MULTIPLE-RANK UPDATES TO MATRIX FACTORIZATIONS FOR NONLINEAR ANALYSIS AND CIRCUIT DESIGN

A DISSERTATION
SUBMITTED TO THE INSTITUTE FOR
COMPUTATIONAL AND MATHEMATICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Linzhong Deng
June 2010

Abstract

For the numerical solution of ODE/PDEs that describe the basic laws of science, such as Kirchhoff's laws in circuit analysis, or nonlinear deformation equations in structural analysis, a series of linear equations $Ax = b$ are generated to represent the basic laws. The size of A could reach 10^{10} in full-chip verification, or 10^9 in structure-foundation seismic simulation. The traditional method is to solve these linear equations independently, which leads to a computational bottleneck when we simulate large and complicated objects because more than 95% of the simulation time is consumed in solving the equations.

The aim of this thesis is to address the challenge of the computational bottleneck by modifying the matrix A's factorization at each simulation step based on the similarities between two consecutive linear systems. To achieve this aim, we define the similarities as an initial matrix A with multiple-rank corrections. Then we design algorithms to solve the linear systems based on updating concepts.

The basic algorithm proposed in this study is a rank-1 update of the factors of an unsymmetric matrix. To solve the well known instability problem, new algorithms are designed to implement partial or full pivoting strategies for the update procedure.

To improve performance and overcome parallelization difficulties during traditional factorization, a pipeline is designed to do multiple-rank updates of the matrix factors. We prove two theorems for this parallel computing. One states that the multiple-rank update sequences are independent. The other, called the "base camp" theorem, describes a multiple-rank merge rule to reduce the computing effort. A divide-and-conquer algorithm for sparse matrix factorization is also described.

Finally, we apply the multiple-rank modification of matrix factorization for nonlinear analysis in structural engineering and functionality verification in circuit design. The same process can be applied in other time-domain problems and nonlinear systems, including fluid dynamics, aerospace, chemical processing, structure analysis, graphics rendering, MEMS, seismic, biotech, and electromagnetic field analyses.

Acknowledgements

GG? Does it mean God's Gift or God Given? Or does it just mean Gene Golub? When this question was discussed in a SCCM lunch seminar, people found many explanations. To me, GG means Prof Gene Golub, who is a Gift from God. Gene is godfather of numerical analysis. His contributions in computer science and applied mathematics are far ahead of others. I was very fortunate to have him as my advisor. According to Chinese philosophy ("Da En Bu Yan Xie", which means you should not say thanks to a person who gave you immense help because you cannot pay him back for his invaluable support), I should speak with actions rather than words. But I still want to present my thanks from the deepest part of my heart. It is he who guided me into this wonderful numerical world. There are many sweet stories between us. We went to every Chinese spicy restaurant in the Bay Area. We found and reviewed the computational core in engineering subjects. Gene was sharp in engineering application and supported my crazy ideas. I felt his love when he said he missed his students when he was away from The Farm. I especially thank him for his support when I wrote the very original NSF proposal entitled "Digital Representation of Structural Response for the Reliability Assessment of Complex Systems". This project finally received full NSF support (grant CMS-0218594) in 2003.

After Gene's death, I was down for quite a while. Prof Michael Saunders gave me a lot of encouragement, and helped me overcome this difficult time because Gene had become the dearest person in my life. Michael then kindly stepped in to fill Gene's giant shoes (1.5 times those of mine), to hold the torch that Gene lit in Stanford and in the world, and to adopt me as his student. Michael is a great advisor inside and outside of school. His guidance and help make me feel at home. His marvelous job let me know the beauty of accuracy in math, and find the joy in research. I knew Michael long before he became my advisor. He helped me in a "Property A" problem in memory simulation, in proposal preparation, and much other writing. When publisher VDM Verlag invited me to write a book "Artificial Intelligence Techniques for Bridge Reliability Assessment: Digital Representation of Structure Response and Safety" in 2009, I really appreciated Michael's support to make this book reach the shelf. When I founded my company Revosys Inc with my partner Bob Xu to do IC yield analysis, Michael gave me huge support in matrix analysis. Without him, my life would be different. Especially this thesis: it would have been impossible without his amazing depth and breadth of knowledge and untiring enthusiasm. The difference between Michael and Gene is that Gene liked spicy food while Michael likes steak and lamb.

I owe special thanks to Prof Walter Murray. I know Walter from my third day at Stanford. My deep impression is that he wrote a formula $\min f$ on a piece of paper instead of fancy computer slides during the student orientation to introduce optimization. It is marvelous that many people devote their whole life to these four letters $\min f$. His leadership guarantees that our study at Stanford proceeds smoothly. His knowledge and connection to industry let me know the solution of the toughest problems in the world. His email written at 2:00am kept me working till 3:00am. He watched me grow up in SCCM and ICME. He is demanding but very nice.

I also want to present my thanks to Dr Zhihong Liu. Dr Liu is a vice president of Cadence Design Systems Inc, and a pioneer in circuit design. He shared with me his academic and industrial experience. I consulted him on the simulation in this work. He taught me strategy and leadership in start-ups. I am so proud to have such a friendly, knowledgeable, and capable person to lead my way.

Special thanks go to Prof Yinyu Ye and Prof Simon Wong, my committee members. They have patience to read and support my work. Their personal characteristics light my way.

I owe many thanks to Indira Choudhury, ICME's "governor". After Gene's death, she gave me an email that said "I hope you are coming to terms with his death. I cannot imagine how profoundly this has affected you. ... You are his legacy and your life will be a testament to his efforts and brilliance. He has passed the torch to all of you and now it is your turn to lead the way." Thank you, dear Indira. Your words made me feel better and understand my responsibility. I am so proud to have you as our "governor". You made it easy for me to focus on research.

Let me continue my thanks to my research collaborators and friends. Without them, it would be difficult to imagine my research and life. It is hard to acknowledge all of them here. Some of them are Ray Sun, Anwei Chai, Zhisu Zhu, and Sou-Cheng Choi.

When my company Revosys merged with Proplus Design Solutions Inc in 2008, we started to schedule product release of a circuit design system. Although I work fulltime in ProPlus, Proplus supported my Stanford research by giving me time and resources. My special thanks go to Proplus CEO James Ma, CTO Bruce McGaughy, and COO Bob Xu for their invaluable technical and financial support.

During the final year, I have been deeply grateful for the love and friendship that Ms. Martina Ma brought to me. Her sunny smile and cheerful laugh raised my spirits and enabled me to complete this research. Her ribeye steaks helped compose many key sentences.

Finally, I would like to present my unconditional gratitude to my parents for their support and encouragement during my study period in the USA.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Solving a linear system $Ax = b$ is an ancient topic. Leibniz did research on determinants in 1693, and Gauss introduced the elimination method to solve linear equations in 1820. Although matrices have a long history of application in solving linear systems, the advent of the computer led to rapid development of this field and to modern matrix theory. Research in linear algebra has become the center of scientific computing today to satisfy academic and industrial requirements to solve more complicated systems with more accurate models by using the exponentially increasing power of the computer.

The core of numerical linear algebra is the algorithms for solving the linear systems that arise frequently in engineering and scientific computing [GVL96]. Linear system algorithms can be divided into two categories: direct methods and iterative methods. The focus of research changes with hardware progress. Twenty years ago, CPU speed was the critical factor. Ten years ago when memory was a bottleneck, iterative methods received more attention. As breakthroughs in the design of memory chips partly remove the bottleneck today, direct methods are now favored by researchers and engineers for their accuracy and robustness. In this study, we propose a new direct linear solver based on *multiple-rank updates* to achieve better performance and use memory in a more efficient way. Applications of this solver in nonlinear analysis and circuit design are presented.

1.1 Factorization methods for $Ax = b$

Direct methods can be divided into left-looking methods and right-looking methods [Dav06]. Methods for solving structured linear equations, such as Toeplitz and Fourier systems, are beyond our discussion [BP94].

A left-looking method is one that calculates factors L and U one column at a time. At step k, the method accesses only columns 1 to k of L, and the kth column of the original matrix. A right-looking method subtracts an outer product of the kth column of L and the kth row of U from the lower-right submatrix of the remaining matrix and gets the next column of L and the next row of U from the updated submatrix. Many techniques such as the BLAS level 1, 2 and 3 have been introduced to obtain better accuracy, capacity, and performance [Dem97]. Parallelism is the focus of recent developments in favor of computers' multi-core architecture. In general, left-looking methods have an advantage in memory access, while right-looking methods have an advantage in parallel computing.

There are two types of matrices: dense and sparse. Both left- and right-looking methods can be used to factorize these two types of matrices, but the rules for choosing a factorization method for a given matrix vary—they depend on the computer architecture, matrix properties, and customer accuracy requirements. In general, the computation effort for dense matrix factorization is $O(n^3)$ floating-point operations, and the computation effort for a sparse matrix is $O(n^{1.05})$–$O(n^2)$. Ordering strategies and stability rules are critical in solving a sparse linear system [DER86]. Much research has been done on strategies to reduce fill-ins. Finding a good (fill-reducing) ordering is essential for reducing time and memory requirements for sparse factorization methods [Dav06].

Many successful factorization algorithms have been developed based on left-looking or right-looking methods. However, they are all quite expensive for linear systems of high dimension. Although computation power has increased substantially in recent years, the computational requirement also increases dramatically when more accurate models are introduced or more complicated systems need to be simulated. In EDA (Electronic Design Automation), the problem size for simulation has increased exponentially according to Moore's law (doubling every two years). It is not unusual for a test case to run for a month. How to achieve good performance without losing accuracy has become an important topic in academics and engineering. Much research has been done on the algebraic or geometric properties of a matrix to improve performance. Our research addresses this subject by considering latency—an intrinsic physical phenomenon in engineering applications.

1.2 Stability of factorizations

Both left-looking and right-looking factorization methods may have stability difficulties when the matrix is not positive definite. To obtain acceptable performance and stability, various strategies have been introduced such as threshold partial pivoting [Mar57] and threshold rook pivoting and threshold complete pivoting [GMS05]. These strategies seek permutations of the rows and columns of the matrix that lead to sparse factors while preserving adequate stability. Practical application has shown these strategies to be effective.

The standard pivot strategies ensure that $|L_{ij}| \leq L_{\max}$ when we process column j, for some stability parameter $L_{\max}$. To favor stability, $L_{\max}$ could be set to 10 or 5 or 2 [Sau09]. To permit greater sparsity, $L_{\max}$ is often chosen to be 1000. A positive-definite matrix does not need any such control because its factorization step is stable. (The row/column permutation can be chosen purely to enhance sparsity.)

Stability has received more attention in large and complex simulations, where the highly nonlinear equations are very sensitive to parameter changes and have high condition number. Refinement processes are used to obtain a relatively reliable solution. In this thesis we also devote much effort to the stability concerns.

1.3 Sequences of linear systems

One of the most effective ways to improve performance is to consider properties of the linear system that depend on the application, discretization methods, and simulation processes. The basic laws of science such as Maxwell's equation, Kirchhoff's laws, and Newton's second law of motion, are described by differential equations. Discretization of the associated ODEs or PDEs inside a practical problem leads to a series of linear equations. Their matrix properties such as symmetry, positive definiteness, data structure, sparsity, and scaling are considered in the design of an algorithm for solving each linear system.

Traditionally, the series of linear systems is solved one by one by a direct or iterative method without consideration of the relationship between consecutive systems, or between the base system and later systems. However, from physical facts and the law of continuity, we know that the present simulation step evolved from the preceding step. This clone relationship should be used in the simulation procedure to achieve accuracy and robustness (no core dump!). It should also be helpful in discontinuous problems and highly nonlinear problems. The following chapters will discuss this again in more detail.

1.4 Updating review

How to model this clone relationship efficiently and stably has received much attention from mathematicians and engineers. Bennett proposed an LU updating strategy for the solution of modified matrices [Ben65], and Gill, Golub, Murray, and Saunders published a landmark paper about matrix factorization updating methods [GM74]. These two papers led to an entire field of research that is still going on within scientific computing.

For example, Law's sparse updating method [Law85, Law89] and the method of Chan and Brandwajn [CB86] are based on Bennett's strategy. For simplex-type active-set optimization methods, Bartels and Golub [BG69] showed how to update LU factors with L kept in product form and U updated explicitly. Reid [Bar71] showed how to implement this method efficiently in the sparse case. Fletcher and Matthews [FM84] gave an LU update that keeps both L and U triangular. Davis and Hager [DH99] implemented a sparse version of the symmetric updating method of Gill, Golub, Murray, and Saunders [GM74] based on manipulation of the underlying graph structure. Makode, Corotis and Ramirez [MR99] proposed the Pseudodistortion method, which finds a right-hand side b_1 such that solving $Ax = b_1$ is equivalent to solving $(A + \Delta A)x = b$. Deng and Ghosn [DG01] extended this method to general engineering applications.

Most of the updating methods have some limitations on their stability, or require considerable computation effort, or only apply to symmetric systems, or need additional physical information.

Multiple-rank updates require similar care. They can use memory more efficiently, and have the same degree of stability as rank-1 updates.

1.5 A new approach

Based on previous work in research and industrial applications, this thesis develops dense and sparse linear solvers and applies them in circuit simulation and structural analysis. The aim is to eliminate the computational bottleneck by developing update procedures for the matrix factors at each simulation step. A certain clone relationship exists between two consecutive linear systems. We represent this relationship as a multiple-rank update. The problem is expressed in general as

$$\text{Given } A = LU, \text{ solve } (A + \Delta A)x = b,$$

where A is $n \times n$ and ΔA has low rank relative to n. The whole simulation flow can be built on the above equation by recursively forming matrices A, L, and U. If ΔA has rank r, it can be decomposed as the summation of r rank-1 matrices:

$$\Delta A = \sum_{j=1}^{r} u_j v_j^T.$$

The basic algorithm proposed in this study is the rank-1 update of L and U factors of an unsymmetric matrix. To ensure stability, we propose algorithms to implement partial or full pivoting strategies in such a way that we can transform the "interchange-for-stability problem" into two rank-1 update problems. In other words, we solve the stability problem by regarding a row interchange as two rank-1 updates.

1.6 Parallelization

One attractive property of multi-rank updates is that the data flow is suitable for parallel computing. To obtain good performance and achieve parallelism, we design a pipeline for the updates. We also prove two theorems for parallel multiple rank-1 computation:

1. One theorem states that a sequence of rank-1 updates can be implemented independently in different columns or in different stages of a pipeline. Both the result and the computational effort are the same for any order.

2. The other so-called "base-camp" theorem states that a large group of rank-1 updates can be merged (coalesced or condensed) into a small group of rank-1 updates to reduce computation. This theorem makes multiple sparse-matrix updates practical.

Later, a divide-and-conquer algorithm for sparse matrix factorization is discussed, which may lead to fully parallelized LU factorization.

1.7 Applications

Finally, we apply the multiple-rank factorization updates to nonlinear analysis in structural engineering and functional verification in circuit design. Also they can be applied to other time-domain and nonlinear systems, including fluid dynamics, aerospace, chemical processing, structural analysis, graphics rendering, MEMS, seismology, biotech, and electromagnetic field analyses.

1.8 Overview

Chapter 2 introduces the basic numerical algorithms. Their properties, including stability, robustness, capacity, parallelism, and high performance are introduced in Chapter 3. Chapter 4 gives algorithms for updating sparse matrix factors. Chapter 5 discusses the symmetric updating problem. Chapter 6 gives some application examples to show the advantages of our proposed method, and Chapter 7 gives a summary and looks ahead to future work.

Chapter 2

Matrices at Work

Matrices exist in every corner of engineering and science practice. They are at the core of numerical analysis. When we analyze a new problem, one of the best ways to proceed is to express it in matrix form. Numerical linear algebra algorithms can be applied to the problem by finding the solution through scientific computation.

The matrix is a vital invention to help describe and solve massive and complex problems. Circuit simulation, structure analysis, and the optimization of jet engine design are typical examples. The matrix is also an effective tool to store, process, and view data. For example, Google regularly builds a linkage matrix to store the data contained in billions of web pages, and interprets this data using an eigenvector of the matrix.

Matrix computation can be time-consuming. One would need days of work to invert a general 10×10 matrix by hand. Before automatic computing devices were invented, people believed matrix theorems were not useful in application because of the associated unacceptable computational effort. The advent of computers led to modern numerical linear algebra, and the advances there helped the development of computer science. It is the computer that is used to solve huge problems expressed in matrices. When the problem is large and complicated, such as in global weather forecasting, we need a computer having high speed and immense storage. When computing hardware and operating systems reach the required power as electrical engineering and computer science advance, we ask for more power to solve even bigger problems using more accurate models and implementing more features to understand the investigated problem in more detail. These processes form the development trajectories of computer science and matrix algebra in the past 60 years.

The core of linear algebra is matrix computation. Why do matrices exist in almost all fields? How do we form a matrix in a practical problem? How do we use the resulting matrix to find a solution? This chapter will discuss these problems in general. Please refer to [Bat96, ZT05, GKO95] for details.

2.1 Matrix formation

Many of the basic laws of science are expressed as ODE/PDEs [Hea02], including:

- Newton's second law of motion
- Kirchhoff's laws
- Heat equation
- Maxwell's equation
- Navier-Stokes equations
- Linear elasticity equations
- Schrödinger's equations
- Einstein's equations.

These ODE/PDEs describe the intrinsic properties in nature, engineering, physics, and more. In some simple conceptual models, we may obtain their solutions in analytical form. In real application problems, numerical methods are the only means for obtaining a solution [GKO95, OF02]. The common numerical methods include modified nodal analysis (MNA), finite element methods (FEM), difference methods, boundary element methods (BEM), and linear programming (LP). The following sections summarize the numerical analysis processes of MNA in circuit simulation and FEM in structural analysis that illustrate the basic numerical procedures.

2.1.1 Circuit simulation

Suppose there is a circuit with n nodes. Define v_i to be the voltage of node i, where $i = 1, 2, \ldots, n$. Let $v = [v_1, v_2, \ldots, v_n]^T$ be a vector to represent the voltages at all nodes, and $t \in [0, t_{\max}]$ be the instantaneous simulation time up to the stopping point $t_{\max}$. Let us define $Q_i(v,t)$, $f_i(v,t)$, and $u_i(t)$ to be the charge, the conductance current, and the current source at node i for $i = 1, 2, \ldots, n$. According to Kirchhoff's laws, the summation of all branch currents at node i is equal to zero:

$$\frac{dQ_i(v,t)}{dt} + f_i(v,t) + u_i = 0. \tag{2.1}$$

Let

$$Q = [Q_1, Q_2, \ldots, Q_n]^T$$

be a vector to represent the charge at every node,

$$f = [f_1, f_2, \ldots, f_n]^T$$

be a vector to represent the conductance current at every node, and

$$u = [u_1, u_2, \ldots, u_n]^T$$

be a vector to represent the known current source at every node. The n equations arising from the application of Kirchhoff's current law for the n nodes can be expressed as

$$\frac{dQ}{dt} + f(v,t) + u = 0. \tag{2.2}$$

Equation (2.2) is the basic ODE for all big signal simulations in circuit design. Management and optimization of timing, power, and reliability in nanometer design require its exact solution because the traditional timing, power, and reliability analysis methods handle new features in the nanometer era inappropriately in IC design practice. Equation (2.2) must be solved numerically because of its stiffness and complex boundary conditions (and because of nonlinearity in the device), and it must be solved to high accuracy for the solution to be useful.

When we use the Backward Euler method to discretize (2.2), we have

$$\frac{Q^{t+1} - Q^t}{\Delta t} + f(v^{t+1}, t^{t+1}) + u^{t+1} = 0 \tag{2.3}$$

at each timestep t. Equation (2.3) is highly nonlinear. The Newton-Raphson method is an ideal tool to solve it. Define

$$F(v,t) \equiv \frac{Q^{t+1} - Q^t}{\Delta t} + f(v^{t+1}, t^{t+1}) + u^{t+1} = 0. \tag{2.4}$$

Its derivative is

$$\begin{aligned} \frac{dF}{dv} &= \frac{1}{\Delta t}\frac{dQ^{t+1}}{dv} + \frac{df(v^{t+1}, t^{t+1})}{dv} \\ &= \frac{1}{\Delta t}C + G, \text{ where } C = \frac{dQ^{t+1}}{\Delta v} \text{ and } G = \frac{df(v^{t+1}, t^{t+1})}{dv}. \end{aligned} \tag{2.5}$$

Applying the Newton-Raphson method to solve (2.4), we have

$$\frac{dF(v,t)}{dv}\Delta v = -F(v,t). \tag{2.6}$$

Substituting $\frac{dF(v,t)}{dt}$ from (2.5) into (2.6), we have

$$\left(G + \frac{1}{\Delta t}C\right)\Delta v = -\left(\frac{Q^{t+1} - Q^t}{\Delta t} + f(v^{t+1}, t^{t+1}) + u^{t+1}\right). \tag{2.7}$$

Alternatively, we can solve (2.2) by the classical trapezoidal method:

$$\left(G + \frac{2}{\Delta t}C\right)\Delta v = -\left(\frac{2}{\Delta t}(Q^{t+1} - Q^t) - \frac{dQ(t)}{dt} + f(v^{t+1}, t^{t+1}) + u^{t+1}\right). \tag{2.8}$$

In general, each Newton-Raphson iteration solves a linear equation $Ax = b$, where

$$\begin{aligned} A &= G + \frac{1}{\Delta t} C, \\ b &= -\left(\frac{Q^{t+1} - Q^t}{\Delta t} + f(v^{t+1}, t^{t+1}) + u^{t+1} \right), \\ x &= \Delta v \end{aligned} \tag{2.9}$$

for the Backward Euler method, and

$$\begin{aligned} A &= G + \frac{2}{\Delta t} C, \\ b &= -\left(\frac{2}{\Delta t}(Q^{t+1} - Q^t) - \frac{dQ(t)}{dt} + f(v^{t+1}, t^{t+1}) + u^{t+1} \right), \\ x &= \Delta v \end{aligned} \tag{2.10}$$

for the trapezoidal method. In both cases, $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. The dimension n could be more than 10^9. For example, the Intel I7-920 processor has 731 million MOSes (transistors). Every MOS has 4–9 nodes, and the whole chip will have around 10^9 nodes. Verifying the I7's functionality by simulation is clearly a major challenge.

In today's technology, circuits are operated at 10GHz frequency, which means Δt should be less than $\frac{1}{10G} = 10^{-10}$ seconds to verify its functionality. If $t_{\max} = 10^{-6}$, we need at least $\frac{10^{-6}}{10^{-10}} = 10^4$ timesteps to finish the simulation. If a Newton-Raphson step needs 2 iterations to converge on average, we need to solve $Ax = b$ 20,000 times (a very conservative estimate). It is common for a circuit to need multiple millions of timesteps or to solve $Ax = b$ multiple millions of times for its basic functionality verification.

Abstractly, the circuit simulation procedure solves a series of linear systems

$$A^{(i)} x^{(i)} = b^{(i)}, \quad i = 1, 2, \ldots, n_t \tag{2.11}$$

in which n_t is the total number of times to solve $Ax = b$. $A^{(i)} \in \mathbb{R}^{n \times n}$ defines the matrix at step i. We will refer to $A^{(i)}$ and $b^{(i)}$ as A and b in the following sections and chapters.

Solving even one of the systems of linear equations is time-consuming. Solving all equations (2.11) will consume most of the simulation time. This forms the bottleneck in circuit simulation.

It is not practical to solve equations (2.11) directly when $n > 10^6$. In industry, a direct solver is used when $n < 50,000$, and a physically or numerically approximate method when $n < 10$ million. There are not many success stories about solving (2.11) when $n > 10$ million.

2.2 Hazard analysis

Structure safety is a serious public issue that has received much attention recently. For example, events like

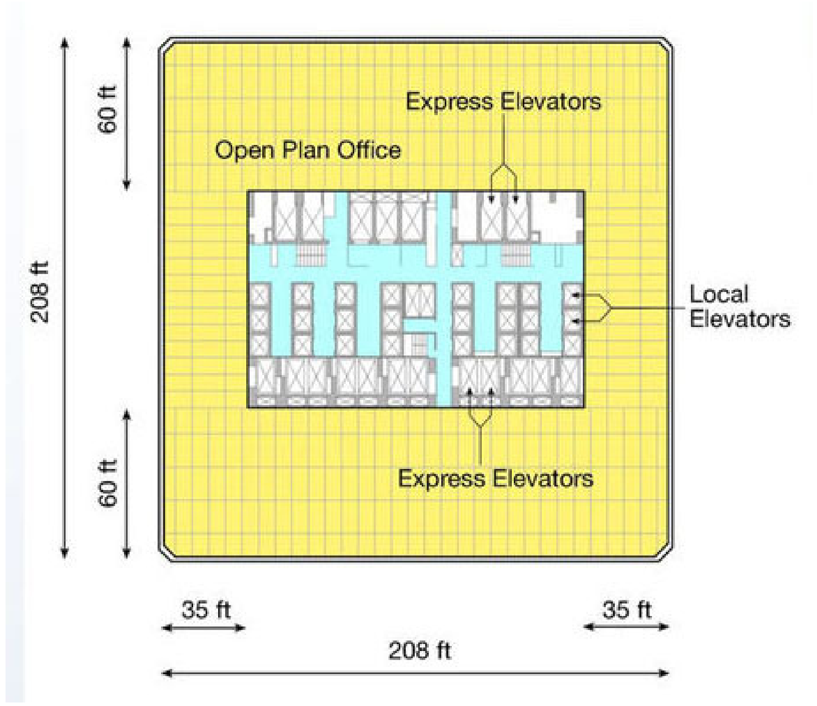


Figure 2.1: World Trade Center front view.

- a bridge collapsing after an unexpected accident,
- a high-rise building resisting sudden hazard forces such as seismic load or terrorist attack,

need serious consideration because of their deep effect on our society.

What is the redundancy in a structure to ensure that it can safely resist the impact of some accident? How can we design a bridge with enough redundancy to resist the impact caused by an accident like a fully loaded ship colliding with a pier of the bridge? How can we design a building that remains standing after a terrorist attack such as on the World Trade Center (WTC). All these questions can be answered partially by structural simulation. This is another example where linear algebra forms the core in design and analysis.

Immediately after the 9/11 attack, Prof. Gene Golub talked to me about the public safety problems in our society. He believed the collapsing phenomenon is a typical implosive process. If we can simulate this process and understand the change of physical states, we will have the ability to prevent such a tragedy happening again. In memory of him, we present a prototype for FEM simulation of the WTC collapse.

WTC had a tube-in-tube structure. The inner tube was formed by the elevator wall. The outer tube was formed by the dense columns around its outside wall. The inner tube and outer tube were connected by trusses and beams at every floor to form an innovative skeleton to bear the vertical loads such as the building's weight and to resist horizontal forces such as wind and seismic forces.

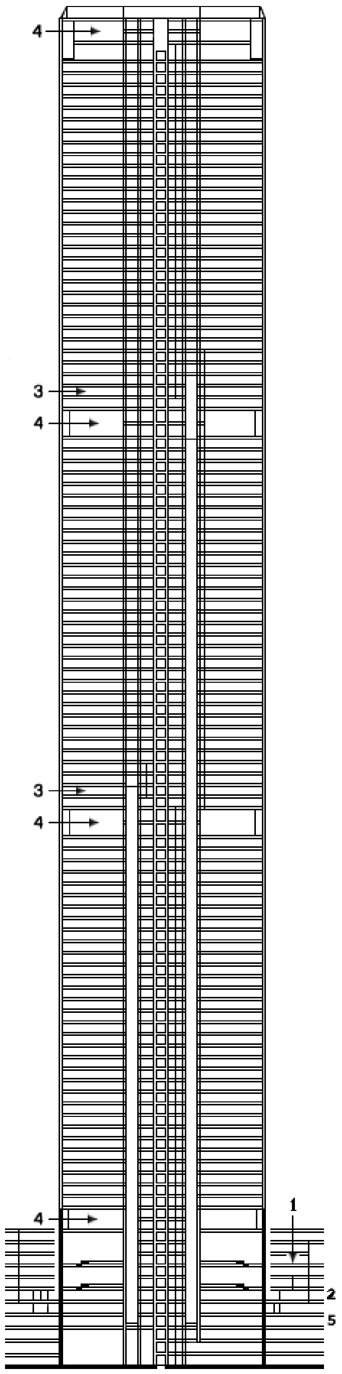


Figure 2.2: World Trade Center floor plan.

Figures 2.1 and 2.2 [God85, Wik05] show WTC's analysis models of the floor and vertical columns.

To understand the implosive process that happened in WTC, we need to investigate the damage and collapse trajectory, including

- the structure damage due to the high-speed impact of the Boeing 767 airplane;
- the fire damage and decrease in the structure capacities after the aircraft broke into fragments;
- the collapse mechanism;
- interactions between two towers.

Initial investigations have been conducted [ALT+03, BZ01]. In the following sections we introduce abstract models of implosive simulation through FEM dynamic analysis, and derive the corresponding linear equations.

2.2.1 Discretization

The first step and the critical step in FEM structure analysis is to discretize a structure into an assemblage of elements that are interconnected at structure joints (nodes). The node properties (typically deformations or displacements) are then calculated from an equilibrium equation $K\delta = f$, where K is a stiffness matrix that is assembled from individual element matrices, δ is a vector of global displacement, and f is a vector of forces acting in the direction of these displacement. (The element matrices describe the relationship between the element node deformations and the forces at the element nodes.) The global characteristics of the structure can be obtained from the node properties.

The WTC consisted of columns, beams, trusses, and tubes as shown in Figures 2.1 and 2.2. Trusses are modeled by truss elements, and tubes can be modeled by shell elements and/or triangular elements. Thus we can discretize WTC into rod, column, beam, shell, and triangular elements. The total number of elements is approximately 1 million.

2.2.2 Forming the stiffness matrix

Assume an element i has n^e nodes and k unknown variables. From [ZT05], the force-deformation relationship for the element is written as

$$K_i^e(p,t,T_t)\delta_i^e = f_i^e, \tag{2.12}$$

in which the element matrix $K_i^e(p,t,T_t) \in R^{k\times k}$ describes the effect of physical parameters p, time t, and temperature T_t; the vector $\delta_i^e \in R^k$ describes the magnitude (deformation) of the k variables; and the force vector $f_i^e \in R^k$ expresses the forces associated with the k variables. The structure stiffness matrix can be assembled from the stiffness matrices of the individual elements using the direct stiffness method [Bat96]. In this procedure the structure stiffness matrix $K \in R^{N\times N}$ is calculated by

$$K = \sum_{i=1}^{N_{elem}} K^{(i)}, \tag{2.13}$$

where N is the total degrees of freedom, N_{elem} is the total elements, and $K^{(i)}$ is a matrix obtained by expanding K_i^e to the same size as K. Note that all the entries in $K^{(i)}$ are zero except those corresponding to an element degree of freedom. Thus the equilibrium equation can be written as

$$K\delta = f, \tag{2.14}$$

where $\delta \in R^N$ is a vector of the system global displacements, and $f \in R^N$ is a vector of forces acting in the direction of these displacements.

2.2.3 Dynamic analysis

According to Newton's law, the statics at time t can be expressed as [Bat96]

$$F_t(t) + F_D(t) + F_E(t) = R(t), \tag{2.15}$$

where $F_t(t) = M\ddot{\delta}$ are the inertial forces, $F_D(t) = C\dot{\delta}$ are the damping forces, and $F_E(t) = K\delta$ are the elastic forces; M, C, K are the mass, damping and stiffness matrices formed from element matrices by following standard FEM assembly procedure; $R(t)$ is the vector of external applied loads; and $\ddot{\delta}$, $\dot{\delta}$, and δ are the acceleration, velocity, and displacement vectors defined at nodes. Thus, (2.15) can be rewritten as

$$M\ddot{\delta} + C\dot{\delta} + K\delta = R. \tag{2.16}$$

Equation (2.16) is a highly nonlinear ODE equation. In common numerical simulation procedures, an explicit or implicit approach is chosen for time descretization, and a Newton-Raphson method is used to solve the resulting nonlinear system. To achieve analysis accuracy and trace a complete physical path such as a collapse trajectory, many timesteps are needed to find accurate details of the stress redistribution. At every timestep, we create a nonlinear equation and use several linear equations to solve the nonlinear equation by Newton-Raphson. Thus, the core of simulation flow, including all timesteps, is to solve a long sequence of linear systems $A^{(i)}x^{(i)} = b^{(i)}$ (2.11) in which $A^{(i)} \in \mathbb{R}^{N\times N}$ defines the matrix at step i. As mentioned earlier, $A^{(i)}$ and $b^{(i)}$ will be written as A and b in the following sections and chapters.

An accurate model to simulate the WTC needs more than 10 million variables and more than 100 thousand timesteps to trace the failure states. We hope it will be possible to complete this task as hardware and linear algebra algorithms advance in the near future.

2.3 Simulation core

In general, simulation procedures in physics, science, and engineering can be abstracted as a series of linear equations (2.11). The main concerns are

- accuracy,
- capacity,
- performance.

We need high accuracy to make simulation results trustworthy. We need high capacity so that large and complicated problems can be analyzed. We need high performance so that the time-to-market can be achieved. As a multi-gigabyte memory chip is now available and 64-bit machines are being introduced, the capacity requirement is partly met. But accuracy and performance are still a vital concern. To reach high accuracy, we need to build more accurate models and use more conservative integration methods, which lead to large N and n_t in (2.11) and substantially lower performance.

