontal matrices A and B if and only if there is a path from A to B consisting of all L edges. In this case B is called an L ancestor of A and A is an L predecessor of B. When such a path contains at least one edge, the notation

$$A \overset{L^+}{\rightsquigarrow} B$$

is used to represent the path. Correspondingly, a U path from A to B exists if and only if the is a path from A to B of all U edges with B the U ancestor of A and A the U predecessor of B. Such a path is denoted by

$$A \overset{U^+}{\rightsquigarrow} B.$$

An LU ancestor of a frontal matrix A is another frontal matrix such that there is both an L path and a U path from A to this other frontal matrix. The L and U paths may or may not share frontal matrices as path nodes. Such relationships are denoted by

$$A \overset{LU^+}{\rightsquigarrow} B.$$

In Figure 7-2, there is an L path from A to D and a U path from A to D which means there is an LU path from A to D.

Furthermore, in some situations, paths of zero or more edges will need to be considered (where a zero length path exists only if both the source and destination frontal matrices are the same). Such paths are denoted by

$$A \overset{L^*}{\rightsquigarrow} B \text{ and } A \overset{U^*}{\rightsquigarrow} B.$$

The definitions of L and U edges represent the true relationships between frontal matrices and thus the set of all of these edges is called the true edge set and denoted: $\mathcal{E}_T$. As $\mathcal{E}_T$ really consists of two types of edges (L edges and U edges), it can be decomposed into two disjoint subsets: $\mathcal{E}_T^L$, which contains only the L edges, and $\mathcal{E}_T^U$, which contains only the U edges. This is similar to a formulation of elimination DAGs done by Gilbert and Liu [75], which is used to predict the structure of the LU factors.

The edge set used to define the assembly DAG (which is provided as input) is called the assembly edge set and denoted by $\mathcal{E}_A$. In general, $\mathcal{E}_A$ is a transitive

reduction of $\mathcal{E}_T$ and it is frequently useful to think of it that way. However, later results in this section will show that not all relationships represented by $\mathcal{E}_T$ need exist in $\mathcal{E}_A$ (unless block upper triangular form is imposed). Thus, the need for a separate notation.

The transitive closure of an edge set is the extension of the edge set created by adding an edge A to B whenever there is a path from A to B in the original edge set. The transitive closure of an edge set $\mathcal{E}$ is denoted by $\mathcal{E}^+$.

Finally, an *entry* is defined as any originally nonzero element of the matrix or any element a_ij such that $i \in \mathcal{L}_A$ and $j \in \mathcal{U}_A$ for some frontal matrix A. That is, all nonzeros of the original matrix and any element in an active submatrix are considered entries. While such entries may become zero, in general they are nonzero and must assumed to be nonzero.

With the preceding notation and definitions established, development of the necessary theoretical results may commence. This development starts with some foundational theorems that will be useful throughout the rest of the section.

7.1.2 Foundational Theorems

Three theorems and two corollaries form the foundation upon which most of the lost pivot recovery results are built. These are provided below:

Theorem 7.1 (L Ancestor Fill-In) *If* $A \overset{L^+}{\rightsquigarrow} B$ *then* $\mathcal{U}''_A(B) \subseteq \mathcal{U}_B$.

PROOF: The proof of this theorem follows an induction argument based on L edges of the L path from A to B.

BASIS: For $A \overset{L}{\rightarrow} B$ (direct L edge), the entries a_{i_1,j_1}, a_{i_2,j_1}, and a_{i_1,j_2} must exist for $i_1 \in \mathcal{L}'_A$, $i_2 \in \mathcal{L}'_B \cap \mathcal{L}''_A$, $j_1 \in \mathcal{U}'_A$, and $j_2 \in \mathcal{U}''_A$. This implies $i_2 \in \mathcal{L}_B$ and $j_2 \in \mathcal{U}_B$ which implies that a_{i_2,j_2} is an entry and $\mathcal{U}''_A(B) \subseteq \mathcal{U}_B$.

INDUCTION: If $A \overset{L^+}{\rightsquigarrow} C \overset{L}{\rightarrow} B$, then by inductive hypothesis $\mathcal{U}''_A(C) \subseteq \mathcal{U}_C$. Furthermore, as $C \overset{L}{\rightarrow} B$, the argument of the induction basis provides $\mathcal{U}''_C(B) \subseteq \mathcal{U}_B$ and an edge from C to B implies that $C < B$. Thus, the following set of inclusions are established

$$\mathcal{U}''_A(B) \subseteq \mathcal{U}''_C(B) \subseteq \mathcal{U}_B. \square$$

Corollary 7.2 *If* $A \overset{L^*}{\rightsquigarrow} B$ *then* $\mathcal{U}''_A(B) \subseteq \mathcal{U}_B$.

This corollary simply adds the null edge case, which is true via the definition $\mathcal{U}''_A \subset \mathcal{U}_A$.

Theorem 7.3 (U Ancestor Fill-In) *If* $A \overset{U^+}{\rightsquigarrow} B$ *then* $\mathcal{L}''_A(B) \subseteq \mathcal{L}_B$.

PROOF: Follows directly as the transpose of the proof of Theorem 7.1.

Corollary 7.4 *If* $A \overset{U^*}{\rightsquigarrow} B$ *then* $\mathcal{L}''_A(B) \subseteq \mathcal{L}_B$.

This corollary simply adds the null edge case, which is true via the definition $\mathcal{L}''_A \subset \mathcal{L}_A$.

Theorem 7.5 (Paths Due to Fill-In) *Assume that* $A \overset{L^+}{\rightsquigarrow} B$ *and* $A \overset{U^+}{\rightsquigarrow} C$, *then (a) if* $B < C$ *there is an U path* $B \overset{U^+}{\rightsquigarrow} C$ *or (b) if* $B > C$ *there is an L path* $C \overset{L^+}{\rightsquigarrow} B$.

PROOF: The proof of this theorem is an induction argument based on the L and U paths. Furthermore, the basis is strengthened to include a path of zero edges which is needed to facilitate the proof.

BASIS: Assume $A \overset{L}{\rightarrow} B$ and $A \overset{U^*}{\rightsquigarrow} C$. Then consider the case of $B < C$ and all C_1 and C_2 such that

$$A \overset{U^*}{\rightsquigarrow} C_1 \overset{U}{\rightarrow} C_2 \overset{U^*}{\rightsquigarrow} C \text{ and } C_1 < B < C_2.$$

(Note that C_1 could be A and C_2 could be C). As $A \overset{U^*}{\rightsquigarrow} C_1$, the inclusion: $\mathcal{L}''_A(C_1) \subseteq \mathcal{L}_{C_1}$, holds per Corollary 7.4. Then $C_1 < B$ and $A \overset{L}{\rightarrow} B$ implies that $\mathcal{L}'_B \cap \mathcal{L}''_{C_1} \neq \emptyset$. Then by definition the edge $C_1 \overset{L}{\rightarrow} B$ exists. The existance of the edge $C_1 \overset{L}{\rightarrow} B$ together with $C_1 < B$ and Corollary 7.2 implies $\mathcal{U}''_{C_1}(B) \subseteq \mathcal{U}_B$. Due to $C_1 \overset{U}{\rightarrow} C_2$ and $B < C_2$, $\mathcal{U}'_{C_2} \cap \mathcal{U}''_{C_1}(B) \neq \emptyset$ and thus $\mathcal{U}'_{C_2} \cap \mathcal{U}_B \neq \emptyset$. Hence the edge $B \overset{U}{\rightarrow} C_2$ exists and, as $C_2 \overset{U^*}{\rightsquigarrow} C$, the result $B \overset{U^+}{\rightsquigarrow} C$ follows.

If $B > C$ then $A \overset{U^*}{\rightsquigarrow} C$ implies $\mathcal{L}''_A(C) \subseteq \mathcal{L}_C$ by Corollary 7.4. Since $A \overset{L}{\rightarrow} B$ and $B > C$, the results $B \in \mathcal{L}''_C$ and $C \overset{L}{\rightarrow} B$ follow.

Furthermore, the transpose of this argument establishes that the results also hold for $A \overset{L^*}{\rightsquigarrow} B$ and $A \overset{U}{\rightarrow} C$.

INDUCTION: For the inductive step, the argument assumes paths of length one or more as follows: $A \overset{L^+}{\rightsquigarrow} B$ and $A \overset{U^+}{\rightsquigarrow} C$. First the case of $B < C$ is considered with any three frontal matrices: C_1, C_2, and B_1 such $C_1 < B_1 < C_2$ and $A \overset{U^*}{\rightsquigarrow} C_1 \overset{U}{\rightarrow} C_2 \overset{U^*}{\rightsquigarrow} C$ and $A \overset{L^+}{\rightsquigarrow} B_1 \overset{L^*}{\rightsquigarrow} B$. By inductive hypothesis, the edge $C_1 \overset{L}{\rightarrow} B_1$ exists. This result, together with Theorem 7.1, implies that $\mathcal{U}''_{C_1}(B_1) \subseteq \mathcal{U}_{B_1}$. Hence, as $C_1 \overset{U}{\rightarrow} C_2$ and $B_1 < C_2$, one finds that $C_2 \in \mathcal{U}''_{B_1}$ and the edge $B_1 \overset{U}{\rightarrow} C_2$ exists, which establishes the path $B_1 \overset{U^+}{\rightsquigarrow} C$. As none of the selection criteria on C_1, B_1, or C_2 preclude $B_1 = B$, these results include $B \overset{U^+}{\rightsquigarrow} C$.

The transpose of the argument just provided establishes that if $B > C$ the results $C \overset{L^+}{\rightsquigarrow} B$ also hold. □

With this foundational theory established, construction may commence on the specific lost pivot recovery theory.

7.1.3 Fill-In due to Recovery

In this section, theoretical results are developed to describe how and where new fill-in is produced due to the recovery of a frontal matrix's lost pivot rows and columns. Specifically, new fill-in resulting from the recovery of lost pivots occurs in the lost pivot rows and columns. New fill-in the lost pivot columns is limited in scope to the row patterns of L ancestors. New fill-in in the lost pivot rows is limited in scope to the column patterns of U ancestors. Furthermore, fill-in in a particular row of the lost pivot columns (or column of the lost pivot rows) can result from

several frontal matrices. This will have significant impacts on the distributed memory implementation to be described in the next section as the fill-in contributions become decentralized and must be combined in a fashion that eliminates duplicates and insures all distinct contributions are included.

A final, yet very significant, result is that the first LU ancestor will be able to absorb the remaining nonzero rows of the lost pivot columns and nonzero columns of the lost pivot rows by simply extending its pivot block (defined by its $\mathcal{L}'$ and $\mathcal{U}'$ patterns) to include the lost pivot rows and columns. Hence, the failed frontal matrix is completely recovered by its first LU ancestor and will have no additional direct effects on the remaining frontal matrices and the assembly DAG.

Throughout the theorems and corollaries of this section, the reader should recall that the recovery frontal matrix for any particular failed frontal matrix is defined as the first LU ancestor, if such exists. Otherwise the recovery frontal matrix is the last of all frontal matrices.

Theorems 7.6 and 7.9 describe how fill-in in the lost pivot columns and rows are produced by L and U ancestors, respectively. Such fill-in was illustrated earlier in the example of Figures 7-1 through 7-3 with the L parent B and the U parent C. Corollaries 7.7 and 7.10 describe how parts of the lost pivot rows and columns are absorbed by L and U ancestors. Also, Corollaries 7.8 and 7.11 establish the possibility of multiple contributions to lost pivot rows and columns entries.

Theorem 7.6 (L Ancestor Recovery Fill-In) *For $A, B, R_A \in \mathcal{F}$ such that A contains failed pivots, $A \overset{L^+}{\rightsquigarrow} B$, R_A is the recovery frontal matrix of A, and $B < R_A$, there will be fill-in in the intersection of the rows of $\mathcal{L}_B$ and columns of $\mathcal{U}_A^{lost}$.*

PROOF: The proof follows an induction argument on the L edges of the L path from A to B.

BASIS: If $A \overset{L}{\rightarrow} B$ (direct L edge), then $a_{i,j}$ is an entry for all $i \in \mathcal{L}'_B \bigcap \mathcal{L}''_A$ and all $j \in \mathcal{U}'_A$ including $j \in \mathcal{U}_A^{lost}$. Thus, when the lost pivot columns ($\mathcal{U}_A^{lost}$) are permuted to extend the pivot block of R_A, these entries will cause an extension of the $\mathcal{U}''_B$ by the columns of $\mathcal{U}_A^{lost}$ and the resulting fill-in will cause the intersection of the $\mathcal{L}_B$ rows and the $\mathcal{U}_A^{lost}$ columns to become entries.

INDUCTION: If $A \overset{L^+}{\rightsquigarrow} B$ (L path) with $B < R_A$, then for $C \in \mathcal{F}$ such that $A \overset{L^*}{\rightsquigarrow} C \overset{L}{\rightarrow} B$, the inductive hypothesis provides that $\mathcal{U}''_C$ has been extended by $\mathcal{U}_A^{lost}$. Since $C \overset{L}{\rightarrow} B$ and $B < R_A$, Theorem 7.1 provides that $\mathcal{U}_A^{lost} \subseteq \mathcal{U}''_C(B) \subseteq \mathcal{U}''_B$, which means frontal matrix B must be extended by the lost pivot columns. Hence, there will be new fill-in in the intersection of the $\mathcal{L}_B$ rows and the columns of $\mathcal{U}_A^{lost}$. □

Corollary 7.7 *The entries in the nonzero rows of the $\mathcal{U}_A^{lost}$ columns that are contained in $\mathcal{L}'_B$ pattern for B as defined in Theorem 7.6 will be fully absorbed by the frontal matrix B and will play no further role in the recovery of A.*

This is true as these entries go into the pivot rows of B and will be now considered as part of the frontal matrix B. If good pivots are found in these rows, they will become part of the U factor associated with frontal matrix B, else they will be handled as B's failed pivot rows.

Corollary 7.8 *For $B, C \in \mathcal{F}$ such that $A \overset{L^+}{\rightsquigarrow} B$ and $A \overset{L^+}{\rightsquigarrow} C$ with $B < R_A$ and $C < R_A$ and R_A the recovery matrix for A, there is the possibility that either $\mathcal{L}_B \bigcap \mathcal{L}''_C \neq \emptyset$ and/or $\mathcal{L}''_B \bigcap \mathcal{L}_C \neq \emptyset$. Thus, there may be multiple contributions made to a particular row of the $\mathcal{U}_A^{lost}$ columns as a result of the recovery of A.*

Theorem 7.9 (U Ancestor Recovery Fill-In) *For $A, B, R_A \in \mathcal{F}$ such that A contains failed pivots, $A \overset{U^+}{\rightsquigarrow} B$, R_A is the recovery frontal matrix of A, and $B < R_A$, there will be fill-in in the intersection of the columns of $\mathcal{U}_B$ and the rows of $\mathcal{L}_A^{lost}$.*

PROOF: Follows the transpose of the argument for Theorem 7.6. Furthermore, the transpose of Corollaries 7.7 and 7.8 also follow.

Corollary 7.10 *The entries in the nonzero columns of the $\mathcal{L}_A^{lost}$ rows that are contained in $\mathcal{U}'_B$ pattern for B as defined in Theorem 7.9 will be fully absorbed by the frontal matrix B and will play no further role in the recovery of A.*

Corollary 7.11 *For $B, C \in \mathcal{F}$ such that $A \overset{U^+}{\rightsquigarrow} B$ and $A \overset{U^+}{\rightsquigarrow} C$ with $B < R_A$ and $C < R_A$ and R_A the recovery matrix for A, there is the possibility that either $\mathcal{U}_B \bigcap \mathcal{U}''_C \neq \emptyset$ and/or $\mathcal{U}''_B \bigcap \mathcal{U}_C \neq \emptyset$. Thus, there may be multiple contributions made to a particular column of the $\mathcal{L}_A^{lost}$ rows as a result of the recovery of A.*

Theorem 7.12 (Limits of Lost Pivot Column Fill-In) *The only fill-in in the lost pivot columns of a frontal matrix A is due to L ancestors of A.*

PROOF: Assume $a_{i,k}$ is new fill-in created due to the recovery of A where $k \in \mathcal{U}_A^{lost}$ but $i \notin \mathcal{L}_B$ for any $B \in \mathcal{F}$ such that $A \overset{L^+}{\rightsquigarrow} B$. Then, there must exist a j such that $A < j < R_A$ such that $a_{i,j} \neq 0$, and $a_{j,j} \neq 0$, and $a_{j,k} \neq 0$ otherwise the fill-in would not have occurred. If $a_{j,k} \neq 0$ regardless of the pivot failure in A then the edge $A \overset{L}{\rightarrow} J$ with $j \in \mathcal{L}'_J$ must exist, which raises a contradiction. Otherwise, if $a_{j,k}$ became an entry due to fill-in from the recovery of A then this argument is inductively applied to the $a_{j,k}$ entry. The induction is terminated by either finding an entry in column k that existed even before the recovery (which results in the contradiction previously described) or if no such entry is found the fill-in could not of occurred (another contradiction of the assumptions). □

Theorem 7.13 (Limits of Lost Pivot Row Fill-In) *The only fill-in in the lost pivot rows of a frontal matrix A is due to U ancestors of A.*

PROOF: Follows the transpose of the argument for Theorem 7.12.

Theorem 7.14 *The recovery frontal matrix R_A for a failed frontal matrix A needs only to have its pivot block (defined by $\mathcal{L}'_{R_A}$ and $\mathcal{U}'_{R_A}$) extended to accommodate the recovery of A. That is, $\mathcal{L}'_{R_A} \leftarrow \mathcal{L}'_{R_A} \cup \mathcal{L}_A^{lost}$ and $\mathcal{U}'_{R_A} \leftarrow \mathcal{U}'_{R_A} \cup \mathcal{U}_A^{lost}$.*

PROOF: For any $B \in \mathcal{F}$ such that $A \overset{L^*}{\rightsquigarrow} B \overset{L^+}{\rightsquigarrow} R_A$, Theorem 7.1 provides that $\mathcal{U}''_B(R_A) \subseteq \mathcal{U}_{R_A}$. Similarly, for any $C \in \mathcal{F}$ such that $A \overset{U^*}{\rightsquigarrow} C \overset{U^+}{\rightsquigarrow} R_A$, Theorem 7.3 provides that $\mathcal{L}''_C(R_A) \subseteq \mathcal{L}_{R_A}$. As $\mathcal{U}''_B$ and $\mathcal{L}''_C$ have only been extended by $\mathcal{U}^{lost}_A$ and $\mathcal{L}^{lost}_A$, respectively, these are the only extensions needed to the recovery frontal matrix R_A. □

Corollary 7.15 *All edges $A \overset{L}{\rightarrow} B$ and $A \overset{U}{\rightarrow} B$ such that $B > C$ and $A \overset{LU^+}{\rightsquigarrow} C$ are redundant due to the existence of corresponding L and U paths from C to B, respectively.*

The next theorem establishes that by using the first LU ancestor as the recovery matrix, all contributions to the lost pivot block by earlier ancestors are avoided. This result allows the implementation to preclude the multiple contribution handling for the lost pivot block portion of the lost pivot rows and columns. Corollaries 7.8 and 7.11 established the need for this processing for the rest of the lost pivot row and column entries.

Theorem 7.16 *If the first LU ancestor B of A (that is, $A \overset{LU^+}{\rightsquigarrow} B$ with $A \overset{LU^+}{\rightsquigarrow} C$ for no $C < B$) is used for the recovery matrix for A (designated R_A), then there will be no contributions made to the lost pivot block of A by L or U ancestors of A.*

PROOF: The lost pivot block of the failed frontal matrix A is defined by the rows in $\mathcal{L}^{lost}_A$ and the columns in $\mathcal{U}^{lost}_A$. By Theorems 7.12 the columns of $\mathcal{U}^{lost}_A$ are only updated via L ancestors and by Theorem 7.13 the rows of $\mathcal{L}^{lost}_A$ are only updated by U ancestors. Thus, for updates to the lost pivot block of A to occur, the updating frontal matrix must be both an L and a U ancestor of the failed frontal matrix. As the recovery matrix, R_A, is the first such LU ancestor and as the patterns $\mathcal{L}'_{R_A}$ and $\mathcal{U}'_{R_A}$ are extended by $\mathcal{L}^{lost}_A$ and $\mathcal{U}^{lost}_A$, respectively, Corollaries 7.7 and 7.10 provided that no further contributions to A's lost pivot block will be made as a result of the recovery of A. □

The practical results of the theory just developed are that the recovery of a failed frontal matrix can be facilitated by (a) extending all L ancestors that occur before the first LU ancestor by the lost pivot rows, (b) extending all U ancestors that occur before the first LU ancestor by the lost pivot columns, and (c) extending the pivot block of the recovery frontal matrix by the lost pivot rows and columns. Furthermore, L ancestors absorb the entries of the lost pivot columns that occur in the ancestor's pivot rows and, likewise, U ancestors absorb the entries of the lost pivot rows that occur in the ancestor's pivot columns.