Some methods for achieving high performance while keeping reasonable accuracy are

- the Fast Multipole Method of Greengard and Rokhlin [GR87],
- the H-matrix method [BGH03],
- cluster analysis and matrix partition [KK98],
- reuse of Krylov vectors (iterative method) [YZP08],
- consideration of advanced hardware such as cache [LaM96], GPU, and FPGA,
- parallel computing.

All these methods have accuracy limitations still, or severe performance degradation in corner cases.

In contrast to the above-mentioned methods, the following chapters establish a system of theorems and algorithms to solve systems (2.11) in a global context instead of solving them one by one as we consider the intrinsic properties of the problems involved: physical latency, physical continuity, and physical geometry. We will extract these properties at the matrix level instead of the related physical level. These algorithms will also overcome the inconsistency between performance and accuracy.

Chapter 3

Multiple-rank Update Theory

3.1 Introduction

In physical applications, engineering simulation, or financial analysis, the essential task is to solve an equation

$$f(t, p, x) = 0 \tag{3.1}$$

in which t is time, p is a set of parameters with valid range, and x is a vector of variables to be computed. The equation may be an ODE or a PDE or a stochastic equation. To solve the equation numerically, we discretize t and possibly x [Bat96, ZT05, GKO95, HN87a, HN87b, TH00, BO99, PTVF99], thus obtaining a system of linear or nonlinear equations. As we discussed in last chapter, this process can be abstracted to solving a series of linear systems

$$A^{(i)}x^{(i)} = b^{(i)}, \quad i = 1, \dots, n_t \tag{3.2}$$

for some number n_t, to get a solution over some physical domain.

It is quite expensive to solve (3.2) when n_t is large, and/or $A^{(i)}$ has high dimension. Although computation power has increased substantially in recent years, the computational requirement also increases dramatically when more accurate models are introduced or more complicated systems need to be simulated. In some fields such as EDA (Electronic Design Automation), the problem size for simulation has been increasing exponentially according to Moore's law (doubling every two years).

Given a typical time-to-market of six months, it is not acceptable to take weeks to solve (3.2). Many efforts [GR87, BGH03, KK98, YZP08, LaM96] have been made to answer industry's request for innovative algorithms to drive the advance of technology.

In general, the methods used to solve (3.2) can be divided into two families. One is to treat the $A^{(i)}$ as independent matrices. The common methods are Cholesky decomposition, Gaussian elimination, QR, SVD, or iterative methods like CG, LSQR, and so on [GVL96, PS82]. Another is to take advantage of the relationship between each $A^{(i)}$ and a reference matrix A_0 and to consider $A^{(i)}$, $i = 1, \dots, n_t$ systematically. If $A^{(i)} = A_0 + \Delta A$ and we know A_0's decomposition, the effort to solve $A^{(i)}x^{(i)} = b^{(i)}$ may be reduced. This problem is expressed in general as to solve

$$(A_0 + \Delta A)x = b, \quad \text{given } A_0 = L_0U_0. \tag{3.3}$$

How to solve (3.3) efficiently and stably has received much attention from mathematicians and engineers. Bennett [Ben65] proposed an LU updating strategy for the solution of modified matrices, and Gill, Golub, Murray, and Saunders [GGMS74] published a landmark paper about matrix factorization updating methods. These two papers led to an entire field of research that is still ongoing within scientific computing and other areas.

Based on previous work in research and industrial applications, this chapter proposes general and robust update algorithms to solve (3.3) when ΔA has low rank. The next sections demonstrate that these proposed algorithms have many advantages, including stability, robustness, capacity, parallel properties, and high performance.

3.2 General simulation strategies through matrix updating

If $A^{(i)}$ in (3.2) is expressed as

$$A^{(i)} = A^{(i-1)} + \Delta A^{(i)}, \tag{3.4}$$

where $\Delta A^{(i)}$ stands for the difference between two consecutive simulation matrices, (3.4) can be simplified as

$$A = A_0 + \Delta A. \tag{3.5}$$

where the reference matrix A_0 is from the previous step, and ΔA represents the part of the matrix that has changed its physical states. From the intrinsic properties of physics, ΔA is highly sparse and has low rank.

For example, the simulation matrix A_0 represents the device states in a timestep in circuit simulation, and ΔA represents a part of the matrix that has changed its states between the previous and current timesteps. The following three examples show how ΔA changes in an engineering application. They also show that ΔA is sparse and of low rank. We usually refer to ΔA as the active region.

Example 1. A Spice Simulation. A customer circuit with 765 nodes and 1000 bsim4 MOSes was simulated from 0 to 1.6×10^{-7} seconds by SPICE3 [spi, Nag75]. SPICE3 takes 368 timesteps to execute the simulation to compute the 765 nodes' voltages $V(i,j)$ at each timestep ($i = 1, \ldots, 368$, $j = 1, \ldots, 765$). To model node voltage changes between two consecutive timesteps, we define $dV(1,:) = 0$ and

$$dV(i,:) = V(i,:) - V(i-1,:), \quad i = 2, \ldots, 368. \tag{3.6}$$

Figure 3.1 shows dV. The x-coordinate represents time $i \in [0, 1.6\text{e}{-7}]$, the y-coordinate represents the node index $j \in [0, 765]$, and the z-coordinate represents $dV(i,j)$.

We see that the voltages of most nodes do not change: $dV(i,j) \approx 0$ for most i and j. We define such nodes to be *idle*. If a node's voltage changes significantly during some timesteps, we define the node to be *active*. In practice, the percentage of active nodes is very low. To see this better, we sort $dV(i,j)$ by rows and replot their absolute values in Figure 3.2. Clearly there are three spikes of interest to circuit designers and simulation algorithm developers.

Example 2. Here we consider another customer circuit that is 10 times bigger than in Example 1. It has 5642 nodes and 12,000 bsim4 MOSes. SPICE3 takes 822 timesteps to simulates it from 0 to 1.0×10^{-7} seconds. We define $V(i,j), i = 1,\dots,822, j = 1,\dots,5642$ to be the node voltages, where i represents the timestep and j represents the node index. The voltage change between two consecutive timesteps can be calculated from $dV[1,j] = 0$, $j = 1,\dots,5642$ and

$$dV(i,j) = V(i,j) - V(i-1,j), \quad i = 2,\dots,822, \quad j = 1,\dots,5642.$$

Figure 3.3 shows dV. As in Example 1, the x-coordinate represents time $i \in [0,\ 1.6\text{e}{-7}]$, the y-coordinate represents the node index $j \in [0, 5642]$, and the z-coordinate represents $dV(i,j)$. Figure 3.4 shows the sorted dV (absolute values) to illustrate the idle nodes in a more direct way.

We found that the percentage of idle nodes in Example 2 is higher than in Example 1. In general, the bigger the circuit, the higher the percentage of idle nodes and the lower the percentage of active nodes. This is a general property of sequential circuits. ΔA in (3.5) is designed to represent the changes caused by active nodes. It is necessary to use this physical information to enhance the accuracy and performance of circuit simulation.

Traditionally, transistor-level simulation can be divided into two families of methods: SPICE and fast SPICE [SJN94, PRV95]. The SPICE methods consider the circuit as an undivided object. A huge sparse Jacobian matrix will be formed when we numerically discretize the circuit to analyze instant current [New79, NPVS81]. The matrix dimension is of the same order as the number of the nodes in the circuit. For transient analysis, this huge nonlinear system needs to be solved hundreds of thousand times, thus restricting the capacity and performance of SPICE methods. SPICE in general can simulate a chip with up to about 50,000 nodes. Therefore it is not practical to use this method in full chip design. It is widely used in cell design, library building, and accuracy verification.

With appropriate accuracy, Fast SPICE methods developed in the early 1990s provided capacity and speed that was two orders of magnitude greater than SPICE [SJN94, ROTH94, HZDS95]. The performance gain was made by employing simplified models [CD86], circuit partition methods [Ter83, TS03], and event-driven algorithms [New79, ST75], and by taking advantage of circuit latency [Haj80, LKMS93, RH76, RSVH79]. Today, this assumption about circuit latency becomes questionable for nanometer designs because some subcircuits might have been functionally latent and yet electrically active because of voltage variation in Vdd and Gnd busses or in small crosstalk coupling signals. Also, the event-driven algorithm is generally insufficient to handle analog signal propagation. Fast SPICE's capacity is limited to a circuit size considerably below 10 million transistors. It is therefore still inadequate for full chip simulations for large circuits. Furthermore, the simulation time increases drastically with the presence of many BJTs, inductors, diodes, or a

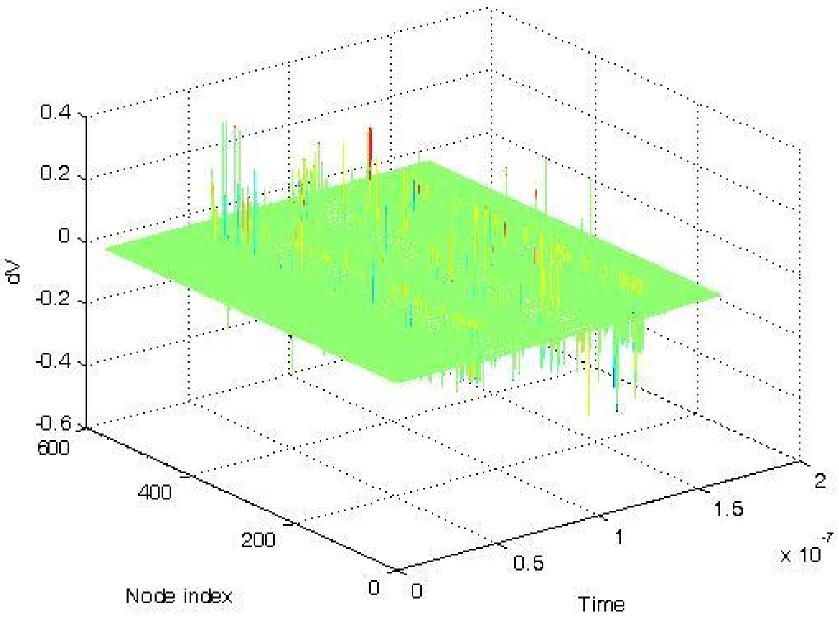


Figure 3.1: Node voltage changes dV for a 765-node circuit.

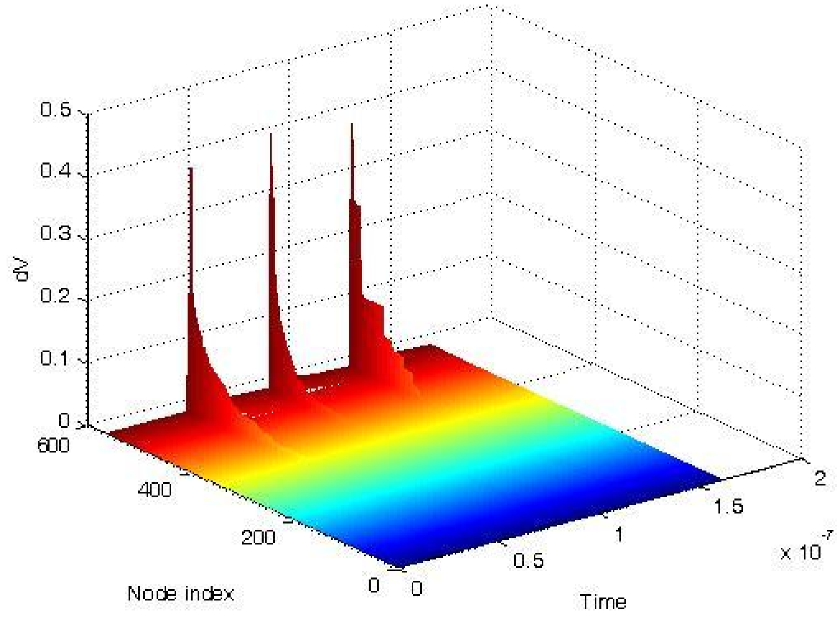


Figure 3.2: Sorted node voltage changes $|dV|$ for a 765-node circuit.

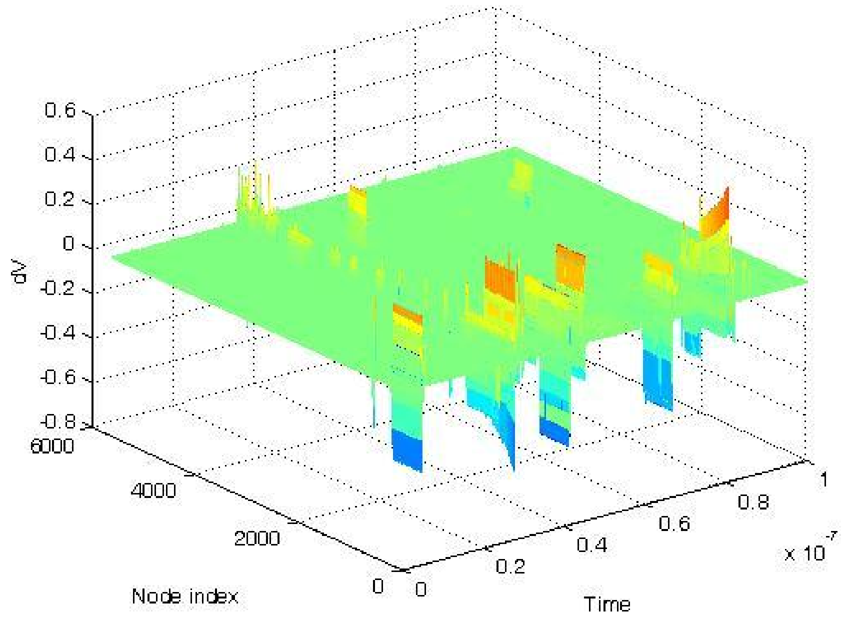


Figure 3.3: Node voltage changes for a 5642-node circuit.

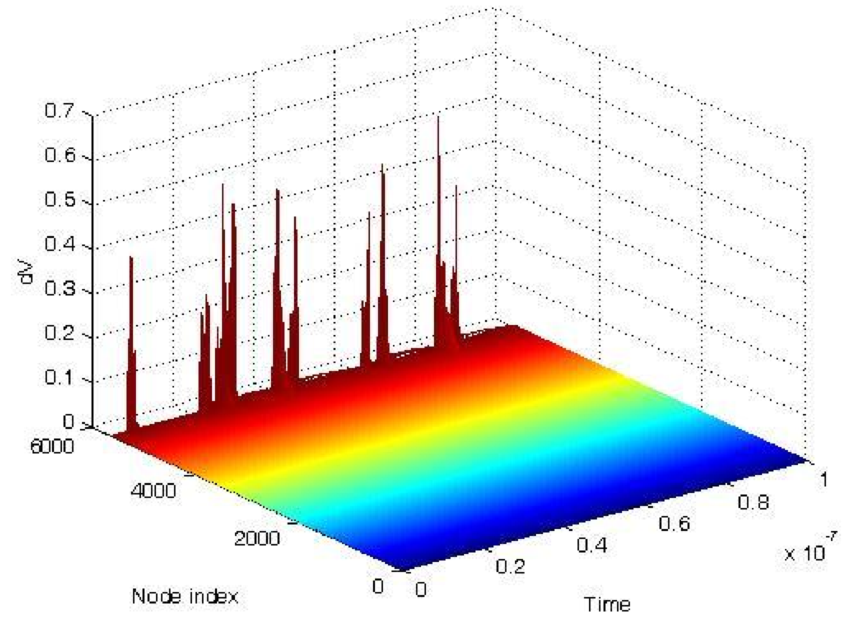


Figure 3.4: Sorted node voltage changes for a 5642-node circuit.

substantial number of cross-coupling capacitors.

Because of integration capacity in the nanometer era, SOC is widely used at present in the main design stream [Hut98]. It is possible now to integrate 1 billion transistors on a single chip. The corresponding matrix size would reach $O(10^9)$ if we used SPICE to simulate the full chip. This is not practical.

Fast SPICE families also face difficulties in simulating SOC because of the size difficulty and modeling inaccuracy. Driven by the market, EDA companies have developed a series of methods that partly satisfy the performance and accuracy requirements. Those methods include cut algorithms [Haj80, Wu76], hierarchical reduction [Syn03], RC reduction [KY97], and hierarchical storage strategies [Nas02]. In 65-nm design or below, the above-mentioned algorithms face big challenges in accuracy, capacity, and performance.

Because of the above limitations, some designers may choose to cut the entire circuit into small blocks and simulate the sub-circuits in different modes and levels. It is a tedious and error-prone process. Only full-chip simulation can make the designer understand the entire circuit behavior, if such a tool were to exist for very large circuits. Other designers may choose not to analyze nanometer effects but instead add guardbands [JHW$^+$01]. In effect, they overdesign their circuits to avoid problems caused by parasitics in the power bus, signal interconnects, and transistor devices. Guardbanding merely hides those effects in present technology, and is designed purely by experience. With the newest generation of nanometer design, guardbanding will prove to be ineffective and unreliable [KWC00]. We need solutions that provide insight into nanometer effects, and the ability to use this insight to control the performance of the design. This leads the functionality of EDA tools into the next generation.

Our proposed algorithms are different from the two traditional families of methods. We solve the simulation systems (3.2) in a global context instead of solving them one by one as we consider the intrinsic properties of the problems involved: physical latency, physical continuity, and physical geometry. We extract these properties at the matrix level instead of the related physical level through ΔA computation. These algorithms overcome the inconsistency between performance and accuracy.

Example 3. A model bridge ($\frac{1}{3}$ scale) with three girders was tested by Znidaric and Moses (1997, 1998). The load configuration used, shown in Figure 3.5, simulates the rated loading scheme with a 5-axle 42-ton semi-trailer used for safety assessment of existing road bridges in Slovenia.

As the load increases from zero to the ultimate load, the plastic region increases and ripples out from the middle toward the two ends. Figure 3.6 shows this in detail. The black region represents the original ΔA, and each colored region represents another ΔA.

Finding ΔA itself is a challenging problem. There are two ways to form ΔA. The first is to use algebraic computations: $\Delta A = A - A_0$. The second is to use physical measurements. In this approach, the active region is defined using device models, or through some engineering estimation. In this way, ΔA can be assembled more efficiently and more accurately, and some truncation errors [BYCM93, She99] are avoided.

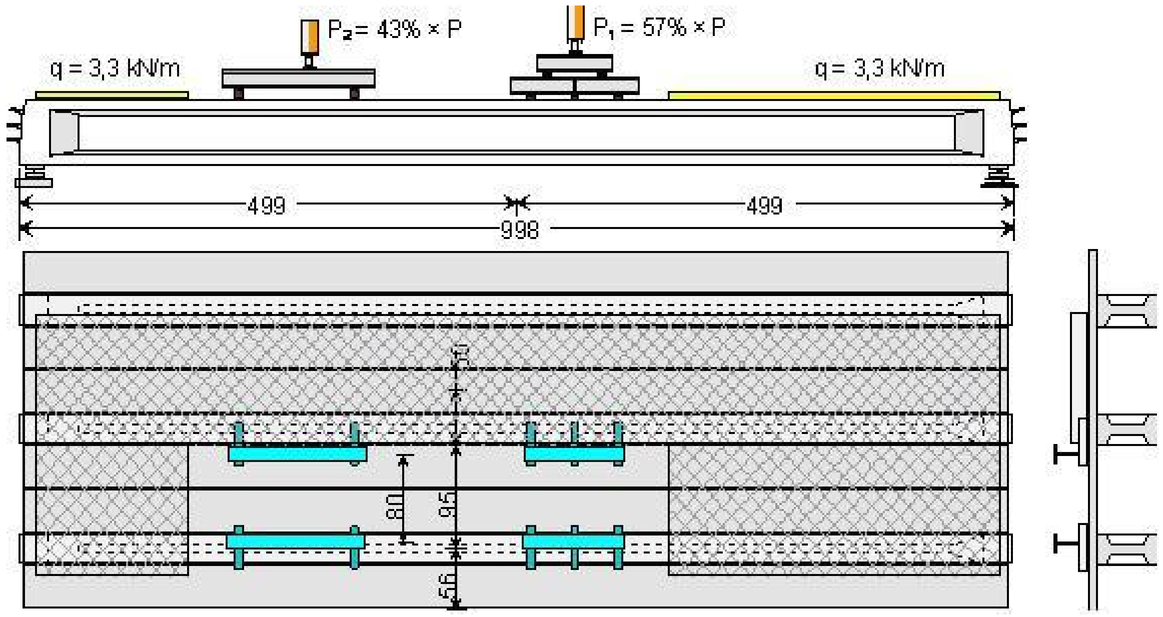


Figure 3.5: Bridge ultimate loading test.

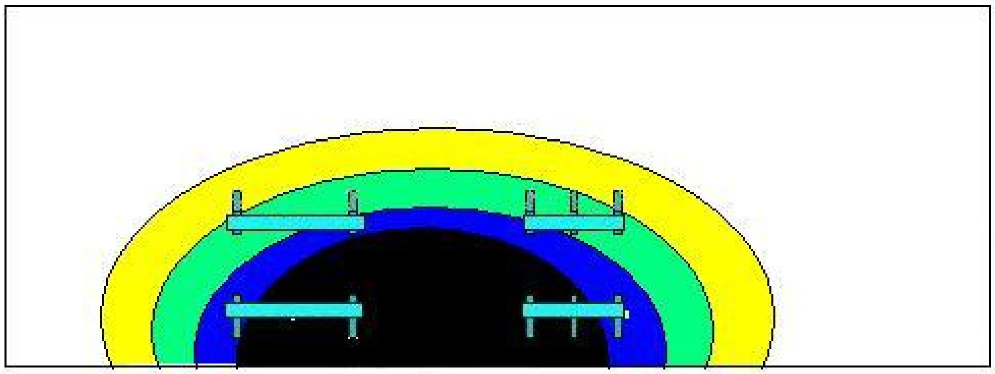


Figure 3.6: Changes of plastic region.

```
Given $A_0 = LU$ and $\Delta A = \sum_{j=1}^{r} \alpha_j x_j y_j^T$.
For $i = 1, 2, \ldots, r$
    $[L, U] = \text{update}(L, U, \alpha_j, x_j, y_j)$
```

Figure 3.7: Algorithm for factorization of $A_0 + \Delta A$.

If ΔA has rank r, it can be decomposed as the sum of r rank-1 matrices:

$$\Delta A = \sum_{j=1}^{r} u_j v_j^T.$$

Some methods for decomposing ΔA are singular value decomposition (SVD), CUR [MMD08], and elementary matrix computation. The SVD guarantees the minimum value of r:

$$\Delta A = U \Sigma V^T = \sum_{j=1}^{r} (\sigma_j u_j) v_j^T,$$

where U, V are orthogonal matrices, Σ is diagonal, r is the rank of ΔA, σ_i is the ith diagonal of Σ, and u_j, v_j are the jth columns of U and V. In other implementations, ΔA may be formed by algebraic transformation. For example, ΔA can be expressed as

$$\Delta A = \begin{bmatrix} a_1 & a_2 & \ldots & a_r \end{bmatrix} = \sum_{j=1}^{r} a_j e_j^T,$$

where r could be the rank of ΔA or greater, and e_j is the jth column of I.

In general, (3.5) can be expressed as

$$A = A_0 + \sum_{j=1}^{r} \alpha_j x_j y_j^T,$$

in which α_j is a parameter, and x_j and y_j are vectors of dimension n.

The factorization of A may be updated from A_0's factorization instead of factorizing A directly. The next sections discuss a rank-1 update procedure. When such a procedure is known, the LU with r rank-1 updates can be computed as follows:

$$\begin{aligned} A &= A_0 + \Delta A \\ &= L_0 U_0 + \sum_{j=1}^{r} \alpha_j x_j y_j^T = LU, \end{aligned}$$

in which L and U are obtained by repeated calls to the rank-1 update method described in this chapter. This process can be easily implemented as illustrated in Figure 3.7.

To obtain an accurate decomposition of ΔA, all r singular values should be considered. However, for practical purposes and to improve the efficiency of the process, accurate results can often be reached if we use a matrix ΔA_s of rank s to represent the actual ΔA matrix, where s is much smaller than r. Suppose ΔA has its singular values σ_1, σ_2, $\ldots$, σ_r. According to the SVD theorem, ΔA_k of rank k can be used as an approximation to ΔA:

$$\Delta A_k = \sum_{i=1}^{k} \sigma_j x_j y_j^T, \quad \text{and} \quad \|\Delta A - \Delta A_k\| \leq \sigma_{k+1}. \tag{3.7}$$

Thus, we can reduce the number of rank-1 terms in ΔA while keeping good accuracy by specifying an error threshold ϵ_T, so that $\|\Delta A - \Delta A_k\| \leq \epsilon_T$, where ϵ_T can be specified as a percentage of the largest singular value depending on how much accuracy is required. This is very important in engineering applications.

In this chapter, we discuss dense general matrix updates. Sparse rank-1 updates are discussed in chapter 4.

3.3 Unsymmetric rank-1 update

As mentioned earlier, a simulation matrix A is modified by a low-rank matrix ΔA that can be represented as k rank-1 updates. The full updating procedure is built on a rank-1 update algorithm. In the notation of [GGMS74], the basic rank-1 update problem can be described as

$$\overline{A} = A + \alpha x y^T, \tag{3.8}$$

in which α is a parameter, and x and y are vectors of dimension n. Assuming that A has been decomposed as $A = LU$, we wish to determine the factorization $\overline{A} = \overline{L}\,\overline{U}$.

Through a transformation, (3.8) can be rewritten as

$$\overline{A} = A + \alpha x y^T = L(I + \alpha p q^T)U, \text{ where } Lp = x, \ U^T q = y. \tag{3.9}$$

If we form the factorization

$$I + \alpha p q^T = \widetilde{L}\widetilde{U} \tag{3.10}$$

(assuming permutations are not needed), the modified factors become

$$\overline{A} = L\widetilde{L}\widetilde{U}U, \quad \text{giving} \quad \overline{L} = L\widetilde{L} \quad \text{and} \quad \overline{U} = \widetilde{U}U. \tag{3.11}$$

In the literature [GGMS74], we have rank-1 update algorithms for symmetric positive-definite matrices A and $\overline{A}$. Here we derive an unsymmetric update.

3.3.1 Factoring $I + \alpha pq^T$

$I + \alpha pq^T$ in (3.9)–(3.10) is a structured matrix. Assume its factors $\widetilde{L}$ and $\widetilde{U}$ are structured matrices with $\widetilde{L}_{j+1:n,j} = \beta_j p_{j+1:n}$, $\widetilde{U}_{i,i+1:n} = \gamma_i q^T_{i,i+1}$, $\widetilde{L}_{j,j} = 1$, and $\widetilde{U}_{i,i} = \theta_i$, for $i = 1, \dots, n$ and $j = 1, \dots, n$.

When $i = 1$, by comparing both sides of (3.10), we have

$$\begin{aligned} \theta_1 &= 1 + \alpha p_1 q_1 \\ \beta_1 p_i \theta_1 &= \alpha p_i q_1, \qquad i = 2, \dots, n, \\ \gamma_1 p_1 q_j &= \alpha p_1 q_j, \qquad j = 2, \dots, n. \end{aligned}$$

Rearranging terms, we have

$$\begin{aligned} \theta_1 &= 1 + \alpha p_1 q_1 \\ \beta_1 &= \frac{\alpha q_1}{\theta_1} \\ \gamma_1 &= \alpha. \end{aligned}$$

Therefore we get the first column $\widetilde{L}(:,1)$ and first row $\widetilde{U}(1,:)$ from the first column and row of $I + \alpha pq^T$. Assume we know $\widetilde{L}(:,1\!:\!i)$ and $\widetilde{U}(1\!:\!i,:)$, the first i columns and rows of the factors of $I + \alpha pq^T$. Then we can get the updated factors by induction. For diagonal $(i+1, i+1)$, we have

$$\begin{aligned} \sum_{k=1}^{i} (\beta_k p_{i+1})(\gamma_k q_{i+1}) + \theta_{i+1} &= 1 + \alpha p_{i+1} q_{i+1} \\ \Rightarrow \quad \theta_{i+1} &= 1 + p_{i+1} q_{i+1} \left(\alpha - \sum_{k=1}^{i} \beta_k \gamma_k \right). \end{aligned}$$

Comparing entries $(i+2\!:\!n, i+1)$ related to L:

$$\begin{aligned} \sum_{k=1}^{i} (\beta_k p_m)(\gamma_k q_{i+1}) + \beta_{i+1} p_m \theta_{i+1} &= \alpha p_m q_{i+1}, \qquad m = i+2, \dots, n \\ \Rightarrow \quad \beta_{i+1} &= \left(\alpha - \sum_{k=1}^{i} \beta_k \gamma_k \right) \frac{q_{i+1}}{\theta_{i+1}}. \end{aligned}$$

Comparing entries $(i+1, i+2\!:\!n)$ related to U,

$$\begin{aligned} \sum_{k=1}^{i} (\beta_k p_{i+1})(\gamma_k q_m) + \gamma_{i+1} q_m &= \alpha p_{i+1} q_m, \qquad m = i+2, \dots, n \\ \Rightarrow \quad \gamma_{i+1} &= \left(\alpha - \sum_{k=1}^{i} \beta_k \gamma_k \right) p_{i+1}. \end{aligned}$$

Therefore, β_i, γ_i, θ_i for $i = 1, \dots n$ are determined uniquely, so that $\widetilde{L}$ and $\widetilde{U}$ can be determined. Because LU factorization is unique, our assumption that $\widetilde{L}$ and $\widetilde{U}$ are structured matrices stands.

Note that $\alpha - \sum_{k=1}^{i} \beta_k \gamma_k$ can be formed incrementally during the update process, and the total computational effort to factor $I + \alpha p q^T$ is $O(n)$. The pseudo-code can be expressed as follows:

```
$\omega = \alpha$
do $i = 1:n$
    $\theta_i = 1 + \omega p_i q_i$
    $\beta_i = \omega q_i / \theta_i$
    $\gamma_i = \omega p_i$
    $\omega = \omega - \beta_i \gamma_i$
end
```

3.3.2 Structured matrix multiplication

Eq. (3.11) performs the product of two triangular matrices: $\overline{L} = L\widetilde{L}$ and $\overline{U} = \widetilde{U}U$. From the structure of $\widetilde{L}$ and $\widetilde{U}$, the computational effort in this process is $O(n^2)$ instead of $O(n^3)$. To form $\overline{L}$ in $O(n^2)$ operations, we start from the equation

$$\begin{bmatrix} 1 & & & & \\ \bar{l}_{21} & 1 & & & \\ \bar{l}_{31} & \bar{l}_{32} & 1 & & \\ \vdots & & & \ddots & \\ \bar{l}_{n1} & \bar{l}_{n2} & \bar{l}_{n3} & \cdots & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & & & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ \beta_1 p_2 & 1 & & & \\ \beta_1 p_3 & \beta_2 p_3 & 1 & & \\ \vdots & & & \ddots & \\ \beta_1 p_n & \beta_2 p_n & \beta_3 p_n & \cdots & 1 \end{bmatrix}.$$

To calculate p, we solve $Lp = x$ by forward substitution:

$$\begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & & & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}.$$

We can reorder the calculation column-wise with the help of a work vector w, which will play an important role in the update:

```
$w = x$
do $j = 1:n$
    $w_{j+1:n} = w_{j+1:n} - w_j \, l_{j+1:n,j}$
end
$p = w.$
```

To do the structured matrix multiplication, we need to rewrite w's pattern as follows. When

$j = 1$, w has the following value and properties:

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 - l_{21}p_1 \\ x_3 - l_{31}p_1 \\ \vdots \\ x_n - l_{n1}p_1 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ l_{32}p_2 + p_3 \\ \vdots \\ l_{n2}p_2 + l_{n3}p_3 + \cdots + p_n \end{bmatrix}.$$

Using these properties, we can calculate the first column of $\overline{L}$ as

$$\begin{aligned} \begin{bmatrix} 1 \\ \bar{l}_{21} \\ \bar{l}_{31} \\ \vdots \\ \bar{l}_{n1} \end{bmatrix} &= \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & & & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} 1 \\ \beta_1 p_2 \\ \beta_1 p_3 \\ \vdots \\ \beta_1 p_n \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ l_{21} + \beta_1(p_2) \\ l_{31} + \beta_1(l_{32}p_2 + p_3) \\ \vdots \\ l_{n1} + \beta_1(l_{n2}p_2 + l_{n3}p_3 + \cdots + p_n) \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ l_{21} \\ l_{31} \\ \vdots \\ l_{n1} \end{bmatrix} + \beta_1 \begin{bmatrix} \\ p_2 \\ l_{32}p_2 + p_3 \\ \vdots \\ l_{n2}p_2 + l_{n3}p_3 + \cdots + p_n \end{bmatrix} = \begin{bmatrix} 1 \\ l_{21} + \beta_1 w_2 \\ l_{31} + \beta_1 w_3 \\ \vdots \\ l_{n1} + \beta_1 w_n \end{bmatrix}. \end{aligned}$$

By repeating the above process, we can update the elements of $\overline{L}$ according to

$$\bar{l}_{ij} = l_{ij} + \beta_i w_j, \quad j = 1 : n, \quad i = j + 1 : n,$$

where w is updated for each j. The total computational effort is $O(n^2)$.

Similarly, to calculate $\overline{U} = \widetilde{U}U$ we need to perform the matrix multiplication

$$\begin{bmatrix} \bar{u}_{11} & \bar{u}_{12} & \bar{u}_{13} & \cdots & \bar{u}_{1n} \\ & \bar{u}_{22} & \bar{u}_{23} & & \bar{u}_{2n} \\ & & \bar{u}_{33} & & \bar{u}_{3n} \\ & & & \ddots & \\ & & & & \bar{u}_{nn} \end{bmatrix} = \begin{bmatrix} \theta_{11} & \gamma_1 q_2 & \gamma_1 q_3 & \cdots & \gamma_1 q_n \\ & \theta_{22} & \gamma_2 q_3 & & \gamma_2 q_n \\ & & \theta_{33} & & \gamma_3 q_n \\ & & & \ddots & \\ & & & & \theta_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ & u_{22} & u_{23} & & u_{2n} \\ & & u_{33} & & u_{3n} \\ & & & \ddots & \\ & & & & u_{nn} \end{bmatrix}.$$

Clearly, $\bar{u}_{ii} = \theta_i u_{ii}$. To calculate q, we solve $U^T q = y$ by forward substitution:

$$\begin{bmatrix} u_{11} & & & & \\ u_{12} & u_{22} & & & \\ u_{13} & u_{23} & u_{33} & & \\ \vdots & & & \ddots & \\ u_{1n} & u_{2n} & u_{3n} & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}.$$

Analogously to the computation of w, we perform the calculation row-wise as

```
$s = y$
do $i = 1 : n$
        $s_i = s_i / u_{ii}$
    $s_{i+1:n} = s_{i+1:n} - s_i u_{i,i+1:n}$
end
$q = s.$
```

In a similar way to calculating $\overline{L}$, we have $\overline{U}$ from

$$\begin{aligned} \bar{u}_{ii} &= \theta_i u_{ii} \\ \bar{u}_{ij} &= \theta_i u_{ij} + \gamma_i s_i \quad \text{for } i \neq j, \end{aligned}$$

where s is updated for each i. We now have the algorithm for updating $\overline{A}$.

Algorithm 1: Update LU factors of $A + \alpha x y^T$

```
$\omega = \alpha$
do $j = 1 : n$
    $y(j) = y(j)/U(j,j)$
    $\theta = 1 + x(j) * y(j) * \omega$
    $\beta = \omega * y(j)/\theta$
    $\gamma = \omega * x(j)$
    $\omega = \omega - \beta * \gamma$
    $U(j,j) = \theta * U(j,j)$
    do $i = j + 1 : n$
        $x(i) = x(i) - L(i,j) * x(j)$
        $y(i) = y(i) - U(j,i) * y(j)$
        $L(i,j) = L(i,j) + \beta * x(i)$
        $U(j,i) = \theta * U(j,i) + \gamma * y(i)$
    end
end
```

With some reordering of operations in Algorithm 1, one multiplication can be saved in the inner loop. We call this modification Algorithm 1a.

Like the symmetric update "Method C1" in [GGMS74], Algorithms 1 and 1a are simple and neat.

Algorithm 1a: Update LU factors of $A + \alpha x y^T$

```
$\omega = \alpha$
do $j = 1 : n$
    $y(j) = y(j)/U(j,j)$
    $\theta = 1 + x(j) * y(j) * \omega$
    $\beta = \omega * y(j)/\theta$
    $\gamma = \omega * x(j)$
    $\omega = \omega - \beta * \gamma$
    $U(j,j) = \theta * U(j,j)$
    do $i = j+1 : n$
        $x(i) = x(i) - L(i,j) * x(j)$
        $tmp = y(i)$
        $y(i) = y(i) - U(j,i) * y(j)$
        $L(i,j) = L(i,j) + \beta * x(i)$
        $U(j,i) = U(j,i) + \gamma * tmp$
    end
end
```

Only 4 internal variables are needed to complete the update. The benefit in memory utilization and parallel computation are described in the following sections.

3.4 Stability

A numerical algorithm should be stable. As in partial pivoting implementations of LU factorization, we can use permutations in our rank-1 update to control the size of the elements of L and perhaps U in order achieve stability.

Let $L_{\max}$ be a given stability tolerance ($L_{\max} \geq 1$). Consider a typical case where the growth condition $l_{kj} < L_{\max}$ will not be satisfied at column j. We can interchange rows j and k of L for some $k > j$. The matrices L and U become

$$A = LU = \begin{bmatrix} 1 & & & & & \\ \times & 1 & & & & \\ \times & \times & 1 & & & \\ \times & \times & \times & 1 & & \\ \times & \times & \times & \times & 1 & \\ \times & \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{bmatrix}$$

$$= T^T \begin{bmatrix} 1 & & & & & \\ \times & 1 & & & & \\ \times & \times & \times & \times & 1 & \\ \times & \times & \times & 1 & & \\ \times & \times & 1 & & & \\ \times & \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{bmatrix},$$

where T is a permutation. We now split L into the sum of a triangular matrix and a rank-2 correction, which we implement as two rank-1 updates:

$$TA = \left(\begin{bmatrix} 1 & & & & & \\ \times & 1 & & & & \\ \times & \times & 1 & & & \\ \times & \times & \times & 1 & & \\ \times & \times & 1 & & 1 & \\ \times & \times & \times & \times & \times & 1 \end{bmatrix} + \begin{bmatrix} & & & & & \\ & & & & & \\ & & \times{-1} & \times & 1 & \\ & & & & & \\ & & & & -1 & \\ & & & & & \end{bmatrix} \right) \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & & & \\ \times & 1 & & & & \\ \times & \times & 1 & & & \\ \times & \times & \times & 1 & & \\ \times & \times & 1 & & 1 & \\ \times & \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{bmatrix} + e_j v^T + e_k w^T,$$

where v and w are the corresponding correction vectors multiplied by U.

Using the rank-1 update procedure described in section 2, the factors of A can be formed stably through two rank-1 updates. We have to allow that each rank-1 update might lead to two more (and each of them two more, and so on!), but in practice such a cascading effect does not seem to happen. This procedure is shown as

$$\begin{aligned} A + \alpha xy^T &= LU + \alpha xy^T \\ &= T^T(TLU + \alpha Txy^T) \\ &= T^T(L'U' + \alpha Txy^T + e_j v^T + e_k w^T) \end{aligned}$$

If partial pivoting fails to give small residuals when the updated factors are used to solve a linear system, greater stability can be obtained by reducing $L_{\max}$ toward 1, or by using a form of complete pivoting in which we find $L_{i1,j1}$, the largest element of L.

Based on the above discussion, multiple rank-1 updates have the same stability characteristics as Gaussian elimination with partial or complete pivoting.

The above algorithm can be integrated into rank-1 update algorithms naturally. When we finish the column j's update, we have the update value of $L(:,j)$. We can check the stability condition $l_{kj} < L_{\max}$. If the condition is not satisfied, we exchange the rows j and k of L, exchange the j^{th} and $k^t h$ entrie in w, form two ranks-1s joining the update procedure as the compensation of the permutation. We now have the algorithm for updating $\overline{A}$. We call this as Algorithm 1b.

Algorithm 1b: Stably update LU factors of $A + \alpha xy^T$

```
$r = 1$
$x(1) \leftarrow x; \quad y(1) \leftarrow y$
do $j = 1:n$
    do $p = 1:r$
        $\omega(p) = \alpha$
        $y(p)(j) = y(p)(j)/U(j,j)$
        $\theta(p) = 1 + x(p)(j) * y(p)(j) * \omega(p)$
        $\beta(p) = \omega(p) * y(p)(j)/\theta(p)$
        $\gamma(p) = \omega(p) * x(p)(j)$
        $\omega(p) = \omega(p) - \beta(p) * \gamma(p)$
        $U(j,j) = \theta(p) * U(j,j)$
        do $i = j+1:n$
            $x(p)(i) = x(p)(i) - L(i,j) * x(p)(j)$
            $y(p)(i) = y(p)(i) - U(j,i) * y(p)(j)$
            $L(i,j) = L(i,j) + \beta(p) * x(p)(i)$
            $U(j,i) = \theta(p) * U(j,i) + \gamma(p) * y(i)$
        end
        checkAndFixStability()
            if not stable, rebase, r = r + 2
    end
end
```

3.5 Parallel update

During rank-1 updates, matrices L and U with dimension $n \times n$ are updated from column 1 to column n, or from top-left to bottom-right, tick by tick as shown in Figure 3.8.

Because the update sequence is a one-way movement, each column is no longer accessed after we finish its update. If we have k rank-1 updates, we can use this property to define a pipeline algorithm for k rank-1 updates, as indicated in Figure 3.9, where R_i means the ith rank-1 update to A_0. Because of the pipeline, multiple updates may proceed in parallel using multiple CPUs, multi-core CPUs, GPUs, and stream processors, or any combination of these processes. Specifically, in step 1, the first rank-1 update modifies the first column of L and the first row of U. In step 2, the second rank-1 update further modifies the first column of L and the first row of U, while in parallel, the first update moves to the *second* column of L and row or U.

From step k on, the pipeline is loaded, and a moving window of k columns of L and k rows of U are updated in parallel. The total time for k updates is

$$T = (k + n)(t_u + t_c), \tag{3.12}$$

where t_u is the average update time per column, t_c is the average communication time for the pipeline to move and synchronize. When $k \ll n$ and t_c is ignored, $T \approx nt_u$, which means k pipelined rank-1

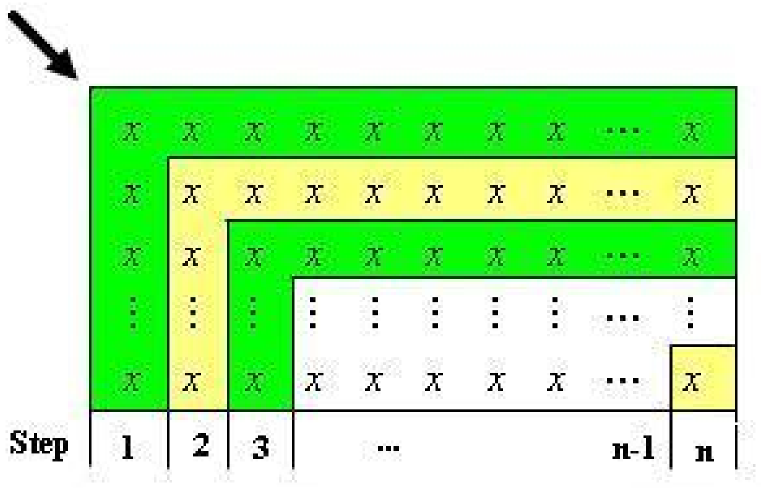


Figure 3.8: Update sequence during a rank-1 modification.

updates have the same performance as 1 rank-1 update in theory. When $k = n$ and t_c is ignored, $T = 2nt_u$, which means k pipelined updates have the performance as 2 updates in theory. $k = n$ is the worst case for pipeline calculation. We should point out that this performance estimation is not realistic because t_c, which includes the synchronizing time and memory/data fetching time, is much bigger than t_u for common dense and sparse matrices. To take full advantage of the pipeline algorithm, we can either increase the computation time in each step by updating several columns in that step, or combine with the group update described in following section.

3.6 Group update

When $A = A_0 + \Delta A$ with $\text{rank}(\Delta A) = r$, we define two types of update to the LU factors of A_0.

multiple rank-1 updates L and U are updated by calling a rank-1 update method r times as described in Figure 3.7. This involves an outer loop of length r and an inner loop over n columns of L and n rows of U.

group update Here the outer and inner loops are reversed. Each column of L and each row of U is updated r times by the next step of the rank-1 updates $R_1, R_2, \ldots, R_r$ sequentially before the outer loop moves to the next column and row. See Figure 3.10 and Algorithm 2.

Davis and Hager [DH01] discuss both approaches for the sparse symmetric positive-definite case (updating a factor L). They explain that multiple-rank updates access memory repeatedly because each rank-1 update requires access to the whole of L. They conclude that a group update has better memory traffic and executes much faster (2x) than the equivalent series of rank-1 updates because the group update makes only one pass through L.

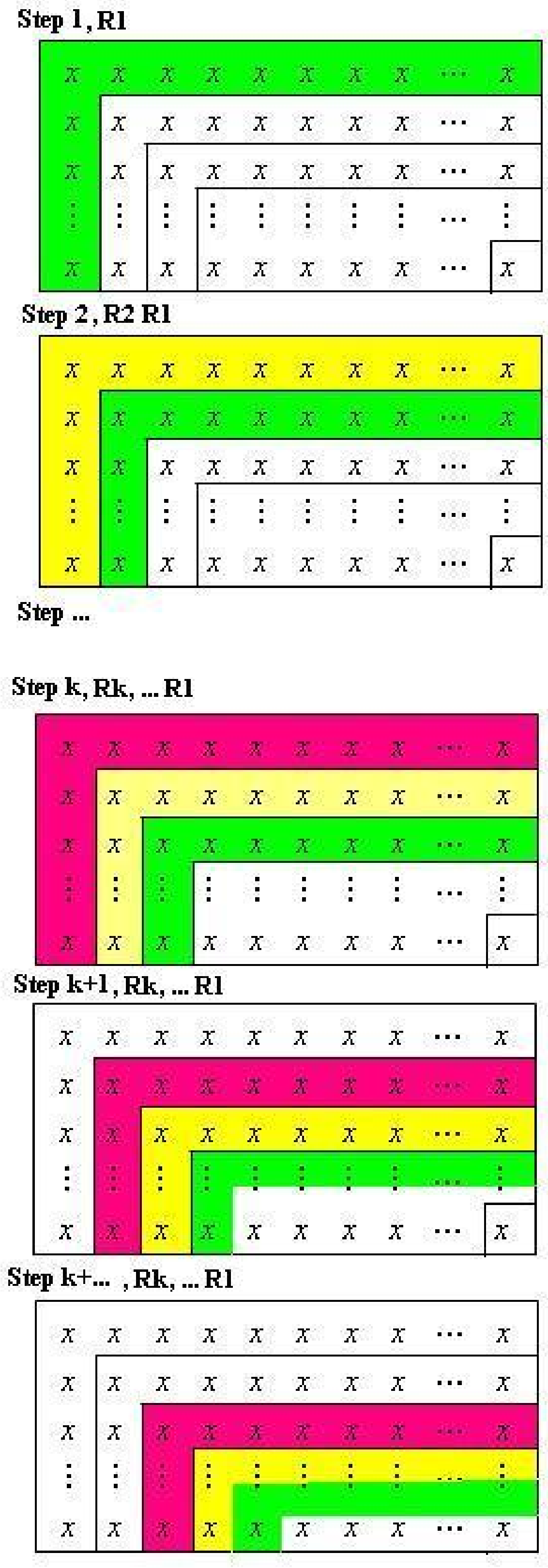


Figure 3.9: Pipeline for k rank-1 updates.

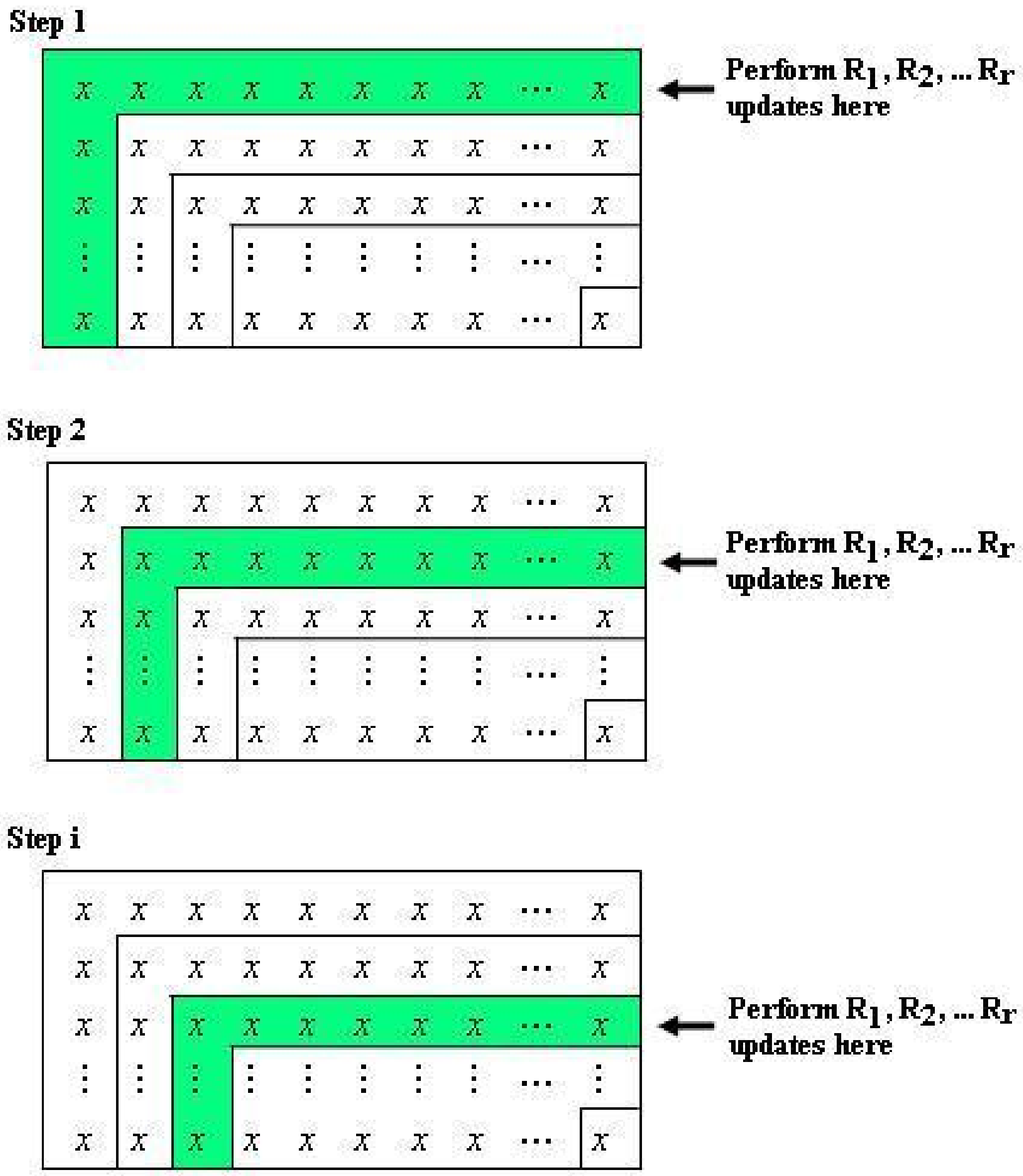

Step 1
Perform $R_1, R_2, \ldots R_r$ updates here
Step 2
Perform $R_1, R_2, \ldots R_r$ updates here
Step i
Perform $R_1, R_2, \ldots R_r$ updates here

Figure 3.10: Group update for applying r rank-1 updates.

Algorithm 2: Group update for LU factors of $A + \sum_{p=1}^{r} \alpha_p x_p y_p^T$

```
$do\ j = 1 : n$
    $do\ p = 1 : r$
        $\omega(p) = \alpha(p)$
        $y(p)(j) = y(p)(j)/U(j,j)$
        $\theta(p) = 1 + x(p)(j) * y(p)(j) * \omega(p)$
        $\beta(p) = \omega(p) * y(p)(j)/\theta(p)$
        $\gamma(p) = \omega(p) * x(p)(j)$
        $\omega(p) = \omega(p) - \beta(p) * \gamma(p)$
        $U(j,j) = \theta(p) * U(j,j)$
        $do\ i = j + 1 : n$
            $x(p)(i) = x(p)(i) - L(i,j) * x(p)(j)$
            $y(p)(i) = y(p)(i) - U(j,i) * y(p)(j)$
            $L(i,j) = L(i,j) + \beta(p) * x(p)(i)$
            $U(j,i) = \theta(p) * U(j,i) + \gamma(p) * y(i)$
        $end$
        //Check stability, and fix stability problems.
        checkAndFixStability()
    $end$
$end$
```

This conclusion stands also for multiple unsymmetric rank-1 updates and unsymmetric group updating. The disadvantage of the group update is the extra $2rn$ storage needed for intermediate results, while for serial rank-1 updates, only $2n$ workspace is needed. If A_0 is stored as a dense matrix, this disadvantage can be avoided, but it will remain a serious problem for sparse group updates.

We now point out that the group update has an advantage in parallel computing also. In the pipeline algorithm (previous section), the computation time may be much less than the communication time if one pipe processes just one rank-1 update and one column of L and one row of U. Even if a pipe updates several columns of L and rows of U, the pipeline will be inefficient because L and U will be accessed repeatedly. But the pipeline will show advantage if a pipe performs a group update on one or several columns and rows.

It is a serious task to design a synchronous algorithm in parallel computing. In a group update, every column will be updated by r rank-1s in the sequence of $R_1, R_2, \ldots, R_r$ if we set up the pipeline as $R_1, R_2, \ldots, R_r$ sequentially at the beginning and force full synchronization. The performance will be affected if the update times among pipes are different. If we update asynchronously, even if we set up the pipeline as $R_1, R_2, \ldots R_r$, the arriving update sequence at later steps will be random, as shown in Figure 3.11. The following theory tells us that group updates can be designed using asynchronous computing. This theory gives group updates a huge advantage in parallel computing.

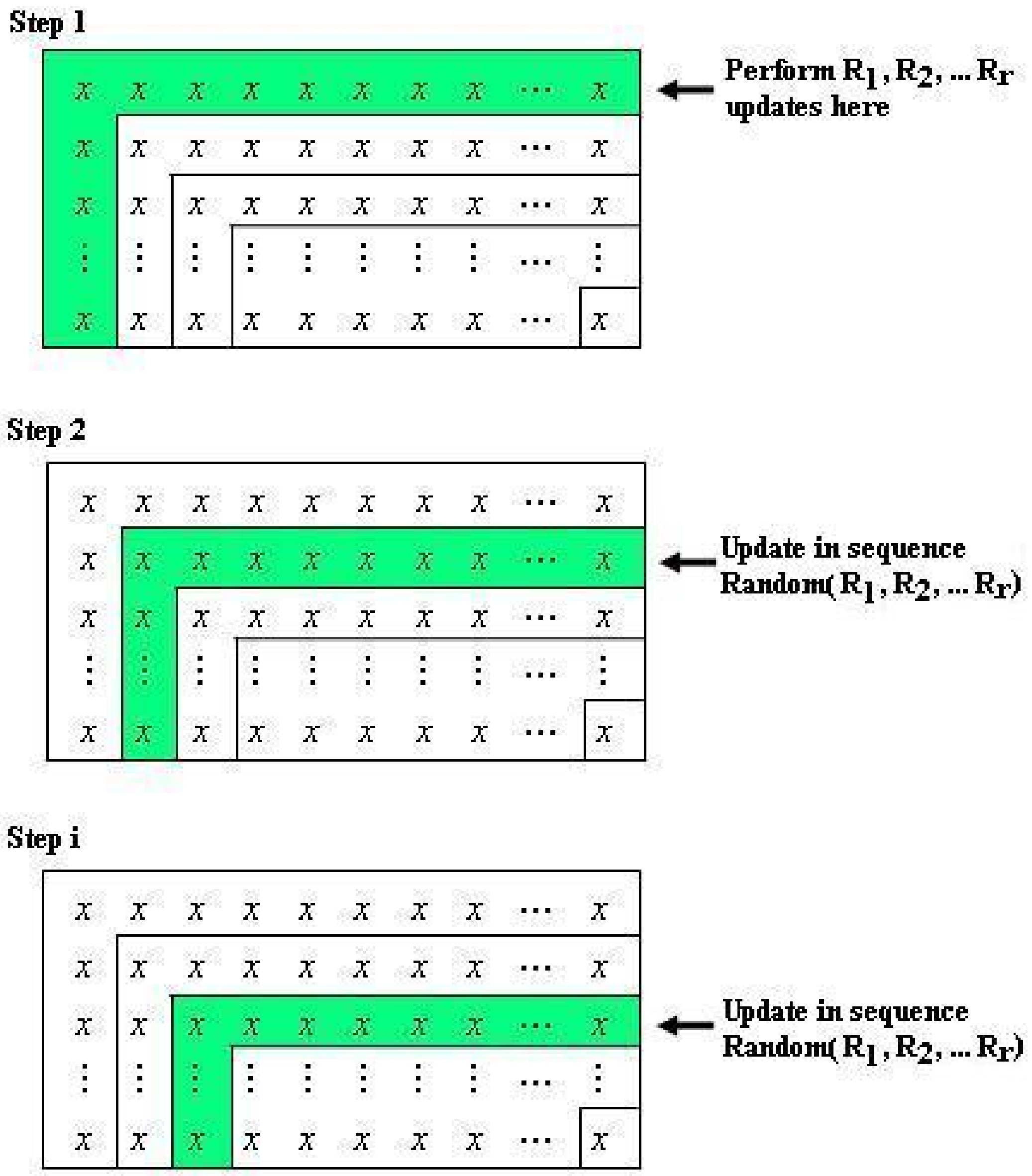

Step 1
Perform $R_1, R_2, \dots R_r$ updates here
Step 2
Update in sequence Random($R_1, R_2, \dots R_r$)
Step i
Update in sequence Random($R_1, R_2, \dots R_r$)

Figure 3.11: Random update sequence in group update of r rank-1s. It does not matter if each step processes the next part of each rank-1 update in a different order.

Theorem 3.1. *In a group update, the sequence of rank-1s at the first step can be ordered randomly as we update the subsequent columns and rows. The different sequences update the original factors into unique final factors.*

Proof. Assume there are two rank-1 updates $\alpha_1 x_1 y_1^T$ and $\alpha_2 x_2 y_2^T$. When the $(j-1)$th column of L has been updated, the first rank-1 has internal variable ω_1, and the second has internal variable ω_2. As we perform the first rank-1 update, we have new values for the diagonal $U_1(j,j)$ and for $L_1(:,j)$ and $U_1(j,:)$:

$$\begin{aligned}
\theta_1 &= 1 + x_1(j)y_1(j)\omega_1/U(j,j)\\
\beta_1 &= \frac{\omega_1 y_1(j)}{U(j,j)+x_1(j)y_1(j)}\\
\gamma_1 &= \omega_1 x_1(j)\\
U_1(j,j) &= \theta_1 U(j,j)\\
L_1(i,j) &= L(i,j) + \frac{\omega_1 y_1(j)}{U(j,j)+x_1(j)y_1(j)\omega_1}\left(x_1(i) - L(i,j)x_1(j)\right)\\
U_1(j,i) &= \theta_1 U(j,j) + \omega_1 x_1(j)\left(y_1(i) - U(j,i)y_1(j)\right)
\end{aligned}$$

When we do the second rank-1 update after we finish the first, we have new values for the diagonal $U_2(j,j)$ and for $L_2(:,j)$ and $U_2(j,:)$ as follows:

$$\begin{aligned}
\theta_2 &= 1 + x_2(j)y_2(j)\omega_2/U_1(j,j)\\
\beta_2 &= \frac{\omega_2 y_2(j)}{U_1(j,j)+x_2(j)y_2(j)}\\
\gamma_2 &= \omega_2 x_2(j)\\
U_2(j,j) &= \theta_2 U_1(j,j)\\
L_2(i,j) &= L_1(i,j) + \frac{\omega_2 y_2(j)}{U_1(j,j)+x_2(j)y_2(j)\omega_2}\left(x_2(i) - L_1(i,j)x_2(j)\right)\\
U_2(j,i) &= \theta_2 U_1(j,j) + \omega_2 x_2(j)\left(y_2(i) - U_1(j,i)y_2(j)\right)
\end{aligned}$$

The diagonal $U_2(j,j)$ can be represented as

$$\begin{aligned}
U_2(j,j) &= \theta_2 U_1(j,j)\\
&= U_1(j,j) + x_2(j)y_2(j)\omega_2\\
&= U(j,j) + x_1(j)y_1(j)\omega_1 + x_2(j)y_2(j)\omega_2
\end{aligned}$$

Thus, the diagonal value is not related to the update sequence.

$L_2(i,j)$ can be represented as follows:

$$\begin{aligned}
L_2(i,j) &= L_1(i,j) + \frac{\omega_2 y_2(j)}{U_1(j,j) + x_2(j)y_2(j)\omega_2}\left(x_2(i) - L_1(i,j)x_2(j)\right) \\
&= L(i,j) + \frac{\omega_1 y_1(j)}{U(j,j) + x_1(j)y_1(j)\omega_1}\left(x_1(i) - L(i,j)x_1(j)\right) \\
&\quad + \frac{\omega_2 y_2(j)}{U_1(j,j) + x_2(j)y_2(j)\omega_2} \times \\
&\quad \left(x_2(i) - \left(L(i,j) + \frac{\omega_1 y_1(j)}{U(j,j) + x_1(j)y_1(j)\omega_1}\left(x_1(i) - L(i,j)x_1(j)\right)\right)x_2(j)\right) \\
&= L(i,j) \\
&\quad + \frac{\omega_1 y_1(j)}{U(j,j) + x_1(j)y_1(j)\omega_1 + x_2(j)y_2(j)\omega_2} x_1(i) \\
&\quad + \frac{\omega_2 y_2(j)}{U(j,j) + x_1(j)y_1(j)\omega_1 + x_2(j)y_2(j)\omega_2} x_2(i) \\
&\quad + \left(\frac{-\omega_1 x_1(j)y_1(j) - \omega_2 x_2(j)y_2(j)}{U(j,j) + x_1(j)y_1(j)\omega_1 + \omega_2 x_2(j)y_2(j)}\right) L(i,j).
\end{aligned}$$

This can be further simplified as

$$L_2(i,j) = L(i,j) + \frac{\omega_1 y_1(j)}{U_2(j,j)}\left(x_1(i) - L(i,j)x_1(j)\right) + \frac{\omega_2 y_2(j)}{U_2(j,j)}\left(x_2(i) - L(i,j)x_2(j)\right).$$

Thus, the updated L does not depend on the update sequence.

On the other hand, we have

$$\begin{aligned}
U_2(j,i) &= \theta_2 U_1(j,j) + \omega_2 x_2(j)\left(y_2(i) - U_1(j,i)y_2(j)\right) \\
&= \theta_2\left(U_1(j,j) + \frac{\omega_2 x_2(j)}{U(j,j) + x_1(j)y_1(j)\omega_1 + x_2(j)y_2(j)\omega_2}\left(y_2(i) - U_1(j,i)y_2(j)\right)\right).
\end{aligned}$$

Thus, in a similar way, we can conclude that the update U is not related to the update sequence by noticing that θ_2 is not dependent on the update sequence and the term inside parentheses has the same symbolic structure as $L_2(:,j)$.

Although this algorithm is derived by using two rank-1 updates, it is straightforward to extend to r rank-1 updates by induction.

3.7 Examples

Example 1. Verify Algorithm 1. Assuming N is the dimension of A and ΔA, we create at first diagonally dominant matrices A_0 and ΔA by setting randomly

$$A_0(i,j) \in [0,1], \quad \Delta A(i,j) \in [0,1], \quad i = 1, \ldots, N, \quad j = 1, \ldots, N,$$
$$A_0(i,i) = \sum_{j=1}^{N} A(i,j), \quad \Delta A(i,i) = \sum_{j=1}^{N} \Delta A(i,j), \quad i = 1, \ldots, N,$$

and computing $A_0 = L_0 U_0$ and $A_0 + \Delta A = LU$ by left-looking and right-looking algorithms [Dav06]. When ΔA is decomposed as

$$\Delta A = \sum_{j=1}^{N} x_j y_j = \sum_{j=1}^{N} \Delta A(:,j) e_j^T \tag{3.13}$$

where e_j is the jth column of I, we then compute the factors of $A_0 + \Delta A$ through updating by (3.7) and Algorithm 1 (multiple-rank updates).

Define

$$\epsilon_{\max} = \max\left(|(A - LU)(i,j)|, \quad i = 1, \ldots, N, \quad j = 1, \ldots, N\right) \tag{3.14}$$

to describe the accuracy of a matrix A's factors L and U. The accuracies of left-looking, right-looking, and multiple-rank update methods are as follows:

N	Method	$\epsilon_{\max}$
500	Left-looking	9.663×10^{-13}
	Right-looking	9.663×10^{-13}
	Multiple-rank update	2.160×10^{-12}
1000	Left-looking	3.069×10^{-12}
	Right-looking	3.069×10^{-12}
	Multiple-rank update	7.048×10^{-12}
4000	Left-looking	1.592×10^{-11}
	Right-looking	1.592×10^{-11}
	Multiple-rank update	4.820×10^{-11}

Example 2. Verify the stability algorithm. We create A_0, ΔA, L_0 and U_0 as in Example 1. Then we compute the factors L and U for a downdated problem $A_0 - \Delta A$ by multiple-rank updates. The results below are obtained by setting $L_{\max} = 10$.