The next section shall establish that all new contributions resulting from a lost pivot recovery can be passed between frontal matrices using the existing edges in the edge set $\mathcal{E}_T$ when an LU ancestor exists and that only a well defined set of additional edges could be needed if there is no LU ancestor. Later results will establish the relationship between $\mathcal{E}_T$ and the edge set of the assembly DAG provided by the UMFPACK software [32, 34, 35].

7.1.4 Impacts on Assembly DAG

One of the most significant features of the lost pivot recovery mechanisms developed in this chapter is the minimal impact on the relationships between frontal matrices as defined by the edge set $\mathcal{E}_T$. Specifically, when LU ancestors are available to serve as recovery matrices, no additional edges need be added to $\mathcal{E}_T$ as a result of the recovery. If an LU ancestor is not available for a particular failed frontal matrix, then a specific frontal matrix is selected for the recovery matrix and only edges to that frontal matrix may need to be added to $\mathcal{E}_T$.

The first theorem in this section establishes the result that no additional edges are necessary if an LU ancestor is available as a recovery matrix.

Theorem 7.17 (No Additional Edges For LU Ancestor Recoveries) Only redundant edges are created due to recovery fill-in when the recovery frontal matrix R_A is an LU ancestor of the failed frontal matrix A.

PROOF: Since $A \overset{LU^+}{\rightsquigarrow} R_A$, Theorem 7.3 provides that $\mathcal{L}''_A(R_A) \subseteq \mathcal{L}_{R_A}$. Furthermore, for each $B \in \mathcal{F}$ such that $A \overset{L^+}{\rightsquigarrow} B$ and $B < R_A$, Theorem 7.5 insures the existence of a U path: $B \overset{U^+}{\rightsquigarrow} R_A$. This path, together with Theorem 7.3, provides that $\mathcal{L}''_B(R_A) \subseteq \mathcal{L}_{R_A}$. By Theorem 7.12, this accounts for all the recovery fill-in in the pattern $\mathcal{L}_{R_A}$, so no new L edges are created from R_A. Furthermore, the transpose of this argument insures that no new U edges are created from R_A.

Next, the new edges to R_A are considered. If $A \overset{L^+}{\rightsquigarrow} B$ with $B < R_A$, then there will be potentially new fill-in in the $\mathcal{L}''_B$ rows of the $\mathcal{U}^{lost}_A$ columns. But for each row $c \in \mathcal{L}''_B$, by definition, there exists and edge $B \overset{L}{\rightarrow} C$ where $c \in \mathcal{L}'_C$. Hence, the path $A \overset{L^+}{\rightsquigarrow} C$ exists and if $C < R_A$ then by Theorem 7.5 the path $C \overset{U^+}{\rightsquigarrow} R_A$. Thus, if the new fill-in of row $c \in \mathcal{L}''_B$ creates a new edge $C \overset{U}{\rightarrow} R_A$, this edge will be redundant to the path $C \overset{U^+}{\rightsquigarrow} R_A$ that already exists. In a similar fashion and by a transpose of this argument, any new L edge will be redundant to an existing L path.□

Hence only redundant edges are created due to the recovery to the LU ancestor R_A and the edge set $\mathcal{E}_T$ still correctly defines the relationships between frontal matrices.

If a failed frontal matrix does not have an LU ancestor, then its recovery matrix is defined to be the last of all frontal matrices. In this case, new edges will be created as established in the next theorem.

Theorem 7.18 (New Edges If No LU Ancestor) If a failed frontal matrix A does not have an LU ancestor, new edges will be created to its recovery frontal matrix R_A (which is defined to be the last of all frontal matrices) as follows:

1. *New L and U edges from A to R_A.*
2. *New U edges ($B \overset{U}{\rightarrow} R_A$) for all $B \in \mathcal{F}$ such that $A \overset{L^+}{\rightsquigarrow} B$.*
3. *New L edges ($B \overset{L}{\rightarrow} R_A$) for all $B \in \mathcal{F}$ such that $A \overset{U^+}{\rightsquigarrow} B$.*

PROOF: Part (a) is true since the $\mathcal{L}_A^{good}$ by $\mathcal{U}_A^{lost}$ portion of A's original pivot block has been permuted to create the U edge $A \xrightarrow{U} R_A$ and since the $\mathcal{L}_A^{lost}$ by $\mathcal{U}_A^{good}$ portion of A's original pivot block has been permuted to create the L edge $A \xrightarrow{L} R_A$

Parts (b) and (c) are true directly from part (a) and Theorem 7.5. □

In a later section on the communication paths required for lost pivot recovery, results will be established as to how $\mathcal{E}_T$ needs to be extended to provide the potential new edges as specified by Theorem 7.18 and thus be suitable for handling all inter-frontal matrix communication required for lost pivot recovery. In the interim, the edge set $\mathcal{E}_T^a$ is defined as $\mathcal{E}_T$ augmented the all of the potential edges defined by Theorem 7.18 and will be useful in proving some results of the next section.

7.1.5 Effects of Multiple Recoveries

Up to this point, all the results established have focused on the effects of a single failed frontal matrix. Nothing, however, precludes the possibility of multiple failed frontal matrices. Furthermore, several failed frontal matrices may all affect a particular frontal matrix. If the failed frontal matrices are all L predecessors or all U predecessors of the affected frontal matrix then the results thus far provide for the appropriate recovery actions. However, if the affected frontal matrix is an L ancestor of one failed frontal matrix and the U ancestor of another, it will be expanded in both row and column patterns. The theory developed thus far fails to provide direction on how to handle the new contributions in the overlap of the new recovery columns and new recovery rows.

Furthermore, as the row and column patterns of a frontal matrix's pivot block defines its relationship to predecessor frontal matrices and as this pivot block can be extended by the recovery of LU predecessors, it is necessary to complete all pivot block expansions prior to determining the effects of recovering failed frontal matrices that the current frontal matrix does not recover. Ordering the handling of recoveries by increasing topological order of their recovery matrices will facilitate both overlap resolution and the necessary prerequisite pivot block expansions.

This section provides the necessary theoretical foundation to specify how these problems should be handled. First, a relationship is established between the recovery matrices of the failed L and U ancestors. From this results, the handling of the overlap area contributions can be easily specified.

Theorem 7.19 (Paths Between Recovery Matrices) For A, B, C, R_A, and $R_B \in \mathcal{F}$ with A and B failed frontal matrices, R_A and R_B their respective recovery matrices, $C < R_A$, $C < R_B$, and the $\mathcal{E}_T^a$ edge set, if $A \overset{L^+}{\rightsquigarrow} C$ and $B \overset{U^+}{\rightsquigarrow} C$, then one and only one of $R_A = R_B$, $R_A \overset{L^+}{\rightsquigarrow} R_B$, or $R_B \overset{U^+}{\rightsquigarrow} R_A$ will be true.

PROOF: Due to the definition of the $\mathcal{E}_T^a$ edge set and that of the recovery matrices, the paths $A \overset{LU^+}{\rightsquigarrow} R_A$ and $B \overset{LU^+}{\rightsquigarrow} R_B$ exist. Thus, as $B \overset{U^+}{\rightsquigarrow} C$, $B \overset{L^+}{\rightsquigarrow} R_B$, and $C < R_B$, Theorem 7.5 provides the existence of $C \overset{L^+}{\rightsquigarrow} R_B$. Combined with the assumption of $A \overset{L^+}{\rightsquigarrow} C$, the path $A \overset{L^+}{\rightsquigarrow} R_B$ is established. But, the existence of $A \overset{U^+}{\rightsquigarrow} R_A$

is also known, so by Theorem 7.5 and the trichotomy of the ordered set $\mathcal{F}$ either (a) $R_A < R_B$ and $R_A \overset{L^+}{\rightsquigarrow} R_B$, (b) $R_B < R_A$ and $R_B \overset{U^+}{\rightsquigarrow} R_A$, or (c) $R_A = R_B$. □

Now that a relationship between the recovery matrices of the failed frontal matrices has been established by Theorem 7.19, the handling of the overlap area contributions of the multiply affected frontal matrix can be defined.

Theorem 7.20 (Overlap Area Contribution Handling) If A, $B \in \mathcal{F}$ are failed frontal matrices with R_A and R_B their respective recovery matrices and $A \overset{L^+}{\rightsquigarrow} C$ and $B \overset{U^+}{\rightsquigarrow} C$ (but no U path from A to C or L path from B to C), then the contributions made by C in the portion of the extended contribution block defined by the row of $\mathcal{L}_B^{lost}$ and the columns of $\mathcal{U}_A^{lost}$ need to be incorporated into the pivot rows/columns of either R_A or R_B whichever occurs first in the assembly DAG.

PROOF: If A and B have the same recovery matrix then there is no issue to resolve. If $R_A < R_B$, then Theorem 7.19 establishes the existence of a path from R_A to R_B and Corollaries 7.7 and 7.10 require that the overlap area contributions be absorbed by R_A. Likewise, if $R_B < R_A$, then the overlap area contributions will be absorbed by R_B. □

Corollary 7.21 Resolution of the overlap area contribution handling described in Theorem 7.20 can also be achieved by absorbing the contributions in whichever recovery frontal matrix (R_A or R_B) has the lower topological level.

This latest corollary provides a way of organizing multiple recoveries that will simplify the implementation (to be discussed in detail later).

The second major issue of this section is to define the need to complete all the pivot block expansions due to failed LU predecessors prior to accommodating any additional effects due to other failed frontal matrices.

Theorem 7.22 (Pivot Block Recoveries First) When multiple failed frontal matrices affect a particular frontal matrix, it is necessary that the effects of recoveries for which the current frontal matrix is the recovery frontal matrix be resolved before any other recoveries are addressed.

PROOF: Corollaries 7.7 and 7.10 establish the importance of an L ancestor's pivot block pattern of rows ($\mathcal{L}'$) and a U ancestor's pivot block pattern of columns ($\mathcal{U}'$) in determining how a failed frontal matrix's recovery will affect the ancestor frontal matrix. When a frontal matrix is the LU ancestor recovery matrix for a failed frontal matrix, its $\mathcal{L}'$ and $\mathcal{U}'$ patterns will be expanded to accommodate the lost pivots. Since the recoveries of other failed frontal matrices will depend on these patterns, all of the effects of failed frontal matrices that the current frontal matrix recovers must be first resolved before other recovery effects can be addressed. Furthermore, as the relationship between the failed frontal matrix and its recovery matrix is well defined a priori, use of the recovery frontal matrix's pivot block pattern is not necessary in recovering these failed frontal matrices. □

This theorem will be combined with Corollary 7.21 and the results of the next section to prescribe the ordering for resolving the effects of multiple failed frontal matrices.

Besides the possibility of distinct frontal matrices having their original anticipated pivots fail, there is the possibility that these same pivots will fail once they have been absorbed into their recovery matrix's pivot block. The next section deals with this possibility.

7.1.6 Repeated Failures

Once a failed frontal matrix's pivot block as been absorbed into a recovery matrix, there are no guarantees that the previously lost pivots will be acceptable in their new frontal matrix. Hence, the question of whether the recovery is now complete or not needs to be asked.

The answer is simple and elegant and aids significantly in both the theoretical development and implementation of lost pivot recovery. The following theorem provides this answer:

Theorem 7.23 (Recovery Completion) *Once the lost pivot block of a failed frontal matrix (A) has been absorbed into the specified recovery frontal matrix (R_A), the recovery of the failed frontal matrix is complete. Any subsequent failures of the lost pivot block of A can be handled as failures in the recovery frontal matrix.*

PROOF: For all $B \in \mathcal{F}$ such that $A \overset{L^+}{\rightsquigarrow} B$ and $B < R_A$, the rows of $\mathcal{L}'_B$ in the $\mathcal{U}_A^{lost}$ columns have become part of the pivot rows of B by Corollary 7.7. Furthermore, any other nonzero rows of the $\mathcal{U}_A^{lost}$ columns that are not contained in $\mathcal{L}'_B$ for one of the B's defined above are absorbed in $\mathcal{L}_{R_A}$ by Theorem 7.14.

In a similar fashion, all $C \in \mathcal{F}$ such that $A \overset{U^+}{\rightsquigarrow} C$ and $C < R_A$, the columns of $\mathcal{U}'_C$ in the $\mathcal{L}_A^{lost}$ rows have become part of the pivot columns of C by Corollary 7.10. Furthermore, any other nonzero columns of the $\mathcal{L}_A^{lost}$ rows that are not contained in $\mathcal{U}'_C$ for one of the C's defined above are absorbed in $\mathcal{U}_{R_A}$ by Theorem 7.14.

Thus, all the effects of A's lost pivot recovery have been absorbed into other frontal matrices and subsequent failures of the pivots in R_A can be handled as failures in R_A since no new edges from R_A have been added to $\mathcal{E}_T^a$ based on the results of Theorem 7.15. □

Corollary 7.24 *The recovery of a particular failed frontal matrix need not progress past the topological level of its recovery matrix.*

The significance of this result is that the scope of a failed frontal matrix's recovery is now limited to a well defined subset of the frontal matrices. This result can also be used to specify the ordering of recovery resolutions necessary when multiple failed frontal matrices affect a particular frontal matrix. This is done in the next section.

7.1.7 Ordering Recovery Resolutions

When multiple failed frontal matrices affect a particular frontal matrix, the ordering with which the effects of each recovery upon the current frontal matrix are

determined is important. In Theorem 7.20, the ordering of failed L and U ancestors' effects is based on the positions of the ancestors' recovery matrices. Furthermore, Theorem 7.22 requires that all recovery operations for failed frontal matrices that are recovered by the current frontal matrix be resolved before any other failed frontal matrices' recoveries. With Theorem 7.23 now also established, a composite ordering of the recovery resolutions is defined by the following theorem.

Theorem 7.25 (Ordering Recovery Resolutions) If resolution of the effects of multiple failed frontal matrices on a particular frontal matrix are ordered by ascending topological level of the failed frontal matrices' recovery matrices, the ordering criteria of Theorems 7.20 and 7.22 will be met.

PROOF: By Theorem 7.23 all recovery effects are terminated at the recovery matrix. Theorems 7.6, 7.9, and 7.5 insure paths from all affected frontal matrices to an LU ancestor recovery matrix. Inclusion of the potential edges from a failed frontal matrix to its non-LU ancestor recovery matrix (specified by Theorem 7.18) insures that any affected frontal matrix will have a path to the recovery matrix of the failed frontal matrix that affected it. Thus, any frontal matrix affected by a failed frontal matrix must be at a topological level lower than the recovery matrix of the failed frontal matrix. Combining this result with the results of Corollary 7.21 and Theorem 7.22 completes the proof. □

With this result established, all the necessary mechanisms can be built to accommodate any number of failed frontal matrices. Yet, the communication paths upon which frontal matrices will share the information necessary for these recovery mechanisms have not been fully defined. The next section completes the theoretical development by establishing the requirements for these communication paths.

7.1.8 Communication Paths

During the recovery of a failed frontal matrix, the lost pivot block, $\mathcal{L}''$ rows of the lost pivot columns, and the $\mathcal{U}''$ columns of the lost pivot rows must be passed to and absorbed by other frontal matrices. In addition, new fill-in can occur in the lost pivot rows and columns and this fill-in must also be passed to and absorbed by other frontal matrices. This section will show that the communication paths necessary for this data passing can be built from $\mathcal{E}_T$ in a simple manner. The resulting edge set is called the *control edge set* and is designated as $\mathcal{E}_C$. Results are also provided that allow for the construction of a smaller but equally sufficient control edge set. Furthermore, the assembly edge set ($\mathcal{E}_A$) is shown to be insufficient for the control edge set in the general case.

The first theorem of this section defines the manner in which $\mathcal{E}_T$ may alone be insufficient for $\mathcal{E}_C$. From this first result, clear direction is given as to how $\mathcal{E}_T$ must be augmented to create $\mathcal{E}_C$, which is explicitly defined in the second theorem.

Theorem 7.26 ($\mathcal{E}_T$ Path Failures) $\mathcal{E}_T$ will fail to contain a path from a failed frontal matrix A to its recovery matrix R_A only if there exists a frontal matrix B such that $A \leq B < R_A$ and there are no edges from B in $\mathcal{E}_T$.

PROOF: If A has an LU ancestor as its recovery matrix, then by definition there will be a path from A to R_A in $\mathcal{E}_T$. Hence, consideration can be limited to frontal matrices A that have no LU ancestor and use the default recovery matrix as defined earlier.

The proof thus assumes no path exists from A to R_A but that also there are no frontal matrices B such that $A \leq B < R_A$ and B has no outward going edges in $\mathcal{E}_T$. A contradiction is thus reached indicating one of these two assumptions must be false.

The argument proceeds inductively starting with the frontal matrix A. Since A must have an outgoing edge in $\mathcal{E}_T$ by assumption, there will exist a frontal matrix B such that $A \rightarrow B \in \mathcal{E}_T$. If $B = R_A$ then a path exists from A to R_A, which contradicts the assumption that such a path does not exist. Otherwise, the definition of the default frontal matrix R_A insures that $B < R_A$ so the argument can be applied recursively starting at B. Since the potential path lengths between A and R_A are bounded, the induction can progress for only a finite number of steps and eventually one of the two assumptions must be violated. □

If $\mathcal{E}_T$ is augmented by adding edges from any frontal matrix with no outgoing edges to the default recovery frontal matrix (last of all frontal matrices) then the resulting edge set (called $\mathcal{E}_C$) will be sufficient to accommodate all communication required by the lost pivot recovery mechanisms. The following theorem formalizes this result.

Theorem 7.27 ($\mathcal{E}_C$ Built From $\mathcal{E}_T$) *$\mathcal{E}_C$ (as built in the discussion of the previous paragraph) is sufficient to accommodate all necessary communication for lost pivot recovery.*

PROOF: Theorem 7.26 and the definition of $\mathcal{E}_C$ insure that there are paths from each frontal matrix to its recovery frontal matrix. Theorems 7.12, 7.13, 7.17, and 7.18 insure that all recovery effects are then limited to frontal matrices on paths in $\mathcal{E}_C$. □

This definition of $\mathcal{E}_C$ does, however, provide one serious drawback. Specifically, the $\mathcal{E}_T$ edge set (upon which this definition of $\mathcal{E}_C$ is based) is very large and thus will impose serious performance impacts. The next theorem establishes the sufficiency of a much smaller edge set to be used as $\mathcal{E}_C$.

Theorem 7.28 ($\mathcal{E}_C$ Based On A Transitive Reduction of $\mathcal{E}_T$) *If a transitive reduction of $\mathcal{E}_T$ is used as the base edge set for the construction of $\mathcal{E}_C$ then the resulting edge set will still be sufficient for all necessary lost pivot recovery.*

PROOF: Follows directly from the definition of a transitive reduction. That is, if there is an edge $A \rightarrow B$ in $\mathcal{E}_T$, then there is also a path from A to B in the transitive reduction of $\mathcal{E}_T$. When combined with the results of Theorem 7.27, this completes the proof.□

In general, the transitive reduction of an edge set is costly to compute. However, some of the earlier results of this chapter can be used to significantly trim down the number of edges that must be considered in the transitive reduction process.