N	$\epsilon_{\max}$ Without Pivoting	$\epsilon_{\max}$ With Pivoting
500	7.41e-9	4.65e-10
1000	2.86e-7	8.98e-8
4000	2.18e-6	6.25e-9

Example 3. Verify Theorem 1. We create A_0, ΔA, L_0 and U_0 as in Example 1. Then we define the group size $r = 10$, 20, 50 respectively. To compute the factor of $A_0 + \Delta A$ by group update, we define the update sequence in column j randomly when $j > r$. The accuracy of the group update is shown here:

N	r	$\epsilon_{\max}$
500	10	2.274×10^{-12}
	20	2.046×10^{-12}
	50	2.046×10^{-12}
1000	10	7.162×10^{-12}
	20	7.958×10^{-12}
	50	6.812×10^{-12}
4000	10	3.910×10^{-11}
	20	4.047×10^{-11}
	50	3.774×10^{-11}

3.8 Performance comparison

The flops needed for LU factorization of a dense $n \times n$ matrix are:

Methods	Flops	Comments
Left-looking LU	$\frac{2}{3}n^3$	$O(n^2)$ memory write, bad in parallel
Right-looking LU	$\frac{2}{3}n^3$	$O(n^3)$ memory write, good in parallel
QR	$\frac{4}{3}n^3$	$O(n^3)$ memory write, good in parallel
n rank-1 updates	$\frac{4}{3}n^3$	$O(n^3)$ memory write, good in parallel

.

We can see that the computational effort is the same for the left-looking and right-looking methods. But performance does not only depend on flops. The memory access time is significant. In sparse-matrix computation, it is much more than the computation time.

In modern computer architecture, memory has a hierarchical structure as follows:

Memory Type	Common Size	Relative Speed
Register	limit	Very fast
Level 1 Cache	32K–64K	1
Level 2 Cache	~2M	0.1
Main Memory	~4G	0.01
Virtual Memory	>500G	0.001

At the top level are the CPU's registers, which provide the fastest way to access data. They number in the dozens. Even including MMX and SIMD registers and other CPU registers, the number is still limited. Thus we usually do not consider the register usage when designing an algorithm, although its effect should be considered during implementation.

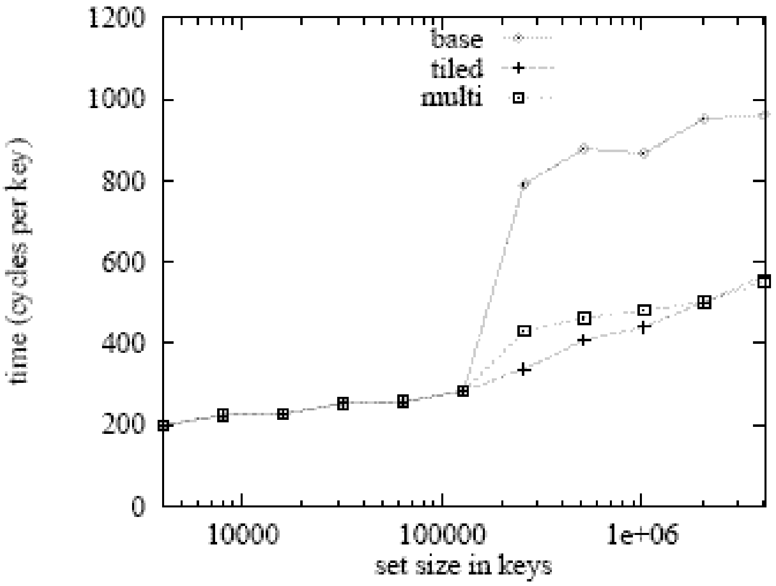


Figure 3.12: Performance of merge sort on sets between 4,000 and 4,096,000 keys.

From the above table, the performance in the worst case and the best case in implementing an algorithm could be different by a factor of 1000. A parameter to measure the efficiency of memory usage in an algorithm is the cache miss rate. We call it a cache miss if an attempt is made to reference data that has not been cached. Two important factors that affect miss rate are preload efficiency and reuse rate. Designing an algorithm with respect to hardware is an active research field [LaM96]. As computation power continuously improves through multi-core or new hardware architecture, flop counts become less important than memory usage.

When there is a cache miss, a computer will locate 128 bytes (or 8 doubles, including the one needed) into the cache. Ideally the extra data will be used next. If referenced data is already present, one memory operation is saved. The operating system preload algorithm tries to load the data so that the miss rate is as low as possible. Storing data contiguously is important to lower the missing rate. Data in cache should be used as many times as possible before it is flushed out. A good implementation of an algorithm, by improving the reuse rate, can lower the cache miss rate fundamentally. An example of cache usage from [LL99] is repeated in Figure 3.12, where performance drops suddenly when $L2$ cache has high miss rate in the base (text book) algorithm. When merge sort is designed by considering $L2$ size, 2x performance was shown there. In general, algorithms with $O(n \ln n)$ or $O(n^{0 \sim 2})$ performance need to consider cache effects seriously.

Suppose a system has 2MB cache and 4GB main memory. When we factorize a dense matrix A of dimension $n = 5000$, the matrix can not be located within L2 because it needs $5000 \times 5000 \times 8 = 200MB$. Common LU factorization needs to access A $O(n)$ times, which leads to L2 flushing $O(n)$ times. According to the above analysis, we know that the memory usage is not efficient at all.

N	LL	RL	GU5	GU10	GU20	GU50
100	0	0.00	0	0.0	0	0
200	0.02	0.02	0.03	0.03	0.03	0.03
300	0.06	0.06	0.10	0.10	0.10	0.10
400	0.14	0.15	0.24	0.24	0.24	0.24
500	0.28	0.28	0.47	0.47	0.47	0.47
600	0.48	0.50	0.81	0.81	0.81	0.80
700	0.76	0.79	1.28	1.28	1.28	1.28
800	1.14	1.17	1.92	1.92	1.92	1.92
900	1.62	1.69	2.73	2.72	2.72	2.72
1000	2.24	2.32	3.74	3.73	3.74	3.74
2000	18.78	18.70	29.78	29.80	29.80	29.78
3000	63.56	63.40	100.77	100.53	100.43	100.37
4000	150.46	150.55	238.57	238.11	237.92	237.84
5000	293.49	291.95	465.73	464.86	464.51	464.54
6000	507.10	507.05	804.59	803.29	802.49	803.29
7000	800.72	805.02	1277.87	1275.15	1274.12	1276.34
8000	1199.71	1203.55	1899.43	1903.35	1902.35	1907.15
9000	1697.86	1723.93	2705.12	2700.23	2708.66	2718.88
10000	2328.01	2341.05	3725.42	3719.39	3701.73	3722.92

Figure 3.13: Performance data for left-looking, right-looking, and group updating for $A = LU$. N: Matrix dimension. LL: Left-Looking. RL: Right-Looking. GU5, GU10, GU20, GU50: Group update with group size 5, 10, 20, 50.

If we do n rank-1 updates to get A's LU factorization, we can decompose A as

$$
\begin{aligned}
A &= \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix} \\
&= \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{bmatrix} \\
&\quad + A[n, n-1]e_{n-1}^T + \cdots + A[3:n, 2]e_2^T + A[2:n, 1]e_1^T.
\end{aligned}
$$

If we construct A's LU factors by group LU updates based on a computer's hardware, we can achieve better performance in memory usage. For example, if we update a group of 100 columns, we only need to access A 50 times. The performance difference is interesting to review, especially in a SIMD structure. Figure 3.13 shows the performance comparison for factorizing $A = LU$ by left- and right-looking methods and four group-updates of varying size.

In Figure 3.14, we see that n rank-1 updates to factor a dense matrix performed better than the theoretical estimate when $n > 400$. The time for n rank-1 updates is 1.5 times that for the LL

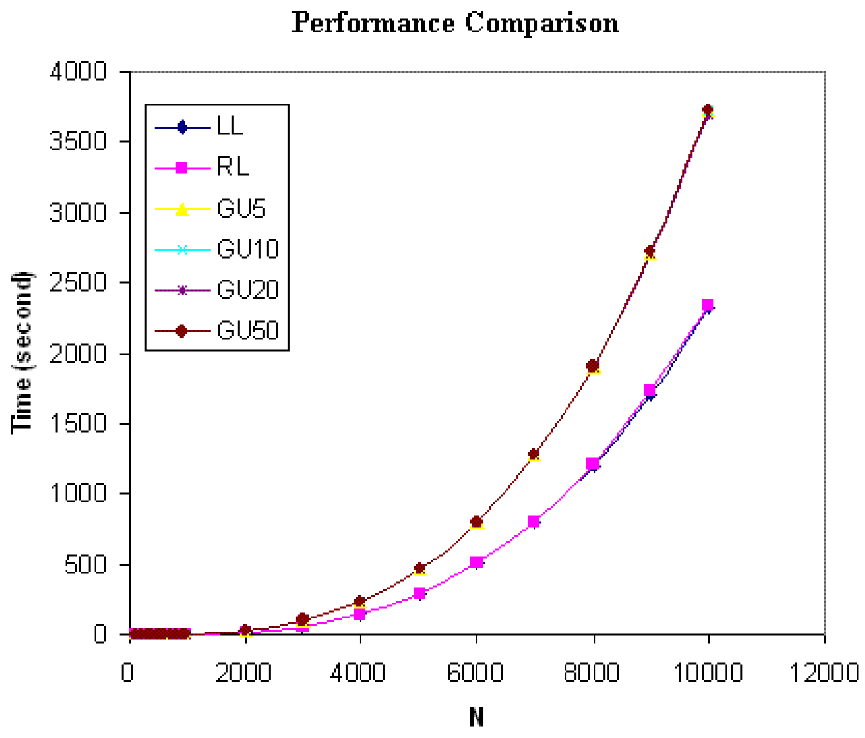


Figure 3.14: Performance comparison among left-looking, right-looking, and group update for $A = LU$. N: Matrix dimension. LL: Left-Looking. RL: Right-Looking. GU5, GU10, GU20, GU50: Group update with group size 5, 10, 20, 50.

or RL methods, not the two times that theory predicts. Also, in engineering applications involving r rank-1 updates, r is usually only $O(\sqrt{n})$, and $O(n^{2.5})$ operations are needed instead of $O(n^3)$. Further, as discussed in the last section, multiple-rank updates are fully parallel. Thus, they can be a powerful tool for dense-matrix factorization.

To show the cache influence on LL matrix factorization, we order the matrix by rows instead of by columns. We estimate that the performance will be 4 times slower than the standard implementation because we use only one of the four data that are brought in by a cache operation. Figure 3.15 shows our estimate is correct.

3.9 Conclusion

We have presented a new general matrix factorization update algorithm. It is stable, and it is efficient when the matrix size is bigger than L2 cache. Its parallel potential is much higher than standard LL and RL methods.

Dense matrices are assembled in the Boundary Element Method (BEM), which is the common

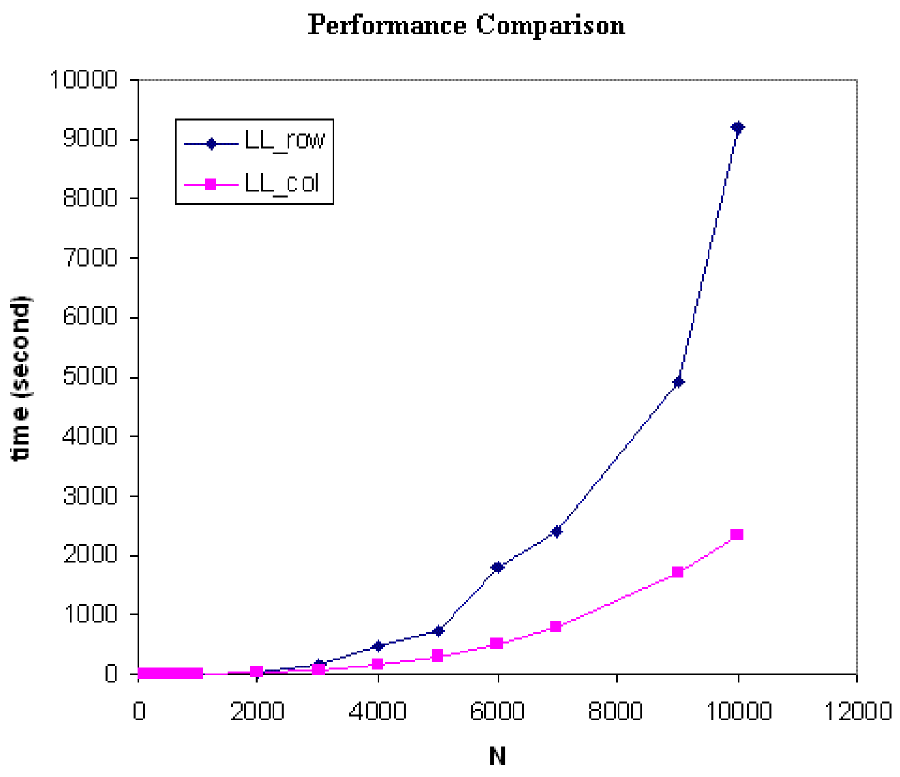


Figure 3.15: Performance comparison between two LL implementations. LL_row and LL_col mean the matrix is ordered by row and col respectively.

method for magnetic field analysis and Schur-complement factorization. Multiple-rank update algorithms will have application in these fields. They are also well suited to matrix factorization on GPUs because many independent jobs are created by multiple-rank updates are ideal for feeding the GPUs' computation cells.

Chapter 4

Sparse Matrix Factorization by Multiple-Rank Updates

4.1 Introduction

Many of the basic laws of science are expressed as ODE/PDEs [Hea02], including:

- Newton's second law of motion
- Kirchhoff's laws
- Heat equation
- Maxwell's equation
- Navier-Stokes equations
- Linear elasticity equations
- Schrödinger's equations
- Einstein's equations.

These ODE/PDEs describe the intrinsic properties in nature, engineering, physics, and more. In some simple conceptual models, we may get their theoretical solutions. In real problems where the complicated boundaries and/or geometries make the associated ODE/PDEs impossible to express in explicit or implicit format, numerical methods are the only tools available to solve them. Some numerical methods like the Boundary Method eventually lead to a system of dense matrices. Other numerical methods like FEM [Bat96] and difference methods [GKO95] discretize the domain and lead to a series of large linear but *sparse* equations. Solving these large linear equations is still a computing bottleneck. This chapter develops a sparse solver to eliminate the computational bottleneck.

Research in sparse matrices has been continuing for decades [Dav06, Sau09, DL, Dava, MUM, Davb]. Most researchers find applicable pivots to reduce the fill-in in direct methods, or to build an effective preconditioner in iterative methods. In this chapter, we discuss solving a sparse linear system in various contexts as a multiple-rank matrix update.

Sparse matrices have properties related to their application. Any algorithm dealing with sparse matrices should respect these properties. Before introducing our update algorithm, let us study the properties of some typical sparse matrices.

4.1.1 Sparse-matrix properties

Sparse linear systems exist in almost all fields. Research on improving the their performance and accuracy is active in the past 40 years. It becomes more attractive and and more essential as we need algorithms to solve the large and complicate problems in circuit design and structural analysis. Sparse matrices exhibit different properties in different applications. We conclude some examples in ths section.

In EDA (Electronic Design Automation) industry, the computational effort to solve a linear system is O($n^{1.05\sim 2}$). EDA involves a long sequence of linear systems in the time domain. Direct methods factorize a given matrix into two triangular matrices L and U, and then solve the system by forward and backward substitution with L and U. It is common for the ratio of factorization time to solve time to be below 5. It is also common for fill-in to change dramatically with different pivot sequences.

A circuit simulation matrix has the following properties [DS04]:

- Extremely sparse: Average entries per column around 5.
- Nearly symmetric.
- Not positive definite.
- Condition number is high.
- Several columns have almost full entries.
- High dimension: During full chip simulation, the dimension may reach $O(10^9)$.
- Latency: Solutions for two consecutive systems are almost the same except for the values of a small number of variables.

Stiffness matrices in structure analysis have these properties:

- Extremely sparse, especially in 3D problems.
- Symmetric positive definite for loading states; symmetric negative definite for unloading states.
- High dimension: For complicate structures, it may reach $O(10^6)$.
- Latency: During two consecutive simulation steps, most of the structure is kept in the same state.
- Continuity: The plastic regions increase continuously and ripple out.
- Elastic regions have the same stiffness matrices during different timesteps.

- Most parts of the simulated structure are in an elastic state. building law requires that structure should operate in elastic region in normal usage.

In optimization, structure reanalysis or reliability analysis, a sparse matrix has the following properties:

- A base matrix exists.
- Update matrices have extremely low rank relative to n (but possibly large rank in absolute terms).
- A trajectory is formed toward the converged solution.
- Latency: During two consecutive search steps along the trajectory, only a small part of the solution changes significantly.

Among these properties, latency stands out as a common property in system analysis. If we subtract two consecutive matrices in the simulation process, we will get a matrix ΔA that has low rank because of latency. This also explains why the intrinsic properties between two consecutive steps do not change much. The changed solutions form a trajectory that simulations usually follow in order to reduce the complexity to a tractable level.

The general simulation flow could be described in Figure 4.1. In the section of preprocessing, we read in the geometry data, analyze options and set up the analysis environment. Then we discretize the time domain to form a seriel of nonlinear equations. Each of the nonlinear equations is linearlized and solved through the methods like Newton method. Thus, the core of simulation flow, including all timesteps, is to solve a long sequence of linear systems. As described in Chapter 2, this series of linear systems is solved traditionally one by one by a direct or iterative method without considering the relationship existing between consecutive systems, or between the base system and later systems. From physical facts and the law of continuity, we know that the present simulation step is evolved from the preceding step. This clone relationship should be used in the simulation procedure to achieve robustness and accuracy. It should be helpful also in discontinuous problems and highly nonlinear problems.

Our proposed multi-rank update simulation flow is to solve

$$(A + \Delta A)x = b \tag{4.1}$$

at each stage, where A is the previous $A + \Delta A$. In this study, we focus on solving (4.1).

The multiple-rank update algorithm developed in Chapter 3 may take full advantage of latency. The relationship between latency and ΔA is shown in Figure 4.2. The higher the latency, the smaller the rank of ΔA. In circuit simulation, more than 90% of the devices in a circuit are in a latent state, which means that rank(ΔA) is quite small. Therefore we can get good performance via low-rank updating.

On the other hand, multiple-rank updates may not have performance advantages even if ΔA has low rank. Example 1 shows such a case.

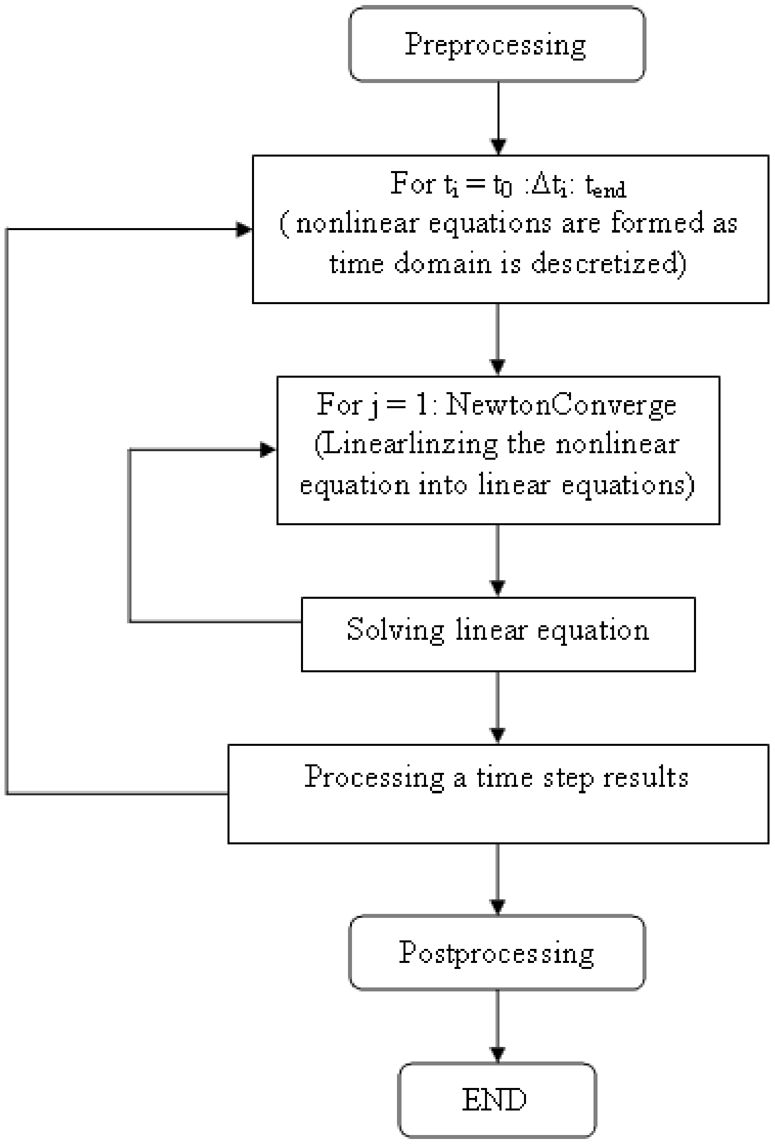

Preprocessing
For $t_i = t_0 : \Delta t_i : t_{end}$
(nonlinear equations are formed as time domain is descretized)
For j = 1: NewtonConverge
(Linearlinzing the nonlinear equation into linear equations)
Solving linear equation
Processing a time step results
Postprocessing
END

Figure 4.1: General simulation flow.

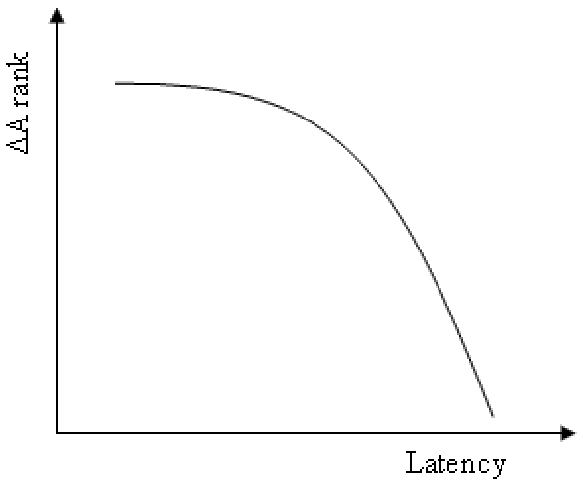


Figure 4.2: Relationship between latency and ΔA.

Example 1: Flops for a full-rank update of a tridiagonal matrix

Let $A \in \mathbb{R}^{n\times n}$ be a tridiagonal matrix. If ΔA is tridiagonal and has rank n, let us regard ΔA as n rank-1 updates, where the first update changes column 1, the second update changes column 2, and so on. The jth update needs $5(n-j+1)$ flops. The total flops for all n updates is therefore $\sum_{j=1}^{n} 5(n+1-j) = 5n^2/2 + O(n)$ flops.

In contrast, direct factorization of a tridiagonal matrix needs only $3n$ flops as show below.

```
U(1,1) = A(1,1)
U(1,2) = A(1,2)
L(2,1) = A(2,1)/U(1,1)
for j = 2:n
   U(j,j)   = A(j,j) - L(j,j-1)*U(j-1,j)
   L(j+1,j) = A(j+1,j)/U(j,j)
   U(j,j+1) = A(j,j+1)
```

Thus, even if ΔA has very low rank, there may be no performance gain from treating it as an update.

We now consider how to update sparse LU factors for a general sparse matrix.

4.2 Multiple-rank update algorithms for sparse LU factors

Suppose A and its LU factors have the following structure:

$$\begin{bmatrix} A_1 & & & B_1 \\ & A_2 & & B_2 \\ & & A_3 & B_3 \\ C_1 & C_2 & C_3 & D \end{bmatrix} = \begin{bmatrix} L_1 & & & \\ & L_2 & & \\ & & L_3 & \\ M_1 & M_2 & M_3 & L_D \end{bmatrix} \begin{bmatrix} U_1 & & & V_1 \\ & U_2 & & V_2 \\ & & U_3 & V_3 \\ & & & U_D \end{bmatrix},$$

and suppose there is a rank-1 update to A_1. The parts L_1, M_1, L_D, U_1, V_1, U_D need to be updated. The other parts do not change. It is clear that D depends on A_1, A_2, A_3, while A_1, A_2, A_3 are independent. This complicated relationship is expressed by an elimination tree. Before introducing the complex elimination process for a sparse matrix, let us introduce the notation introduced by George and Liu [GL81] and Amestoy, Davis, and Duff [ADD96].

4.2.1 Notation

Matrices are capital letters like A or L. Vectors are lower-case letters like x or v. Sets and multisets are in calligraphic style like $\mathcal{A}$, $\mathcal{L}$, or $\mathcal{P}$. Scalars are either lower-case Greek letters or italic style like σ, κ, or m.

The main diagonals of L and U are always nonzero for factors of a nonsingular matrix. The nonzero pattern of column j of L is denoted by

$$\mathcal{L}_j = \{i : l_{ij} \neq 0\}, \tag{4.2}$$

and $\mathcal{L}$ denotes the collection of patterns:

$$\mathcal{L} = \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n\}. \tag{4.3}$$

Correspondingly, the nonzero pattern of row i of U is denoted by

$$\mathcal{U}_i = \{j : u_{ij} \neq 0\}, \tag{4.4}$$

and $\mathcal{U}$ denotes the collection of patterns:

$$\mathcal{U} = \{\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_n\}. \tag{4.5}$$

In KLU, U is stored in column which makes $\mathcal{U}_i$ not shown there. Similarly, $\mathcal{A}_j$ denotes the nonzero pattern of column j of A:

$$\mathcal{A}_j = \{i : a_{ij} \neq 0\}, \tag{4.6}$$

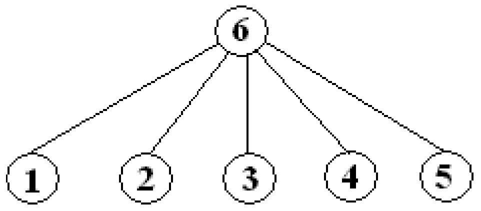


Figure 4.3: A shallow elimination tree.

and $\mathcal{A}$ denotes the collection of patterns:

$$\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n\}. \tag{4.7}$$

The elimination tree can be defined in terms of a parent map π [ADD96]. For any node j, $\pi(j)$ is the row index of the first nonzero element in the jth column of L beneath the diagonal element:

$$\pi(j) = \min\ (\mathcal{L}_j \setminus \{j\}). \tag{4.8}$$

The children of node k are the set defined by

$$\pi^{-1}(j) = \{j : \pi(j) = k\}. \tag{4.9}$$

The ancestors of a node j, denoted by $\mathcal{P}(j)$, are the set of successive parents:

$$\mathcal{P}(j) = \{\pi(j), \pi(\pi(j)), \dots\} \equiv \{\pi^1(j), \pi^2(j), \dots\}. \tag{4.10}$$

4.2.2 Rank-1 update path

The path of a rank-1 update follows exactly the path of the elimination tree. If there is a rank-1 change in column j, all the columns in set $\mathcal{P}(j)$ need to be updated.

Assume there is a rank-1 change in node 4 of the elimination tree shown in Figure 4.3. The rank-1 update process will update column 6 in L and U.

The elimination tree for a tridiagonal matrix is shown in Figure 4.4. If there is a rank-1 change in column 1, the rank-1 update process will modify all remaining columns 2, 3, . . . , n of the original L and U. This process needs $5n$ flops. Thus, even a *single* rank-1 update cannot outperform direct LU factorization (which needs $3n$ flops). For more general matrices, as mentioned in the last section, updating will lead to poor performance if there are $O(n)$ rank-1 updates.

From these two examples, we see that the depth of the elimination tree is a critical factor for update performance.

Davis and Hager [DH99] developed sparse low-rank update techniques for symmetric positive

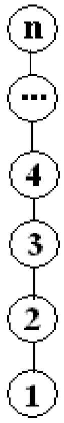


Figure 4.4: The elimination tree for a tridiagonal matrix.

definite matrices. Their techniques are based on an analysis and manipulation of the elimination tree and Algorithm C1 of Gill, Golub, Murray, and Saunders [GGMS74]. They provide a system of algorithms to handle fill-in in sparse factors, using symbolic updating and downdating. Later, Davis and Hager [DH01] showed 2× performance gain with their multiple-rank update algorithm, which makes only one pass through L. (A series of rank-1 updates requires multiple passes through L.)

The rank-1 update algorithm is optimal in the sense that it takes time proportional to the number of nonzero entries that are changed in L and U, and optimal in the sense of managing traffic according to the computer architecture. However, the multiple-rank update algorithm is not optimal in this sense, because the difficulty shown in Example 1 still exists.

Let us discuss the tridiagonal matrix in Example 1 again. Referring to its elimination tree in Figure 4.4, we see that the rank-1 updates starting from column 1, 2, ..., $j-1$ will all pass through column j shown in Figure 4.5.

The sum of r rank-1 updates which pass through a column j depend on the elimination tree and the location of rank-1s. We define this sum to be

$$s(j) = \sum_{i=1}^{r} \hat{\pi}(i) \tag{4.11}$$

where $\hat{\pi}(i)$ be null if the rank-1 update started from column i does not modify column j, and be 1 if the rank-1 update started from column i does modify column j

The difficulty described in Example 1 still exists when $s(j)$ is bigger than the number of nonzeros in the jth column of L. In Example 1, $s(j)$ is $O(n)$ even if the number of nonzeros in the jth column of L is one. This explains why we cannot get good performance in tridiagonal update procedures. A new algorithm described in the next section is designed to solve this problem.

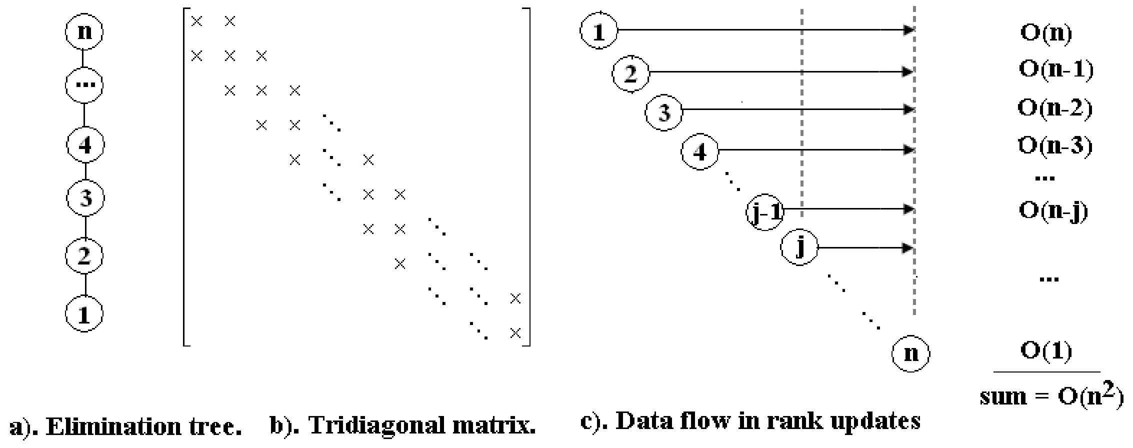


Figure 4.5: Illustration of tree travel in update procedures for a tridiagonal matrix.

4.3 Base-camp theory

Suppose we have the factorization $A = LU$. For any given k, the factors can be partitioned as

$$A = \begin{bmatrix} F & B \\ C & D \end{bmatrix} = \begin{bmatrix} L_F & \\ L_C & L_D \end{bmatrix} \begin{bmatrix} U_F & U_B \\ & U_D \end{bmatrix} = \begin{bmatrix} L_F & \\ L_C & I \end{bmatrix} \begin{bmatrix} U_F & U_B \\ & S \end{bmatrix}, \tag{4.12}$$

where $F \in \mathbb{R}^{(k-1)\times(k-1)}$ and $S = D - L_C U_B$ is the Schur complement of F. The factorization $S = L_D U_D$ is part of the factorization of A. Conversely, if we have S and a factorization $S = L_S U_S$, we can define $L_D = L_S$ and $U_D = U_S$.

Now suppose the factorization of $A + \Delta A$ in (4.1) has reached column k, giving

$$\begin{aligned} A + \Delta A &= \begin{bmatrix} F & B \\ C & D \end{bmatrix} + \begin{bmatrix} \Delta F & \Delta B \\ \Delta C & \Delta D \end{bmatrix} \\ &= \begin{bmatrix} L_{\hat F} & \\ L_{\hat C} & I \end{bmatrix} \begin{bmatrix} U_{\hat F} & U_{\hat B} \\ & \hat S \end{bmatrix}, \end{aligned}$$

where $L_{\hat F}$, $L_{\hat C}$, $U_{\hat F}$, and $U_{\hat B}$ are known. The new Schur complement is $\hat S = D + \Delta D - L_{\hat C} U_{\hat B}$, and its factorization $\hat S = L_{\hat S} U_{\hat S}$ can be used to complete the factors of $A + \Delta A$:

$$A + \Delta A = \begin{bmatrix} L_{\hat F} & \\ L_{\hat C} & L_{\hat S} \end{bmatrix} \begin{bmatrix} U_{\hat F} & U_{\hat B} \\ & U_{\hat S} \end{bmatrix}.$$

Any method could be used to factorize $\hat S$. Our approach is to consider $\hat S$ as an update of S:

$$\hat S = S + \Delta S, \qquad \Delta S = \Delta D + L_C U_B - L_{\hat C} U_{\hat B}. \tag{4.13}$$

If ΔS has rank r, we can factorize $S + \Delta S$ by applying r rank-1 updates to $S = L_S U_S$. If $s(k)$ is greater than r, a performance gain is achieved. A key point is how to choose k such that ΔS *has low rank.* Before introducing a theorem to express this key point, we need some definitions.