Theorem 7.29 (Transitive Edges In $\mathcal{E}_T$) *If B is the lowest frontal matrix in the ordered set $\mathcal{F}$ such that $A \xrightarrow{L} B$ and C is the lowest frontal matrix such that $A \xrightarrow{U} C$, then for $B \leq C$ only L edges $A \xrightarrow{L} D$ such that $D < C$ need to be considered in the transitive reduction of edges from A and for $C \leq B$ only U edges $A \xrightarrow{U} D$ such that $D < B$ need to be considered in the transitive reduction of edges from A.*

PROOF: Consider the case of $B \leq C$. If $B = C$ then B is a direct LU parent of A and by Theorems 7.1 and 7.3, any other edges from A will have corresponding edges from B and would thus be eliminated by a transitive reduction. If $B < C$, then by Theorem 7.5 there exists an edge $B \xrightarrow{U} C$ and the $A \xrightarrow{U} C$ edge is eliminated as transitive. This result similarly holds for all other U edges from A. For any L edge $A \xrightarrow{L} D$ such that $D \geq C$, if $D = C$ then the path from A to C per the earlier argument implies that $A \xrightarrow{L} C$ can be eliminated. Furthermore, if $D > C$ then $A \xrightarrow{U} C$ edge insures the existence of an L path from C to D via Theorem 7.5 and the L edge of A to D can be eliminated from consideration.

The transpose of the argument above establishes the result if $B > C$. □

Thus one way to build a suitable control edge set is from a transitive reduction of the $\mathcal{E}_T$ edge set. However, it may also be possible to construct a suitable edge set from the assembly edge set ($\mathcal{E}_A$) provided in the definition of the assembly DAG. The next theorem shows that $\mathcal{E}_A$ is not suitable for control edge set construction in the general case.

Theorem 7.30 (Absence of Edges In $\mathcal{E}_C$) *There can exist edges in $\mathcal{E}_T$ for which there is no corresponding path in $\mathcal{E}_A$.*

PROOF: Consider the matrix pattern shown below

$$A = \begin{pmatrix} X & X \\ & X \end{pmatrix}.$$

Two frontal matrices can be defined within A with the first consisting of row 1 and columns 1 and 2 and the second consisting of just row 2 and column 2. Clearly, the $a_{1,2}$ nonzero entry establishes the existence of a U edge from the first to the second frontal matrix in $\mathcal{E}_T$. However, no contribution is passed on this edge as the $\mathcal{L}''$ pattern of the first frontal matrix is empty. Thus, there will be no edge from the first to the second frontal matrix in $\mathcal{E}_A$.□

Corollary 7.31 *$\mathcal{E}_A$ is not sufficient to handle the communication required by lost pivot recovery, in the general case.*

By way of conclusions, this section establishes that all necessary communication required for lost pivot recovery can be facilitated by a control edge set that can be completely defined prior to the factorization. This edge set however must be constructed from a transitive reduction of $\mathcal{E}_T$ and preprocessing is required for both the creation of $\mathcal{E}_T$ and the computation of its transitive reduction. Furthermore, some

additional edges may be necessary to complete the construction of $\mathcal{E}_C$. Unfortunately, the assembly edge set ($\mathcal{E}_A$), that is provided as input via the assembly DAG, is insufficient for the control edge set in this general setting.

However, in the next section, results will be established that show how an assumption of block upper triangular form will allow the control edge set to be defined as exactly the given assembly edge set. Thus, the required preprocessing to define sufficient communication paths for lost pivot recovery is essentially eliminated.

7.1.9 Consequences of Block Triangular Form

In the previous chapter some of the benefits of a block triangular form of the matrix to be factored were discussed. Specifically, only the portion of the matrix within the diagonal blocks needs to be factored (which also limits fill-in) and each diagonal block provides an independent sub-DAG within the assembly DAG, which promotes parallelism. In this section, further benefits of block triangular form will be realized that greatly simplify determination of the communication paths required for lost pivot recovery.

These additional benefits all result from a single but powerful property of the frontal matrices in a block upper triangular form of the matrix, which is established in the following theorem.

Theorem 7.32 (Impacts On Frontal Matrices of Block Triangular Form) *If the matrix has been permuted to block upper triangular form, then the frontal matrices within each diagonal block i (defined by the set of frontal matrices $\mathcal{F}_i \subseteq \mathcal{F}$) except for the last frontal matrix in the block will have $\mathcal{L}'' \neq \emptyset$ and $\mathcal{U}'' \neq \emptyset$.*

PROOF: Suppose a frontal matrix $C \in \mathcal{F}_i$ exists such that C is not the last frontal matrix in $\mathcal{F}_i$ but $\mathcal{U}''_C = \emptyset$. Then, the argument that follows will show that permutations are possible that will divide $\mathcal{F}_i$ into two distinct diagonal blocks. This implies that the original matrix was not in block upper triangular form as this form has been proven to be unique for nonsingular matrices up to permutations strictly within the blocks and reorderings of the blocks on the diagonal [48]. Furthermore, the transpose of the argument that follows establishes the same result if $\mathcal{L}''_C = \emptyset$.

To aid in the understanding of the argument that follows an illustration of a frontal matrix pattern is provided below for which C has $\mathcal{U}''_C = \emptyset$ and its permuted form after the permutations to be described have been applied.

$$
\begin{array}{c}
\mathcal{L}'_{A_1} \\ \mathcal{L}'_B \\ \mathcal{L}'_{A_2} \\ \mathcal{L}'_C \\ \mathcal{L}'_{D_1} \\ \mathcal{L}'_{D_2}
\end{array}
\begin{array}{c}
\begin{array}{cccccc} \mathcal{U}'_{A_1} & \mathcal{U}'_B & \mathcal{U}'_{A_2} & \mathcal{U}'_C & \mathcal{U}'_{D_1} & \mathcal{U}'_{D_2} \end{array} \\
\begin{pmatrix} x & & x & x & & \\ x & x & x & x & x & x \\ x & & x & x & & \\ x & & x & x & & \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{pmatrix}
\end{array}
\Rightarrow
\begin{array}{c}
\begin{array}{cccccc} \mathcal{L}'_B & \mathcal{L}'_{D_1} & \mathcal{L}'_{D_2} & \mathcal{L}'_{A_1} & \mathcal{L}'_{A_2} & \mathcal{L}'_C \end{array} \\
\begin{pmatrix} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ & & & x & x & x \\ & & & x & x & x \\ & & & x & x & x \end{pmatrix}
\end{array}
$$

Suppose that $\mathcal{U}''_C = \emptyset$. This implies that all of the entries in the rows $\mathcal{L}'_C$ and the columns of $\bigcup_{(D \in \mathcal{F}_i) \wedge (D > C)} \mathcal{U}'_D$ are zero. Furthermore, for all $A \in \mathcal{F}_i$ such that $A \overset{L^+}{\rightsquigarrow} C$, the entries in the $\mathcal{L}'_A$ rows and $\bigcup_{(D \in \mathcal{F}_i) \wedge (D > C)} \mathcal{U}'_D$ columns are zero. Otherwise, the

entries in these columns and the $\mathcal{L}'_C$ rows would be nonzero due to fill-in per the results of Theorem 7.1.

If there is no L path from a frontal matrix $B \in \mathcal{F}_i$ to C, then there must be zero entries in the columns of $\mathcal{U}'_B$ and the rows in $\mathcal{L}'_C$ and $\mathcal{L}'_A$ for all $A \in \mathcal{F}_i$ such that $A \overset{L^+}{\rightsquigarrow} C$. Otherwise, if $A < B$, the edge $A \overset{U}{\rightarrow} B$ would exist and by Theorem 7.5 the $B \overset{L}{\rightarrow} C$ would exist, which violates the assumptions. If $A > B$, then a nonzero entry in the $\mathcal{L}'_A$ rows and the $\mathcal{U}'_B$ columns would imply the edge $B \overset{L}{\rightarrow} A$ and thus the path $B \overset{L^+}{\rightsquigarrow} C$, which also violates the assumptions.

Thus, for C and each L predecessor of C, the entries in their pivot rows and the columns

$$\mathcal{U}(\mathcal{F}_i^1) = \left(\bigcup_{B \in \mathcal{F}_i \text{ such that } (B<C) \wedge (B \overset{L^+}{\rightsquigarrow} C \notin \mathcal{E}_T)} \mathcal{U}'_B \right) \bigcup \left(\bigcup_{(D \in \mathcal{F}_i) \wedge (D>C)} \mathcal{U}'_D \right)$$

are zero. (Notice that $\mathcal{U}(\mathcal{F}_i^1)$ includes columns that occur prior to C in the current pivot ordering). Hence, the $\mathcal{U}(\mathcal{F}_i^1)$ columns can be permuted to the front of the $\mathcal{F}_i$ block and the rows designated by

$$\mathcal{L}(\mathcal{F}_i^2) = \left(\bigcup_{A \in \mathcal{F}_i \text{ such that } A \overset{L^+}{\rightsquigarrow} C} \mathcal{L}'_A \right) \bigcup \mathcal{L}'_C$$

can be permuted to the bottom of the $\mathcal{F}_i$ block. Hence, a new independent diagonal block (referred to as $\mathcal{F}_i^2$) has been created and is defined by the rows $\mathcal{L}(\mathcal{F}_i^2)$ and the columns

$$\mathcal{U}(\mathcal{F}_i^2) = \left(\bigcup_{A \in \mathcal{F}_i \text{ such that } A \overset{L^+}{\rightsquigarrow} C} \mathcal{U}'_A \right) \bigcup \mathcal{U}'_C .$$

The independence of $\mathcal{F}_i^2$ is verified by the observation that the entries in the $\mathcal{U}(\mathcal{F}_i^1)$ columns and the $\mathcal{L}(\mathcal{F}_i^2)$ rows are all zero and that

$$\mathcal{U}(\mathcal{F}_i) - \mathcal{U}(\mathcal{F}_i^1) = \mathcal{U}(\mathcal{F}_i^2).$$

The breaking of $\mathcal{F}_i$ into two independent diagonal blocks implies that the matrix was not in block upper triangular form. □

A more elegant proof of Theorem 7.32 is possible using the results of Rose and Tarjan [128, 127] but a different graph formulation of the matrix would be required.

The results of Theorem 7.32 are very powerful in that they provide for a number of simplifications for lost pivot recovery, which are defined in the following theorems.

Theorem 7.33 ($\mathcal{E}_T$ Sufficient For Lost Pivot Recovery) Assuming block upper triangular form, no edges will need to be added to $\mathcal{E}_T$ for handling of lost pivot recoveries.

PROOF: Follows directly from Theorems 7.26 and 7.32. □

Theorem 7.34 ($\mathcal{E}_A$ Sufficient For Lost Pivot Recovery) Assuming block upper triangular form, there will be no edge in $\mathcal{E}_T$ that is not also available via a path in $\mathcal{E}_A$ and thus $\mathcal{E}_A$ can fully support lost pivot recovery.

PROOF: The existance of edges in $\mathcal{E}_T$ that do not have corresponding paths in $\mathcal{E}_A$ as illustrated in Theorem 7.30 results from an empty contribution block in the earlier frontal matrix. Theorem 7.32 precludes the possibility of an empty contribution block and thus the situation of Theorem 7.30 cannot occur. □

Theorem 7.35 (Always An LU Ancestor) Assuming block upper triangular form, every frontal matrix will have either an LU ancestor or no ancestors at all.

PROOF: Assume that there exists a frontal matrix A with ancestors but no LU ancestor. By Theorem 7.32, the frontal matrix must have edges to both L and U parents in $\mathcal{E}_T$. Likewise, each L parent must have L edges to later frontal matrices in $\mathcal{F}_i$ unless the parent is the last frontal matrix in the block. As each $\mathcal{F}_i$ has only a finite number of frontal matrices in it, there will be an L path from A to the last frontal matrix in the block. In the same fashion, an argument based on the U edges that follows the same logic will establish the existence of a U path from A to the last frontal matrix in the block. Hence, there are both L and U paths from A to the last frontal matrix in the block and thus this frontal matrix is an LU ancestor of A. □

In summary, block upper triangular form has several very significant impacts on the mechanisms necessary for lost pivot recovery. First, $\mathcal{E}_T$ (or a transitively reduced $\mathcal{E}_T$) can be used without modification to facilitate all inter-frontal communication necessary for lost pivot recovery. More importantly, $\mathcal{E}_A$ can be used for this same purpose and a valid $\mathcal{E}_A$ is provided as an input to the refactorization routine and does not require computational overhead for its definition. Finally, the need for a default recovery matrix is completely eliminated by permuting to block upper triangular form.

7.1.10 Summary of the Lost Pivot Recovery Theory

The theory development of this section completely specifies the mechanisms necessary for the recovery of lost pivots. Fill-in due to the recovery process in the lost pivot columns results only from the L ancestors of the failed frontal matrix, while fill-in in the lost pivot rows is the result of U ancestors only. Hence, augmentation of the L ancestors by the lost pivot columns and the U ancestors by the lost pivot rows will accommodate any resulting new contributions. Furthermore, such augmentation need only occur until the failed frontal matrix's recovery matrix has been encountered at which point the recovery operation is complete.

These mechanism extend easily to handling multiple pivot failures with just a few minor refinements. Specifically, each frontal matrix must account for the effects of failed frontal matrices for which it is the recovery matrix before handling the effects of any other recoveries. Furthermore, if a frontal matrix is the L ancestor of one failed frontal matrix and the U ancestor of another, care must be taken in handling the new contributions in the overlap area of its extended column and row patterns. A specific ordering strategy was defined to aid in satisfying these requirements.

The communication paths necessary for lost pivot recovery are easily established, in the general case, by a control edge set ($\mathcal{E}_C$) built by augmenting a transitive reduction of the $\mathcal{E}_T$ edge set, since the assembly edge set $\mathcal{E}_A$ is insufficient for this purpose in general. However, if a permutation to block upper triangular form is first accomplished, $\mathcal{E}_A$ is sufficient for handling the communication required by lost pivot recovery. (This permutation to block upper triangular form takes place as preprocessing during the analyse-factorize operation).

With this theoretical groundwork established, the necessary lost pivot recovery mechanisms can be implemented. The next section deals with this implementation which is built to execute in a parallel distributed memory environment. As such implementations are statically scheduled, the ability to fully define the necessary communication paths prior to refactorization that is provided by this theory is especially important.

7.2 Implementation Specifics

In the previous section, the theoretical development of lost pivot recovery established what mechanisms are needed. The next step is to define how these mechanisms can be implemented within the context of a distributed memory, parallel environment. This section addresses these practical implementation issues as they were resolved by extending the host preprocessing and parallel refactor code described in Chapter 6. While this discussion focuses on the implementation done on the nCUBE 2 distributed memory system, many of issues are also relevant to sequential, parallel shared memory, and other distributed memory implementations.

This section starts with a discussion of some additional synchronization requirements imposed by the lost pivot recovery mechanisms. This is followed by a discussion of the additional host preprocessing needed. Then an overview of the frontal matrix task processing as revised for lost pivot recovery is provided. Specific implementation details follow on the composition and format of the recovery structures, the handling of recovery structures, and the creation and forwarding of recovery structures. At the end of this section, implementation issues specific to sequential and parallel shared memory implementations will be briefly discussed.

7.2.1 Lost Pivot Recovery Synchronization

As discussed in the previous section on the theoretical development, a particular frontal matrix can be the L, U, or LU ancestor of any number of failed frontal matrices. The effects of failed predecessors can cause modifications to each component row and column pattern of the effected frontal matrix, which will impact its dimensions, the amount of memory required for frontal matrix storage, and how original values and contributions are mapped into the frontal matrix storage area. Hence, there is a need to determine all of the effects on the particular frontal matrix prior to allocating its storage and commencing the assembly of original values and contributions. Furthermore, per Theorems 7.20, 7.22, and 7.25, the process of determining these effects must follow a specified ordering on the recoveries. In addition, all the processors involved in the partial factorization of the frontal matrix must agree upon the resolution of the effects of all failed predecessors.

As a result, two levels of additional synchronization are needed. At the first level, a frontal matrix's processing should not begin until all the information required for recovering failed predecessors has been received and the corresponding effects resolved. This is best handled in a centralized manner, so the processing node 0 of the frontal matrix's assigned subcube (here after called the *root processor*) will receive all the incoming recovery information and resolve all the effects. The second level of synchronization then calls for the other processors in the frontal matrix's subcube to wait for the root processor to broadcast the results of the resolution process.

The broadcast required of the second level of synchronization is straight forward. However, the communication patterns required of the first level of synchronization require further discussion. As presented in the theoretical development, an augmented version of the true edge set $\mathcal{E}_T$ (called the control edge set, $\mathcal{E}_C$) is sufficient for these communication in the general case and the true edge set itself sufficient for $\mathcal{E}_C$ if block upper triangular form is provided. However, these edge sets are large and messages along the edges are needed for synchronization even if no failures occur, which would result in excessive overheads. (Note that the full edge sets would not really be needed as recoveries are complete upon reaching the recovery matrix, per Theorem 7.23, and any edges to later frontal matrices could and should be eliminated).

The alternative is to use appropriate transitive reductions of the base edge sets for the control edges and provide the necessary communication via paths instead of direct edges, which would significantly reduce the required message passing. Theorems 7.27 and 7.28 define the adequacy of such a control edge set in the general case. With an assumption of block upper triangular form, Theorems 7.33 and 7.28 show the adequacy of a transitive reduction of just the true edges and Theorem 7.34 justifies the use of the assembly edge set (which is typically already partially transitively reduced) for the control edge set.

The use of a path-based control edge set is further justified by the observation that the recovery operations of an L (or U) ancestor can produce recovery contributions that must be assembled into other frontal matrices on an L (or U) path from the failed frontal matrix. Thus, the use of a path-based control edge set is not only more efficient but also required as a necessary criteria.

The conclusion is to use one of the transitively reduced edge sets for the control edges and have a path based recovery mechanism. But this leaves open the question of which transitively reduced edge set to use. If block upper triangular form is provided, the assembly edge set would be easy to use as it is provided as input to the problem. Furthermore, these $\mathcal{E}_A$ edges are already used for passing messages that contain the normal inter-frontal contributions. The recovery information could be piggy-backed onto these existing messages to further reduce the lost pivot recovery communication overhead. However, the recovery information needs to be passed between the root processors of the communicating frontal matrices and the contribution messages are passed between processors in the respective frontal matrix's subcube based on data locality. Hence, there may not be a contribution message going between the root processors and a new message must be added anyway. Furthermore, this piggy-backing only applies if block upper triangular form is present.

The alternative (which is also what has been implemented) is to maintain the control edges and the contribution passing assembly edges as distinct edge sets. While increasing the required message passing, this approach has the benefits of being able to handle the most general case (where block upper triangular form is not assumed) and allows outgoing control messages to be released before forwarding contributions so remote processors can get an earlier start on the resolution of received recovery structures. With this approach, a transitive reduction of the possibly augmented true edge set provides for the smallest control edge set and covers the general case, which justifies its selection for implementation. Table 7-1 provides a comparison of the sizes of the original true edge set (original $\mathcal{E}_T$), a transitively reduced true edge set (reduced $\mathcal{E}_T$), and the assembly edges ($\mathcal{E}_A$) provided by UMFPACK for five test matrices that have been permuted to block upper triangular form.

Table 7-1. Edge Count Comparison

MATRIX	ORIGINAL $\mathcal{E}_T$	REDUCED $\mathcal{E}_T$	UMFPACK $\mathcal{E}_A$
GEMAT11	7993	793	834
EXTR1	10180	1588	2127
RDIST1	17479	315	406
RDIST2	22917	857	1043
RDIST3A	13632	325	393

With the additional synchronization requirements for lost pivot recovery specified and the edge set defined that will satisfy these requirements, a discussion of how this edge set is created and the other additions to the host preprocessing is appropriate. This will be followed by a discussion of the lost pivot recovery processing that occurs within the parallel refactorization code.

7.2.2 Host Preprocessing

There are two additional requirements of the host preprocessing code imposed by the inclusion of lost pivot recovery. First, the control edges must be defined and associated with the individual frontal matrix descriptions upon which they are incident. Second, the recovery matrix and its level must be determined for each frontal matrix.