Definition: Let $\mathcal{L}_j$ and $\mathcal{U}_i$ be known from the factors of A for all $i, j = 1, \ldots, n$. The same $\mathcal{L}_j$ and $\mathcal{U}_i$ apply to $A + \Delta A$ because we assume that A and ΔA has the same symbolic structure in this study. A *base-camp matrix* is a dense matrix $\Lambda_k \in \mathbb{R}^{p\times q}$ associated with column k of A, where $p = \text{card}(\mathcal{L}_k)$, $q = \text{card}(\mathcal{U}_k)$. Column k is called the base-camp's *master column.* The map $i : \mathcal{L}_k(i)$ relates row i of Λ_k to row $\mathcal{L}_k(i)$ of L. Similarly, the map $j : \mathcal{U}_k(j)$ relates column j of Λ_k to column $\mathcal{U}_k(j)$ of U. Λ_k is obtained from rows $\mathcal{L}_k$ and columns $\mathcal{U}_k$ of $L_C U_B$.

Note that $\Lambda_k \subseteq L_C U_B \subseteq S$, and column k has not yet been included in the LU factorization. The elimination path to column j and the path that does not come to column j are independent. For simplicity we can reorder L_C and U_B so that Λ_k is in the upper-left corner of $L_C U_B$. Thus, $L_C U_B$ becomes

$$L_C U_B = \begin{bmatrix} L_{C1} & L_{C2} \\ L_{C3} & L_{C4} \end{bmatrix} \begin{bmatrix} U_{B1} & U_{B3} \\ U_{B2} & U_{B4} \end{bmatrix} = \begin{bmatrix} L_{C1}U_{B1} + L_{C2}U_{B2} & L_{C1}U_{B3} + L_{C2}U_{B4} \\ L_{C3}U_{B1} + L_{C4}U_{B2} & L_{C3}U_{B3} + L_{C4}U_{B4} \end{bmatrix}. \tag{4.14}$$

When A becomes $A + \Delta A$, multiple-rank changes in rows $\mathcal{L}_k$ and columns $\mathcal{U}_k$ are accumulated into Λ_k to give $\hat{\Lambda}_k$, and $\hat{L}_C \hat{U}_B$ becomes

$$\hat{L}_C \hat{U}_B = \begin{bmatrix} \hat{L}_{C1} & \hat{L}_{C2} \\ \hat{L}_{C3} & \hat{L}_{C4} \end{bmatrix} \begin{bmatrix} \hat{U}_{B1} & \hat{U}_{B3} \\ \hat{U}_{B2} & \hat{U}_{B4} \end{bmatrix} = \begin{bmatrix} \hat{\hat{L}}_{C1}\hat{U}_{B1} + \hat{L}_{C2}\hat{U}_{B2} & \hat{L}_{C1}\hat{U}_{B3} + \hat{L}_{C2}\hat{U}_{B4} \\ \hat{L}_{C3}\hat{U}_{B1} + \hat{L}_{C4}\hat{U}_{B2} & \hat{L}_{C3}\hat{U}_{B3} + \hat{L}_{C4}\hat{U}_{B4} \end{bmatrix}. \tag{4.15}$$

We regard ΔA as a sum of rank-1 updates. If an update is about to alter column k, then we terminate the update, the column k is not changed. According to symbolic analysis [DH01],

$$\mathcal{L}_k = \{k\} \cup \left(\bigcup_{c:k=\pi(c)} \mathcal{L}_c \backslash \{c\} \right) \cup \left(\bigcup_{\min \mathcal{A}_j = k} \mathcal{A}_j \right),$$

the columns in L_{C2} have "k" as ancestor, and $L_{c4} = \hat{L}_{c4} = 0$. In a similar way, we can conclude that $U_{B4} = \hat{U}_{B4} = 0$.

There are $s(k)$ such updates that will alter column k. Other updates continue till the end (or until they reach a later base-camp). When $s(k)$ updates have been terminated at column k, those updates do not modify the term $L_{C1}, L_{C3}, U_{C1}, U_{C3}$, so we have

$$L_{C1} = \hat{L}_{C1},\ L_{C3} = \hat{L}_{C3},\ U_{C1} = \hat{U}_{C1},\ U_{C3} = \hat{U}_{C3}.$$

Thus we build base-camp matrices $\Lambda_k = L_{C2}U_{B2}$ and $\hat{\Lambda}_k = \hat{L}_{C2}\hat{U}_{B2}$ to consider the effect of column k's update and form

$$\Delta\Lambda_k \equiv \Lambda_k - \hat{\Lambda}_k, \tag{4.16}$$

which will be part of ΔS in (4.13) exactly. $\Delta\Lambda_k$ has much less computation effort than the full ΔS. This is the one of the advantages of base-camp algorithm. We proceed to update the LU factors of A, starting at column k, by repeating the preceding process with only $\min(p, q)$ rank-1 updates waiting.

The processing of $\Delta\Lambda_k$ is explained in Theorem 4.1 and its proof.

Definition: Let $\kappa \equiv \min\{\mathrm{card}(\mathcal{L}_k), \mathrm{card}(\mathcal{U}_k)\}$ be the maximum rank of the base-camp matrix associated with master column k.

Theorem 4.1. *$s(k)$ rank-1 updates entering a base-camp associated with master column k (i.e., waiting to contribute to the kth column of L and the kth row of U) can be merged into at most κ rank-1 updates when no other rank-1 updates are proceeding inside Λ_k (i.e., waiting to modify some of the remaining columns of Λ_k).*

Proof. Suppose there are r rank-1 updates to be made. Initially, assume that the updates generate only one base-camp in A with master column k. We divide the updates into two groups. Group 1 contains $s(k)$ rank-1 updates that modify column k, and together they define a matrix ΔA_1. The other rank-1 updates are put into Group 2, and together they form ΔA_2.

We now consider two scenarios that are equivalent in the sense that they lead to the same final L and U:

Scenario 1: $(A + \Delta A_1) + \Delta A_2 = LU$;

Scenario 2: $(A + \Delta A_2) + \Delta A_1 = LU$.

In scenario 1, we process the group 1 updates first. By definition, all of them stop at column k. In (4.12), D does not yet change. Instead, the D update information from group 1 is contained in $\hat{\Lambda}_k$ (which will be computed), while L_{C1}, L_{C3}, U_{B1} and U_{B3} do not change. Thus we can merge the $s(k)$ group 1 updates by forming $\Delta\Lambda_k \equiv \Lambda_k - \hat{\Lambda}_k$ (4.16). Note that if $t \equiv \mathrm{rank}(\Delta\Lambda_k)$, then $t \le \kappa$ and possibly $t \ll s(k)$. We process $\Delta\Lambda_k$ as t rank-1 updates to complete the updates for group 1. We then process group 2 as normal updates to complete the full set of updates.

In scenario 2, we process group 2 first. We analyze this scenario as cases 2a and 2b. In case 2a we assume that no group 2 update modifies Λ_k, so that L_{C1}, L_{C3}, U_{B1} and U_{B3} are not changed during the group 2 updates. Since group 1 and group 2 are in different subtrees of the elimination tree, we can then process group 1 using the base-camp strategy described in the previous paragraph.

In case 2b, the group 2 updates do change Λ_k, but after they are completed, we can reset Λ_k using the present L and U. Following the above argument, in order to apply the group 1 updates we only need to calculate the change in Λ_k. Suppose $\Lambda_k = (L_{C2})_0(U_{B2})_0$ after the group 2 updates, and $\hat{\Lambda}_k = L_{C2}U_{B2}$ after the group 1 updates reach column k. As in (4.16), the correction matrix is derived as

$$\Delta\Lambda_k = (L_{C2})_0(U_{B2})_0 - L_{C2}U_{B2}, \tag{4.17}$$

and it will have low rank. We can now complete the group 1 updates by taking ΔA from (4.17) to complete the update procedures for group 1. This conclusion can be extended to multi-group and multi-base-camp update scenarios by superposition. □

The base-camp theorem can be explained in an algebraic way also. Let $p = \mathrm{card}(\mathcal{L}_k)$, $q = \mathrm{card}(\mathcal{U}_k)$, $r = s(k)$. Each of the r rank-1 updates enters the base-camp as a parameter α, a p-vector x used to update $L(:,k)$, and a q-vector y used to update $U(k,:)$:

$$\begin{array}{lll}
\alpha_1 & x(1:p)_1 & y(1:q)_1 \\
\alpha_2 & x(1:p)_2 & y(1:q)_2 \\
\alpha_3 & x(1:p)_3 & y(1:q)_3 \\
\cdots & & \\
\alpha_r & x(1:p)_r & y(1:q)_r\,.
\end{array}$$

In the multiple-rank update algorithm, x, y, $L(:,k)$, and $U(k,:)$ are updated as follows:

```
calculate $\beta_1$ $\gamma_1$
    $x(1:p)_1 = x(1:p)_1 - L(:,k) * x(k)_1$    $y(1:q)_1 = y(1:q)_1 - U(k,:) * y(k)_1$
    $L(:,k) \;= L(:,k) + \beta_1 x(1:p)_1$         $U(k,:) \;= U(k,:) + \gamma_1 y(1:q)_1$
calculate $\beta_2$ $\gamma_2$
    $x(1:p)_2 = x(1:p)_2 - L(:,k) * x(k)_2$    $y(1:q)_2 = y(1:q)_2 - U(k,:) * y(k)_2$
    $L(:,k) \;= L(:,k) + \beta_2 x(1:p)_2$         $U(k,:) \;= U(k,:) + \gamma_2 y(1:q)_2$
                 ...
calculate $\beta_r$ $\gamma_r$
    $x(1:p)_r = x(1:p)_r - L(:,k) * x(k)_r$    $y(1:q)_r = y(1:q)_r - U(k,:) * y(k)_r$
    $L(:,k) \;= L(:,k) + \beta_r x(1:p)_r$         $U(k,:) \;= U(k,:) + \gamma_r y(1:q)_r$ .
```

In other words, $L(:,k)$ is a linear combination of the initial $L(:,k)$ and $x_1, \ldots, x_r$, and $U(k,:)$ is a linear combination of the initial $U(k,:)$ and $y_1, \ldots, y_r$. Assuming $a_1, \ldots, a_p \in \mathbb{R}^p$ are independent, we can express $L(:,k)$, $x(1:p)_1, \ldots, x(1:p)_r$ as a linear combination of these vectors:

$$\begin{aligned}
L(:,k) \; &\in \mathrm{span}\{a_1, \ldots, a_p\} \\
x(1:p)_1 &\in \mathrm{span}\{a_1, \ldots, a_p\} \\
x(1:p)_2 &\in \mathrm{span}\{a_1, \ldots, a_p\} \\
&\cdots \\
x(1:p)_r &\in \mathrm{span}\{a_1, \ldots, a_p\}\,.
\end{aligned} \tag{4.18}$$

Therefore, the final updated $L(:,k)$ can be expressed as a linear combination of $a_1, \ldots, a_p$, which are equivalent to p rank-1 updates. A similar conclusion can be reached for $U(k,:)$. If r is bigger than p, the input rank-1s can be merged to improve performance.

4.4 Sparse matrix factorization

When solving a system of the form (4.1), it is effective to get a factorization of $A+\Delta A$ by the update strategy when ΔA has low rank. Performance can be improved fundamentally if we build base-camps during the update procedure. The parallel update procedure can have better load balance because the data flow is reorganized when base-camps are built along critical paths.

When $s(k) > \kappa$, it is obvious that we will get good performance if we build a base-camp at column k, because computational effort is saved $O(\frac{s(k)}{\kappa})$ times. If there are m base-camps at columns k_1, k_2, …, k_m in a path of the elimination tree, the total computation effort that could be saved

$$\sum_{i=1}^{m} O\left(\frac{s(k_i)}{\kappa}\right)$$

times just for this path in elimination tree. For example, a tritriangular matrix with $O(n)$ updates can be done in $O(n)$ by base-camp algorithm instead of $O(n^2)$ by multiple rank-1 updates.

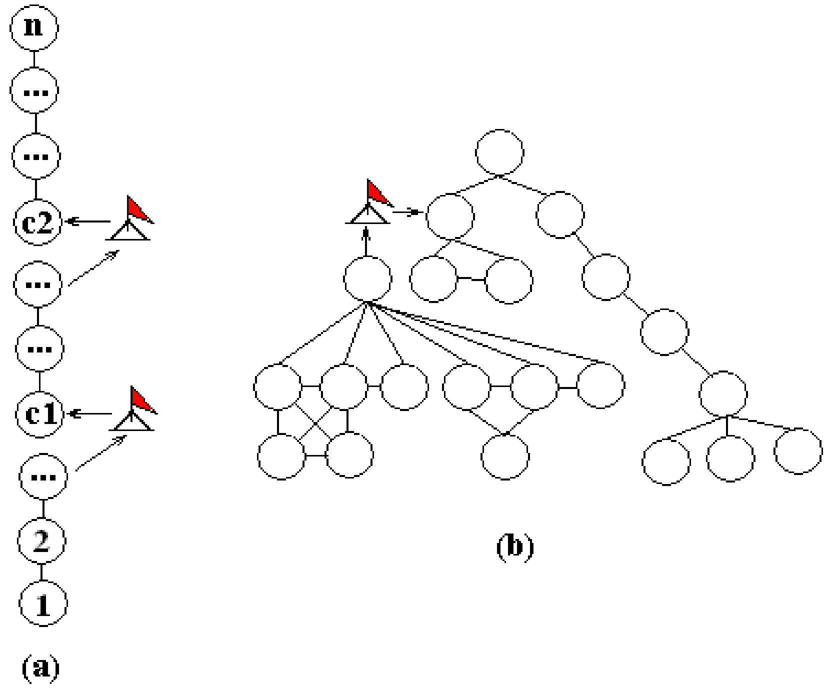


Figure 4.6: Illustration of the base-camp setup.

It is not obvious how to get good *parallel* performance by building base-camps. Suppose there is a long thin path like the one for a tridiagonal matrix. There is not much opportunity to factorize this matrix in parallel. The optimized computing effort for its factorization is $3n$ flops. If we build m base-camps as shown in Figure 4.6(a) and apply truncation when the update vectors become very small, the total computational effort will be $O(\frac{n}{m} + log(\frac{1}{\epsilon}))$, where ϵ is the convergence tolerance. This may be the fastest algorithm for factorization of tridiagonal matrix. In general, base-camps can be used to define parallel regions, to balance load, and to control data traffic. Many update flops can be saved if we set up a base-camp in the elimination graph shown in Figure 4.6(b).

To give base-camps the ability to lower computational effort and improve load balance, the rules to build base-camps on candidate master columns k and j (for example), are:

- $s(k) > \kappa$. Performance gain is equal to $O(\frac{s(k)}{\kappa})$.
- Λ_k can not overlap with Λ_j.
- Limiting the length of a long and critical computation path. We can insert base-camps along a long and critical path to make the computation parallel on this path.

$\Delta\Lambda_k = \Lambda_k - \hat{\Lambda}_k$ in (4.15) can be calculated inexpensively if we take advantage of memory localization. The performance can be enhanced further if we consider how to use the cache. In some situations, forming $\Delta\Lambda_k$ by

$$\Delta\Lambda_k = L_{\hat{S}}U_{\hat{S}} - L_S U_S \tag{4.19}$$

is more effective if κ is very small.

When $\Delta\Lambda_k$ is formed, multiple rank-1s can be formed in the following ways:

- Treat one column or one row in $\Delta\Lambda_k$ as a rank-1 update.
- Use matrix decompositions like Cholesky, SVD, or QR.
- Use SVD approximation.

The first is a straightforward one that leads to the exact update. The last can be the fastest, and the accuracy can be controlled.

Finally, Table 4.1 gives the sparse-matrix update algorithm, Algorithm 2, to solve (4.1).

Table 4.1: Algorithm 2: Sparse-matrix update algorithm.

```
Define base-camps.
Define multiple rank-1 update groups.
do {
    multiple rank-1 updates
    if (base-camp is not full)
        continue
    merge base-camp rank-1s
    form new multiple rank-1 update group
} while (base-camp is not empty)
```

4.5 Left- and right-looking LU factorization

There are two main types of method to do LU factorization: left-looking and right-looking. For the left-looking method, LU decomposition is performed column by column. For example, when column j is processed, only data in columns to the left of column j are needed. The pseudo-code in Table 4.2 illustrates the left-looking method.

Table 4.2: A typical left-looking method.

```
for column j = 1 to n do
    f = A( : ,j)
    Symbolic factor: determine which columns of L will update f
    for each updating column $r < j$ in topological order do
        Col-col update: f = f − f(r)L(r+1:n,r)
    end for
    Pivot: interchange f(j) and f(m), where f(m) = max(f(j:n))
    Separate L and U: U(1:j, j) = f(1:j); L(j:n, j) = f(j:n)
    Scale: L( : , j) = L( : ,j)/L(j,j)
 Prune symbolic structure based on column j
end for
```

The main computation effort in a left-looking method is to solve a triangular system

$$\begin{bmatrix} 1 & & & & & \\ \times & 1 & & & & \\ \times & \times & 1 & & & \\ \times & \times & \times & 1 & & \\ \times & \times & \times & 0 & 1 & \\ \times & \times & \times & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix}.$$

For a right-looking method, LU decomposition is performed in a left-to-right and top-to-bottom manner (hence the name right-looking). The pseudo-code shown in Table 4.3 illustrates the right-looking method.

Table 4.3: A typical right-looking method.

```
Loop k from 1 to N:
    Loop i from k+1 to N:
        A(i,k) = A(i,k) / A(k,k);
    endLoop
    Loop i from k+1 to N:
        Loop j from k+1 to N:
            A(i,j) = A(i,j) − A(i,k) * A(k,j);
        endLoop
    endLoop
endLoop
```

In a typical step of dense matrix factorization, a left-looking method needs $O(n^2)$ reads and $O(n)$ writes, while a right-looking method needs $O(n^2)$ writes and $O(n)$ reads. Left-looking is more memory efficient than right-looking, while right-looking has a better parallel computing pattern.

In sparse-matrix factorization, some sub-trees are good candidates for left-looking processing.

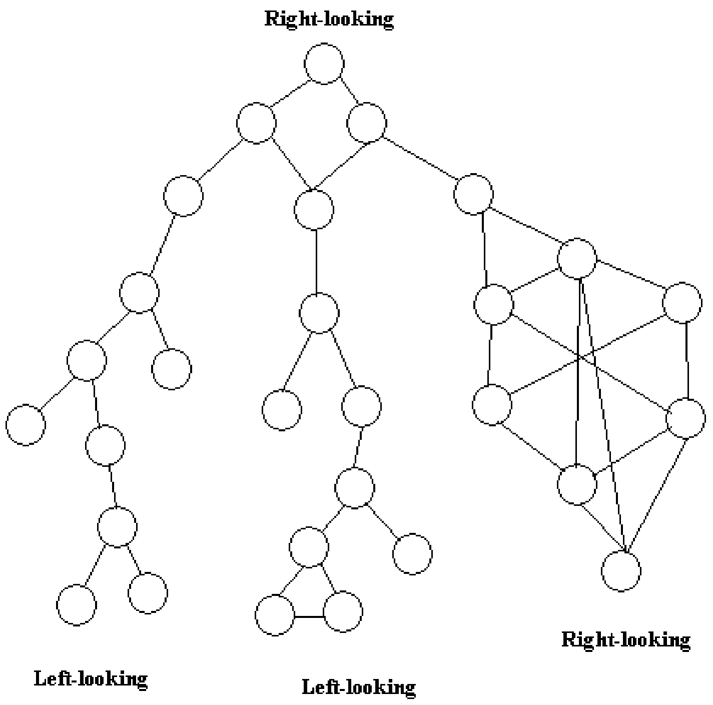


Figure 4.7: Elimination tree with different features for left- and right-looking methods.

Other sub-trees may be good for right-looking processing. For example, Figure 4.7 shows the ideal blocks for different methods. In general, the parts having few fill-ins and small bandwidth are suitable for a left-looking method. The parts that depend on many other parts are suitable for a right-looking method so that the surrounding parallel computing can take advantage.

To take full advantage of left-looking and right-looking methods, we need an algorithm that uses both methods, switching between the methods at judicious times. The details are explained below.

When the LU factorization of a matrix A reaches column k, the factors are of the form

$$A = \begin{bmatrix} A_1 & B \\ C & D \end{bmatrix} = \begin{bmatrix} L_{A_1} & \\ L_C & I \end{bmatrix} \begin{bmatrix} U_{A_1} & U_B \\ & S \end{bmatrix},$$

where A_1 is $(k-1) \times (k-1)$ and $S = D - L_C U_B$ is the Schur complement of A_1. If we factorize $S = L_S U_S$, we can get the full factorization as

$$\begin{bmatrix} L_{A_1} & \\ L_C & I \end{bmatrix} \begin{bmatrix} U_{A_1} & U_B \\ & S \end{bmatrix} = \begin{bmatrix} L_{A_1} & \\ L_C & L_S \end{bmatrix} \begin{bmatrix} U_{A_1} & U_B \\ & U_S \end{bmatrix}. \tag{4.20}$$

Thus, S is an independent block in the factorization. Any method can be used to get its factorization. At each stage, a left-looking method forms only the first column of U_B and S in order to proceed.

A right-looking method forms all of S, and its first row becomes a new row of U_B. Thus, when we reach stage k, U_B and S are not available in a left-looking method, but both are complete in a right-looking method.

The key to transfer between left-looking and right-looking is to form S. If we use right-looking method to process the first $k-1$ steps, S is formed automatically, and we can transfer to a left-looking factorization of S without cost to get L_sU_s and complete the process. If we use a left-looking method to process the first $k-1$ columns, we have $\{L_{A1},\ L_C,\ U_{A1}\}$, and $\{U_B,\ S\}$ can be computed implicitly or explicitly as

$$\begin{aligned} U_B &= L_{A1}^{-1}B \\ S &= D - L_C U_B. \end{aligned}$$

Then we can use a right-looking method to get L_sU_s from S.

Thus, the complete sparse-matrix factorization is a combination of left-looking, right-looking, and multiple rank-1 updates, as shown in Table 4.4.

Table 4.4: Algorithm 3. Sparse matrix hybrid factorization method.

```
Symbolic analysis including MD, AMD, cluster analysis
Analysis elimination tree to get M blocks
Build elimination tree for each blocks
Use DF to travel blocks
    For each block, process factorization by one of following methods:
        left-looking
        right-looking
        multiple-updates
        group update
```

4.6 Distributed computing

A divide-and-conquer algorithm is ideal for shared-memory or distributed parallel computing. Matrix factorization can be implemented by this approach through multiple-rank updates.

We can use a cluster analysis algorithm to partition the matrix into blocks, and build a hierarchical matrix of the form

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}.$$

In cluster analysis, the connections inside clusters (blocks) are stronger than the connections between clusters, which means that the A_{ij} are much sparser than A_{ii}. In other words, A_{ij} are low-rank matrices. We can rewrite A_{ij} as the summation of rank-1 matrices as described in earlier chapters.

This is the divide process. Then the conquer process can be implemented by multiple-rank updates. The core of the conquer process is merging 2×2 blocks into one block. Let us first discuss the 2×2 block

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.$$

This can be presented as $A + \delta A$ in the form

$$\begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} + \begin{bmatrix} 0 & A_{12} \\ A_{21} & 0 \end{bmatrix}.$$

We can get the factorization

$$\begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ 0 & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & 0 \\ 0 & U_{22} \end{bmatrix}$$

while factoring A_{11} and A_{11} independently. The block matrix

$$\begin{bmatrix} 0 & A_{12} \\ A_{21} & 0 \end{bmatrix}$$

can be rewrittene as the sum of rank-1 matrices as $\sum_{i=1}^{r} x_i y_i^T$. Thus we can use multiple-rank updates to get the factors of this 2×2 blocks as

$$\begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} + \begin{bmatrix} 0 & A_{12} \\ A_{21} & 0 \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ 0 & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & 0 \\ 0 & U_{22} \end{bmatrix} + \sum_{i=1}^{r} x_i y_i^T = LU.$$

By following the standard (recursive) divide-and-conquer procedure, we obtain the factors of the full matrix.

4.7 Symmetric update

The general symmetric matrix A update procedure can be defined as

$$A = LL^T + uu^T.$$

Some of the original work was done by Gill, Golub, Murray and Saunders [GGMS74]. More recently, Davis and Hager [DH99] extended such methods to sparse A. In applications, how to form rank-1 matrices from ΔA is a big challenge. Details are presented in Chapter 5.

Chapter 5

Symmetric Factorization by Multiple-Rank Updates

5.1 Introduction

Structural design, reliability analysis, and design optimization require repeated analyses of progressively modified structures. This leads to the need to solve systems of equations of the form $(K + \Delta K)\bar{\delta} = R$, where K is the stiffness matrix, ΔK represents structural changes in the matrix, $\bar{\delta}$ is the deformation vector, and R is the applied force vector. Both K and ΔK are symmetric matrices. $K + \Delta K$ is symmetric positive definite when the structure analyzed is in a stable state. In general, ΔK is a low-rank matrix and highly sparse.

Traditionally, $(K + \Delta K)\bar{\delta} = R$ is solved by factoring $K + \Delta K$. Although computational power has increased substantially in recent years, the computational requirement is still excessive when reliability analysis or optimization of nonlinear structures is performed, or when extremely fine meshes are needed for the analysis of large-scale structures, or when more accurate models are introduced. To reduce the computational effort in these situations, researchers have developed various reanalysis techniques, as described by Deng and Ghosn [DG01]. The most common techniques may be grouped into three categories:

a) distortion-based methods;

b) methods based on the Sherman-Morrison identity;

c) methods based on updating matrix factors.

The basic concept in the "distortion" family is that the response of the modified structure is modeled as the sum of the response of the applied loads on the original structure and the response of the original structure to virtual distortions, where the distortions are imposed on the members whose properties are being modified. In these methods, the types of virtual (or pseudo) distortions vary with the type of structure. Thus, different distortion equations need to be developed for trusses, frames,

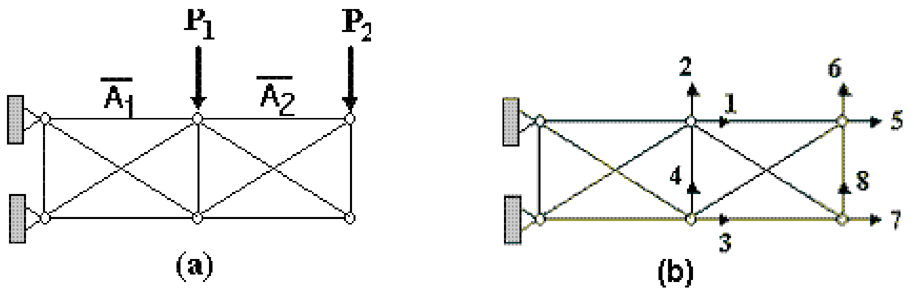


Figure 5.1: Illustration of updating strategy for simple truss problem.

or continuous structures like plates and shells. (See for example the work of Gierlinski, Sears and Shetty [GSS93]; Arora [Aro76]; McGuire and Gallagher [MG79]; Abu-Kassim and Topping [AKT87]; Kirsh and Moses [KM95]; Kirsh and Rubinstein [KR70]; Makode, Corotis and SnRamirez [MR99] and Holnichi [HS91])

Researchers who developed methods based on the Sherman-Morrison identity [SM49] include Abu-Kassim and Topping [AKT87], Arora [Aro76], Sack, Carpenter, and Hatch [SCH67], Kirsch and Rubinstein [KR70, KR72], Mohraz and Wright [MW73], Wang and Pilkey [WP80, WP81], Wang, Pilkey,and Palazzola [WPP83], Hirai, Wang and Pilkey [HWP84], and more recently, Deng and Ghosn [DG01].

The last group of methods, which provide exact solutions applicable to nonlinear structural reanalysis problems, were developed by Bennett [Ben65], Gill, Golub, Murray and Saunders [GM74], Law [Law85, Law89], Chan and Brandwajn [CB86], and Davis and Hager [DH99]. These methods had some limitations on their stability, or still required considerable computational effort.

Deng and Ghosn [DG01] proposed a solver by defining the response of the modified structure as the difference between the response of the original structure to a set of applied loads and to a set of pseudoforces. Although this method showed good improvement in efficiency, the method was found not to be stable as the stiffness matrix became ill-conditioned. In addition, it was found that the size of the inner condensed matrix was growing as the analysis process continued over hundreds of iterations. This required the establishment of a restart strategy to reinitiate the algorithm.

Building on the experience gained from previous work on structural reanalysis techniques and matrix solution methods, we propose a new solver that updates matrix factors based on eigenvalue decomposition (EVD) of the change to the matrix. The next sections demonstrate that the proposed method has many advantages, including stability, efficiency comparable to other methods, and ability to handle sequences of modified stiffness coefficients without requiring the restart strategy of Deng and Ghosn [DG01].

5.2 Matrix updating strategies

To demonstrate matrix-updating strategies in structural applications consider a truss whose original members' cross sectional areas are given as A_1, A_2, A_3, ..., A_f. Suppose that the cross-section areas of members 1 and 2 are changed from A_1 and A_2 to $\bar{A}_1$ and $\bar{A}_2$ respectively, as shown in Figure 5.1. The stiffness matrix of the modified structure $\bar{K}$ is equal to the stiffness matrix of the original structure K plus a correction matrix ΔK. The finite element stiffness equation can be expressed as

$$\bar{K}\bar{\delta} = (K + \Delta K)\bar{\delta} = R, \tag{5.1}$$

where $\bar{\delta}$ is the deflection of the modified structure and R is the set of applied forces. Usually, only a few members change their properties during one reanalysis step. Thus, the correction matrix ΔK is highly sparse. For example, when the unknown displacements are numbered as shown in Figure 5.1b, the stiffness matrix can be expressed as

$$\bar{K} = K + \Delta K \tag{5.2}$$

or

$$\begin{bmatrix} \bar{k}_{11} & \bar{k}_{21} & \dots & \dots & \bar{k}_{18} \\ \bar{k}_{21} & \bar{k}_{22} & \dots & \dots & \bar{k}_{28} \\ \dots & & & & \\ \dots & & & & \\ \bar{k}_{81} & \bar{k}_{82} & \dots & \dots & \bar{k}_{88} \end{bmatrix} = \begin{bmatrix} k_{11} & k_{21} & \dots & \dots & k_{18} \\ k_{21} & k_{22} & \dots & \dots & k_{28} \\ \dots & & & & \\ \dots & & & & \\ k_{81} & k_{82} & \dots & \dots & k_{88} \end{bmatrix} + \begin{bmatrix} s_{11} & \dots & s_{15} & \dots & 0 \\ \dots & 0 & 0 & 0 & \dots \\ s_{51} & \dots & s_{55} & \dots & 0 \\ \dots & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{5.3}$$

For illustration, and without loss of generality, the matrix ΔK in (5.3) has only four nonzero entries. For large-scale systems, the number of nonzero entries in ΔK is much smaller than the dimension of K, and the matrix ΔK can be expressed as the multiplication of two low-rank matrices P and W:

$$\Delta K = PWP^T, \tag{5.4}$$

where P is an $n \times d$ pointer matrix, n is the dimension of ΔK, and d is the number of nonzero columns in ΔK. P is called a pointer matrix because it points to the modified Degrees of Freedom (DOF) of the modified members. The elements of P are either 0 or 1. The matrix W is the compressed stiffness matrix, which contains the nonzero entries of ΔK. The rank of P is d, and rank$(W) \leq d$. As an illustration, ΔK for the structure in Figure 5.1 can be expressed as

$$\begin{bmatrix} s_{11} & 0 & 0 & 0 & s_{15} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ s_{11} & 0 & 0 & 0 & s_{15} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} s_{11} & s_{15} \\ s_{51} & s_{55} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}^T . \tag{5.5}$$

The rank of W is usually much less than n. Therefore, its EVD can be executed at low cost as follows:

$$W = V\Lambda V^T = \sum_{i=1}^{d} \lambda_i v_i v_i^T. \tag{5.6}$$

This implies that ΔK can be decomposed as a series of rank-one modifications:

$$\begin{aligned} \Delta K &= PWP^T \\ &= P\left(\sum_{i=1}^{d} \lambda_i v_i {v_i}^T\right) P^T \\ &= \sum_{i=1}^{d} \lambda_i z_i {z_i}^T, \quad \text{in which } z_i = Pv_i. \end{aligned} \tag{5.7}$$

If W is not symmetric, which may be the case when solving other engineering simulation problems such as circuit function verification, the Singular Value Decomposition (SVD) can be used to decompose W as

$$W = U\Sigma V^T = \sum_{i=1}^{d} \sigma_i u_i v_i^T. \tag{5.8}$$

As in the symmetric case, ΔK can be decomposed as a series of rank-one modifications:

$$\begin{aligned} \Delta K &= PWP^T \\ &= P\left(\sum_{i=1}^{d} \sigma_i u_i {v_i}^T\right) P^T \\ &= \sum_{i=1}^{d} \sigma_i z_i {y_i}^T, \quad \text{in which } z_i = Pu_i,\ y_i = Pv_i. \end{aligned} \tag{5.9}$$

Thus, the general simulation matrix $K + \Delta K$, where K and ΔK are not symmetric, can be written as

$$K + \Delta K = K + \sigma_1 z_1 y_1^T + \sigma_2 z_2 y_2^T + \cdots + \sigma_d z_d y_d^T. \tag{5.10}$$

Since K and ΔK are symmetric in structural analysis problems, it is sufficient to use EVD to represent ΔK as described in equation (5.7). Let α_1, α_2, …, be the eigenvalues of ΔK ordered according to decreasing absolute value. Equation (5.2) becomes

$$K + \Delta K = K + \alpha_1 z_1 z_1^T + \alpha_2 z_2 z_2^T + \cdots + \alpha_d z_d z_d^T. \tag{5.11}$$

The reason to order the α_i according to absolute value is to minimize truncation error when fewer than d terms are included.