In this implementation, the control edges will be created from the transitive reduction of the true edge set and augmented, if necessary, with additional edges to the last frontal matrix from any intermediate frontal matrix that has no outgoing edges in $\mathcal{E}_T$. Theorem 7.29 provides a very useful result that can significantly improve the performance of the transitive reduction process. Specifically, all edges to a frontal matrix numbered higher than both the first L parent and the first U parent (together with the higher of these two parents) can be immediately eliminated as transitive edges. This significantly reduced set of parents is the initial set of directly reachable parents in the reduced $\mathcal{E}_T$. Each frontal matrix in this set is then taken in ascending order and its outgoing edges in $\mathcal{E}_T$ checked. If an edge exists to another

frontal matrix in the directly reachable set, the edge to this frontal matrix from the original frontal matrix is eliminated and the destination frontal matrix removed from the directly reachable set. The edges remaining after this process are specified as outgoing control edges with the description of the original frontal matrix and incoming control edges with each destination frontal matrix's description. Counts of incoming and outgoing control messages are also maintained for each frontal matrix. These counts and the IDs of the frontal matrices connected by these edges are passed to the parallel processing nodes as part of the frontal matrix descriptions.

In addition to specification of the control edges, host preprocessing is also extended to include determination of the recovery matrices and their topological levels for each frontal matrix. This is done for each frontal matrix by first specifying the last frontal matrix in the encompassing diagonal block as the initial recovery matrix. (If block upper triangular form is not presumed, there will be only one such block and the appropriate semantics will be preserved). The direct L and U edges from the frontal matrix are then scanned to determine if a direct LU parent exists. If so, the recovery matrix is updated to be the LU parent. The remaining open question is whether or not there are LU ancestors that occur before the current recovery matrix. This can be answered by computing a partial transitive closure. Specifically, all of the frontal matrices that are reachable via L paths and have IDs less than the current recovery matrix will be determined and called the L-reachable set. Likewise a U-reachable set will be determined using U paths. The lowest frontal matrix in the intersection of these two sets, if one exists, will be the first LU ancestor and become the recovery matrix. The L-reachable set is initialized with the direct L parents whose IDs are less than that of the current recovery matrix. Each frontal matrix placed in the L-reachable set will have its direct L parents scanned and those with IDs less than the current recovery matrix will also be added (if they are not already present). This process continues until all frontal matrices in the L-reachable set have been checked. The same process is done to build the U-reachable set. Ascending order scans of the two sets are used to find the first LU ancestor with the scan terminated upon either finding the first LU ancestor or exhausting one of the sets.

In earlier processing, the topological level of each frontal matrix was determined via a topological sort. Thus, once the recovery matrix has been found, its topological level is readily available. Both the recovery matrix ID and level will be added to the frontal matrix description and forwarded to the parallel processing nodes for use in accomplishing any necessary recovery of the frontal matrix.

Both the determination of the control edge set and of the recovery matrices and levels are done within the SCHEDULE_REFACTOR routine of the host preprocessing that was described in Chapter 6. With this preprocessing done, the parallel refactorization is launched. The next discussion will overview how the parallel refactorization frontal matrix tasks will function with the addition of lost pivot recovery.

7.2.3 Enhanced Task Processing

Incorporation of the mechanisms for lost pivot recovery into the frontal matrix task processing will require significant amounts of new processing to be performed by the root processor of a frontal matrix's assigned subcube together with additional

processing that is unique to non-root processors and some that is required of all subcube processors. A sequential ordering of the enhanced frontal matrix task processing is provided below with a prefix specification stating which of the frontal matrix's subcube processors will perform each major processing step. A definition of the recovery structure used to facilitate the recovery of a particular failed frontal matrix and the remaining details of the lost pivot recovery mechanisms will be described later.

1. (On the root processor): Accept all the expected control messages. Discard any recovery structures that do not affect the current frontal matrix and are recovered at or below the current topological level. Combine multiple instances of particular recovery structures to include marking all absorbed rows and columns and eliminating redundant recovery contributions. Determine the effects of each combined recovery structure on the frontal matrix and broadcast the results to the frontal matrix's assigned subcube.

2. (On the non-root processors): Wait for and accept the broadcast from the root processor that contains the resolution of all recovery effects.

3. (On all subcube processors): Determine how the updates to the frontal matrix required by recoveries impact the locally held portions of the frontal matrix. Allocate the appropriate frontal matrix storage and assemble in all the recovery contributions.

4. (On all subcube processors): Assemble in the original values and normal contributions from the other frontal matrices. If the pivot block of the current frontal matrix has been expanded (i.e., this is the recovery matrix of a failed frontal matrix), a remapping of the entries' location in the local frontal matrix storage area will be required.

5. (On all subcube processors): Perform the partial dense factorization of the frontal matrix performing lost pivot recovery within the pivot block as necessary. Record all required permutations within the lost pivot block. This processing is done by the P4 partial dense factorization kernel described in Chapter 5.

6. (On the non-root processors): Pack the locally held portions of any lost pivot rows and columns and any row or column extensions of the contribution block required by lost pivot recovery into a message and send the message to the root processor. (If there are no failed pivots and no row or columns extensions, this message passing need not and is not done).

7. (On the root processor): Receive the lost pivot rows and columns and any contribution block row or columns extensions from the non-root processors, as necessary. Update any received recovery structures with the effects produced by the current frontal matrix. If pivots were lost in the current frontal matrix, create a new recovery structure. Forward all recovery structures that are recovered at a higher topological level.

8. (On all subcube processors): Forward normal contributions via the established contribution message passing mechanisms.

9. (On all subcube processors): Allocate space for the locally held portions of the LU factors from the local LU buffer and save the local LU factors.

Within the original routines of the parallel refactor code, first described in Chapter 6, little has changed. The routines for assembling original values and contributions and for forwarding contributions were slightly modified to include remapping of the local row and column indices if the pivot block of the current frontal matrix has grown. The routine that assembles contributions must also remap the local indices of a local source frontal matrix if that frontal matrix's pivot block has grown. The partial dense factorization kernel is simply replaced by the P4 kernel described in Chapter 5.

Finally, the routine that saves the local LU factors is changed as the individual LU factor storage for a particular processor's portion of a frontal matrix's LU factors is no longer distinctly allocated from the processor's heap. Instead, a single buffer area has been preallocated for the storage of all the LU factors that are held locally by the processor. This single LU buffer area is expanded by a linear multiple of the original LU buffer requirement plus a fixed expansion amount to accommodate growth in the LU factors due to lost pivot recovery. Both the linear multiple and fixed expansion factors are compile time parameters. (They can be easily changed to run time parameters). The space required for the LU factors of a particular frontal matrix is simply and efficiently allocated consecutively from this single LU buffer just prior to the saving of the LU factors. Furthermore, this LU buffer can be reset for the saving of the LU factors for a new matrix in the sequence with a simple reinitialization of a next allocation pointer.

In terms of data structures, the only real alterations of existing structures are the reorganization of the LU factor storage just described and some additional fields added to the frontal matrix descriptions. Specifically the frontal matrix descriptions have new fields for the identification of the recovery frontal matrix and its level, for the numbers of control child (incoming) and parent (outgoing) edges, for the lists of control child edge sources and of control parent edge destinations, and some for the preservation of the original contents of fields that can be modified during lost pivot recovery. These preservation fields are needed to restore the modifiable fields to their original contents for subsequent refactorizations. The fields preserved in this manner are the global and local dimensions: *m, n, g, my_g_cnt,* and *my_cols*; the global and local patterns: *row_ids, col_ids,* and *my_matrix_cols*; and the additional vector fields: *turns, cube_assigns,* and *local2frontal.* (All of these fields were defined in detail in Chapter 6). There are some other fields modified by lost pivot recovery but they can be easily regenerated from other information in the frontal matrix description.

In addition to the changes required in the existing data structures, lost pivot recovery implementation will also require some new data structures, which are mostly based on the concept of a recovery structure.

7.2.4 Recovery Structures

A *recovery structure* is a data object that holds the information necessary to recover a failed frontal matrix. As there can be multiple control paths in $\mathcal{E}_C$ from a failed frontal matrix to its recovery matrix, multiple instances of the corresponding recovery structure may exist and need to be joined into a combined recovery structure. Thus, there are two varieties of recovery structures: (1) the recovery structures as sent along the control edges and (2) the combined recovery structures resulting from the joining of multiple instances of the first type of recovery structure. Both varieties will include some scalar data, the row and column pattern data defining the lost pivot rows and columns and their respective nonzero patterns, the values of the lost pivot block, and the nonpivot block values in the lost pivot rows and columns, which can be augmented by additional contributions due to L and U ancestors respectively.

The scalar information includes the dimensions of the lost pivot block, the number of nonpivotal rows in the lost pivot columns, the number of nonpivotal columns in the lost pivot rows, the recovery matrix ID and level, and the numbers of contributions held for the lost pivot rows and columns, respectively.

The pattern data of the recovery structure consists of a vector of row IDs and a vector of column IDs. The pivot block rows and columns appear at the beginning of the respective vectors. The nonpivot block row and column patterns can grow as new contributions are made by L and U ancestors of the failed frontal matrix. Also, the nonpivot block rows and columns are absorbed by L and U ancestors as defined by Corollaries 7.7 and 7.10. When this occurs, the corresponding pattern entries are marked (by negation). If a row or column is marked in any one instance, it must also be marked in the combined recovery structure.

Theorem 7.16 insures that no new contributions will be made to the lost pivot block during recovery, so the recovery structures can simply hold the lost pivot block values in a column-major storage format without additional indexing information.

The situation is more complicated for entries of the lost pivot rows and columns that lie outside of the lost pivot block. Theorems 7.6 and 7.9 and Corollaries 7.8 and 7.11 show that both new and multiple contributions can occur within the nonpivot block portions of the lost pivot rows and columns. Furthermore, the multiple instances of a recovery structure that pass on different control paths may pick up distinct new contributions. When these instances are combined, duplicate contributions must be eliminated and all new and distinct contributions retained. Hence, both the original entry value and all subsequent recovery contributions must be held as 4-tuples consisting of row, column, source frontal matrix ID, and value. Such 4-tuples uniquely define each original entry and contribution, thus allowing the necessary combining operations. In addition, these 4-tuples can be ordered to expedite the combining process. Specifically, the 4-tuples of the rows of lost pivot columns are ordered first by ascending column and within each column by ascending source frontal matrix ID. Likewise, the 4-tuples of the columns of the lost pivot rows are ordered first by ascending row and then by ascending source frontal matrix ID within rows. The specifics of the combining process will be described later.

In a combined recovery structure, the nonpivot block contribution 4-tuples are held in a large buffer area with all the entries for a particular row of the lost pivot columns and for a particular column of the lost pivot rows organized into a singly linked list. Two additional vector fields in the combined recovery structure contain the pointers to the first element in each such list. These pointer vectors are ordered exactly as the corresponding pattern data to facilitate the access to each list. In the other variety of recovery structure, the nonpivot block contribution 4-tuples for a particular row of the lost pivot columns or a particular column of the lost pivot rows are held sequentially in the order prescribed above.

In addition to the recovery structures, lost pivot recovery requires some other significant data objects. Specifically, there are integer vectors for the accumulation of row and column patterns during the combining of multiple recovery structure instances and during the resolution of effects on a frontal matrix by multiple failed frontal matrices. An integer accumulation vector is also required for the assignment of columns to processors as new columns are added to a frontal matrix by failed L and LU predecessors. Integer vectors are also required for the remapping of contributions and original values when the pivot block of a frontal matrix grows. An integer vector is also created to globally specify the root processors for each frontal matrix and will be used for passing control messages with recovery structures between frontal matrices.

Finally, a distinct heap of composite data structures is required for the organization of the multiple distinct recovery structures and multiple instances of an individual recovery structure received by a particular frontal matrix. The composite data object has three integer fields and two pointer fields. These data objects are used to build a multilevel linked list structure that organizes the received recovery structures. The highest level list has one entry for each recovery structure received from a distinct failed frontal matrix. This list is ordered by ascending recovery level of the failed frontal matrices and provides for the resolution ordering required by Theorem 7.25. Each entry in this high level list uses one of its pointers to point to a secondary list, which contains one entry for each instance of the corresponding failed frontal matrix's recovery structure. A pointer in these secondary list elements will point to the specific recovery structure as it resides in the received control message. This avoids having to copy the received recovery structures. Figure 7-4 illustrates an example of this multi-level list structure before the multiple instances have been combined.

When the multiple instances of a particular recovery structure are joined into a combined recovery structure, the high level list element for that recovery structure has its pointer updated to point to the combined recovery structure. Figure 7-5 illustrates the state of the multi-level list structure after multiple instances have been combined.

In addition to the new data structures required by lost pivot recovery, three new types of messages are necessary. Control messages provide for the necessary additional synchronization as discussed earlier in this section and pass the recovery structures between frontal matrices. These control messages include the destination

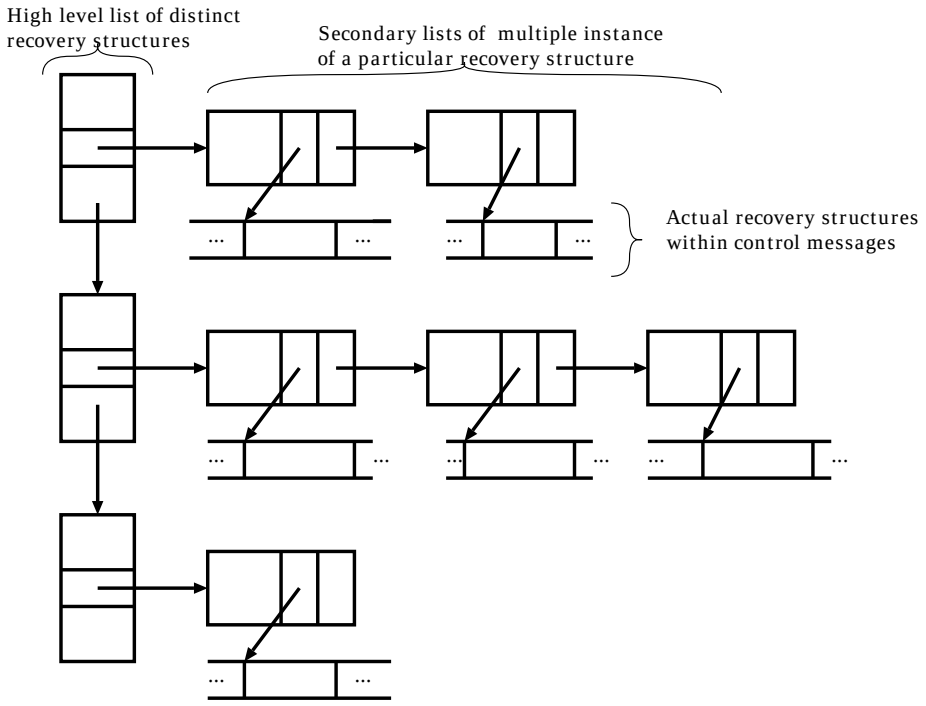

Figure 7-4. Multi-Level List Structure Prior to Combining

and source frontal matrix IDs, the number of recovery structures contained in the message, and the number of receivers of the message followed by the individual recovery structures. The number of receivers is necessary as only a single instance is created of a control message to be sent to multiple frontal matrices that all have root processors that are the same as the sending processor. The receiving frontal matrices have a pointer set to the single instance of the control message. The number of receivers field is decremented as each receiver accepts the message. The storage for the message is released when the counter goes to zero.

Control messages sent to other processors are given a message type out of a specific range, which correlates (via a modulus function) to the ID of the destination frontal matrix. (This is done exactly as for the contribution messages described in Chapter 6 except that a distinct range of message types is used). This allows the receiving frontal matrix to only accept the desired control messages. A check of the destination frontal ID in the received message insures its proper reception and allows those messages that are destined for other local frontal matrices to be queued up to a pre-read list of control messages for the proper destination frontal matrix.

The root processor broadcasts a recovery update message to the rest of the frontal matrix's assigned subcube. This message contains the new numbers of rows, columns, and pivots, together will the new row and column patterns, column to processor assignments, the number of recovery contributions, and the actual recovery contributions in a triplet format. The row, column, and value triplet format is sufficient as all distinct contributions to an entry absorbed by the current frontal matrix have already been combined by the root processor.

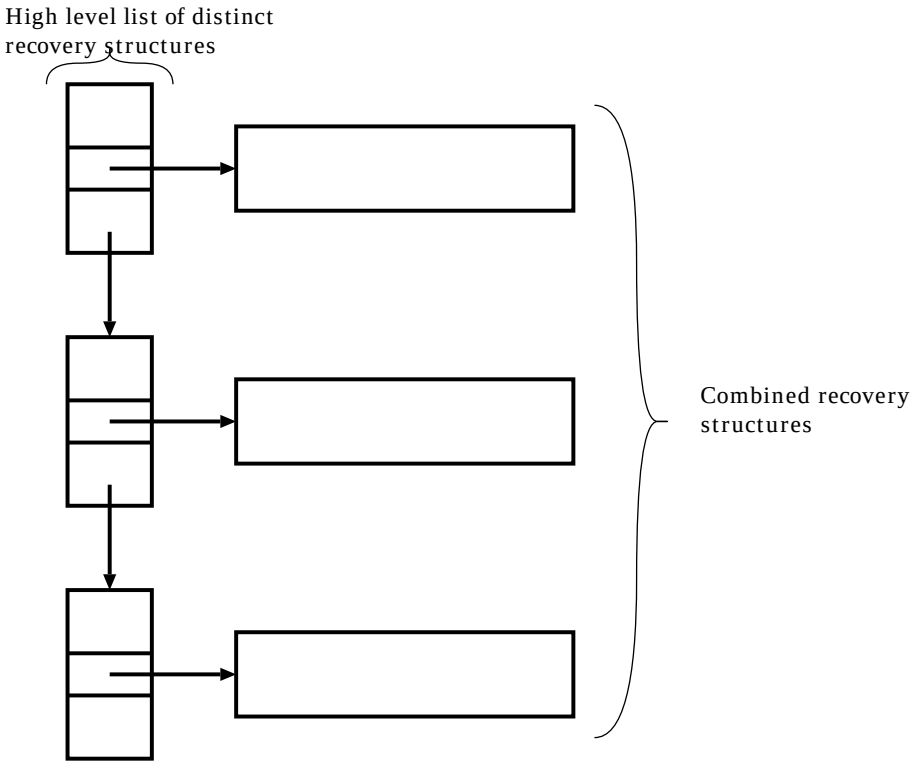

Figure 7-5. Multi-Level List Structure After Combining

The last new message type is used to accumulate the recovery contributions resulting from the partial factorization attempt. This message is sent from all non-root processors to the root processor. The contributions in the message come from the locally held portion of the lost pivot rows and columns, if any pivots failed, and from any locally held contribution block extensions resulting from the recovery of earlier failed frontal matrices. All these contributions are held in a column-major format with scalar and local pattern data prepended to facilitate access by the receiving processor.

With specification of the data objects required for lost pivot recovery complete, a discussion of the new routines for lost pivot recovery can commence. The handling of received recovery structures is addressed first, followed by a discussion of the creation and forwarding of recovery structures.

7.2.5 Recovery Handling

Recovery handling encompasses the processing required to accept the incoming control messages, extract the included recovery structures, combine and organize the recovery structures, and determine the effects of these recoveries on the current frontal matrix. This processing is performed by four new routines.

The ACCUMULATE_MODS routine is the top level routine, which has separate logic paths depending on whether or not the executing processor is the root processor for the current frontal matrix. If executed by the root processor, this routine will first wait for and accept the incoming control messages. The messages are typed based

on the destination frontal matrix ID, which allows for selective reads, but checks are required of the internal message destination field since the message typing facility is limited. Control messages destined for other frontal matrices are queued up to a pre-read control message queue at the appropriate frontal matrix. Once all the expected control messages have been received, they are scanned for recovery structures by a call to the ACCEPT_AND_LOAD_RECOVERY_STRUCTURES routine. In this routine, the multi-level list structure shown in Figure 7-4 is built and then the multiple instances of each distinct recovery structure are joined into a combined recovery structure as shown in Figure 7-5.

If any recovery structures were received, the ACCUMULATE_MODS routine then calls the RESOLVE_IMPACTS_FROM_RECOVERY_STRUCTURES routine to determine the effects of the recovery structures on the current frontal matrix and to broadcast the accumulated effects to the other non-root processors in the subcube assigned to the current frontal matrix. After any recovery effects are resolved, the storage for the local portion of the frontal matrix is allocated and the control messages have their receiver counter decremented. Storage is released if the counter goes to zero.

When ACCUMULATE_MODS is called by a non-root processor, a blocking read is posted for the broadcast from the root processor. The received broadcast message will indicate the length of the accumulated recovery effects. If this length is zero, there are no effects and local frontal matrix storage can be allocated and normal processing commenced. If the length is nonzero, another read is posted to receive a second broadcast from the root processor that contains the actual accumulated recovery effects. This second broadcast and read are necessary since the nCUBE requires the availability of an appropriately sized receive buffer before a broadcast message can be read. The second read is actually done by the INTEGRATE_MODS_FOR_RECOVERY routine, which also updates the local frontal matrix description as required by the accumulated modifications.

Within the ACCEPT_AND_LOAD_RECOVERY_STRUCTURES routine, only recovery structures recovered by the current frontal matrix or having a recovery level higher than the current level are retained. Each distinct recovery structure (for a distinct failed frontal matrix) causes a new entry to be created in the high level list. This list is ordered by ascending recovery level to facilitate the later resolution ordering required by Theorem 7.25. The high level list entry for each distinct recovery structure is built to include the recovery matrix, recovery level, relationship of the failed frontal matrix to the current frontal matrix (i.e., L predecessor, U predecessor, or LU predecessor), and a list of the received instances of this particular recovery structure. The relationship determination is done after the particular instances have been combined. Specifically, the relationship is established by a comparison of the recovery structure's combined nonpivot block pattern with the current frontal matrix's pivot block pattern. The list of instances use pointers to the recovery structure instances as they appear in control messages to avoid memory-to-memory transfers.

Once all the appropriate recovery structures have been added to the multi-level list structure, the instances of each distinct recovery structure are joined into a combined recovery structure, which will replace the list of instances in the multi-level list structure. The most interesting operations in this joining process are the accumulation of the recovery structure pattern data and the integration of recovery contributions. The recovery row and column patterns are accumulated into both scattered and gathered vectors. The scattered form allows duplicate pattern entries to be ignored unless they indicate the marking of a previously unmarked entry. Marking, done by negation of the pattern entry, indicates that the particular row or column of the recovery structure has been absorbed into an earlier frontal matrix as specified by Corollaries 7.7 and 7.10. Once a marked pattern entry is encountered, that entry must be marked in the combined recovery structure.

With the combined patterns completed, the next step is to integrate the recovery contributions of the various instances into the combined recovery structure. A heap space is reserved in the combined recovery structure that will be used for holding the linked lists of recovery contributions for each row of the lost pivot columns and each column of the lost pivot rows. Pointer arrays that correspond to the nonpivot block row and column patterns of the recovery structure are also reserved in the combined recovery structure and will be used to point to the individual linked lists of recovery contributions. The recovery contributions for the unmarked rows and columns are scanned for each recovery structure instance. During this scan, the uniqueness of each contribution is determined based on the row, column, and source frontal matrix ID fields. Distinct recovery contributions not yet added to the combined recovery structure are inserted into the appropriate list, while duplicates are eliminated. The recovery contributions for a particular row or column adhere to a specified ordering in both the recovery structure instances and the combined recovery structure. This allows for improved efficiency in the insertion and duplicate elimination processes since a merge pointer can be used within the combined recovery structure list for each scan. The specified ordering requires organization by ascending column ID and ascending source frontal matrix IDs within columns for each row of the lost pivot columns. Likewise, the ordering of the columns of the lost pivot rows is by ascending row ID and then ascending source frontal matrix ID.

With construction of the combined recovery structures complete, the impacts of these recoveries on the current frontal matrix can be determined. This is done by the RESOLVE_IMPACTS_FROM_RECOVERY_STRUCTURES routine and requires two passes. In the first pass, the magnitude of the effects are determined so that storage for the combined effects can be allocated. In the second pass, the combined effects are actually determined. Within both passes, the combined recovery structures are processed in the proper order as imposed by the multi-level list structure. This requires that all recovery structures recovered in the current frontal matrix be handled first. In this case the current frontal matrix need only have its pivot block expanded (per the results of Theorem 7.14). In the second pass of these

recovery structures, new pivot columns are assigned to processors in a scattered manner to maintain load balance. All recovery contributions are queued up for assembly into the current frontal matrix.

Once the recovery structures recovered by the current frontal matrix have been visited, the remaining recovery structures are processed in order of ascending recovery levels. During the first pass, those nonpivot rows and columns of the recovery structure that are absorbed by the current frontal matrix are marked in the patterns of the combined recovery structure. Furthermore, the growth of the current frontal matrix's $\mathcal{U}''$ pattern by L predecessors and of its $\mathcal{L}''$ pattern by U predecessors is determined. In the second pass, the actual pattern extensions are specified with reallocations of the pattern vectors in the frontal matrix description performed as necessary. (These vectors were initially allocated with room for growth, so that reallocations may be avoided). Any new columns are assigned to processors in a scattered manner. The recovery contributions for each absorbed row and column of the combined recovery structure are also queued for assembly into the current frontal matrix.

With all of the effects determined, the recovery update message is constructed and sent to the non-root processors. During this construction process, recovery contributions to be assembled into the local frontal matrix are directly assembled and not included in the message. Those destined for other processors are assembled into a single contribution for each distinct entry and placed in the message buffer. The recovery update message also contains scalar, pattern, and assignment information that will be used by the receiving processors to update their local frontal matrix descriptions. Furthermore, the recovery update message is only built if more than one processor will participate in the factorization of the current frontal matrix.

The RESOLVE_IMPACTS_FROM_RECOVERY_STRUCTURES routine also performs the local updates to the frontal matrix description required by the combined recovery effects, which includes allocation of an appropriately sized local frontal matrix storage area.

The INTEGRATE_MODS_FOR_RECOVERY routine performs the local updates to the frontal matrix description as required per the information in the received recovery update message. This includes any necessary reallocations of frontal matrix description pattern and assignment vectors, together with the allocation of a local frontal matrix storage area, and the assembly of the appropriate recovery contributions into that area.

As has just been described, recovery handling requires the acceptance of all expected control messages as required for synchronization and lost pivot recovery, the organization and combining of recovery structures, the determination of how the received recovery structures affect the current frontal matrix, and an updating of the current frontal matrix's description in each assigned processor.

7.2.6 Recovery Creation and Forwarding

Once all of the effects of lost pivot recovery on the current frontal matrix have been determined and the frontal matrix descriptions updated, the normal assembly and partial factorization may commence. When the partial factorization is complete, the outgoing control messages may be built and sent out. These will include the

forwarding of all received recovery structures not recovered by the current frontal matrix with updates that reflect the effects and additional recovery contributions made by the current frontal matrix. Also, pivots may have failed in the current frontal matrix, which will require the creation of a new recovery structure.

However, as the current frontal matrix may be distributed across the processors of an assigned subcube, any lost pivot rows and columns and the new recovery contributions of the current frontal matrix might be spread across the processors and must be accumulated at the root processor before the outgoing control messages can be built. The SEND_RECOVERY_CONTRIBUTIONS routine is executed by non-root processors and creates the "accumulate recovery contribution" messages that are forwarded to the root processor. These messages include all local frontal matrix entries that will need to be included into a recovery structure. The entries are packed into the message in a column-major format with scalar and local pattern data also included to assist in accessing the entries. Each non-root processor forwards its message the to root processor.

Within the root processor, the PASS_RESULTS routine handles all recovery structure creation and forwarding. Specifically, all the received recovery structures to be forwarded must be updated with the effects of the current frontal matrix's factorization attempt and a new recovery structure must be created if there are failed pivots in the current frontal matrix. All these recovery structures must then be built into a single outgoing control message. As the size of this control message is dependent upon the effects and results of the current frontal matrix's factorization, two passes over the recovery structures must be completed. During the first pass the size of the control message is determined. The second pass builds the control message. The building process, however, requires the recovery contributions from the other processors in the assigned subcube. Thus, in between the first and second passes, the root processor must read the accumulated recovery contribution messages from the other processors and put them into an indexed structure so that the recovery contributions can be accessed during the subsequent building of the control message. Postponing the reading of these messages until after the first pass allows for some parallelism between the root and non-root processors.

During the first pass, the major objective is to determine how the existing recovery structures patterns will grow due to new recovery fill-in produced by this frontal matrix per the results of Theorems 7.6 and 7.9. During the second pass, the objective is to build the recovery structures in the allocated control message buffer with all recovery contributions produced by the current frontal matrix integrated into the recovery contribution lists per the specified ordering schemes. Newly created recovery structures resulting from pivot failures in the current frontal matrix have their pivot block patterns sorted in ascending order to aid in preserving the required ordering of recovery contributions.

In addition, both the first and second passes must deal with resolution of the overlap area that can be created if the current frontal is both the L ancestor of one failed frontal matrix and the U ancestor of another failed frontal matrix. When this occurs, the contribution block is expanded by both new rows and columns and the

intersection of these new rows and columns is called the overlap area. Per the results of Theorem 7.20, the recovery contributions in the overlap area must be forwarded with the earliest recovered recovery structure. The ordering of recovery structures specified by Theorem 7.25 and enforced by the recovery handling code is used to insure proper handling of the overlap area. During the first pass, special pattern arrays corresponding to the rows and columns of the overlap area are used to determine how the recovery structures' row and column patterns are affected by the overlap area. In this process, each recovery structure is consecutively numbered and this numbering is used in the special pattern arrays. Specifically, if the current frontal matrix is an L ancestor of a recovery structure, the columns of the corresponding expansion are marked in the special overlap resolution column pattern array with the recovery structure's number. Likewise, the special overlap resolution row pattern array is marked in the expansion rows of a U predecessor with that recovery structure's number. The row patterns of L predecessor recovery structures will be expanded by rows of the overlap area only if these rows are marked by a number higher than the current recovery structure's number. Likewise, the column patterns of U predecessor recovery structures will be expanded only by columns of the overlap area that are numbered higher than the recovery structure's number.

Handling of the overlap area changes however during the second pass. The recovery structures are processed in the same prescribed order but inclusion of the overlap area recovery contributions is determined strictly based on the pattern data. To avoid the possible duplication of recovery contributions that can result, the entries are zeroed once taken as a recovery contribution. Thus, if the entry is included as a separate recovery contribution, its value will be zero and have no effect. Meanwhile, the true recovery contribution will be included with the appropriate recovery structure. The reason for this rather obtuse handling during the second pass is based on a problem that arises from repeated failures of a pivot row or column. Specifically, a nonpivotal row of a lost pivot column may be absorbed from the recovery structure and then that row may fail in the absorbing frontal matrix. A third frontal matrix could be an L ancestor of the original frontal matrix and a U ancestor of the absorbing frontal matrix. Resolution of the overlap area in this third frontal matrix could call for the reintroduction of the absorbed row. However, the pattern handling mechanisms for recovery structures do not allow for such reintroductions and modifications to allow reintroductions would significantly complicate an already complicated process. By disallowing the reintroduction during the pattern updates of the first pass and allowing zeroed duplicate recovery contributions during the second pass, the resulting effect is to carry the "would-be" reintroduced row as part of the absorbing frontal matrix's recovery structure. The transitivity between L and U paths established by Theorem 7.5 insures the contribution will reach its required destination with this approach.

With the control message built and all the appropriate recovery structures included, it can now be forwarded to the control parents. If the control message is to be passed locally to multiple receivers, only a single copy of the message is made. Pointers to the message are queued to each receiver and a counter of the number of

receivers is maintained in the message. Receivers will then decrement this counter and the last receiver frees the allocated message buffer.

An important exception occurs when a frontal matrix experiences failed pivots but has no control parents. In this case the lost pivots are "unrecoverable", which means the matrix is numerically singular. The current implementation simply accumulates the number of such "unrecoverable" failed pivots, which is the rank deficiency or nullity of the matrix, and reports it together with the fact that the factorization failed. An alternative would be to allow the factorization of rank deficient matrices in a manner similar to that done by Davis [32]. In this alternative, the permutation vectors are flagged, via negation, to indicate the position of zeroed pivots. The subsequent triangular solves are made to recognize these zeroed diagonal entries and to set the corresponding solution component to whatever value is specified in that same component of the right hand side $\bar{b}$.

By way of summary, after the factorization attempt has been made for a particular frontal matrix, any received recovery structures must be updated per the effects of the current frontal matrix and forwarded in the outgoing control messages. Also, if the current frontal matrix experiences lost pivots, a new recovery structure must be created and also forwarded in the control messages. Determination of the effects of the current frontal matrix is complicated by the need to resolve the overlap area that can occur in the contribution block. This overlap resolution is further complicated by the possibility that absorbed rows or columns may need to be reintroduced into the recovery structure. Special handling that allows zeroed duplicate recovery contributions and precludes "reintroductions" resolves the problem by essentially keeping the absorbed row or column with the absorbing frontal matrix's recovery structure. Finally, lost pivots can occur in a frontal matrix that has no control parents. This indicates a rank deficiency that is reported and causes a failure in the factorization.

The discussion of the distributed memory implementation of a lost pivot recovery capability is complete. Many of the complications in this capability resulted from the decentralization and duplication of recovery structures forced by the distributed memory environment. In a sequential or parallel shared memory implementation of lost pivot recovery, many of these complications would be dismissed and new approaches made feasible. The next subsection provides some thoughts on how a sequential or shared memory implementation might be addressed.

7.2.7 Sequential and Shared Memory Considerations

Much of the processing for lost pivot recovery results directly from the multiplicity and decentralization of recovery structures, which are necessary in a distributed memory implementation. Hence, a sequential or parallel, shared memory implementation would likely be simpler and more efficient (but might not scale as well). Specifically, only one copy of each failed frontal matrix's recovery structure would need to be maintained. This eliminates the need for the joining of multiple instances as well as the need to perform duplicate elimination on the recovery contributions. Multiple recovery contributions to the same entry could be assembled together as created, which eliminates the additional memory requirements and the list processing code.

Within a shared memory parallel environment, multiple locking mechanisms within a recovery structure could allow parallel operations on a particular recovery structure.

Furthermore, the synchronization requirements for lost pivot recovery are not a consideration in a sequential implementation and could be satisfied by a number of potentially more efficient mechanisms within a shared memory parallel environment. Also, the recovery update and the accumulate recovery contributions messages would no longer be required since a shared memory would allow all processors access to the combined effects of all recovery structures on a particular frontal matrix, as well as to the recovery contributions.

Additional parallelism in the lost pivot recovery process could be investigated in a shared memory implementation. Within the current implementation, parallelism is mostly available only via the independence of distinct frontal matrices. There is little parallelism within the lost pivot recovery processing of a single frontal matrix. While the ordering restrictions on the resolution of recovery effects defined by Theorems 7.20 and 7.22 must be complied with, the full ordering defined by Theorem 7.25 is not always necessary and some parallelism may be exploitable within this resolution process.

Greater parallelism is available in the forwarding and creation of recovery structures. Independence across recovery structures provides one level of parallelism. Furthermore, within the updating of a recovery structure additional parallelism is available and may be exploitable on some architectures. The only real hinderence to parallelism in the forwarding of recovery structures are the complications created in handling recovery contributions from the overlap area due to the possible "reintroduction" of rows or columns into the recovery structure pattern. However, the reintroduction problem goes away in a shared memory environment as the single instance of a recovery structure would allow reintroduction and precludes the need for the special processing that can create redundant, but zeroed, recovery contributions. The overlap area row and column pattern marking mechanism described earlier could be used to uniquely define the handling of each recovery contribution in the overlap area.

While indications are that a shared memory parallel implementation should be simpler and possibly more efficient than the distributed memory implementation developed with this effort, there will still be significant challenges. Careful consideration of the parallel memory access patterns is necessary, not only for the lost pivot recovery mechanisms but, more importantly, for the partial dense factorization kernels as these routines dominate the processing on most problems. Furthermore, whatever synchronization primitives are used to replace the blocking message read must be chosen and implemented wisely. Moreover, in a shared memory environment, dynamic scheduling is possible and more alternative lost pivot recovery strategies are possible.

In conclusion, the lost pivot recovery implementation described in this section obviously adds significantly to the parallel refactorization code. Specifically, the source code expanded from about 4,000 lines to over 9,500 lines (which does not include the relatively minor additions required to the 6,000 lines of host processing

code). The corresponding effects on performance due to the inclusion of lost pivot recovery are thus of concern and a detailed performance evaluation is provided as the next and final section in this chapter

7.3 Performance Evaluation

The inclusion of a lost pivot recovery capability into the distributed memory parallel refactorization added a significant amount of additional processing with much of it being sequential in nature. Moreover, a number of $O(n)$ data structures have been added to each processing node. A detailed analysis of the effects on performance and memory requirements is thus in order. This final section addresses such issues as the overhead introduced by lost pivot recovery, the effectiveness and efficiency with which it can factor a sequence of realistic matrices, the achievable parallelism both when pivots are lost and not lost, and the amounts of additional memory required and how this impacts scalability.

The discussion starts with a definition of the key performance issues. This is followed by a description of how each performance issue will be measured. The test cases used for the evaluation are then briefly described. Finally, the empirically determined performance results are presented and analyzed.

7.3.1 Performance Issues

The performance issues that need to be addressed in this section can be divided into four categories: overhead, sequential execution time, parallel execution time, and memory requirements/scalability.

Overhead: Even when all anticipated pivots are numerically acceptable, significant additional processing is required with lost pivot recovery. Specifically, the control messages used for the synchronization can cause both processing and synchronization delays, as can the root processor's broadcast to initiate a frontal matrix's assigned subcube. Furthermore, the numerical checking added to the partial dense factorization kernel also increases the processing burden. Thus, an assessment of the additional execution time required for factorization using the lost pivot recovery code when all anticipated pivots are valid is necessary. Furthermore, this assessment should consider both sequential and parallel execution.

Sequential Execution Time: The objective of lost pivot recovery is to allow the factorization of a sequence of identically structured sparse matrices without the need to repeat the analyze operation (or to at least have to reaccomplish the analyze less often). To assess how well lost pivot recovery accomplishes this objective, the execution times for the factorizations of a sequence of matrices should be measured and compared (in some fashion) to the execution times necessary without lost pivot recovery. The effects of using the avoidance strategy of relaxing the pivoting threshold should also be included in this assessment.

Parallel Execution Time: A most significant feature of the lost pivot recovery implementation developed is that it runs in a parallel, distributed memory environment. Hence, it is critical to assess how performance is affected when the parallel features of the implementation are utilized.

Memory Requirements and Scalability: Besides improving execution time, another advantage of parallel, distributed memory environments is that they provide larger available memories and memory bandwidths as more processors are added. In order to take advantage of this feature, an algorithm must be able to spread out the bulk of its required data structures across the processors. Such an algorithm allows larger problems to be run when more processors are available and is said to have good scalability. With perfect scalability, a processor set that is twice as large could be able to handle problems that are twice the size.

7.3.2 Performance Measures

In this section, the means of measuring each of the specified performance issues are defined. These means will be discussed individually for each performance issue. Whenever a refactorization execution time is mentioned, it refers to the execution time as measured in Chapter 6 where the matrix values and frontal matrix descriptions have already been distributed to the parallel processing nodes and these nodes have been synchronized via the *synch* nCUBE directive [111]. The actual time measurements are taken using the *n_time* nCUBE directive. Furthermore, each run used the S3 scheduling routine with all edge consideration and a blocked column to processor assignment scheme, which was typically the best configuration per the results of Chapter 6.

Overhead: The overhead of lost pivot recovery will be measured by doing factorizations of the same matrix for which the assembly DAG (and thus pivot order) was built using both the fixed pivot order refactorization code of Chapter 6 and the full lost pivot recovery code described in this chapter. As the anticipated pivots are all guaranteed to be numerically acceptable, the differences in execution time will indicate only the additional overhead due to lost pivot recovery. These differences will be normalized to a percent by dividing them by the fixed pivot order refactorization time. Thus, the measure will be the percent increase in execution time due to lost pivot recovery. This measurement is done for each of the five test matrices using hypercubes of dimensions zero through six (processor sets ranging from 1 to 64 processors).