The setup presented in (5.11) allows for factorization of $\bar{K} = K + \Delta K$ efficiently and in a stable manner. If the factorization $K = LDL^T$ is known, its rank-one modification $K + \alpha zz^T$ can be

```
Given $K = LDL^T$ and $\Delta K = \sum_{i=1}^{d} \alpha_i z_i z_i^T$
For $i = 1, 2, \ldots, d$
    $[L, D] =$ UPDATE$(L, D, \alpha_i, z_i)$
```

Figure 5.2: Algorithm for recursive factorization of $\bar{K}$ matrix.

factorized as $\bar{L}\bar{D}\bar{L}^T$, where $\bar{L}$ and $\bar{D}$ can be easily determined from L and D using the methods described by Gill, Golub, Murray, and Saunders (1974). Thus, the factors of $K + \Delta K$ can be determined incrementally, as shown in Figure 5.2.

To obtain an accurate decomposition of W, all d eigenvalue terms of (5.11) must be included in the expansion. However, for practical purposes and to improve the efficiency of the process, accurate results can often still be reached if we use a matrix W_r of rank r to represent the actual W, where r is much smaller than d:

$$\begin{aligned} W_r &= \sum_{i=1}^{r} \alpha_i v_i v_i^T, \\ \|W - W_r\| &\leq |\alpha_{r+1}|. \end{aligned} \tag{5.12}$$

Thus, we can reduce the rank of W while keeping good accuracy by specifying an error limit τ on the norm, so that $\|W - W_r\| < \tau$, where τ can be specified as a percentage of the largest eigenvalue depending on how much accuracy is required. This is a significant advantage for using the EVD. Similarly for unsymmetric matrices using the SVD.

Based on the above derivations, the updating algorithm for structural re-analysis problems requires the following steps:

1. Assemble the stiffness matrix K for the original structure and determine the loading vector R.

2. Perform the analysis using the LDL^T factorization method.

3. Determine the pointer matrix P associated with the degrees of freedom of the changed structure having the stiffness matrix $K + \Delta K$, with $\Delta K = PWP^T$.

4. Compute the EVD of W and use the results to decompose ΔK by finding the eigenpairs (λ_i, v_i) of W and the associated eigenpairs (λ_i, z_i) of ΔK (with $z_i = Pv_i$), and reorder λ_i based on their absolute values to get α_i.

5. Set $k = 1$ to start the factorization process.

6. Update the factorization $LDL^T \leftarrow LDL^T + \alpha_k z_k z_k^T$.

7. Set $k = k + 1$ and repeat step 6 to cover all the eigenvalue factors. Alternatively, repeat step 6 for $k = 1, \ldots, r$, where r is selected so that W_r gives a reasonable approximation to W such that $\|W - W_r\| < \tau$.

8. Find the solution of the changed structure with the stiffness matrix $K + \Delta K$.

9. Change K into a new matrix $K + \Delta K$ as required by the structural analysis, optimization, or reliability search algorithm, and go to step 3 (until all required structural changes are covered).

The efficiency of the analysis process depends on steps 4 and 6. Step 6 can be speeded up by a "group update" as discussed in Chapter 3 and Chapter 4. As Davis and Hager [DH01] describe, symmetric rank-1 updates require repeated access to the whole matrix L, while the group update has better memory traffic because it makes a single pass through L and therefore is computationally more efficient. Here we described the sequential rank-1 updating procedure for simplicity, even though we recommend multiple-rank updating in practical implementations.

Although several routines are available to execute these steps in an efficient manner, additional approximations can be introduced to speed up the analysis process further. The next sections describe the methods used to perform these analysis steps.

5.3 Implementation of EVD and SVD algorithms

The efficiency of the proposed analysis procedure depends on the EVD or SVD of ΔK. Developing methods for EVD and SVD is an active research field in applied mathematics, and several researchers have proposed efficient algorithms (see for example Demmel [Dem97] or Golub and Van Loan [GVL96]). Some standard algorithms follow.

Jacobi's method This is a classical yet effective algorithm for finding the EVD of a dense matrix [DV07].

QR iteration and its variations Properly implemented, this is the fastest algorithm to find all eigenvalues of a tridiagonal matrix or all singular values of a bidiagonal matrix with high relative accuracy [FB94].

Divide-and-conquer This is currently the fastest method to find all singular values and singular vectors for large matrices (with $n > 25$ according to Demmel [Dem97]).

Bisection and inverse iteration This is used to find only the eigenvalues or singular values that lie within a desired interval.

5.4 Updating LDL^T factorization

As mentioned earlier, the updates (5.10) can be executed recursively by progressively adding the EVD terms one at a time. For each additional term obtained from the EVD one can assume that the symmetric positive definite matrix K of the previous step is modified by a symmetric matrix of rank one such that

$$\bar{K} = K + \alpha z z^T. \tag{5.13}$$

Assuming that K has been previously decomposed as $K = LDL^T$, we wish to determine the factors of $\bar{K}$:

$$\bar{K} = \bar{L}\bar{D}\bar{L}^T. \tag{5.14}$$

Through a basic transformation, (5.13) can be rewritten as

$$\bar{K} = K + \alpha zz^T = L(D + \alpha pp^T)L^T, \text{ where } Lp = z. \tag{5.15}$$

If we perform the factorization

$$D + \alpha pp^T = \tilde{L}\tilde{D}\tilde{L}^T, \tag{5.16}$$

the required modified Cholesky factors will take the form

$$\bar{K} = L\tilde{L}\tilde{D}\tilde{L}^T L^T, \tag{5.17}$$

where $\bar{L} = L\tilde{L}$ and $\bar{D} = \tilde{D}$, since the product of two lower-triangular matrices is a lower triangular matrix. Because $\tilde{L}$ is a structured matrix, $L\tilde{L}$ can be calculated in $O(n^2)$ operations as shown by Gill, Golub, Murray and Saunders [GGMS74]. If the factorization of (5.16) can also be executed within $O(n^2)$ operations, then the total number of operations needed to obtain $\bar{K}$ will remain $O(n^2)$. Thus, the efficiency of the method depends on the efficiency of the factorization performed in (5.16).

Gill, Golub, Murray and Saunders [GGMS74] proposed four different methods to update the LDL^T matrices to obtain the required $\bar{L}\bar{D}\bar{L}^T$ matrix. These methods are described and their stability is thoroughly discussed in their landmark paper. Because of space limitations, only the first two methods are reviewed here.

5.4.1 Application of classical Cholesky factorization

The Cholesky factorization of $D + \alpha pp^T$ in (5.15) can be performed directly knowing that the subdiagonal elements of the jth column of $\tilde{L}$ are multiples of the corresponding elements of the vector p. This can be expressed by (see [GGMS74])

$$\tilde{l}_{rj} = p_r\beta_j, \quad r = j+1, \dots n; \quad j = 1, 2, \dots, n, \tag{5.18}$$

where

$$\begin{aligned} \beta_j &= \frac{p_j}{\tilde{d}_j}\left[\alpha - \sum_{i=1}^{j-1}\tilde{d}_i\beta_i^2\right], \\ \tilde{d}_j &= d_i + \alpha p_i^2 - p_i^2\sum_{i=1}^{j-1}\tilde{d}_i\beta_i^2. \end{aligned} \tag{5.19}$$

```
1   Given $K = LDL^T$, determine the factors of $K + \alpha zz^T$

2   for $j = 1 : N$
        $p = z(j); dj = D(j);$
        $D(j) = D(j) + alfa * p * p;$
        $beta = p * alfa/D(j);$
        $alfa = dj * alfa/D(j);$
        for $k = j + 1 : N$
            $z(k) = z(k) - p * L(k, j);$
            $L(k, j) = L(k, j) + beta * z(k);$
        end
    end
```

Figure 5.3: Updating LDL^T using classic Cholesky method.

This result implies that we need only compute the values of $\tilde{d}_j$, β_j, $j = 1, \dots, n$ in order to obtain the factors of $D + \alpha pp^T$. In practice, we can define the auxiliary quantity

$$\alpha_j = \alpha - \sum_{i=1}^{j-1} \tilde{d}_i \beta_i^2 \tag{5.20}$$

and compute the product $\bar{L} = L\tilde{L}$ in terms of β_i to form the algorithm given in Figure 5.3 [GGMS74].

Note that this algorithm is so simple that only nine lines of code are needed to implement it. The number of operations necessary to compute the modified factorization using this algorithm is $n^2 + O(n)$ multiplications and $n^2 + O(n)$ additions.

If the matrix K is sufficiently positive definite, that is, its smallest ordered eigenvalue is sufficiently large relative to some norm of $\bar{K}$, then this algorithm is numerically stable. However, if $\alpha < 0$ and $\bar{K}$ is nearly singular, it is possible that rounding errors could cause the diagonal elements $\bar{d}_j$ to become zero or arbitrarily small. In such cases, it is also possible that $\bar{d}_j$ could change sign, even when the modification may be known from theoretical analysis to give a positive definite factorization. In such cases, it may be advantageous to use the Householder algorithm described in the next section because its resulting matrix will always be positive definite regardless of any numerical errors. For example, a structure matrix is ill-conditioned and close to singular when the structure bears a load close to its ultimate value. We need to factor this matrix when we calculate the limit state of a structure.

5.4.2 Application of Householder matrix method

As explained by Gill et al. [GGMS74], the factorization of (5.15) can be performed using Householder matrices. This is achieved by first rearranging the equation into the following format:

$$\bar{K} = K + \alpha zz^T = LD^{1/2}(I + \alpha vv^T)D^{1/2}L^T,$$

where v is the solution of $LD^{1/2}v = z$.

The matrix $I + \alpha vv^T$ can be factorized into the form

$$I + \alpha vv^T = (I + \sigma vv^T)(I + \sigma vv^T). \tag{5.21}$$

By choosing

$$\sigma = \frac{\alpha}{(1 + \sigma v^T v)^{1/2}}, \tag{5.22}$$

we ensure that the expression under the root sign is a positive multiple of the determinant of $\bar{K}$. If $\bar{K}$ is positive definite, σ will be real.

The QR factorization $I+\sigma vv^T = QR$ can be computed with Q a product of Householder matrices and $\bar{R}$ upper triangular. Thus, (5.21) can be continuously decomposed as

$$\begin{aligned}\bar{K} &= K + \alpha zz^T \\ &= LD^{1/2}(I + \alpha vv^T)D^{1/2}L^T \\ &= LD^{1/2}(I + \sigma vv^T)(I + \sigma vv^T)D^{1/2}L^T \\ &= LD^{1/2}(I + \sigma vv^T)^T(I + \sigma vv^T)D^{1/2}L^T \\ &= LD^{1/2}R^TQ^TQRD^{1/2}L^T \\ &= LD^{1/2}\widetilde{L}\widetilde{L}^TD^{1/2}L^T \\ &= \bar{L}\bar{D}\bar{L}^T,\end{aligned} \tag{5.23}$$

where $\widetilde{L} = R^T$.

The algorithm to find $\bar{L}$ is described as Algorithm C2 in Gill, Golub, Murray and Saunders [GGMS74]. The algorithm's pseudocode is summarized in Figure 5.4. The advantage of the Householder method over the Cholesky method lies in the fact that the algorithm will always yield a positive-definite factorization even under extreme circumstances. The method requires $3n^2/2+O(n)$ multiplications and $n + 1$ square roots. We usually face singularity problems during limit analysis of nonlinear structures. This algorithm is expected to overcome these issues to help trace the full behavior of a structure from the initiation of loading through ultimate loading and during post-peak unloading until failure.

5.5 Analysis of efficiency

As described in the previous sections, the total number of flops needed for the implementation of the recursive LDL^T algorithm adds up to:

$$2rmn + O(d^3), \quad r \le d \tag{5.24}$$

where d is the size of the compressed matrix W, r is the number of EVD terms used to approximate W, n is the size of matrix K, and m is the average bandwidth of K.

1	Solve $Lp = z$
2	Define
	$w_j^{(1)} = z_j$
	$s_j = \sum_{i=j}^{n} p_i^2/d_i \equiv \sum_{i=j}^{n} q_i, \quad j = 1, 2, \cdots n$
	$\alpha_1 = \alpha$
	$\sigma_1 = \alpha/[1 + (1 + \alpha s_1)^{1/2}]$
3	For $j = 1, 2, \ldots, n$, compute
	$q_j = p_j^2/d_j$
	$\theta_j = 1 + \sigma_j q_j$
	$s_{j+1} = s_j - q_j$
	$\rho_j^2 = \theta_j^2 + \sigma_j^2 q_j s_{j+1}$
	$\bar{d}_j = \rho_j^2 d_j$
	$\beta_j = \alpha_j p_j/\bar{d}_j$
	$\alpha_{j+1} = \alpha_j/\rho_j^2$
	$\sigma_{j+1} = \sigma_j(1 + \rho_j)/[\rho_j(\theta_j + \rho_j)]$
	$w_r^{(j+1)} = w_r^{(j)} - p_j l_r j$
	$l_{rj} = l_{rj} + \beta_j w_r^{(j+1)}, \quad r = j + 1, j + 2, \cdots, n.$

Figure 5.4: Updating LDLT using Householder matrices Calculation of $\bar{L}$ based on Gill et al. [GGMS74].

If the Sherman-Morrison-Woodbury update is used, Equation (5.1) can be solved via the expression (Deng & Ghosn [DG01])

$$\bar{\delta} = K^{-1}R - K^{-1}U(I + VK^{-1}U)^{-1}VK^{-1}R. \tag{5.25}$$

The computational effort needed for (5.25) includes $d+3$ backward and forward operations on matrix K and the solution of a linear system of size d. Thus, the total flops for the Sherman-Morrison-Woodbury update are

$$2(d + 3)mn + O(d^3). \tag{5.26}$$

Comparing (5.24) and (5.26), we see that the proposed new algorithm would be faster than the Sherman-Morrison-Woodbury update by a factor $(d + 3)/r$. The improvement in efficiency depends on the size of r. If no approximation for W is used and all the EVD factors are included in the decomposition, then the two algorithms will have about the same efficiency.

Although the proposed algorithm shows improved efficiency, the primary benefit of the new algorithm as pointed out by Gill, Murray and Wright [GMH91] is mainly in its improved stability as the matrix becomes ill-conditioned. Another advantage of the new algorithm lies in its ability to handle large changes in the stiffness matrix without a need to reinitiate the factorization process regardless of the growth in the compressed matrix W, as was the case in the original algorithm solved by Deng and Ghosn [DG01].

Further improvements to the algorithm's efficiency can be introduced through some additional approximations as described in the next section.

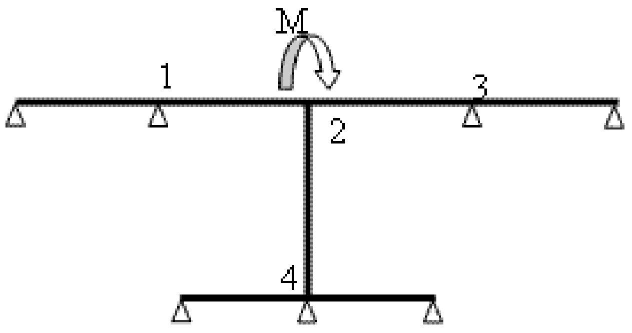


Figure 5.5: Illustration of a structure having high potential for sparsification.

5.6 Stiffness matrix sparsification and decoupling

It is well known that EVD computation for a matrix $W \in \mathbb{R}^{d\times d}$ requires $O(d^3)$ operations. If W can be decoupled into unrelated smaller blocks, then improved efficiency could be obtained. In a nonlinear structural analysis problem, the matrix W represents the stiffness terms that undergo plastification at the particular iteration step. Under normal circumstances, plastification occurs very gradually, affecting only a small number of degrees of freedom that are not connected or only weakly connected to the other DOF's in the system. The matrix W can then be decoupled into smaller matrices. This process requires eliminating the stiffness coefficients of the weakly connected DOF's from the stiffness matrix. The terms in the ΔK and W matrices with very low values are set to 0. This allows us to break the weak inter-block connections, and creates a set of independent sub-block matrices that are decoupled from the original W matrix. This process is known as *sparsification.*

To describe the sparsification process as it applies to structural analysis problems, consider the structure in Figure 5.5, whose stiffness matrix can be assembled as

$$\begin{bmatrix} k_{11} & k_{12} & & \\ k_{21} & k_{22} & k_{23} & k_{24} \\ & k_{32} & k_{33} & \\ & k_{42} & & k_{44} \end{bmatrix}. \tag{5.27}$$

According to the moment distribution method, the moment at node 3 can be approximated as

$$M_{32} \approx M\frac{K_{32}}{K_{22}}. \tag{5.28}$$

If $\frac{K_{32}}{K_{22}}$ is small, then the moment at end 3 of beam 2-3 will be very small. Thus, if we set the stiffness coefficient $K_{32} = 0$, the final response of the structure will not be affected significantly. Therefore, if the sparsification conditions are properly established, the error can be limited and estimated in advance. For example, a possible sparsification threshold value τ can be set in advance as a relatively

```
for i = 1, d
   for j = 1, d
   if $|W(i,j)| \leq \tau W(i,i)$ and $|W(i,j)| \leq \tau W(j,j)$
      $W(i,j) = 0$
   end
end
for i = 1, d
   Form $i(V,E)$
end
block = 0,  tmpBloc = 0,  stack(block) = 0

for i = 1, d
   if node i is not marked
      call DSF(i(V, E))
   end
   block++,  stack(block) =  tmpBlock,  tmpBlock = 0
end
DSF(i(V,E))
   Form tmpBlock whose nodes connect with node i by path search.
   Mark nodes in tmpBLock
end
```

Figure 5.6: Pseudo-code for sparsification and decoupling.

small number, say $\tau = 0.001$. Then a search of the stiffness coefficients of W will replace $W(i,j)$ by 0 whenever $|W(i,j)|$ is found to be less than $W(i,i) * \tau$ and $W(j,j) * \tau$.

The object of matrix sparsification by setting the weak $W(i,j)$ terms equal to zero is to break W into a group of independent matrices. This process, known as decoupling, can be executed on the graph associated with W after sparsification. The graph algorithm designates a row i as a graph node and associates with it the structure $i(V,E)$, where V represents the vertex with value i, and E is a list containing the vertices v for which an edge (i,v) represents the nonzero $W(i,j)$. For example, the matrix in (5.27) has the four structures $1(1,2)$, $2(2,1,3,4)$, $3(3,2)$, $4(4,2)$.

The matrix W can be searched using Depth First Search (DFS) to determine the $i(V,E)$ nodes that will subsequently be used to decouple W into smaller matrices [CLR90]. A pseudo-code for the sparsification and decoupling of matrix W is summarized in Figure 5.6.

5.7 Implementation of sparse matrix updating

It is well known that a rank-one update such as (5.15) can make $\bar{L}$ denser than L. If the data structure is poorly managed, $\bar{L}$ may even become fully dense. On the other hand, since the LDL^T factorization of a matrix is unique, the factors L and $\bar{L}$ should have the same symbolic forms if the two original matrices K and $\bar{K} = K + \Delta K$ have the same symbolic form and the same elimination trees, as explained by Duff, Erisman and Reid [DER86]. Hence, all the extra entries in $\bar{L}$ introduced during the recursive rank-one updating process described in the algorithm of Figure 5.3 should cancel

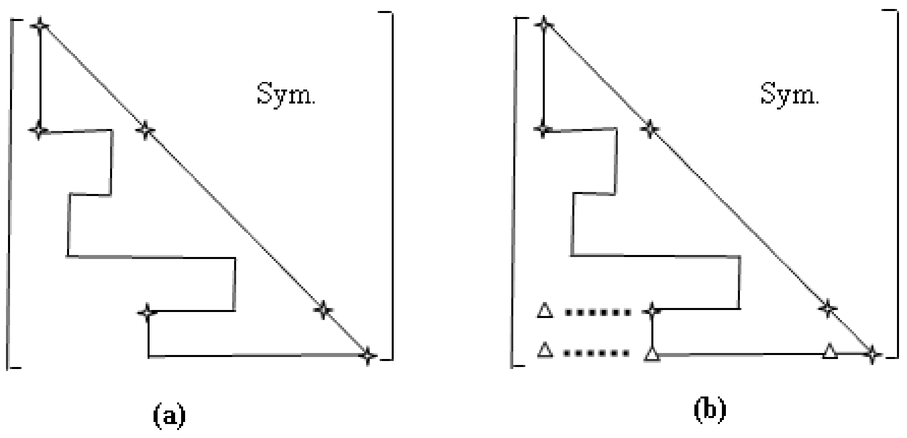


Figure 5.7: Illustration of densification of L during recursive algorithm.

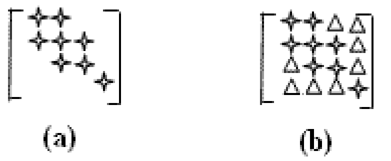


Figure 5.8: Illustration of densification of EVD vectors during recursive algorithm.

when the updating process is complete. It is very helpful to take advantage of this property during the implementation of the recursive factorization algorithm.

The issue with the densification of $\bar{L}$ can be illustrated as shown in Figures 5.7 and 5.8. If matrix K has the varied band structure shown in Figure 5.7(a), L will have the same varied band as K. The stars in Figure 5.7(b) point out the entries that are modified during the nonlinear analysis, so that $K + \Delta K$ has the same terms as K except for the coefficients indicated by the star.

A compressed matrix W can be extracted from K in Figure 5.7(a) and is shown in Figure 5.8(a). Each eigenvector v_i of W (5.6) will produce a full matrix $v_i v_i^T$ as shown in Figure 5.8(b), where the triangles represent extra entries that are introduced by that rank-1 update.

The application of $v_i v_i^T$ to calculate $K + \alpha z z^T$ as in (5.13) during one step of the recursive algorithm will lead to a matrix like that in Figure 5.7(b). New fill-in terms are introduced and represented as dots in Figure 5.7(b). As mentioned above, all the fill-ins introduced at different steps of the recursive algorithm will eventually cancel out as the updating process is completed, so that the final $K + \Delta K$ matrix will have the same structure as that of Figure 5.7(a).

When we use a matrix W_r of rank r to approximate W with rank d as proposed in (5.12), some of the fill-ins will remain in the factors. Furthermore, when W is large, calculating its EVD is not cheap. However, when the compressed W can be decoupled as suggested in section 6, not only will

```
struct column_j {
   double *val;
   double *a_val;
   int *idx;
   int *a_idx;
   int len;
   int a_len;
   int a_active;
};
```

Figure 5.9: Data structure for a typical column of L.

the EVD calculation become more efficient, but also the number of fill-ins introduced during each rank-one updating becomes small.

To make this algorithm practical, implementation is a critical issue. It is necessary to keep $\bar{L}$ as sparse as possible and use cache memory efficiently by considering the structure of vector z. Executing list searches and inserting operations or memory allocation during the algorithm cannot be efficient. Because the entries in a column of L can be computed independently, efficiency can be improved by defining a dynamic column structure for L as described in Figure 5.9.

In Figure 5.9, len gives the nonzero entries in column j of L, and idx and val record these entries' locations and values. Diagonal values are stored as the first elements of vector val. Fill-ins are dynamically changed. It is efficient to allocate a block of memory and free it when the simulation finishes. a_len is the length of this memory. a_active is the actual number of fill-ins introduced during a rank-one update. a_idx and a_val record these fill-ins' locations and values. Based on this structure, a rank-one update can be efficiently executed as described in the algorithm of Figure 5.10. min_index is the smallest global index in W. The first min_index columns in L don't change when $K + \Delta K$ is updated. listL is an array to store L column-wise. This algorithm can be easily implemented to solve nonlinear structural analysis problems in an efficient manner.

5.8 Nonlinear structural analysis

By its very nature, the nonlinear analysis of a structure is an iterative process that can benefit greatly from the improved efficiency and stability that the proposed EVD/SVD-based solver provides. As mentioned earlier, the algorithm remains stable even as the stiffness matrix becomes singular near the ultimate load, as illustrated in Figure 5.11.

The complete nonlinear structural analysis procedure up to iteration i of load step t, as shown in Figure 5.11, can be expressed as

$$\left({}^{0}K^{0} + \sum_{l=1}^{t-1} \sum_{k=1}^{Ts(l)} {}^{l}(\Delta K)^{k} + \sum_{k=1}^{i-1} {}^{t}(\Delta K)^{k} + {}^{t}(\Delta K)^{i} \right) {}^{t}\delta^{i} = {}^{t}R^{i}, \tag{5.29}$$

where ${}^{0}K^{0}$ is the stiffness matrix for the elastic system at load increment $t = 0$ and iteration $i = 0$.

```
// Given $min\_indx, n, listL, z, alfa$.
for ($j = min\_indx;\ j \leq n;\ j{+}{+}$){
    $s = \&listL[j];\ p1 = z[j];\ dj = s \to val[1]$;
    $s \to val[1] \mathrel{+}= alfa * p1 * p1$;
    $beta = p1 * alfa / s \to val[1]$;
    $alfa = dj * alfa / s \to val[1]$;
    for ($k = 2;\ k <= s \to len;\ k{+}{+}$){
        $r = s \to idx[k]$;
        $z[r] \mathrel{-}= p1 * s \to val[k]$;
        $s \to val[k] \mathrel{+}= beta * z[r]$;
    }
    if ($s \to a\_active$){
        for ($k = 1;\ k <= s \to a\_active;\ k{+}{+}$){
            $r = s \to a\_idx[k]$;
            $z[r] \mathrel{-}= p1 * s \to a\_val[k]$;
            $s \to a\_val[k] \mathrel{+}= beta * z[r]$;
        }
    }
}
```

Figure 5.10: Rank-one updating algorithm for a band matrix with EVD modification.

Ts is the total number of iterations needed to find the solution for load step l.

To solve for the displacements ${}^{t}\delta^{i}$, the application of the proposed EVD/SVD-based solver starts by assembling and factorizing the elastic matrix ${}^{0}K^{0}$. The algorithm described above is then followed to find the solution for iteration 1 of load step 1. The process is continued for iteration 2 in load step 1, ..., iteration 1 in load step 2, iteration 2 in load step 2, ... until the nonlinear analysis is completed, or the stiffness matrix becomes not positive definite. The application of the algorithm to solve realistic structural analysis problems is described in the next section.

5.9 Illustrative examples

Example 1 The applicability and efficiency of the proposed solver for the incremental nonlinear analysis of structural systems is demonstrated through the analysis of the same structural grid problem previously solved by Deng and Ghosn [DG01]. The grid represents the model of a 10m bridge (1/3 scale) with three girders, whose elevation and material properties are shown in Figure 5.12. The bridge was tested by Znidaric and Moses [ZM97, ZM98] to investigate its ultimate capacity. In this example, the program NONBAN is used to perform its nonlinear analysis [GDX+97]. Each

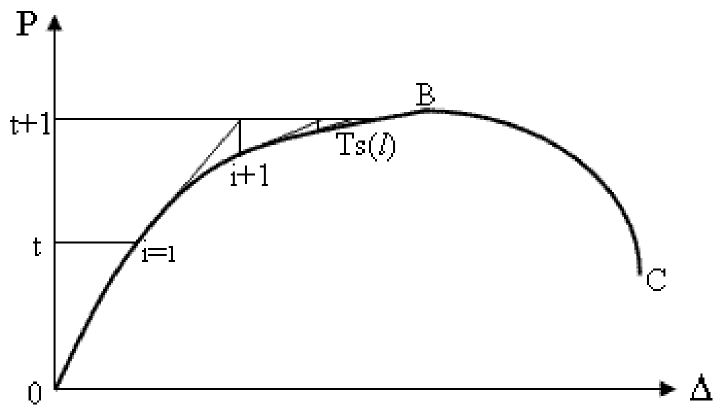
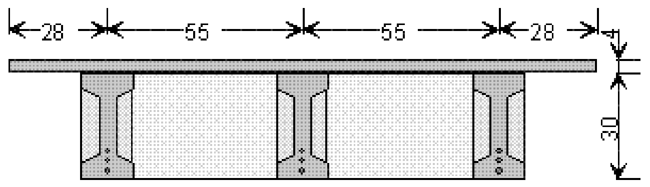


Figure 5.11: Illustration of nonlinear analysis process.



Material	Yield Strength f_y (MPa)	Ultimate Strength f_u (MPa)
Tendon	1600	1838
Deck Concrete		52.0
Beam Concrete		52.2

Figure 5.12: Bridge cross section (units in cm) and table of material properties.

girder is discretized into 20 identical beam elements as shown in Figure 5.13. The moment-curvature relationship for the beam elements is shown in Figure 5.14. The value of the element plastic hinge length *Lgp* associated with this mesh and the main girders' member properties is calculated to be 0.175 m [DGZC01]. Figure 5.15 compares the analysis results to those obtained during laboratory testing.

The program NONBAN has been modified so that it can perform the nonlinear analysis by assembling the global stiffness matrix and solving it at each loading step using the traditional LDL^T method, the pseudoforce method proposed in Deng and Ghosn [DG01], or the solver proposed in that paper in combination with the pseudoforce formulation. All three algorithms reached the same solution. However as observed from Figure 5.16, the computational effort needed to solve the problem at each iteration step is different for each of the three solvers. Figure 5.17 shows the accumulated time as the number of iterations increases. On average, the traditional LDL^T method requires

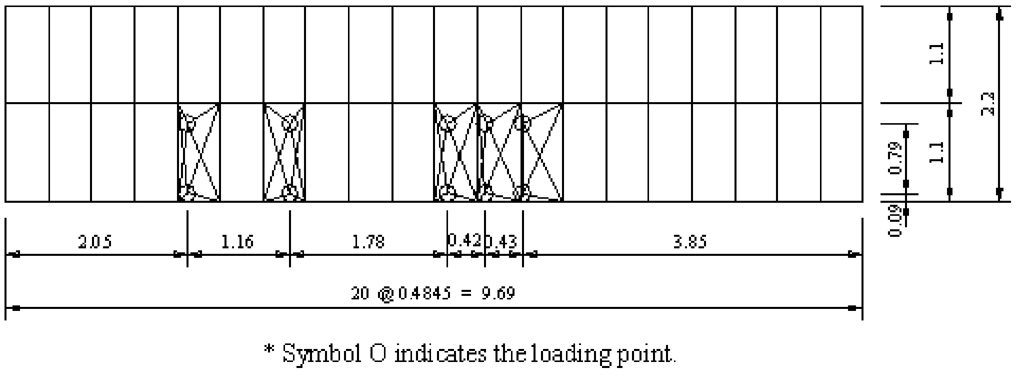


Figure 5.13: Mesh discretization for Slovenia bridge (units in m).

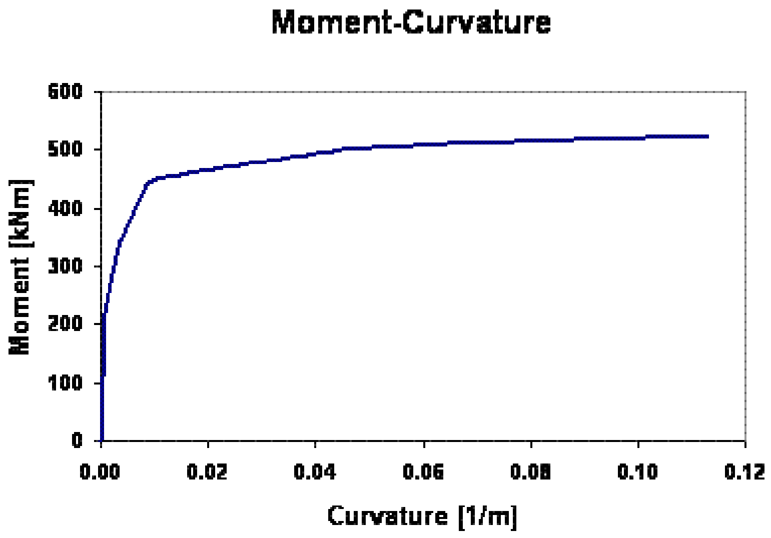


Figure 5.14: Moment-curvature relationship for girders.