Sequential Execution Time: The effectiveness and efficiency of lost pivot recovery on a single processor is measured using single processor refactorizations of each matrix in the RDIST1, RDIST2, and RDIST3A sequences. Separate sets of runs are done with the pivot threshold value set to 0.1 (which is the value used during the analyze-factorize UMFPACK run that built the assembly DAG and thus specified the initial pivot ordering) and to 0.001 (which corresponds to the use of a lost pivot avoidance strategy). To put these times into perspective, they will be compared to an estimation of the analyze-factorize time. Only an estimated analyze-factorize time is possible, since the UMFPACK analyze-factorize routine required too much memory to be run on an single nCUBE processing node. The estimate of the analyze-factorize time was achieved by taking the ratio of the UMFPACK analyze-factorize time to the UMFPACK refactorize time as run on a single processor of a KSR-1 system and

multiplying the ratio by the single nCUBE 2 processing node execution time of the fixed pivot order refactorization.

Parallel Execution Time: Because many of the matrices in any one particular sequence experienced very similar numbers of lost pivots and execution times, only selected matrices within a sequence were tested for parallel performance. The selected matrices were factored on a five dimensional hypercube (32 processors). The parallel execution times are then compared to the corresponding sequential execution time and a speed-up factor is determined. The number of lost pivots for each run is also provided to help put the results in a proper context.

A specific matrix that incurs many lost pivots will also be selected and factored on a full range of hypercube dimensions. The corresponding speed-up curves are plotted for runs both with and without lost pivot avoidance. These are compared to the speed-up curves when no pivots are lost.

Memory Requirements and Scalability: Memory requirements and scalability will be measured using the maximum amount of dynamically allocated memory used by any one parallel processing node. This statistic is measured by special code that traces memory allocations and releases. Using these traces, the code maintains current and maximum usage statistics for each processing node. At the conclusion of the factorization, a global *imax* nCUBE operation determines the maximum usage by any single processing node. Scalability is evident if this memory statistic goes down as the number of processors increases, and a perfect scalability curve is provided for reference.

Errors and Residuals: Lost pivot recovery can have adverse effects on the numerical quality of the factorization. In particular, the avoidance strategy of relaxing the pivot threshold increases the growth bound on individual entries of the LU factors, which has direct effects on the error bounds of the solution. Furthermore, lost pivot recovery can also result in a growth in the size of the LU factors, which increases the required arithmetic and the error bounds. To track the magnitude of these effects, each factorization was followed by a pair of solves with one using a known solution vector of all ones and the other using a randomly generated solution vector. Once the known solution vectors were defined, the corresponding right hand side ($\bar{b}$) was determined by a sparse matrix-vector multiplication. A solve was then done using the LU factors and $\bar{b}$ The relative error was then determined as

$$\text{relative error} = \frac{\|\bar{x}_{true} - \bar{x}_{computed}\|_\infty}{\|\bar{x}_{true}\|_\infty}.$$

Since matrices in sequences frequently become ill-conditioned and as the residual is typically the convergence measure, the relative residuals were also determined as

$$\text{relative residual} = \frac{\|A\bar{x}_{computed} - \bar{b}\|_\infty}{\|b\|_\infty}.$$

References to these observed relative errors and residuals will be made as appropriate in the discussion of the performance results.

7.3.3 Test Cases

The test matrices used in the performance evaluation presented in this chapter are the same as those used in Chapter 6. However, sequences were available for only the RDIST1, RDIST2, and RDIST3A matrices. These sequences consist of 41, 40, and 37 distinct matrices, respectively. These sequences are from three different chemical engineering distillation problems that were originally solved using a unifrontal method [144, 143]. This method maintains fully assembled rows, which allow partial pivoting. The various matrices in the sequence correspond to the iterations of a Newton's method so the values of the next matrix in the sequence will be dependent upon the solution of the current matrix. Hence, these are somewhat artificial sequences as the solutions of the unifrontal method used to generate the sequences could differ from the solutions produced by the parallel, multifrontal refactorization method.

7.3.4 Performance Results

Overhead: The first performance issue addressed is how severely the inclusion of the lost pivot recovery capability effects the performance of the parallel refactorization code when all of the anticipated pivots are numerically acceptable. As only the first matrix in a sequence is required for this determination, all of the five test cases from Chapter 6 were evaluated for overhead. The results of this evaluation are shown in Figure 7-6.

The overhead evaluation results of Figure 7-6 show two distinct patterns. The EXTR1 and GEMAT11 matrices have much higher overheads but the overheads taper off with larger hypercubes. The RDIST matrices however have relatively low lost pivot recovery overheads but the overhead increases with more processors. These two patterns correlate closely with the amount of computation required for factorization. The GEMAT11 and EXTR1 matrices require 5 to 10 times less computations for factorization but are of nearly the same order, have more frontal matrices, and have larger edge sets as compared to the RDIST matrices. While their absolute overhead times are similar to the RDIST matrices, the amount of overhead relative to all required computations are greater for the GEMAT11 and EXTR1 matrices.

The tapering off of overhead for the GEMAT11 and EXTR1 matrices is due to inter-frontal matrix parallelism exploited by the larger hypercubes. These matrices take special advantage of inter-frontal matrix parallelism for lost pivot recovery since most of the frontal matrices are small and only allocated a single processor. The RDIST matrices on the other hand have significantly larger frontal matrices and can take less advantage of inter-frontal matrix parallelism because more frontal matrices are assigned larger subcubes.

In terms of speed-ups, the overheads for the GEMAT11 and EXTR1 matrices translated into speed-ups that were slightly better with lost pivot recovery. The nearly constant overhead for the RDIST matrices across the various hypercubes meant nearly identical speed-ups both with and without lost pivot recovery except for

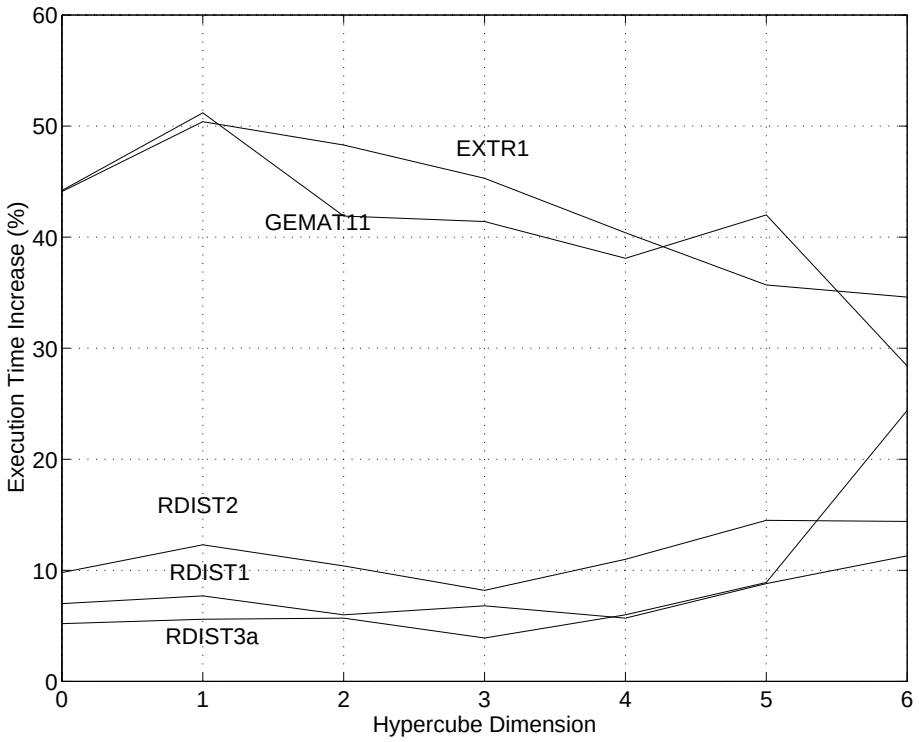

Figure 7-6. Lost Pivot Recovery Overhead

the largest configuration where overhead went up and accordingly speed-up dropped (especially for RDIST3A).

Sequential Execution Time: Sequential execution time was measured for each of the three available RDIST sequences and three distinct behaviors emerged. The results are shown in Figures 7-7, 7-8, and 7-9.

With the RDIST1 matrix sequence of 41 distinct matrices, most of the anticipated pivots were either acceptable or recoverable within the frontal matrices. Specifically, without lost pivot avoidance ($\mu = 0.1$), only 24 pivots had to be recovered across frontal matrices in the worst case. When lost pivot avoidance was enabled ($\mu = 0.001$), only a single lost pivot had to be recovered across frontal matrices in the worst case. Hence, the results in shown in Figure 7-7 indicate that lost pivot recovery always produced better performance than that predicted for an analyze-factorize operation.

With the RDIST2 matrix sequence of 40 distinct matrices, significantly more pivots were lost. Without lost pivot avoidance, as many as 149 anticipated pivots had to be recovered across frontal matrices. When avoidance was enabled, the most lost pivots across frontal matrices dropped to 49. Figure 7-8 shows that with lost pivot avoidance, the refactorization code was always better than doing an analyze-factorize operation. However, by matrix 7 in the sequence, the refactorization time without avoidance began to exceed that predicted for an analyze-factorize operation. At that point an analyze-factorize operation could be run to produce a new pivot order and associated assembly DAG. The refactorization could then be tried on the

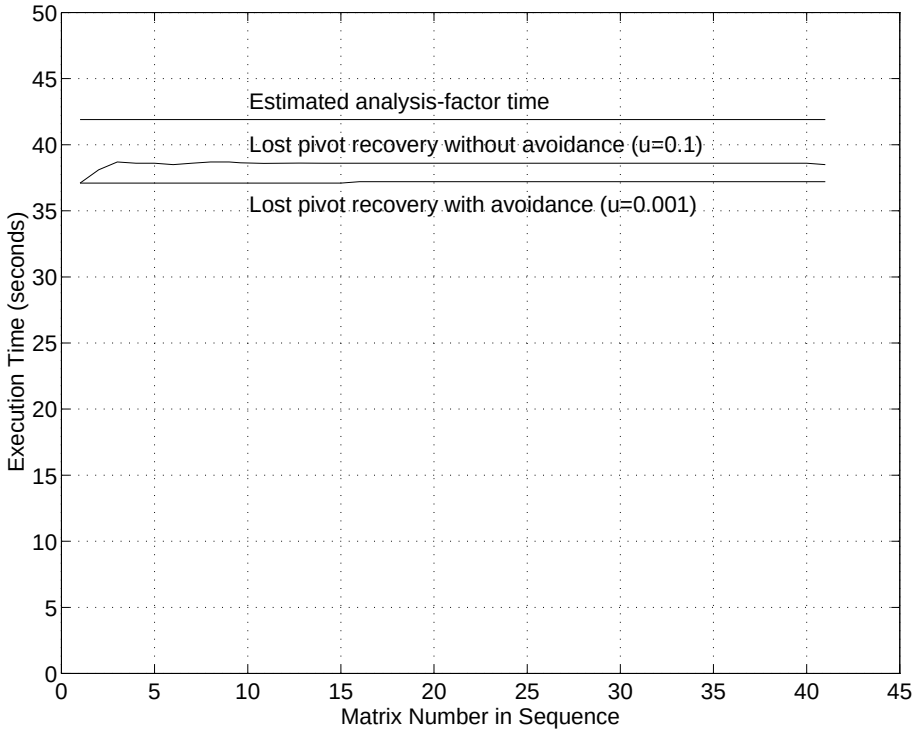

Figure 7-7. Sequential Execution Time for RDIST1 Sequence

subsequent matrices with a greater likelihood that the new anticipated pivots would be acceptable.

The RDIST3A matrix sequence of 37 matrices experienced the worst performance. As many as 223 failed pivots had to be recovered across frontal matrices when no avoidance was employed. Furthermore, Figure 7-9 shows that without avoidance refactorization time began to exceed the estimated analyze-factorize time at only the fourth matrix in the sequence. Again, an analyze-factorize run followed by refactorizations on the subsequent matrices could be tried. However, the picture was much brighter when an avoidance strategy was enabled. In this case all of the matrices were refactored faster than they would have been factored by an analyze-factorize operation. The worst number of lost pivots with avoidance was only 52.

With all of the RDIST sequences, the relative errors became large for matrices in the middle of the sequences as these matrices were extremely ill-conditioned. The residuals however remained low with only the loss of four significant digits in the worst case. Furthermore, these relative error and residual results are nearly identical to those received when the analyze-factorize operation of UMFPACK was done as a cross check on the worst of these matrices. In addition, the results were somewhat worse when the avoidance strategy was employed. However, this was anticipated per the relationship between the growth factor and the pivot threshold as specified in the beginning of this chapter. The differences in results were within the bounds of the growth factors.

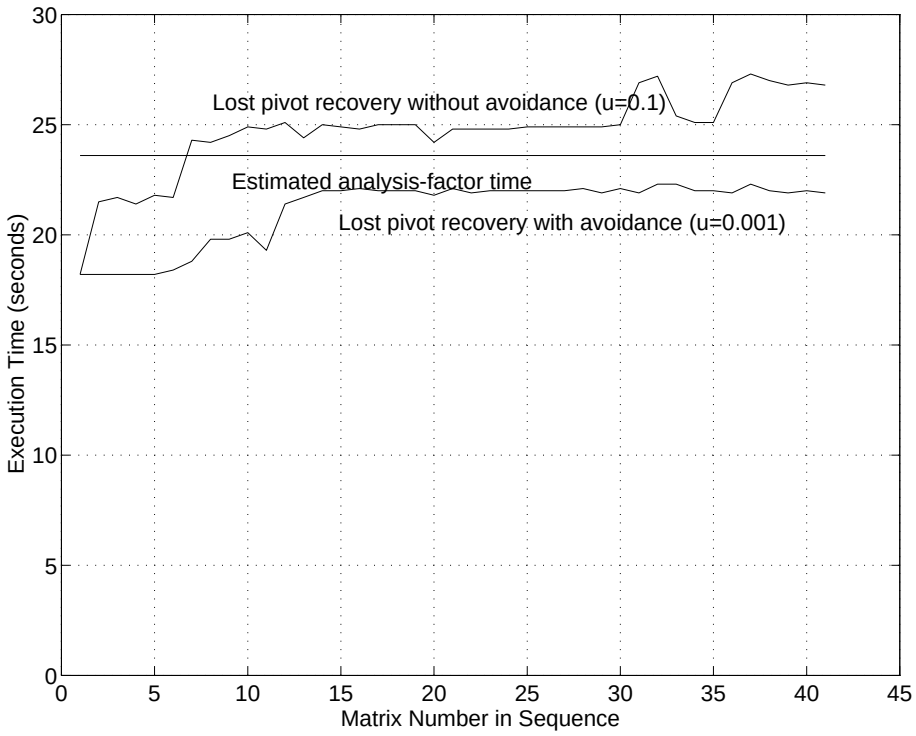

Figure 7-8. Sequential Execution Time for RDIST2 Sequence

Perhaps the most significant result of this sequential execution time analysis is that with the use of a lost pivot avoidance strategy, the refactoring time was always better than the estimated analyze-factorize time. However, the additional and significant advantages of parallelism have not yet been investigated. One of the principal advantages of a factorization-only multifrontal algorithm is that the predefined assembly DAG and dense frontal matrix factorization tasks provide exploitable parallelism. Thus a parallel execution would make the advantages of refactorization with lost pivot recovery (as compared to a sequential analyze-factorize) even more attractive. The open questions are how much better is the parallel execution time and how much does inclusion of lost pivot recovery hurt parallelism when pivots are lost.

Parallel Execution Time: The first assessment of parallel execution time was performed by running representative matrices from each sequence on a 32 processor hypercube. The results for the three matrix sequences are shown in Tables 7-2, 7-3, and 7-4.

With the RDIST1 parallel execution time results shown in Table 7-2, there is little change in the speed-up, which correlates to the relatively few number of lost pivots that had to be recovered across frontal matrices. The results with lost pivot avoidance are not shown since at most only one pivot was lost across frontal matrices and all parallel execution times were essentially the same as that of matrix number 1.

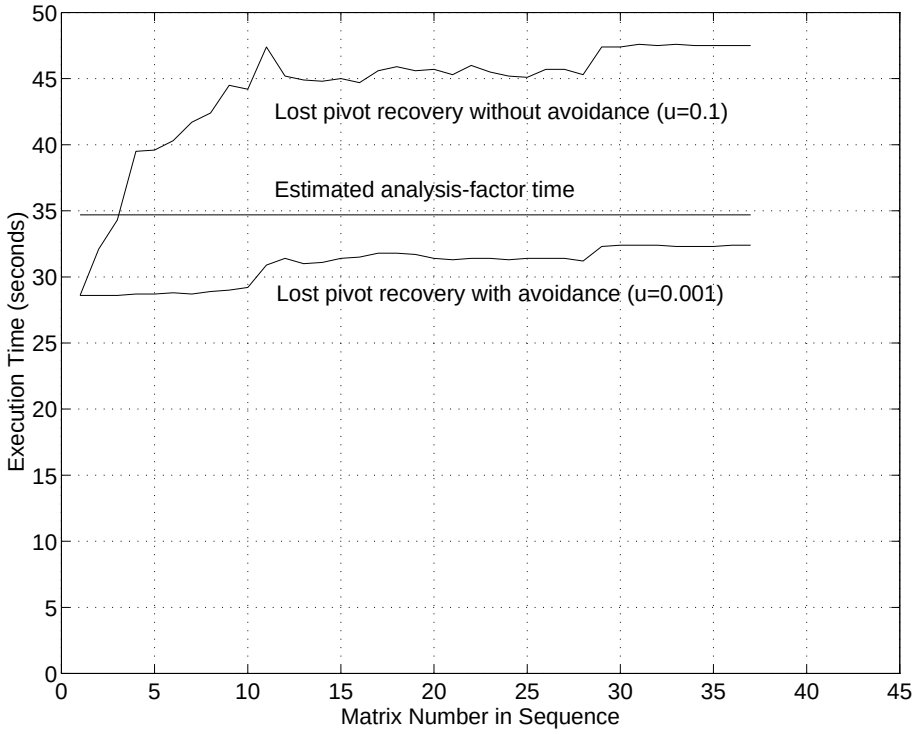

Figure 7-9. Sequential Execution Time for RDIST3A Sequence

Table 7-2. RDIST1 Parallel Execution Time Results (32 processors)

MATRIX NUMBER	PARALLEL TIME (secs)	SPEED-UP	LOST PIVOTS
Without Avoidance:			
1	2.996	12.4	0
9	3.161	12.2	23
20	3.170	12.2	16

In Table 7-3, the RDIST2 results show significant drops in achieved speed-up especially as more pivots are lost across frontal matrices. Moreover, the results are worse when an avoidance strategy is not in use. However, reasonable speed-ups were consistently achieved with very favorable parallel factorization times.

The worst matrix sequence, in terms of pivots lost across frontal matrices, was the RDIST3A sequence. Parallel results for RDIST3A are shown in Table 7-4. Here we see extreme drop-offs in speed-ups when a lost pivot avoidance strategy is not employed. However, with an avoidance strategy, the results are again quite favorable.

While the parallel execution times of the RDIST sequences on a 32 processor hypercube were very favorable compared to the sequential execution times and to the previous estimated analyze-factorize times, there is a significant loss of parallelism

Table 7-3. RDIST2 Parallel Execution Time Results (32 processors)

MATRIX NUMBER	PARALLEL TIME (secs)	SPEED-UP	LOST PIVOTS
Without Avoidance:			
1	1.726	10.6	0
5	2.560	8.5	45
8	3.120	7.8	80
18	3.440	7.3	104
28	3.431	7.3	104
With Avoidance:			
1	1.726	10.6	0
7	1.874	10.0	3
9	2.128	9.3	14
17	2.619	8.4	38
21	2.827	7.8	51
33	2.741	8.1	44
40	2.724	8.1	44

when more pivots are lost. In order to bound the degradations in parallelism due to lost pivot recovery, the speed-up curves for the worst lost pivot recovery test case were determined. This case was matrix 37 of the RDIST3A sequence, which lost 221 pivots without avoidance and 52 pivots with avoidance. These speed-up curves are shown in Figure 7-10. Also shown in that figure are the speed-up curves for matrix 1 of the RDIST3A sequence where no pivots are lost. Both the speed-ups with and without the lost pivot recovery (LPR) code enabled are shown for the matrix 1 case.

Notice, from Figure 7-10, that little parallelism is lost by the lost pivot recovery code when no pivots are actually lost. However, there are significant degradations in parallelism when pivots are actually lost. This difference in exploited parallelism motivates the need to take better advantage of parallelism in lost pivot recovery. While doing so in a distributed memory environment would be exceedingly difficult, the ideas presented earlier on shared memory implementations could successfully enhance parallelism within that environment.

Memory Requirements and Scalability: The final performance issue is how does lost pivot recovery effect the scalability of the refactorization as measured by the memory requirements on a per processing node basis. Figures 7.14 and 7.16 show the memory required per processing node both without and with the inclusion of lost pivot recovery, respectively. In each figure, the perfect scalability curve is also shown for reference. Figure 7-11 shows the results from matrix 1 of RDIST3A where no pivots are lost and the lost pivot recovery code was not enabled. Indeed these results are quite promising. However, Figure 7-12 shows the results when the lost pivot recovery code was enabled. Cases were run and plotted for no lost pivots (by

Table 7-4. RDIST3A Parallel Execution Time Results (32 processors)

MATRIX NUMBER	PARALLEL TIME (secs)	SPEED-UP	LOST PIVOTS
Without Avoidance:			
1	2.388	12.0	0
2	4.435	7.2	120
3	5.126	6.7	100
4	6.553	6.0	120
5	6.561	6.0	120
10	7.749	5.7	132
20	8.364	5.5	201
30	9.180	5.2	218
With Avoidance:			
1	2.388	12.0	0
5	2.464	11.7	4
10	2.767	10.6	12
20	3.801	8.3	33
30	4.160	7.8	52

using matrix 1 of RDIST3A) and for the worst amounts of lost pivots (by using matrix 37 of RDIST3A both with and without avoidance). The closeness of the three measured results curves indicates that the number of lost pivots has a relatively minor effect on memory requirements and scalability (as these cases have 0, 52, and 221 lost pivots respectively). The large difference between these curves and the perfect scalability curve is due rather to the $O(n)$ data structures required by lost pivot recovery that are allocated on each processing node. Currently these data structures are preallocated of sufficient size to handle most all necessary recoveries. Indeed they could be significantly scaled down and/or dynamically allocated to improve scalability, however, their $O(n)$ nature is limiting.

By way of concluding this performance evaluation section, a quick summary of the results is useful. First, the overhead incurred by including the lost pivot recovery code was shown to be very minimal for problems with more significant computational requirements. Furthermore, parallelism was maintained when no pivot loss occurred. Sequential execution time analysis showed that lost pivot recovery with an avoidance strategy was a very effective alternative to the analyze-factorize operation. While even greater advantages are possible by doing the refactorizations in parallel, there are definite degradations as more pivots are lost. Analysis of the scalability of the method illustrated that without lost pivot recovery the scalability was excellent, however, scalability degraded significantly due to the $O(n)$ data structures required on each processing node by lost pivot recovery. The analysis indicates the importance of fine tuning the management of these structures.

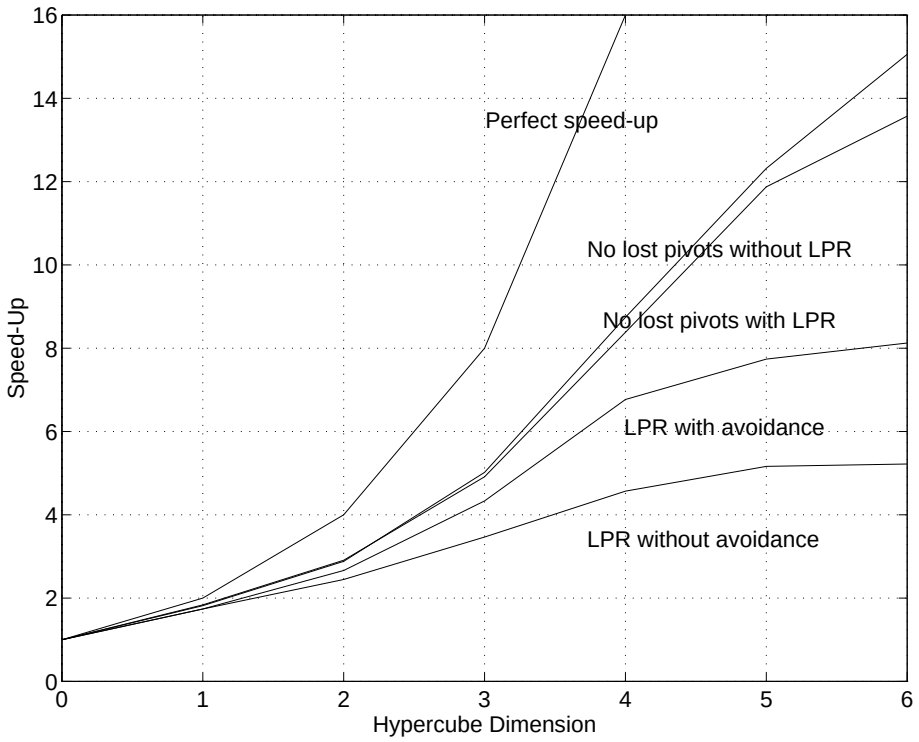

Figure 7-10. RDIST3A Speed-Ups With and Without Lost Pivot Recovery (LPR)

The bottom line, however, is that the lost pivot recovery capability developed and implemented in this chapter does provide a very effective alternative to repeated use of the analyze-factorize operation. These benefits are further improved when exploitation of parallelism is added to lost pivot recovery with lost pivot avoidance.

By way of concluding this chapter, first presented was the theoretical development of a lost pivot recovery strategy that is fully robust and will handle any number of failed pivots. Importantly, the theory provided for a lost pivot recovery mechanism that could completely determine all communication requirements before factorization, which is especially critical for statically scheduled, distributed memory implementations. Moreover, if block upper triangular form is achieved prior to the initial analyze-factorize operation, significant advantages are obtained such as the ability to use existing communication paths only.

The promises of the theoretical development were then realized by implementing the necessary lost pivot recovery mechanisms within the framework of the parallel refactorization code developed in the previous chapter. Subsequent evaluation of this implementation revealed significant advantages for refactorization with lost pivot recovery (and a lost pivot avoidance strategy) even if parallelism was not exploited. Even more benefits were achievable when parallelism was included.

Thus, while an unsymmetric-pattern multifrontal method does not possess the properties of a classical multifrontal method that make lost pivot recovery very straight forward, it can be augmented with a more sophisticated lost pivot recovery

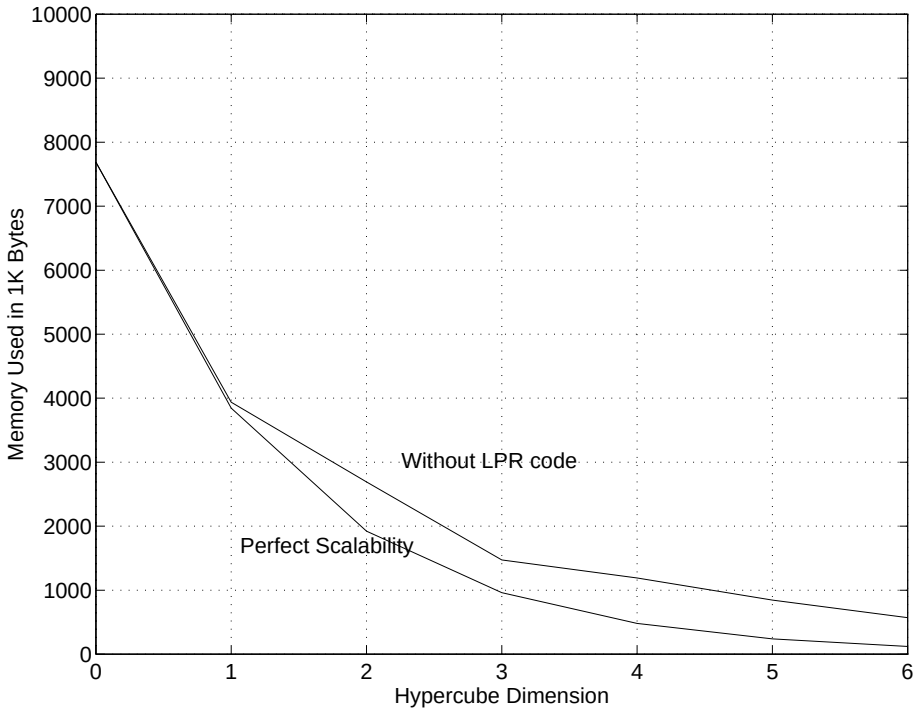

Figure 7-11. Scalability Without Lost Pivot Recovery - RDIST3A

capability that is fully robust and demonstrates significant performance advantages on realistic problems.

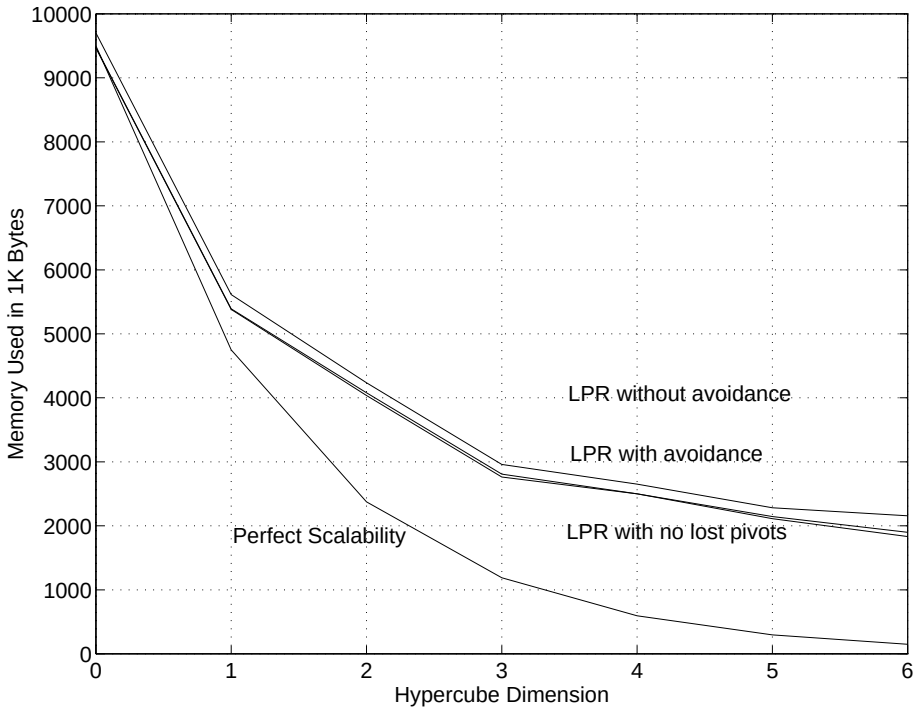

Figure 7-12. Scalability With Lost Pivot Recovery - RDIST3A

CHAPTER 8
CONCLUSION

Sequences of identically structured sparse matrices arise in numerous application areas. When the matrices are not symmetric and positive-definite, the most common solution technique is LU factorization. One method that can be used to perform the LU factorizations is the unsymmetric-pattern multifrontal approach [34, 35]. With this approach, the assembly DAG (and implied pivot ordering) produced by a single analyze operation can be repeatedly used to factorize the subsequent matrices in the sequence and reduce the overall required computation time. Furthermore, the assembly DAG defines a computational structure that exposes available parallelism, which occurs both between frontal matrices (nodes in the assembly DAG) and within the partial factorizations of the dense frontal matrices.

8.1 Summary

This research effort first investigated both the theoretical and achievable parallelism available via the assembly DAG. This was first accomplished by analysis of the assembly DAG where it was found that exploiting parallelism both between and within frontal matrices was critical for the best performance. Theoretical speed-ups of as much as 167 were predicted using inter-frontal parallelism and a medium grain (row-oriented) parallelism within frontal matrices on modestly sized problems (of order less than 5000). When inter-frontal parallelism was combined with fine grain parallelism within frontal matrices, the theoretical speed-ups increased to as much as 6,800 and 11,500 for two of the five matrices tested. These analytical models, however, assumed an unbounded number of available processors. In order to determine the minimal sizes of processor sets needed to achieve these theoretical speed-ups, corresponding simulation models were developed and run assuming processor sets of 1 to 65,536 (using the powers of 2 from 0 to 16). The results of these simulation runs indicated that the theoretical speed-ups with inter-frontal matrix parallelism and medium grain parallelism within frontal matrices could be achieved with at most 512 processors and with fine grain parallelism within frontal matrices at most 65,536 processors were required. Thus, the processor set sizes required to achieve the best theoretical speed-ups are within the range of currently available configurations. However, these results were all achieved based on an underlying parallel random access machine (PRAM) model. The questions of actually achievable parallelism on existing architectures remained an open issue.

In order to assess the amount of parallelism actually exploitable given the current state of the art, the simulation model was modified to represent the parallel distributed memory environment of the nCUBE 2 multiprocessor. The results of an empirical performance characterization of the nCUBE 2 were used to define the

model's parameters and conservative assumptions were the norm. The results of these simulation runs showed that only 20 to 25 percent of the theoretical parallelism of the PRAM-based models could actually be expected on an nCUBE 2 class of multiprocessor. Furthermore, various configurations of the distributed memory model indicated that much of the achievable parallelism was realized with 1024 or fewer processors and that taking advantage of parallelism in the assembly of contributions was very important (resulting in 20 to 50 percent better speed-ups).

The next step in the evaluation of parallelism in the unsymmetric-pattern multifrontal method was to actually implement a parallel refactorization algorithm on the nCUBE 2 multiprocessor. Since much of the computational time for refactorization would be spent actually doing the partial dense factorization of frontal matrices, particular attention was given to these routines. Five varieties were developed and implemented. Two used only the anticipated pivots with no numerical checking or pivoting and two used threshold pivoting with mechanisms to find alternative pivots on other processors if necessary. Within both the fixed pivot and threshold pivoting categories were pipelined and non-pipelined versions. The pipelined versions improved performance by overlapping required communication with computation. The fifth version was tuned for the frequent case of a single pivot frontal matrix.

These partial dense factorization kernels achieved as much as 90 percent of the peak basic linear algebra subroutine (BLAS) performance and 62 to 70 percent of peak machine performance on a single processor. On 32 processors, as much as 45 percent of peak BLAS performance and 32 percent of machine peak performance was achieved.

With efficient partial dense factorization kernels available, a full parallel refactorization capability was built. The speed-ups actually achieved compared very favorably with those predicted earlier with the distributed memory simulation model. For instance, the GEMAT11 matrix had a predicted speed-up of 15.2 and actually achieved 16.7 on 64 processors. The RDIST1 matrix had a predicted speed-up of 23.2 and actually achieved 20.2 on 64 processors. Furthermore, the parallel refactorization routine was seen to be very competitive as compared to the commonly accepted sequential standard of Harwell's MA28B [53]. For example, refactorization of the GEMAT11 matrix on a single processor took 3.103 seconds with MA28B and only 2.521 seconds with the parallel refactorization routine implemented. The scalability of the parallel refactorization was assessed empirically. The memory requirements scale well, diminishing almost linearly with increased numbers of processors.

Within the parallel refactorization routine a number of alternative scheduling, allocation, and assignment strategies were developed and evaluated. Of the two scheduling methods, the one that emphasized reducing overall execution time via subcube management with a secondary consideration of reducing communication always out performed the other method. A variety of bunching and overlapping schemes to reduce communication worked well with as much as an 11 percent reduction over a strictly blocked data assignment and as much as 22.5 percent compared to a scattered data assignment. However, the costs of contribution message passing were low compared to the effects of data assignment on the partial dense factorization kernels'

performance and the best execution times were typically achieved with blocked data assignments for the larger hypercubes (dimensions 4 to 6) and with scattered data assignments for the smaller hypercubes.

With an effective and efficient implementation of a parallel refactorization routine based on the unsymmetric-pattern, multifrontal method completed, the second objective was to remove the assumption of numerically acceptable pivots and allow for threshold pivoting. The idea was to accommodate changing entries in subsequent matrices that may cause the anticipated pivots to no longer be numerically acceptable. A multi-level approach was taken to this problem. First, lost pivots could be avoided by relaxing the pivot threshold. Second, two of the partial dense factorization kernels were built to recover from lost pivots by choosing new pivots from within the pivot block. The last and most difficult level was to recover lost pivots across frontal matrices.

Lost pivot recovery across frontal matrices had not been done within an unsymmetric-pattern, multifrontal approach until this effort. Extensive theoretical development was necessary to define and prove correct the lost pivot recovery capability. The lost pivot recovery capability was then integrated into the parallel refactorization routine and its performance evaluated. As additional message passing and synchronization are required by lost pivot recovery, the amount of additional overhead it actually incurred was important to measure. For the largest and most relavent problems tested, the overheads were very low, typically less than a 10 percent increase in execution time. However, the relative overheads did grow as more processors were added.

The sequential performance of lost pivot recovery on three realistic matrix sequences from chemical engineering distillation processes showed a range of performance. On the first sequence, the refactorization code with lost pivot recovery was always better than that predicted for the analyze-factorize operation. For the second two sequences, the inclusion of a lost pivot avoidance strategy was necessary for the refactorizations to always be better. Moving to an assessment of parallel performance, the achieved parallelism with lost pivot recovery was virtually unchanged when few pivots were lost. However, realizable parallelism quickly dropped off as more lost pivots were encountered. Scalability was also adversely affected by inclusion of lost pivot recovery. This was primarily due to the need to add a number of $O(n)$ data structures to each processing node for lost pivot recovery.

While this initial implementation of lost pivot recovery was quite successful in achieving its goals, it also demonstrated the difficulties of creating an efficient implementation within a distributed memory environment. In fact, a number of suggestions were provided regarding how a sequential or parallel, shared memory implementation might be significantly easier and more efficient.

This research effort was the first study of the theoretical parallelism of the unsymmetric-pattern multifrontal method. These theoretical results were then extended by a realistic distributed memory-based simulation that used empirically derived performance characteristics. The results of this simulation model were then validated by the first parallel implementation of a refactorization capability based on the

unsymmetric-pattern multifrontal method. This implementation was then extended by the development and realization of the first lost pivot recovery capability for an unsymmetric-pattern multifrontal method.

While several new doors have been opened by this research, it has also pointed the way to other paths waiting to be explored. Some of the most significant of these paths are discussed in the next and last section.

8.2 Future Efforts

While the parallel refactorization software resulting from this research provides a significant new capability, there are two important and difficult issues that must be addressed and resolved before it can be practically employed. First, the performance assessments were all based on prepositioned data and frontal matrix descriptions. Realistically, the data is not likely to be initially positioned in this manner and the time to position this data must be taken into account. Furthermore, efficient mechanisms would need to be developed to perform this positioning. Moreover, other operations may need to take place on the data and routines to use the current data positioning or to reposition it would also be needed.

The second major hindrance to practical employment is the poor parallel performance of the distributed triangular solve capability developed. Indeed, the relatively low amount of computation per required communication makes exploitation of parallelism especially difficult. A number of suggestions were provided earlier on how to improve the parallelism of the triangular solves.

Within the parallel refactorization routine there are also additional ideas to explore. The performance advantage of column scattering on smaller hypercubes and column blocking on larger hypercubes suggests that a block scattered assignment of frontal matrix columns to processors might further improve performance. The blocking factor could be set during preprocessing and based on the particular frontal matrices dimensions and number of assigned processors. Experimentation and analysis could be used to find an optimal function for determining the blocking factor.

Refinements to the scheduling mechanisms are also possible. For instance if the desired sized subcube is not available in the S3 method, a smaller available subcube could be used if it would not extend the frontal matrix task's completion time. Moreover, the current implementation uses critical path priorities based on the frontal matrix tasks' sequential weights. Perhaps it would be better to use the tasks' parallel times or parallel times multiplied by the number of assigned processors for node weights in the critical path determination.

Furthermore, all or part of the host preprocessing could be made to run in a distributed and parallel manner, although much of this processing requires centralized data, which leads to the last and most important suggestion for future work: shared memory implementations.

The parallel refactorization implementation of this effort was accomplished on a distributed memory environment for a number of valid reasons such the explicit formulation of parallelism, the use of increased memory size and bandwidth, the scalability of such environments, and the availability of a distributed memory platform. However, as has been mentioned in a number of points in this text, the centralized

data provided in a sequential or parallel shared memory environment could significantly streamline the processing and would likely lead to more efficient and effective implementations, especially for lost pivot recovery. Such an implementation would also preclude the data prepositioning and distributed triangular solve problems mentioned earlier. Thus, a full scale, production-quality parallel refactorization capability with lost pivot recovery would most likely be targeted for a shared memory environment and this is where I most encourage future efforts.