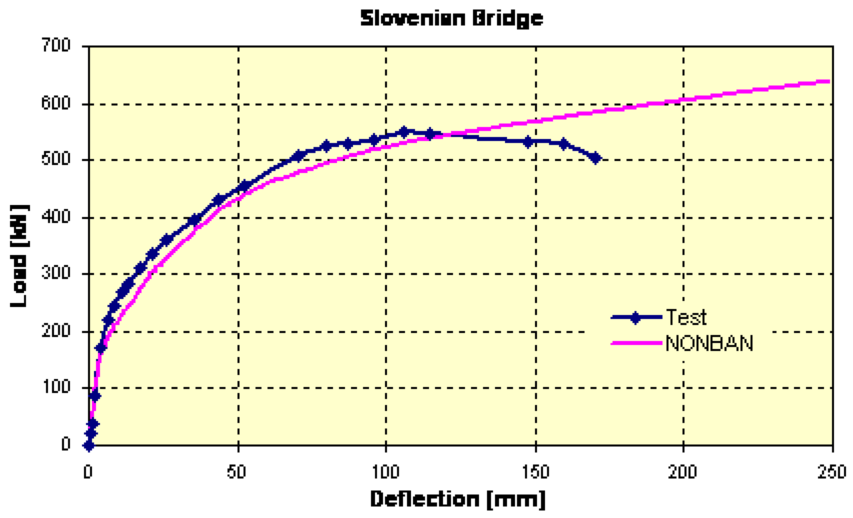


Figure 5.15: Comparison of NONBAN results with experimental results for Slovenia Bridge.

about five times as much time at each iteration step as the other two methods. Twenty milliseconds were needed for the traditional method, while the average time per step for the new method or the original pseudoforce method was 4 milliseconds (with times varying between 1 to about 10 milliseconds). The original pseudoforce solver required about the same time as the proposed EVD-based pseudoforce algorithm, except that the original algorithm had to be restarted at cycles of about fifty iterations each. The new proposed method did not reach the expected efficiency over the original pseudoforce method because in the latter, a special list was used to store the $K^{-1}U$, which saved some computational time at every step [DG01].

Example 2 The same continuous 2-span 4-girder steel bridge previously described by Deng and Ghosn [DG01] is also analyzed in this second example. The girders are spaced at 8.33ft center-to-center and the spans are 100ft each. Two lanes of vehicular loads are applied at the center of each span and incremented to obtain the ultimate capacity of the bridge. The girders' composite moment rotation curves are assumed to be bilinear with initial stiffness of 70750 kip/in and moment capacity of 38,400 kip-in, and the girders are expected to fracture when the plastic hinge rotation reaches a value qp=0.03 radian.

As in Example 1, three methods were used to analyze this bridge. Figure 5.18 shows their results. We see that LDL^T and the proposed EVD-based methods reach identical results in 117 steps, while the pseudoforce method needed 126 iterations to reach almost identical results. This example shows that the application of the Sherman-Morrison-Woodbury-based algorithm (pseudoforce) introduces some minor numerical rounding errors despite its rigorous mathematical formulation.

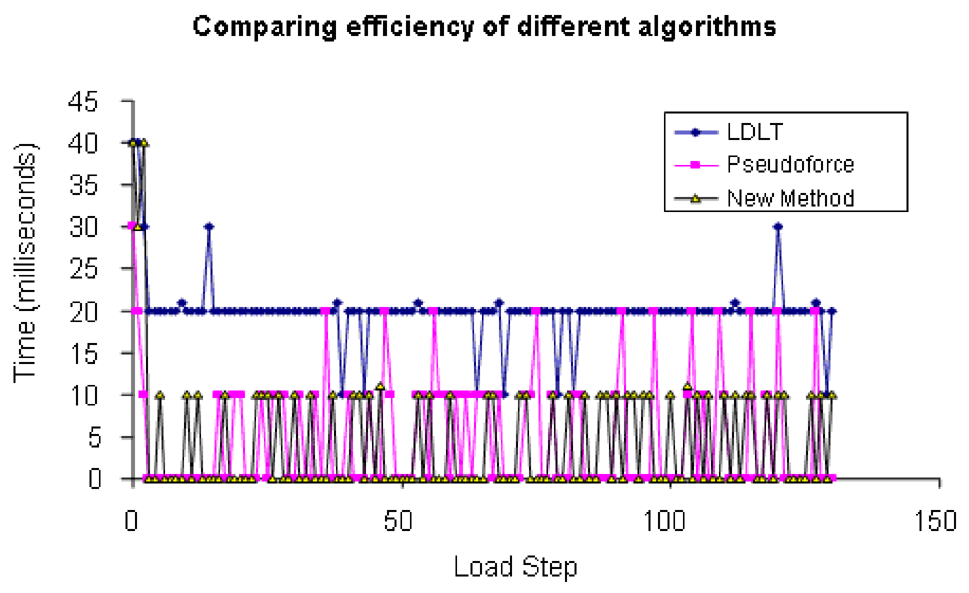


Figure 5.16: Comparison of computational time required at every load step for Example 1.

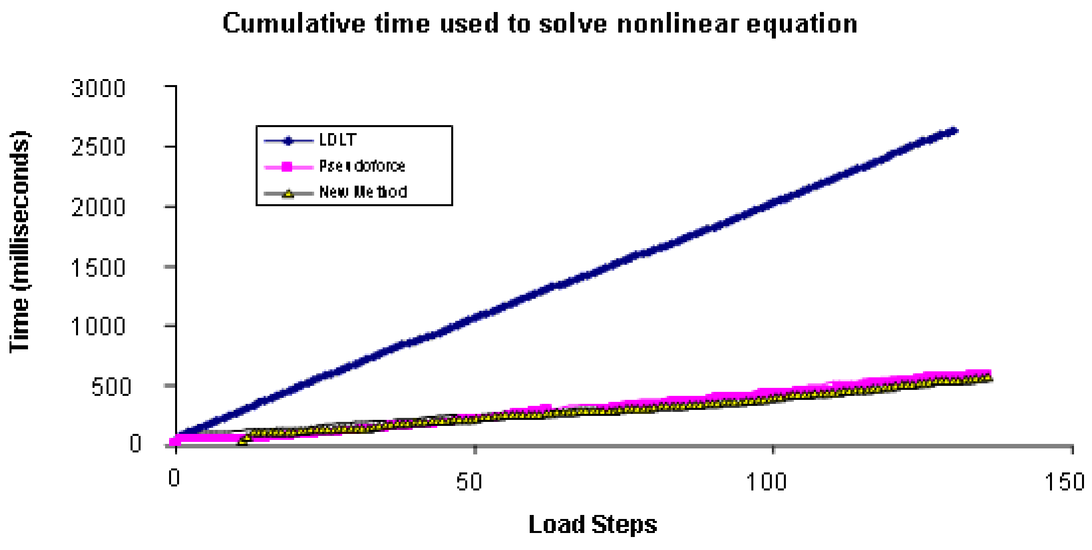


Figure 5.17: Cumulative solver time for LDL^T, pseudoforce, and our proposed method.

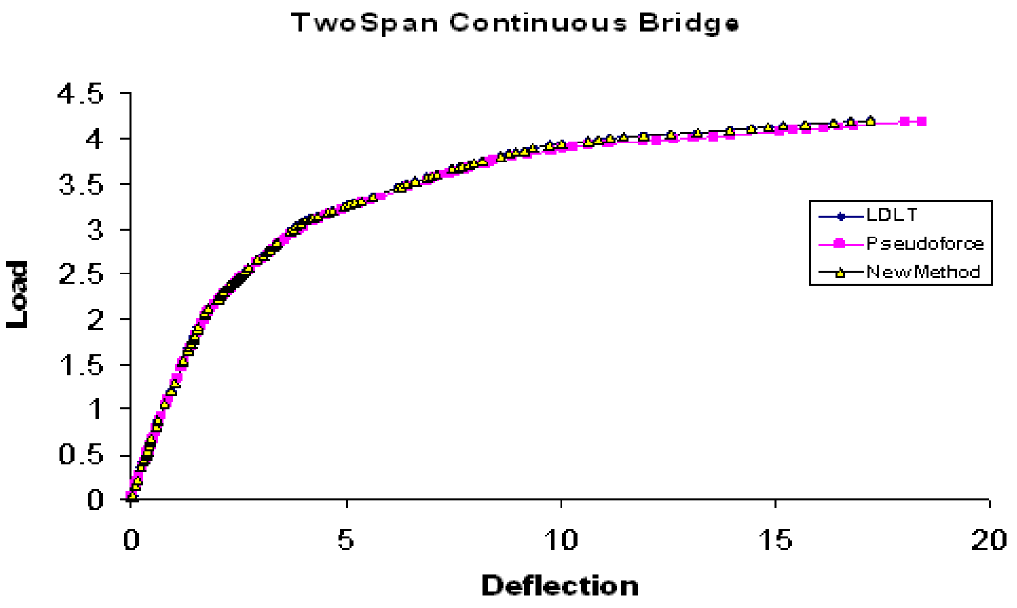


Figure 5.18: Comparison of Example 2 results from different solvers.

Figure 5.19 shows the time needed to solve the stiffness equation at each iteration step. Figure 5.20 shows the cumulative analysis time for this example. In this case, the traditional LDL^T solver was nearly 10 times slower than the proposed method. The latter needed an average of 10 milliseconds per iteration, while the average time per step for the traditional LDL^T was 93 milliseconds, and the average time per step for the original pseudoforce method was 16 milliseconds.

5.10 Conclusions

This chapter introduced a new method to solve iterative finite element-based structural analysis problems. The method has the following features:

1. It uses the pseudoforce formulation previously introduced by the authors [DG01], but includes an LDL^T updating method based on the Eigenvalue Decomposition (EVD) of the original stiffness matrix and the modified matrix that is related to the changes in the stiffness coefficients.

2. It is as stable as the traditional LDL^T method and does not require the restarting that was necessary in the original pseudoforce algorithm as the number of modified stiffness coefficients became large.

3. It is simple to implement, and easy to adapt into existing finite element packages.

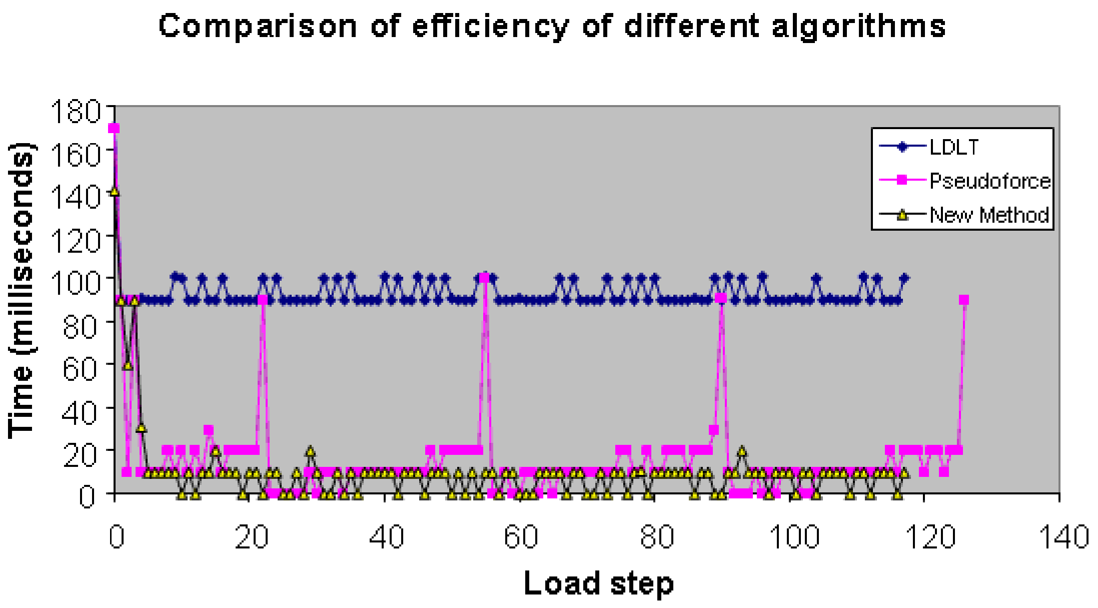


Figure 5.19: Comparison of computational time required at every load step of Example 2.

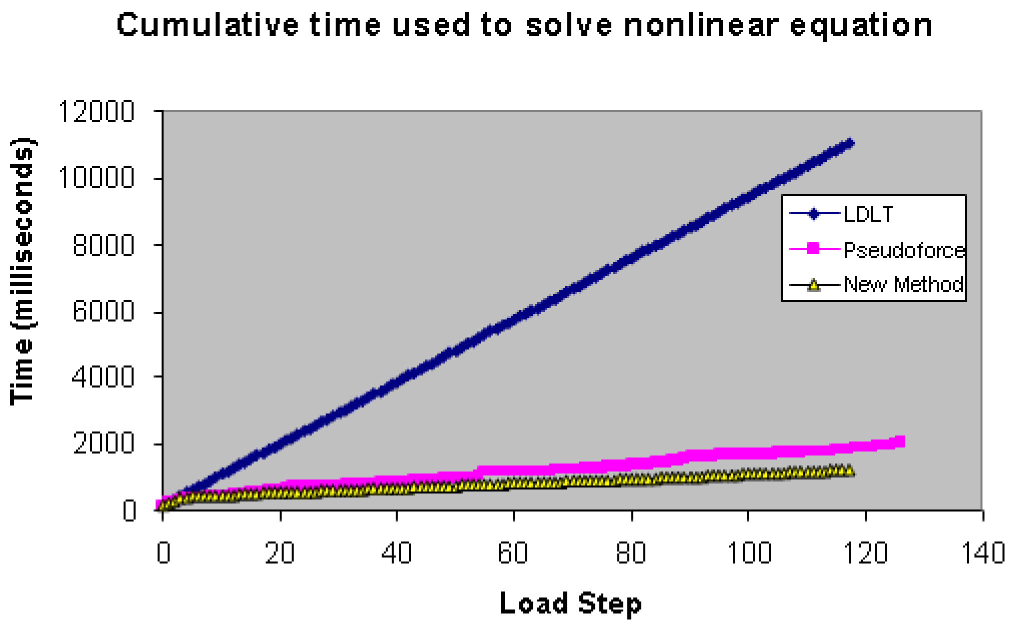


Figure 5.20: Cumulative solver time for Example 2 using LDL^T, pseudoforce, and proposed methods.

Chapter 6

Circuit Simulation

6.1 Introduction

The field of circuit simulation leads IC design in its 50-year history. A device model was established when a device having multiple electrical components and their interconnections manufactured on a single substrate was made in 1958 [Rei01]. A simulation program, SPICE (Simulation Program with Integrated Circuit Emphasis) was developed in 1975 in Berkeley to simulate thousands of transistors manufactured on a single substrate [Nag75]. At present, nearly 10^9 transistors can be integrated on a chip to perform a complex operation while satisfying the economic and power requirements. The field of circuit simulation evolves as an EDA (Electronic Design Automation) industry [Gar09]. It deals with every aspect of IC design from concept design to formal verification, from data path analysis to RC extraction, from net schematic to placement and routing.

Engineering productivity in integrated circuit design and development is limited by the effectiveness of the CAD tools used. For those domains of product design that are highly dependent on transistor-level circuit design and optimization, such as high-speed logic and memory, mixed-signal analog-digital interface, RF functions, power integrated cicuits, and so forth, circuit simulation is perhaps the single most important tool [Kun95]. In the nanometer era, new process technologies need new simulation theory and algorithms to solve the new problems intrinsic to circuit design with nanometer features [DS00]. Modern integrated circuits continually challenge circuit simulation algorithms and implementations in the development of new technology generations. EDA tools are needed with the capacity to simulate a circuit with 10^9 transistors with high accuracy and acceptable performance. The EDA industry is built through inter-disciplinary research from mathematics, circuit theory, graph theory, physics, device modeling, electrical engineering, management, system engineering, and software engineering. What makes circuit simulation outstanding is its multi-disciplinary nature. As described in Chapter 2 and Chapter 4, the core of simulation flow, including all timesteps, is to solve a long sequence of linear systems. A detailed discussion on treating the core is presented in this chapter.

6.1.1 Challenges in nanometer design

Mainstream designs are using 65-nanometer technology. They entered the nanometer realm in the year 2000. Integrated-device manufactures and stand-alone foundries have full production running on 90/65-nanometer parts [IRT].

In design practice, there are new challenges in considering effects such as power leakage, noise, crosstalk, IR drop, and reliability [SH03]. The inherent variability of manufactured nanometer parts also has a great impact on the overall performance of designs. Management and optimization of timing, power, and reliability will become formidable tasks in new nanometer designs as the traditional timing, power, and reliability analysis methods will no longer be viable for any size of module or block of gates [SK99]. Some effects like variability in production, circuit complexity, and significant parasitic effects that introduce huge amounts of interconnect data to be analyzed with current limited computing power, need to be considered in a new light.

In the EDA industry, the sheer increase in circuit size and complexity has broken some EDA tools and hence reduced users' ability to predict design problems accurately. Most EDA tools face the challenge of establishing appropriate models for nanometer features [SK99]. They also face the difficulty of not having the capacity to simulate hundreds of millions of transistors. New tools and methodologies for managing timing, power, and reliability will clearly be in demand.

6.1.2 Transistor-level simulation profile

The semiconductor industry requires EDA software with the ability to analyze nanometer effects such as coupling noise, ground bounce, transmission line wave propagation, dynamic leakage current, supply voltage drop, and nonlinear device and circuit behavior, which are all related to dynamic current. Then the detailed circuit simulation, transistor-level simulation, should become the most effective way to investigate and resolve these problems.

Traditionally, transistor-level simulation can be divided into two families of methods: SPICE and Fast SPICE [SJN94, PRV95]. The SPICE methods consider the circuit as an undivided object. A huge sparse Jacobian matrix will be formed when we numerically discretize the circuit to analyze instant current [New79, NPVS81]. The matrix dimension is of the same order as the number of the nodes in the circuit. For transient analysis, this huge nonlinear system needs to be solved hundreds of thousand times, thus restricting the capacity and performance of SPICE methods. SPICE in general can simulate a chip with up to about 50,000 nodes. Therefore it is not practical to use this method in full chip design. It is widely used in cell design, library building, and accuracy verification.

With appropriate accuracy, Fast SPICE methods developed in the early 1990s provided capacity and speed that were two orders of magnitude greater than SPICE [SJN94, ROTH94, HZDS95]. The performance gain was made by employing simplified models [CD86], circuit partition methods [Ter83, TS03], and event-driven algorithms [New79, ST75], and by taking advantage of circuit latency [Haj80, LKMS93, RH76, RSVH79]. Today, this assumption about circuit latency becomes questionable for nanometer designs because some subcircuits might have been functionally latent and yet electrically active because of voltage variation in Vdd and Gnd busses or in small crosstalk coupling signals. Also, the event-driven algorithm is generally insufficient to handle analog signal propagation. Fast SPICE's capacity is limited to a circuit size considerably below 10 million transistors. It is therefore still inadequate for full chip simulations for large circuits. Furthermore, the simulation time increases drastically with the presence of many BJTs, inductors, diodes, or a substantial number of cross-coupling capacitors.

Because of integration capacity in the nanometer era, SOC is widely used at present in the main design stream [Hut98]. It is possible now to integrate 1 billion transistors on a single chip. The corresponding matrix size would reach $O(10^9)$ if we used SPICE to simulate the full chip. This is not practical.

Fast SPICE families also face difficulties in simulating SOC because of the huge size and the modeling inaccuracy. Driven by the market, EDA companies have developed a series of methods that partly satisfy the performance and accuracy requirements. Those methods include cut algorithms [Haj80, Wu76], hierarchical reduction [Syn03], RC reduction [KY97], and hierarchical storage strategies [Nas02]. In 65-nm design or below, the above-mentioned algorithms face big challenges in accuracy, capacity, and performance.

Because of the above limitations, some designers may choose to cut the entire circuit into small blocks and simulate the sub-circuits in different modes and levels. It is a tedious and error-prone process. Only full-chip simulation can make the designer understand the entire circuit behavior, if such a tool were to exist for very large circuits. Other designers may choose not to analyze nanometer effects but instead add guardbands [JHW$^+$01]. In effect, they overdesign their circuits to avoid problems caused by parasitics in the power bus, signal interconnects, and transistor devices. Guardbanding merely hides those effects in present technology, and is designed purely by experience. With the newest generation of nanometer design, guardbanding will prove to be ineffective and unreliable [KWC00]. We need solutions that provide insight into nanometer effects, and the ability to use this insight to control the performance of the design. This leads the functionality of EDA tools into the next generation.

6.2 A new simulation method

The purpose of simulation is to validate circuit functionality and predict circuit performance. The ability to discover errors early in the design cycle is critically important. Simulation is more than a mere convenience. It allows a designer to explore his circuit in ways that may otherwise be impossible. The effects of manufacturing and environmental parameters can be investigated without actually having to create the required conditions. The ability to detect manufacturing errors can be evaluated beforehand; voltages and currents can be determined without the difficulties associated with attaching a probe. To paraphrase a corporate slogan: without simulation, VLSI itself would be impossible [FM87].

As described in Chapter 2, the core of simulation flow is to solve a long sequence of linear systems

$$A^{(i)}x^{(i)} = b^{(i)}, \quad i = 1, 2, \ldots, n_t. \tag{6.1}$$

```
$A = A^{(1)}$
$A = LU$
Solve $LUx^{(1)} = b^{(1)}$
for $i = 2, 3, \ldots, n_t$
    Calculate $\Delta A$, such as $\Delta A = A^{(i)} - A^{(i-1)}$
    $[L, U] \Leftarrow (A = LU) + \Delta A$
    Solve $LUx^{(i)} = b^{(i)}$
    $A = A + \Delta A$
end
```

Figure 6.1: Algorithm to solve systems (6.1).

In the nanometer era, we need to know exact dynamic currents to manage and optimize timing, power, and reliability. Solving (6.1) approximately, as in Fast SPICE, may make the simulation fail.

Our proposed algorithms are different from the two traditional families of methods. We solve the simulation systems in a global context instead of solving them one by one as we consider the intrinsic properties of the problems involved: physical latency, physical continuity, and physical geometry. We obtain these properties by defining ΔA at the matrix level instead of the related physical level. This process can be easily implemented as illustrated in Figure 6.1. Our algorithms overcome the inconsistency between performance and accuracy, combining the advantages of SPICE and Fast SPICE, and overcoming their disadvantages.

In summary, we address the computational challenge by modifying LU factors of A at each simulation step based on the similarities between two consecutive systems. We define the similarities as an initial matrix A with a low-rank correction ΔA. We then use the algorithms developed in Chapters 3 and 4 to perform multiple-rank updates to the LU factors. Each $x^{(i)}$ is calculated through standard forward and backward substitution. For today's advanced memory chips (8GB per chip available on the market), our algorithms have become increasingly attractive.

6.2.1 Forming ΔA

ΔA preserves the important latency property. It can be calculated in different ways. The direct way is by $\Delta A = A^{(i+1)} - A^{(i)}$. The other way is to find active nodes a_v and calculate

$$\Delta A = f(a_v), \tag{6.2}$$

where a_v is a vector of active nodes, and $f(a_v)$ is a matrix function that calculates ΔA from a_v. How to calculate a_v and how to define $f(a_v)$ are critical problems related to accuracy and performance. We introduce the basic algorithms here. The more robust ones needs further research.

The conditions defining a node with index i as active node are

- $\Delta v(i) = |v(i)^t - v(i)^{t-1}| > dV_{\max}$
- $|I(i)| > I_{\max}$

- $|\frac{\Delta v(i)}{\Delta t}| > dV\, dt_{\max}$
- Satisfying the above conditions in a fixed time interval $\Delta t_{\min}$.

The thesholds $dV_{\max}$, $I_{\max}$, $dV\, dt_{\max}$ and $\Delta t_{\min}$ are defined by experience and/or according to customer specification. We may choose one condition or several combined conditions to define active nodes by considering accuracy requirements and circuit (gate) properties. All the active nodes form a_v.

The function $f(.)$ can be defined in two ways:

- Direct mapping. ΔA can be extracted from $A^{(i+1)} - A^{(i)}$ as $D(A^{(i+1)} - A^{(i)})D$, where $D = \text{diag}(d)$ and d is a pointer vector with

$$d(j) = \begin{cases} 1 & \text{if } j \text{ is an active node,} \\ 0 & \text{if } j \text{ is an idle node,} \end{cases}$$

  or according to

$$\Delta A(:,j) = \begin{cases} A(:,j)^{(i+1)} - A(:,j)^{(i)} & \text{if } j \text{ is an active node,} \\ 0 & \text{if } j \text{ is an idle node.} \end{cases}$$

- Connectivity information. If a device has one active node, we promote all nodes connected to this device to be active nodes. We then form ΔA by direct mapping.

6.2.2 Decomposing ΔA

In multiple-rank updates, we need to decompose ΔA of rank r as $\Delta A = \sum_{j=1}^{r} u_j v_j^T$, where u_j and v_j are sparse vectors. The submatrix containing the nonzeros of ΔA can be decomposed by the following methods (see Chapter 3 for details):

- elementary matrix computation
- QR
- SVD
- LUSOL

6.2.3 Multiple-rank updates

Sparse matrices can be stored in triple form, compressed-column form, or compressed-row form. In triple form, we store every nonzero entry as $(i, j, A(i, j))$, where i is the row number and j the column number. In compressed-column form, a square sparse matrix is described with four parameters [Dav06]:

- n: an integer scalar. The matrix $A \in \mathbb{R}^{n\times n}$.

- Ap: an integer array of size $n+1$. The first entry is $Ap[0]=0$, and the last entry $nz=Ap[n]$ is equal to the number of entries in the matrix. Ap stores the index offset of Ai and Ax.
- Ai: an integer array of size $nz=Ap[n]$. The row indices of entries in column j of A are located in $Ai[Ap[j] : Ap[j+1]-1]$.
- Ax: a double array of size nz . The numerical values in column j are located in $Ax[Ap[j] : Ap[j+1]-1]$.

Compressed-row form uses four similar parameters:

- n: an integer scalar. The matrix $A \in \mathbb{R}^{n\times n}$.
- Ap: an integer array of size $n+1$. The first entry is $Ap[0]=0$, and the last entry $nz=Ap[n]$ is equal to the number of entries in the matrix.
- Ai: an integer array of size $nz=Ap[n]$. The column indices of entries in row i of A are located in $Ai[Ap[i] : Ap[i+1]-1]$.
- Ax: a double array of size nz. The numerical values in row i are located in $Ax[Ap[i] : Ap[i+1]-1]$.

We choose the compressed-column form to represent each $A^{(i)}$ and to express our sparse multiple-rank update algorithms.

Before describing the multiple-rank updates, let us review left-looking LU factorization. Symbolic analysis gives permutation matrices P and Q, and we compute the sparse factorization $PAQ^T = LU$, where L and U are stored in compressed column form (Lp, Li, Lx) and (Up, Ui, Ux). The core of a left-looking LU is implemented as follows [Dava].

```
/* clean work space X */
for (k=0; k<n; k++) {
    X[k] = 0;
}
for (k=0; k<n; k++) {
    oldcol = Q[k];
    pend = Ap[oldcol+1]
    for (p = Ap[oldcol]; p < pend; p++) {
        newrow = Pinv[Ai[p]];
        X[newrow] = Ax[p];
    }
    ulen = Up[k+1] - Up[k];
    Uik = &Ui[Up[k]];
    Uxk = &Ux[Up[k]];
    for (up = 0; up < ulen; up++) {
       j = Uik[up];
       ujk = X[j];
       X[j] = 0;
```

```
         Uxk[up] = ujk;
         Lik = &Li[Lp[j]];
         Lxk = &Lx[Lp[j]];
         llen = Lp[j+1] - Lp[j];
         for (p = 0; p < llen; p++) {
             X[Lik[p]] -= Lxk[p]*ujk;
         }
      }
      /* form L */
      ukk = X[k];
      Diag[k] = ukk;
      llen = Lp[k+1] - Lp[k];
      Lik = &Li[Lp[k]];
      Lxm = &Lx[Lp[k]];
      for (p = 0; p < llen; p++) {
         i = Lik[p];
         Lxk[p] = X[i]/ukk;
         X[i] = 0;
      }
}
```

For simplicity, we store L and U in separate compressed-column forms. In KLU [Dava], the elements of Li and Lx (and those of Ui and Ux) are mixed together for better cache memory usage.

Left-looking algorithms calculate L and U column by column by solving a series of lower-triangular systems. For example, when computing column j of L and U, we know $L(:,1:j-1)$ and solve

$$\begin{bmatrix} 1 & & & & \\ 0 & \ddots & & & \\ l_{31} & \dots & 1 & & \\ \vdots & & 0 & \ddots & \\ 0 & l_{n2} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_j \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} A(1,j) \\ \vdots \\ A(j,j) \\ \vdots \\ A(n,j) \end{bmatrix}.$$

to get $L(:,j)$ and $U(:,j)$ through a working array w, assuming $P = Q = I$. For more general P and Q, permutation is needed to transfer $A(:,j)$ into local indexes, as in the above code. The relationship between w and $L(:,j)$ and $U(:,j)$ is

$$\begin{bmatrix} U(1,j) \\ \vdots \\ U(j-1,j) \\ U(j,j)\ (=\mathrm{diag}(j)) \\ L(j+1,j)\,\mathrm{diag}(j) \\ \vdots \\ L(n,j)\,\mathrm{diag}(j) \end{bmatrix} = \begin{bmatrix} w(1) \\ \vdots \\ w(j-1) \\ w(j) \\ w(j+1) \\ \vdots \\ w(n) \end{bmatrix}.$$

The implementation of the working array w must be considered carefully. If we clear w at every column, the whole clearing effort is $O(n^2)$, which is much higher than the computation effort $O(n^{1.05-2.0})$ for the sparse matrix factorization. An optimized implementation of w is to clear it once, and dynamically clear the nonzero entries in w after they have been used. Thus, the whole clearing effort for w is $O(Lp(n) + Up(n))$. This implementation leads to success for the left-looking algorithm. It is a general technique in sparse matrix analysis. We will use it again and again in multiple-rank updates.

To implement multiple-rank updates, we need to store L in compressed-column form and U in compressed-row form. We use two linked lists Xwi and Xsi to manage working arrays w and s to define which column is the one to update next. The left-looking algorithm needs to go through all the columns, while rank-1 update only goes through the columns defined by the elimination tree. Please refer to Chapter 4 for details.

When we finish updating column j, we have $k1$ and $k2$ as

$$k1 = \min\ \{i \mid w(i) \neq 0,\ i > j\},$$
$$k2 = \min\ \{i \mid s(i) \neq 0,\ i > j\}.$$

The next updated column in L and row in U is `pipe_tick` $\equiv \min(k_1, k_2)$. A sparse rank-1 update αxy^T with $q1$ as the first nonzero entry in x and $q2$ as the first nonzero entry in y can be implemented as follows.

```
Xw = x; /* scatter x into Xw according xi */
setup_list(Xwi, xi)
Xs = y;
setup_list(Xsi, yi)
diag_ws = 0;
pipe_tick = min(q1,q2)
while(1) {
  if (pipe_tick >= n) break;
  if (pipe_tick < 0) break;
```

```
orig_si = Xs[pipe_tick];
Xs[pipe_tick] /= Udiag[pipe_tick];

p1 = Xw[pipe_tick];
p2 = Xs[pipe_tick];

d_ws = alpha - diag_ws;
gama = d_ws*p1;
beta = d_ws*p2/(1.0 + p1*p2*d_ws);
diag_ws += beta*gama;
Xw[pipe_tick] = 0.0;
Xs[pipe_tick] = 0.0;

Udiag[pipe_tick] += p1*orig_si*d_ws;

ulen = Up[pipe_tick+1] - Up[pipe_tick-1];
pUi = &Ui[Up[pipe_tick-1]];
pUx = &Ux[Up[pipe_tick-1]];

merge_list(Xsi, pUi, ulen); /* merge pUi into Xsi. */

for (p = 0; p < ulen; p++) {
    r = *pUi++;
    if (r >= s2) break;
    a11 = Xs[r];
    Xs[r] = a11 - *pUx*p2;
    (*pUx++) += gama*a11;
}

llen = Lp[pipe_tick+1] - Lp[pipe_tick-1];
pLi = &Li[Lp[pipe_tick-1]];
pLx = &Lx[Lp[pipe_tick-1]];

merge_list(Xwi, pLi, llen); /* merge pLi into Xwi. */

for (p = 0; p < llen; p++) {
    r = *pLi++;
    Xw[r] -= *pLx*p1;
    (*pLx++) += beta*Xw[r];
}

tk1 = Xsi->next;
tk2 = xwi->next;
pipe_tick = min(tk1,tk2);
```

```
}
```

Setup and dynamic updating of lists Xsi and Xwi is tedious work. Its computing effort and memory access pattern are similar to updating Ux and Lx. If we take the lists out, it is interesting that the speedup is around 30% rather than double because the computation is memory intensive, and the lists and updating Ux and Lx share memory with Li and Ui.