REFERENCES

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

[2] S. Al-Bassam and H. El-Rewini. Processor allocation for hypercubes. *Journal of Parallel and Distributed Computing*, 16:394–401, 1992.

[3] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Computing*, 11:201–221, 1989.

[4] P. R. Amestoy. *Factorization of large unsymmetric sparse matrices based on a multifrontal approach in a multiprocessor environment.* PhD thesis, L'Institut National Polytechnique de Toulouse, Toulouse, France, November 1990.

[5] P. R. Amestoy, M. J. Dayde, and I. S. Duff. Use of Level-3 BLAS kernels in the solution of full and sparse linear equations. In J.-L. Delhaye and E. Gelenbe, editors, *High Performance Computing*, pages 19–31. North-Holland, Amsterdam, 1989.

[6] P. R. Amestoy and I. S. Duff. Efficient and portable implementation of a multifrontal method on a range of MIMD computers. Technical report, CERFACS, 1989.

[7] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputing Applications*, 3:41–59, 1989.

[8] C. P. Arnold, M. I. Parr, and M. B. Dewe. An efficient parallel algorithm for the solution of large sparse linear matrix equations. *IEEE Transactions on Computers*, C-32(3):265–272, 1983.

[9] C. Ashcraft, S. Eisenstat, and J. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal of Scientific and Statistical Computing*, 11:593–599, 1990.

[10] C. Ashcraft, S. Eisenstat, J. Liu, B. Peyton, and A. Sherman. A compute-based implementation of the fan-in sparse distributed factorization scheme. Technical Report ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[11] C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman. A comparison of three column-based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Department of Computer Science, Yale University, New Haven, CT, 1990.

[12] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[13] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. John Wiley and Sons, New York, NY, second edition, 1989.

[14] U. Banerjee, D. Gajski, and D. J. Kuck. Accessing sparse arrays in parallel memories. *Journal of VLSI and Computer Systems*, 1(1):69–100, 1983.

[15] R. Benner, G. Montry, and G. Weigand. Concurrent multifrontal methods: Shared memory, cache, and frontwidth issues. *International Journal of Supercomputing Applications*, 1:26–44, 1987.

[16] Marsha J. Berger and S. H. Bokhari. A partitioning strategy for non–uniform problems on multiprocessors. *IEEE Transactions on Computers*, 35(5):570–580, 1987.

[17] R. D. Berry. An optimal ordering of electronic circuit equations for a sparse matrix solution. *IEEE Transactions on Circuit Theory*, CT-19(1):40–50, Jan 1971.

[18] A Bjork, R. J. Plemmons, and H. Schneider, editors. *Large Scale Matrix Problems*. New York: North-Holland, 1981.

[19] A. Bojanczyk. Complexity of solving linear systems in different models of computation. *SIAM Journal of Numerical Analysis*, 21(3):591–603, June 1984.

[20] Ming-Syan Chen and Kang G. Shin. Processor allocation in an n-cube multiprocessor using gray codes. *IEEE Transactions on Computers*, C-36(12):1396–1407, December 1987.

[21] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[22] E. Chu and A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[23] E. G. Coffman Jr. and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[24] S. D. Conte and Carl de Boor. *Elementary Numerical Analysis*. McGraw-Hill Book Company, New York, NY, 1980.

[25] A. R. Curtis and J. K. Reid. On the automatic scaling of matrices for Gaussian elimination. *Journal of the Institute of Mathematics and its Applications*, 10:118–124, 1972.

[26] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings 24th National Conference of the Association for Computing Machinery*, pages 157–172, New Jersey, 1969. Brandon Press.

[27] G. Davis. Column LU factorization with pivoting on a message passing multiprocessor. *SIAM Journal of Algebra and Discrete Methods*, 7(4):538–550, 1986.

[28] T. A. Davis. Psolve: A concurrent algorithm for solving sparse systems of linear equations. Technical Report CSRD-612, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, 1986.

[29] T. A. Davis. A parallel algorithm for sparse unsymmetric LU factorization. Technical report, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, 1989.

[30] T. A. Davis. Parallel algorithms for the direct solution of sparse linear systems. In I. St. Doltsinis, editor, *Proc. Second World Congress on Computational Mechanics*, pages 960–963, Stuttgart, Germany, Aug. 1990. Int. Assoc. of Computational Mechanics.

[31] T. A. Davis. Performance of an unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report TR-92-014, Comp. and Info. Sci. Dept., Univ. of Florida, Gainesville, FL, May 1992.

[32] T. A. Davis. Users' guide for the unsymmetric-pattern multifrontal package (UMFPACK). Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, Gainesville, FL, June 1993.

[33] T. A. Davis and E. S. Davidson. Pairwise reduction for the direct, parallel solution of sparse unsymmetric sets of linear equations. *IEEE Trans. Comput.*, 37(12):1648–1654, 1988.

[34] T. A. Davis and I. S. Duff. Unsymmetric-pattern multifrontal methods for parallel sparse LU factorization. Technical Report TR-91-023, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1991.

[35] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for parallel sparse LU factorization. Technical Report TR-93-018, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1993.

[36] T. A. Davis and P. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM Journal of Matrix Applications*, 11:383–402, 1990.

[37] Benjamin Dembart and Albert M. Erisman. Hybrid sparse-matrix methods. *IEEE Transactions on Circuit Theory*, CT-20(6):641–649, November 1973.

[38] R. H. Dodds, Jr. and L. A. Lopez. Substructuring in linear and non-linear analysis. *International Journal of Numerical Method for Engineering*, 15:583–597, 1980.

[39] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[40] J. Dongarra and S. Ostrouchov. LAPACK block factorization algorithms on the Intel iPSC/860. Technical Report LAPACK Working Note 24, University of Tennessee, Knoxville, TN, 1990.

[41] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. Technical Report ORNL/TM-12126, Oak Ridge National Laboratory, Oak Ridge, TN, 1992.

[42] I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software*, 7:315–330, 1981.

[43] I. S. Duff. *Sparse Matrices and their Uses.* Academic Press, New York and London, 1981.

[44] I. S. Duff. The solution of nearly symmetric sparse linear systems. In R. Glowinski and J.-L. Lions, editors, *Computing Methods in Applied Sciences and Engineering.* North-Holland, Amsterdam, New York, and London, 1984.

[45] I. S. Duff. Data structures, algorithms and software for sparse matrices. In D. J. Evans, editor, *Sparsity and Its Applications*, pages 1–29. Cambridge, United Kingdom: Cambridge University Press, 1985.

[46] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.

[47] I. S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *Journal of Computational and Applied Mathematics*, 27:229–239, 1989.

[48] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, New York, NY, 1989.

[49] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix problems. *ACM Transactions on Mathematical Software*, 14:1–14, 1989.

[50] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report TR/PA/92/86, Computer Science and Systems Division, Harwell Laboratory, Oxon, U.K., October 1992.

[51] I. S. Duff and L. S. Johnsson. Node orderings and concurrency in structurally-symmetric sparse problems. In Graham F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms, and Applications*, pages 177–189. John Wiley and Sons Ltd., New York, NY, 1989.

[52] I. S. Duff and U. Nowak. On sparse solvers in a stiff integrator of extrapolation type. *IMA Journal of Numerical Analysis*, 7:391–405, 1987.

[53] I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, 5(18):18–35, 1979.

[54] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[55] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric set of linear equations. *SIAM Journal of Scientific and Statistical Computing*, 5(3):633–641, 1984.

[56] S. Eisenstat, M. Heath, C. Henkel, and C. Romine. Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 9(3):589–600, 1988.

[57] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[58] K. Gallivan, W. Jalby, and U. Meier. The use of BLAS3 in linear algebra on a parallel processor with hierarchical memory. *SIAM Journal of Scientific and Statistical Computing*, 8(6), 1987.

[59] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. In R. J. Plemmons, editor, *Parallel Algorithms for Matrix Computations*, pages 1–82. SIAM, Philadelphia, PA, 1990.

[60] Dennis B. Gannon and John van Rosendale. On the impact of communication complexity on the design of parallel numerical algorithms. *IEEE Transactions on Computers*, C-33(12), 1984.

[61] G. A. Geist. Solving finite element problems with parallel multifrontal schemes. In M. T. Heath, editor, *Hypercube Multiprocessors*, pages 656–661. SIAM, Philadelphia, 1987.

[62] G. A. Geist and M. Heath. Matrix factorization on a hypercube. In M. Heath, editor, *Hypercube Multiprocessors 1986*, pages 161–180. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.

[63] G. A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[64] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal of Scientific and Statistical Computing*, 9(4):639–649, 1988.

[65] A. George. *Computer Implementation of the Finite-Element Method.* PhD thesis, Report STAN CS-71-208, Department of Computer Science, Stanford University, Stanford, CA, 1971.

[66] A. George. Nested dissection of regular finite-element mesh. *SIAM Journal of Numerical Analysis*, 10:345–363, 1973.

[67] A. George. Solution of linear systems of equations: Direct methods for finite-element problems. In V. A. Barker, editor, *Sparse Matrix Techniques. Lecture Notes in Mathematics*, page 572. Springer-Verlag, Berlin, Heidelberg, New York, and Tokyo, 1977.

[68] A. George. An automatic one-way dissection algorithm for irregular finite-element problems. *SIAM Journal of Numerical Analysis*, 17:740–751, 1980.

[69] A. George, M. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM Journal of Scientific and Statistical Computing*, 9:327–340, 1988.

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

[71] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communications results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[72] A. George and E. G.-Y. Ng. Parallel sparse Gaussian elimination with partial pivoting. *Annals of Operations Research*, 22:219–240, 1990.

[73] N. E. Gibbs, W. G. Poole, Jr., and P. K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal of Numerical Analysis*, 13:236–250, 1976.

[74] J. Gilbert and H. Hafsteinsson. Parallel symbolic factorization of sparse linear systems. *Parallel Computing*, 14:151–162, 1990.

[75] J. Gilbert and J. Liu. Elimination structures for unsymmetric sparse LU factors. Technical Report CS-90-11, Department of Computer Science, York University, Ontario, Canada, 1991.

[76] J. R. Gilbert. An efficient parallel sparse partial pivoting algorithm. Technical Report CMI No. 88/45052-1, Christian Michelsen Institute, 1988.

[77] J. R. Gilbert and R. Schreiber. Highly parallel sparse cholesky factorization. *SIAM J. Sci. Comput.*, 13(5):1151–1172, 1992.

[78] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD and London, UK, second edition, 1990.

[79] J. Gustafson, G. Montry, and R. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal of Scientific and Statistical Computing*, 9:609–638, 1988.

[80] F. G. Gustavson. Finding the block lower-triangular form of a sparse matrix. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 275–289. Academic Press, New York and London, 1976.

[81] Gary D. Hachtel. Vector and matrix variability type in sparse matrix algorithms. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 53–64. Plenum Press, New York, NY, 1972.

[82] Gary D. Hachtel, Robert K. Brayton, and Fred G. Gustavson. The sparse tableau approach to network analysis and design. *IEEE Transactions on Circuit Theory*, CT-18(1):101–113, January 1971.

[83] S. M. Hadfield and T. A. Davis. Analysis of potential parallel implementation of the unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report TR-92-017, Department of Computer and Information Systems, University of Florida, Gainesville, FL, 1992.

[84] William W. Hager. *Applied Numerical Linear Algebra*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[85] M. Heath and C. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 9(3):558–588, 1988.

[86] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. In R. J. Plemmons, editor, *Parallel Algorithms for Matrix Computations*, pages 83–124. SIAM, Philadelphia, PA, 1990.

[87] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[88] L. Hulbert and E. Zmijewski. Limiting communications in parallel sparse Cholesky factorization. *SIAM Journal of Scientific and Statistical Computing*, 12(5):1184–1197, 1991.

[89] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, 1989.

[90] K. R. Jackson and W. L. Seward. Adaptive linear equation solvers in codes for large stiff systems of ODEs. *SIAM J. Sci. Comput.*, 14(4):800–823, July 1993.

[91] J. A. G. Jess and H. G. M. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, 31:231–239, 1982.

[92] S. Lenhart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[93] L. Kantorovich. *Functional Analysis in Normed Spaces*. Pergamon Press, Oxford, U.K., 1964.

[94] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11), 1984.

[95] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8:623–625, October 1965.

[96] C. P. Kruskal, L. Rudolph, and M. Snir. Techniques for parallel manipulation of sparse matrices. *Theoretical Computer Science*, 64:135–157, 1989.

[97] K. S. Kundert. Sparse matrix techniques and their applications to circuit simulation. In A. E. Ruehli, editor, *Circuit Analysis, Simulation and Design*. New York: North-Holland, 1986.

[98] Th. Lengauer and Ch. Wieners. Efficient solutions of hierarchical systems of linear equations. *Computing*, 39:111–132, 1987.

[99] G. Li and T. Coleman. A parallel triangular solver for a distributed-memory multiprocessor. *SIAM Journal of Scientific and Statistical Computing*, 9(3):485–502, 1988.

[100] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. *SIAM J. Sci. Comput.*, 14(6):1259–1288, November 1993.

[101] J. W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.

[102] J. W. H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[103] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:249–264, 1986.

[104] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

[105] R. Lucas, T. Blank, and J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):981–991, November 1987.

[106] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, April 1957.

[107] Willard L. Miranker. *Numerical Methods for Stiff Equations*. D. Reidel Publishing Company, Dordrecht, Holland; Boston, USA; London, England, 1981.

[108] M. Nakhla, K. Singhal, and J. Vlach. An optimal pivoting order for the solution of sparse systems of equations. *IEEE Transactions on Circuits and Systems*, CAS-21(2):222–225, Mar 1974.

[109] nCUBE. *nCUBE 2 Processor Manual*. Foster City, CA, 1992.

[110] nCUBE. *nCUBE 2 Programmer's Guide*. Foster City, CA, 1992.

[111] nCUBE. *nCUBE 2 Programmers Reference Manual*. Foster City, CA, 1992.

[112] nCUBE. *nCUBE 2 Systems: Technical Overview*. Foster City, CA, 1992.

[113] nCUBE. ndoc on-line documentation facility, 1992.

[114] R. S. Norin and C. Pottle. Effective ordering of sparse matrices arising from nonlinear electrical networks. *IEEE Transactions on Circuit Theory*, CT-18:139–145, Jan 1971.

[115] G. J. Nutt. *Centralized and Distributed Operating Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[116] Dianne P. O'Leary and G. W. Stewart. Assignment and scheduling in parallel matrix factorization. *Linear Algebra and its Applications*, pages 275–299, 1986.

[117] L. S. Ostrouchov, M. T. Heath, and C. H. Romine. Modeling speedup in parallel sparse matrix factorization. Technical Report ORNL/TM-11786, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[118] S. V. Parter. The use of linear graphs in Gaussian elimination. *SIAM Review*, 3:119–130, 1961.

[119] A. Pothen and C. Sun. A distributed multifrontal algorithm using clique trees. Technical Report CS-91-24, Department of Computer Science, Pennsylvania State University, University Park, PA, 1991.

[120] A. Pothen and C. Sun. A mapping algorithm for parallel sparse cholesky factorization. *SIAM J. Sci. Comput.*, 14(5):1253–1257, September 1993.

[121] Roldan Pozo. Performance modeling of sparse matrix methods for distributed memory architectures. In *CONPAR '92, VAPP-V*, Philadelphia, PA, September 1992. SIAM.

[122] Roldan Pozo and Sharon Smith. Performance evaluation of the parallel multifrontal method in a distributed memory environment. In *SIAM 6th Conference of Parallel Processing for Scientific Computing*. SIAM, March 1993.

[123] M. Prastein. Precedence-constrained scheduling with minimum time and communications. Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[124] Douglas Quinney. *An Introduction to the Numerical Solution of Differential Equations*. Research Studies Press, Ltd, Letchworth, Hertfordshire, England, revised edition, 1987.

[125] V. J. Rayward-Smith. UET scheduling with interprocessor communication delays. Technical Report SYS-C86-06, University of East Anglia, Norwich, U.K., 1986.

[126] J. K. Reid. TREESOLVE. A Fortran package for solving linear sets of linear finite-element equations. Technical Report Report CSS 155, Computer Science and Systems Division, Harwell Laboratory, Oxon, U.K., 1984.

[127] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *Proc. 7th Annual Symposium on the Theory of Computing*, pages 245–254, 1975.

[128] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34(1):176–197, 1978.

[129] Youcef Saad. Communication complexity of Gaussian elimination algorithm on multiprocessors. *Linear Algebra and its Applications*, 77:315–340, 1986.

[130] Youcef Saad and Martin H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

[131] R. Saleh, K. Gallivan, M. Chang, I. Hajj, D. Smart, and T. Trick. Parallel circuit simulation on supercomputers. *Proc. IEEE*, 77(12):1915–1931, 1989.

[132] N. Sato and W. F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Transactions on Power*, PAS-82:944–949, 1963.

[133] B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, 10:222–232, 1990.

[134] Mandayam A. Srinivas. Optimal parallel scheduling of Gaussian elimination. *IEEE Transactions on Computers*, C-32(12):1109–1117, December 1983.

[135] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing, Co., Reading, MA, second edition, 1990.

[136] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of IEEE*, 55:1801–1809, 1967.

[137] J. D. Ullman. NP-Complete scheduling problems. *Journal of Computer System Science*, 10:384–393, 1975.

[138] J. H. Wilkinson. Error analysis of direct methods of matrix inversion. *Journal of the ACM*, 8:281–330, 1961.

[139] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford Press, London, 1965.

[140] Omar Wing and John W. Huang. A computation model of parallel solution of linear equations. *IEEE Transactions on Computers*, C-29(7):632–638, 1980.

[141] M. Yannakakis. Computing the minimum fill-in is NP-Complete. *SIAM Journal of Algebra and Discrete Methods*, 2:77–79, 1981.

[142] Z. Yin, C. Chui, R. Shu, and K. Huang. Two precedence-related task-scheduling algorithms. *International Journal of High Speed Computing*, 3:223–240, 1991.

[143] S. E. Zitney. A frontal code for ASPEN PLUS on advanced architecture computers. In *Proc. AIChE Annual Meeting, Symposium on Parallel Computing*, Chicago, IL, 1990. American Institute of Chemical Engineers.

[144] S. E. Zitney and M. A. Stadtherr. A frontal algorithm for equation-based chemical process flowsheeting on vector and parallel computers. In *Proc. AIChE Annual Meeting*, Washington, DC, 1988. American Institute of Chemical Engineers.

[145] Z. Zlatev. On some pivotal strategies in Gaussian elimination by sparse technique. *SIAM Journal of Numerical Analysis*, 17:18–30, 1980.

[146] Z. Zlatev. Sparse matrix techniques for general matrices with real elements: Pivotal strategies, decompositions and applications in ODE software. In D. J. Evans, editor, *Sparsity and Its Applications*, pages 185–228. Cambridge, United Kingdom: Cambridge University Press, 1985.

[147] Zahari Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, Dordrecht, Boston, London, 1991.

[148] E. Zmijewski. Limiting communications in parallel sparse Cholesky factorization. Technical Report TRCS89-18, Department of Computer Science, University of California, Santa Barbara, CA, 1989.

BIOGRAPHICAL SKETCH

Steven M. Hadfield was born in Milwaukee, WI, on March 6, 1959 and grew up in Clearwater, FL, where he graduated with honors from Clearwater Central Catholic High School in 1977. He received a B.S. degree from Tulane University in 1981 with majors in mathematics and economics and a minor in computer science. In 1982, he received an M.S. degree in computer science from the Air Force Institute of Technology. As an officer in the U.S. Air Force since 1981, he has served in the Pentagon (from 1982 through 1985) and with Space Command (from 1986 to 1989) specializing in communications software and real-time distributed systems. From 1989 to 1991, he was an instructor and course director in the Department of Mathematical Sciences, U.S. Air Force Academy. He currently holds the rank of major and is working towards a Ph.D. in computer science at the University of Florida. He is married to the former Marissa F. Gomez and they have three children, Melissa, Andrew, and Christopher.