It is possible to avoid implementing the lists by considering the pattern of L_j and U_j [DH01]:

$$\mathcal{L}_j = \{j\} \cup \left(\bigcup_{c:j=\pi(c)} \mathcal{L}_c \backslash \{c\} \right) \cup \left(\bigcup_{\min \mathcal{A}_k = j} \mathcal{A}_j \right).$$

We can just trace the parents of L_j and U_j and the $\mathcal{A}_j$ to decide which column to update next. This leads to an algorithm in which we use function `f1` to implement the above formula.

```
Xw = x; /* scatter x into Xw according xi */
setup_list(Xwi, xi)
Xs = y;
setup_list(Xsi, yi)
diag_ws = 0;
pipe_tick = min(q1,q2)
while(1) {
  if (pipe_tick >= n) break;
  if (pipe_tick < 0) break;

  orig_si = Xs[pipe_tick];
  Xs[pipe_tick] /= Udiag[pipe_tick];

  p1 = Xw[pipe_tick];
  p2 = Xs[pipe_tick];

  d_ws = ALFA - diag_ws;
  gama = d_ws*p1;
  beta = d_ws*p2/(1.0 + p1*p2*d_ws);
  diag_ws += beta*gama;
  Xw[pipe_tick] = 0.0;
  Xs[pipe_tick] = 0.0;

  Udiag[pipe_tick] += p1*orig_si*d_ws;

  ulen = Up[pipe_tick+1] - Up[pipe_tick-1];
  pUi = &Ui[Up[pipe_tick-1]];
  pUx = &Ux[Up[pipe_tick-1]];

  for (p = 0; p < ulen; p++) {
        r = *pUi++;
        if (r >= s2) break;
        a11 = Xs[r];
        Xs[r] = a11 - *pUx*p2;
        (*pUx++) += gama*a11;
    }

    llen = Lp[pipe_tick+1] - Lp[pipe_tick-1];
    pLi = &Li[Lp[pipe_tick-1]];
    pLx = &Lx[Lp[pipe_tick-1]];

    for (p = 0; p < llen; p++) {
        r = *pLi++;
        Xw[r] -= *pLx*p1;
        (*pLx++) += beta*Xw[r];
    }

    tk1 = f1(pipe_tick, Li, xi)
    tk2 = f1(pipe_tick, Ui, yi);
    pipe_tick = min(tk1,tk2);
  }
```

Implementing `f1` is cheap if we know a node's children from the elimination tree. Otherwise we need to trace $\mathcal{A}_j$ and $\mathcal{L}_c$ by using a stack and pruning $\mathcal{L}_c$ to save computation. To have an optimized implementation, let us study

$$\bar{L}\bar{U} = LU + ue_j^T \tag{6.3}$$

in another way.

If we partition L and U as block matrices L_{11}, $U_{11} \in \mathbb{R}^{(j-1)\times(j-1)}$, and other submatrices with appropriate dimension, (6.3) becomes

$$\begin{aligned}\bar{L}\bar{U} &= \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} e_j^T \\ &= \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} + L_{11}^{-1}u_1 e_j^T \\ & U_{22} \end{bmatrix} + \begin{bmatrix} 0 \\ u_2 - L_{21}L_{11}^{-1}u_1 \end{bmatrix} e_j^T .\end{aligned}$$

To compute $L_{11}^{-1}u_1$ and $u_2 - L_{21}L_{11}^{-1}u_1$, we need a partial forward computation, which is the standard operation in a left-looking method:

$$\begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \tag{6.4}$$

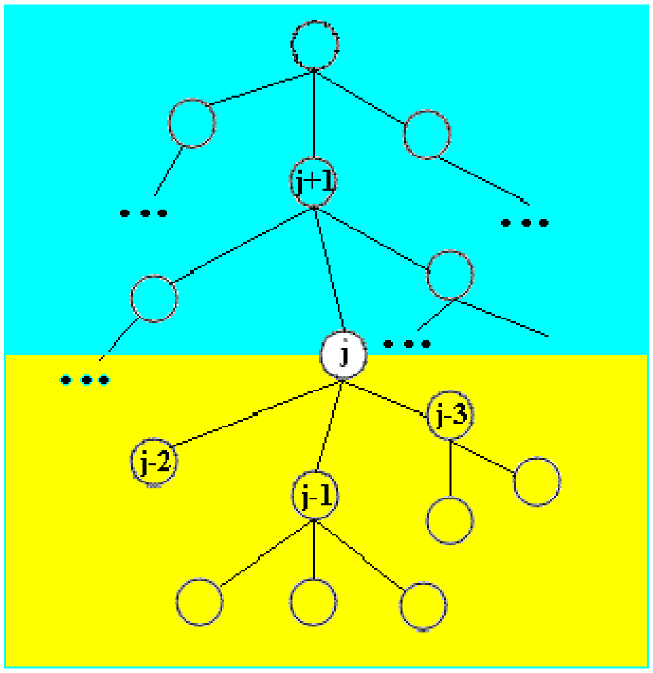


Figure 6.2: Operation in elimination tree.

giving $x_1 = L_{11}^{-1}u_1$ and $x_2 = u_2 - L_{21}x_1$. Thus the standard sparse rank-1 update procedure can be implemented as two steps:

1. Calculate x_1 and x_2 from (6.4).
2. Compute $\begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} + x_1 e_j^T \\ & U_{22} \end{bmatrix} + \begin{bmatrix} 0 \\ x_2 \end{bmatrix} e_j^T$. (6.5)

These two steps represent different elimination steps in the elimination tree shown as in Figure 6.2, where the yellow region represents the first step (called j's children-elimination step), and the blue region represents the second step (called j's parent-elimination step).

Node j may have many children, but only one parent. In other words, j's children-subtree defined in (4.9) is not unique, while its parent-tree defined in (4.8) is unique. During the elimination process, all of the children-subtree must be processed before node j. Although this process is complicated, partial forward computation limits the complexity. We don't need a special (e.g., linked-list) structure to manage the process.

Although step 2 is unique, it is sufficient to follow the parent to do a rank-1 update for a symmetric matrix. But in the unsymmetric case, two cases need special treatment.

The first case as shown in Figure 6.3 is $A(i,k) = 0$ for $k < j$, while $A(i,k)$ may not equal 0 for $k > j$. In this case, $L(:,j)$ is equal to $A(:,j)$. The second case as shown in Figure 6.4 is $A(i,k) = 0$

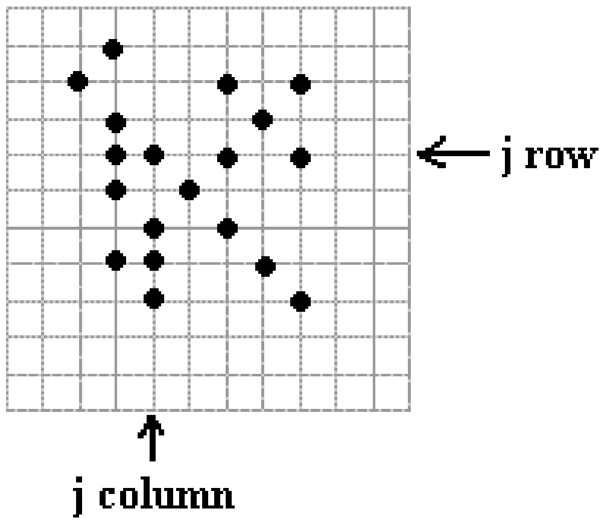


Figure 6.3: Case 1 description.

for $k < j$ and $U(j,k) = 0$ for $k > j$. In this case, the rank-1 update is finished.

Based on our discussion, the optimized sparse multiple-rank update procedure can be designed as follows.

```
Xw = x; /* scatter x into Xw according xi */
setup_list(Xwi, xi)
Xs = y;
setup_list(Xsi, yi)
diag_ws = 0;
pipe_tick = min(q1,q2)
while(1) {
  if (pipe_tick >= n) break;
  if (pipe_tick < 0) break;

  orig_si = Xs[pipe_tick];
  Xs[pipe_tick] /= Udiag[pipe_tick];

  p1 = Xw[pipe_tick];
  p2 = Xs[pipe_tick];

  d_ws = ALFA - diag_ws;
  gama = d_ws*p1;
  beta = d_ws*p2/(1.0 + p1*p2*d_ws);
  diag_ws += beta*gama;
```

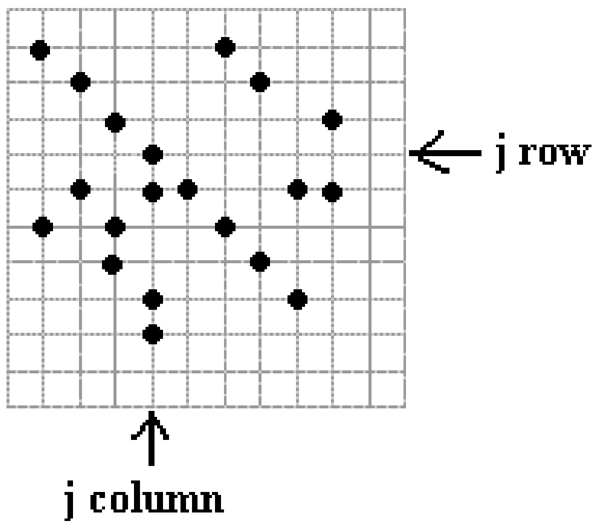


Figure 6.4: Case 2 description.

```
Xw[pipe_tick] = 0.0;
Xs[pipe_tick] = 0.0;

Udiag[pipe_tick] += p1*orig_si*d_ws;

ulen = Up[pipe_tick+1] - Up[pipe_tick-1];
pUi = &Ui[Up[pipe_tick-1]];
pUx = &Ux[Up[pipe_tick-1]];

for (p = 0; p < ulen; p++) {
    r = *pUi++;
    a11 = Xs[r];
    Xs[r] = a11 - *pUx*p2;
    (*pUx++) += gama*a11;
}

llen = Lp[pipe_tick+1] - Lp[pipe_tick-1];
pLi = &Li[Lp[pipe_tick-1]];
pLx = &Lx[Lp[pipe_tick-1]];

for (p = 0; p < llen; p++) {
    r = *pLi++;
    Xw[r] -= *pLx*p1;
    (*pLx++) += beta*Xw[r];
    }

    tk1 = f1(pipe_tick, Li, xi)
    tk2 = f1(pipe_tick, Ui, yi);
    pipe_tick = min(tk1,tk2);
  }
```

According to our experience, the linked list used to calculate the next candidate column to update doubles the computational effort (because it is memory intensive) but reduces overall performance only 30%. It is the most reliable implementation method. Others methods have corner cases or are hard to implement.

Truncation can be used to shorten the update procedure while preserving the engineering accuracy. A voltage will drop when current traverses a long path. This phenomenon causes the values in `Xw` and `Xs` to become smaller and smaller as they proceed column by column. To implement truncation, we can set different thresholds for `p1`, `p2`, `beta`, and `gama`. When their values are smaller than their thresholds, we can terminate the rank-1 update procedure.

If the initial values in x and y are small, truncation may stop the update procedure early, giving higher performance for the rank-1 update. This can be achieved by considering

$$\begin{aligned} LU(:,j) &= A(:,j) \\ L\ (1+\alpha)U(:,j) &= (1+\alpha)A(:,j) \\ L(i,:)U &= A(i,:) \\ (1+\alpha)L(i,:)\ U &= (1+\alpha)A(i,:) \end{aligned}$$

for some scalar α. Thus we can approximate $\Delta A(:,j) \approx \alpha A(:,j)$ or $\Delta A(i,i) \approx \alpha A(i,i)$ to speed up the update procedures. We can calculate α from two possible minimization problems:

$$\min_{\alpha} \|\Delta A(:,j) - \alpha A(:,j)\|$$

or

$$\Delta A(i,i) - \alpha A(i,i) = 0.$$

It is interesting to note that this strategy works perfectly for the nodes connected to pure capacitors. In other words, we can get the LU factors at no cost for some nodes in a circuit. The candidate circuits applying this technology are called charge pump circuits. Another application is for tiny timestep analysis. In SPICE simulation, the C in the Jacobian matrix $G + \frac{C}{\Delta t}$ will make G insignificant as $\Delta t \to 0$. Therefore, we can get the factors at no cost as $\Delta t \to 0$.

In solving $(A + \Delta A)x = b$, we may need iterative refinement to obtain more accurate results. When truncation is used, it is necessary to check the residual $r = b - (A + \Delta A)x$ to make sure we reach the desired accuracy. After truncation and/or approximately using rank-1s to represent ΔA, we have $LU \approx A + \Delta A$. Letting $B = LU$, $Bx_1 = b$, $A_1 = A + \Delta A$, we have

$$r = b - (A + \Delta A)x_1.$$

To improve accuracy, we could make a rank-1 correction to B so that $\bar{B} = LU + uv^T$, where (ideally) we choose u and v to minimize

$$\min_{u,v} \|(uv^T - (A_1 - B))\|_F.$$

To simplify this problem, we choosing $u = r$ and obtain v by solving the optimization problem one column at a time. This proves to be equivalent to one step of iterative refinement, but with a steplength (linesearch) on the correction to x_1:

$$x_2 = x_1 + \theta \Delta x_1,$$

where $LU\Delta x_1 = r$. The new residual is

$$r_2 = b - A_1 x_2 = b - A_1(x_1 + \theta \Delta x_1) = r - \theta c,$$

where $c = A_1 \Delta x_1$, and we choose θ to minimize $\|r_2\|$:

$$\theta = c^T r / c^T c.$$

Our experiments show that r_2 is usually an order of magnitude smaller than r.

Example 1. Verify the multiple-rank update algorithm A sparse matrix of size 63284 is extracted from a circuit simulation. It has 410801 nonzeros as shown in Figure 6.5. After pivoting, we have the symbolic pattern as shown in Figure 6.6. ΔA with $r = 88$ columns has 440 nonzero entries as shown in Figure 6.7. b is a current vector assembled by device stamping. We perform multiple-rank updates as described in the last section, and solve $Ax = b$. We then have residual norm $\|r\|_1 = 0.0019$. After the first refinement, $\|r\|_1 = 2.71\text{e-}6$. This refinement process works well. The time for 88 rank-1 updates is 1e-7 to 0.004 seconds. The time for left-looking LU is 0.03602. We have 9 to 1900 times performance enhancement.

6.2.4 Building base-camps

For simplicity, we separate the base-camp identification procedure (Chapter 4) and the multiple-rank update procedure, although these two procedures can be mixed to improve the data flow so that memory can be used more efficiently. The base-camp in the worst case where ΔA has full rank is not the same as when ΔA has low rank, although there may be some connection. The best way is to find a base-camp for each special ΔA.

Let β denote an integer such as 1, 2, or 3. The candidate columns for building a base-camp are chosen according to one or more of these conditions:

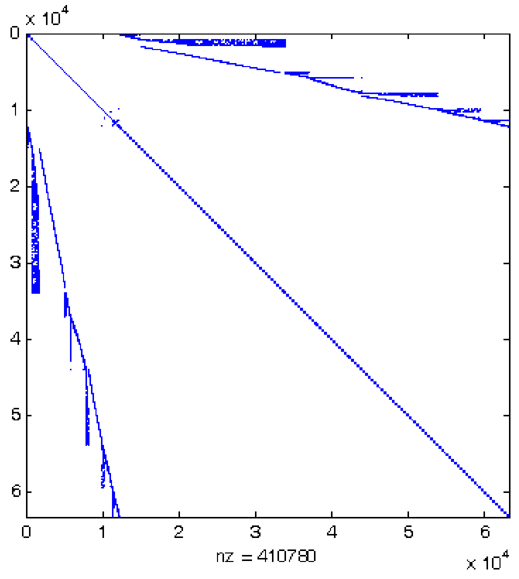


Figure 6.5: Matrix symbolic pattern before pivoting.

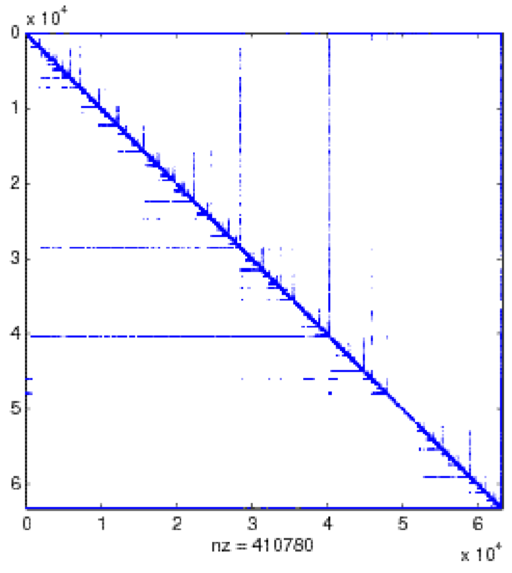


Figure 6.6: Matrix symbolic pattern after pivoting.

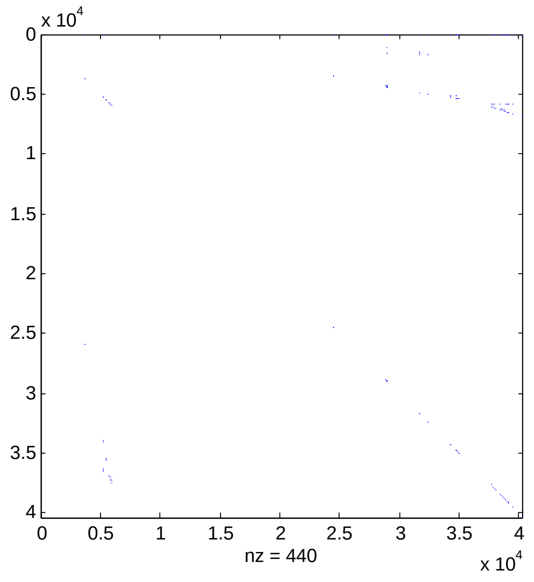


Figure 6.7: Symbolic pattern of ΔA.

- $llen < \beta$
- $ulen < \beta$
- $s(k) > \alpha\ \min(llen, ulen)$
- other big-node, bus, or control-node strategies based on special circuits,

where $s(k)$ refers to the number of rank-1 updates that reached column k, $llen$ is the length of $L(:,k)$, and $ulen$ is the length of $U(k,:)$. In circuit simulation, we recommend $\beta \leq 3$ and $\alpha = 2$.

How to calculate $s(k)$ efficiently and/or optimally is a big challenge. But we can approximate the $s(k)$ when all rank-1s lie in a lower-triangular matrix. In this study we implemented the following steps for sparse multiple rank-1 updates:

1. Trim ΔA into a lower-triangular matrix using the two-step algorithm described in (6.5).
2. Identify the master columns where base-camps will be built.
3. Execute

```
for (j = 0; j < n; j++) {
    if ( j is a_master_column ) {
        build_base_camp()
        update_dA_base_camp()
    }
    if (dA(:, j) == 0) continue; \\ no update for column j
    multiple_rank1_update()
}
```

There are two ways to write function *build_base_camp*(). One is to implement (4.16). The other is to implement (4.19) implicitly. To simplify analysis, rewrite (4.19) as

$$\Delta\Lambda_k = L_{\hat{S}}U_{\hat{S}} - L_S U_S.$$

As we start to build a base-camp at a master column, L_S and U_S are the values before the update. We can compute $L_S U_S$ directly because the base-camp shelters it from being changed. $L_{\hat{S}}$ and $U_{\hat{S}}$ represent the updated values. We do not have these values because the base-camp prevented computing them. But from (4.3), we have

$$L_{\hat{S}}U_{\hat{S}} = D + \Delta D - L_{\hat{C}}U_{\hat{B}},$$

where $L_{\hat{C}}$ and $U_{\hat{B}}$ are the available values updated by the rank-1s entering the base-camp. Thus we have

$$\begin{aligned}\Delta\Lambda_k &= D + \Delta D - \left(L_{\hat{C}}U_{\hat{B}} + L_S U_S\right)\\ &= D + \Delta D - L(D,:)U(:,D),\end{aligned}$$

where $L(D,:)$ and $U(:,D)$ represent the rows of L and columns of U corresponding to the base-camp matrix. This procedure has the same data-flow as the left-looking procedure, and is easy to implement. The examples in the next section are computed by this implementation.

6.3 Examples

Normal case A test case is to do a transient simulation in [0, 1e-07] seconds for a circuit with 5591 nodes and 11559 Bsim4 transistors that create a Jacobian matrix of size 5641 shown in Figure 6.8 in transient analysis. SPICE takes 1850 timesteps to finish the simulation in 4215.12 seconds, of which 370.28 seconds are needed for LU factorization and 19.92 seconds for the solve procedures. Most simulation time is used for finding the DC solution. When SPICE uses its multiple-rank update algorithm to factor the Jacobian matrix, it takes 3988.1 seconds to run 1855 timesteps to finish the same simulation.

The results from a direct solver (KLU) and multiple-rank updates are identical. The average factor time for the direct solver is 0.0431 seconds. The average factor time for multiple-rank updates is 0.00617 seconds. The factorization speedup is 7 times. Because we use sparsification technology to get ΔA with low rank, we use refinement to improve the accuracy of multiple-rank updates. The average number of refinements per step is 1.4.

Memory case A DRAM circuit has 180,483 nodes and 16896 Bsim4 transistors. We need to make a transient simulation in [0, 5e-08] seconds. The Jacobian matrix of size 180486 is shown in Figure 6.9 in transient analysis. SPICE takes 362 timesteps to finish the simulation in 14459.4 seconds, of which 1102 seconds are needed for LU factorization and 61.3 seconds for the solve

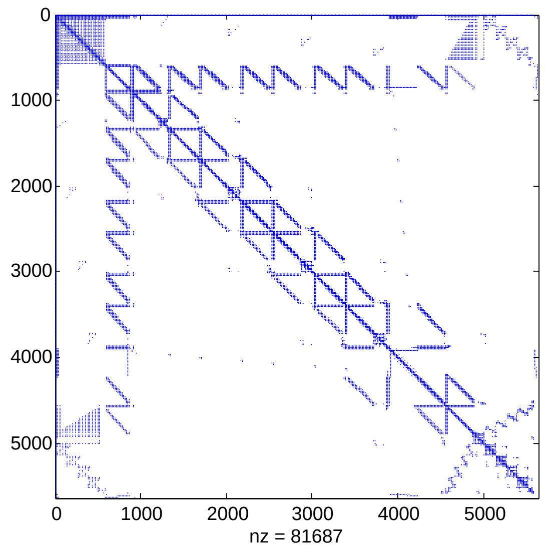


Figure 6.8: Symbolic pattern of original Jacobian matrix.

procedures. When using its multiple-rank update algorithm to factor the Jacobian matrix, SPICE take 12889.6 seconds to run 362 timesteps to finish the same simulation.

The results from the direct solver (KLU) and our multiple-rank update solver are identical. The average time for multiple-rank updates per timestep is 0.0269 seconds. The average time for KLU is 1.113 seconds. The factorization speedup is 41.38 times. The average number of refinements per step is 1.243.

Powernet case A powernet circuit has 160812 nodes with 320801 resistors and 160827 capacitors. We need to make a transient simulation in [0, 4e-07] seconds. The Jacobian matrix of size 160817 is shown in Figure 6.10 in transient analysis. SPICE takes 103 timesteps to finish the simulation in 22016.2 seconds, of which 2596.0 seconds are needed for LU factorization and 38.39 seconds for the solve procedures. When using its multiple-rank update algorithm to factor the Jacobian matrix, SPICE take 15410.1 seconds to run 103 timesteps to finish the same simulation.

The results from the direct solver (KLU) and our multiple-rank update solver are identical. The average number of rank-1 updates per timestep is 0.0468 seconds. The average time for KLU is 9.512 seconds. The factorization speedup per timestep is 203.25. The average number of refinements per step is 4.89.

Simulation Notes Circuit simulation is complex system work. Its accuracy depends on modeling and numerical discretization methods. Typically, the integration method has 0.001 relative error in every timestep. The inner Newton-Ralphson iteration also allows 0.001 relative error in its convergence conditions. If the error in the linear solver is within the integration method's

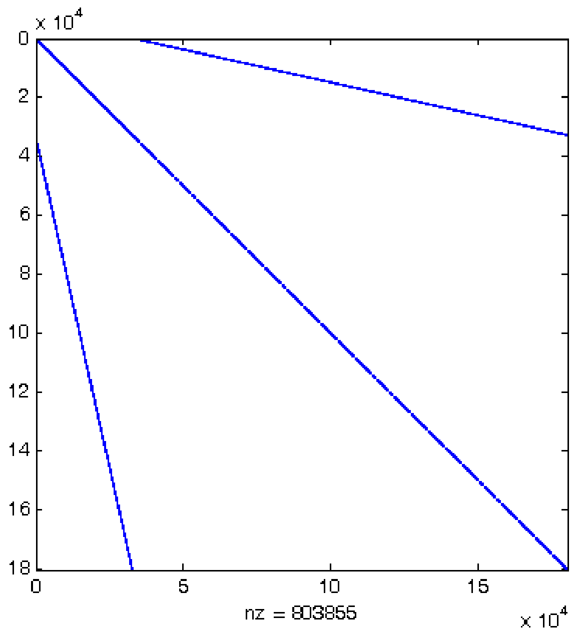


Figure 6.9: Symbolic pattern of original Jacobian matrix.

stability range, the linear solver's error will be damped out or not amplified during the time-domain calculation. In other words, it is not necessary for the linear solver to have high accuracy when we keep the number of simulation iterations unchanged. This is the justification for sparsifying ΔA.

In this study, we form ΔA from linear algebra computation so that it has high dimension during some timesteps, which causes the simulation to be slow. The simulation will have much better performance and retain high accuracy if we identify ΔA from the nature of the circuit functionality. In our algorithm, every part can be parallelized.

In our example, we choose relative tolerance 10^{-10} for terminating iterative refinement. The average number of refinements is less than 5, which indicates that our updated LU factors have high accuracy. In reality, the relative tolerance can be as low as 10^{-6}.

In some cases, SPICE needs a higher number of timesteps with multiple-rank updates than with the direct solver. We believe this matter can be resolved when we consider ΔA more carefully.

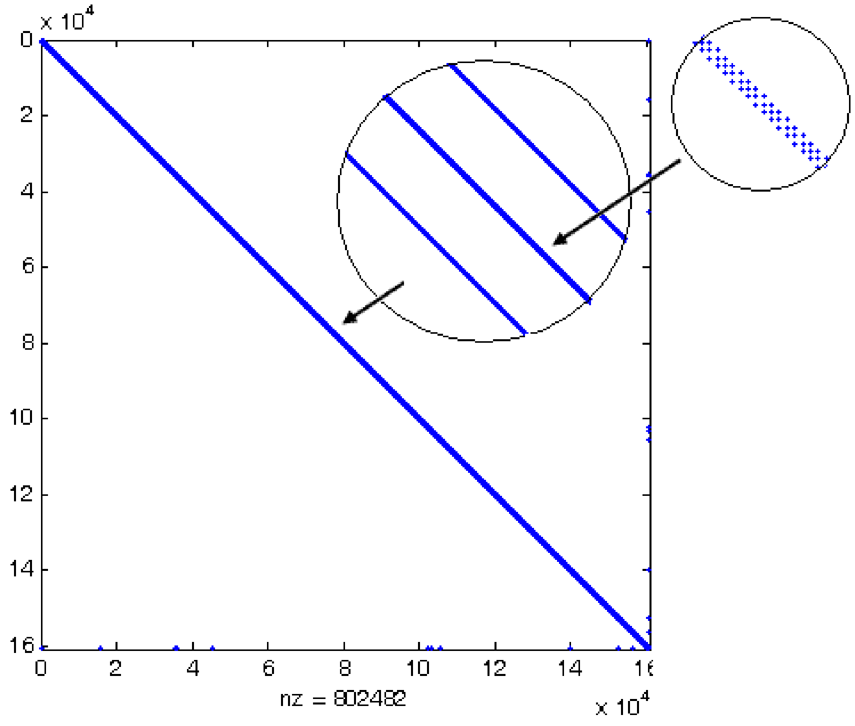


Figure 6.10: Symbolic pattern of original Jacobian matrix.

Chapter 7

Conclusion

7.1 Contributions

In this study, we analyzed general simulation flows in engineering and science, we defined the core of simulation as a series of linear equations $Ax = b$, and we found that the computational bottleneck lies in solving these linear systems. We then addressed the challenge of the computational bottleneck by modifying the matrix A's factorization at each simulation step based on the similarities between two consecutive linear systems, and we defined the similarities as an initial matrix A_0 with multiple-rank corrections. We designed algorithms to solve the linear systems based on updating concepts. The main contributions of this work are as follows:

- Defining the core of a general simulation as

  $$A^{(i)}x^{(i)} = b^{(i)}, \quad i = 1, 2, \ldots, n_t,$$

  and modeling each problem as

  $$(A_0 + \Delta A)x = b, \quad \text{given } A_0 = L_0U_0.$$

- Defining $(A_0 + \Delta A)x = b$ as a multiple-rank update problem, and deriving algorithms to convert to a sum of rank-1 updates:

  $$A_0 + \Delta A = A_0 + \sum u_i v_i^T.$$

- Deriving a new stable algorithm for unsymmetric rank-1 updating:

  $$A_0 + \alpha uv^T = L_0U_0 + \alpha uv^T = LU.$$

- Defining multiple-rank update algorithms and group-update algorithms.

- Designing a pipeline algorithm for parallel multiple-rank updates.
- Analyzing a sparse-matrix update procedure, finding why sparse updating is slow, and obtaining the solution by designing our so-called base-camp algorithm.
- Proving Theorem 1, which eliminates the synchronization difficulty in parallel computing.
- Proving Theorem 2, which is the basis for sparse-matrix multiple/group updates.
- Discussing application problems in structure analysis.
- Discussing application problems in circuit simulation.

The results obtained are confirmed using available public resources, including the circuit simulator SPICE3, the direct sparse solver KLU, our own variable-bandwidth solver, and the bridge structure analyzer NONBAN.

7.2 Future research

The long-term benefits of the proposed algorithms will be in improving the efficiency of computer simulation of devices in mechanical engineering, structural engineering, fluid dynamics, chemical engineering, aerospace, physics, and so on. Our algorithms are suitable for use on GPU hardware, and will assist future development of the GPU hardware itself.

More work is needed to extend the range of applicability of the techniques and to demonstrate their robustness. Specifically, the following tasks should be further developed in the future:

- Forming ΔA more accurately and efficiently.
- Designing an improved refinement procedure that can use physical information from ΔA.
- Running more test cases in circuit simulation.
- Optimizing the base-camp procedure.
- Applying multiple-update algorithms within other fields.
- Performing tests of the parallelization strategies.
- Developing new simulation engines that use multi-rank updates more efficiently. For example, we can use the detailed changes in the diagonal elements of U to identify the latency properties that define ΔA and circuit functionality.

Bibliography

[ADD96] P. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[AKT87] A. M. Abu Kassim and B. H. V. Topping. Static reanalysis: A review. *J. Struct. Engr.*, 113(5):1029–1045, 1987.

[ALT$^+$03] N. Abbound, M. Levy, D. Tennant, J. Mould, H. Levine, S. King, C. Ekwueme, A. Jain, and G. Hart. Anatomy of a disaster: A structural investigation of the World Trade Center collapse. In *Forensic Engineering, Proceedings of the 3rd congress, San Diego, CA*, 2003.

[Aro76] J. S. Arora. Survey of structural reanalysis techniques. *J. Struct. Div., ASCE*, 102(4):782–802, 1976.

[Bar71] R. H. Bartels. A stabilization of the simplex method. *Numer. Math.*, 16:414–434, 1971.

[Bat96] K. J. Bathe. *Finite Element Procedures*. Prentice-Hall, NJ, 1996.

[Ben65] J. M. Bennett. Triangular factors of modified matrices. *Numerische Mathematik*, 7:217–221, 1965.

[BG69] R. H. Bartels and G. H. Golub. The simplex method of linear programming using LU decomposition. *Comm. ACM*, 12(5):266–268, 1969.

[BGH03] S. Börm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *EABE*, 27:403–564, 2003.

[BO99] C. M. Bender and S. A. Orszag. *Advanced Mathematical Methods for Scientists and Engineers I: Asymptotic Methods and Perturbation Theory*. Springer, 1999.

[BP94] D. Bini and V. Y. Pan. *Polynomial and Matrix Computations: Volume 1: Fundamental Algorithms*. Birkuauser, 1994.

[BYCM93] R. Burch, P. Yang, P. Cox, and K. Mayaram. A new matrix solution technique for general circuit simulation. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 12:225–241, 1993.

[BZ01] Z. P. Bazant and Y. Zhou. Why did the World Trade Center collapse? – Simple analysis. *Archive of Applied Mechanics*, 71:802–806, 2001.

[CB86] S. M. Chan and V. Brandwajn. Partial matrix refactorization. *IEEE Trans. on Power Systems*, pages 193–200, 1986.

[CD86] L. O. Chua and A. C. Deng. Canonical piecewise-linear modeling. *IEEE Transactions on Circuits and Systems*, 5:511–525, 1986.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[Dava] T. A. Davis. Klu. `http://www.cise.ufl.edu/research/sparse/klu/`.

[Davb] T. A. Davis. Summary of available software for sparse direct methods. `http://www.cise.ufl.edu/research/sparse/codes/`.

[Dav06] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. SIAM, Philadelphia, 2006.

[Dem97] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[DER86] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[DG01] L. Deng and M. Ghosn. Pseudoforce method for nonlinear analysis and reanalysis of structural systems. *J. of Structural Engineering, ASCE*, 127(5):570–578, 2001.

[DGZC01] L. Deng, M. Ghosn, A. Znidaric, and J Casas. Nonlinear flexural behavior of prestressed concrete girder bridges. *ASCE J. of Bridge Engineering*, 6(4):276–284, 2001.

[DH99] T. A. Davis and W. Hager. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. App.*, 20(3):606–627, 1999.

[DH01] T. A. Davis and W. Hager. Multiple-rank modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 22(4):997–1013, 2001.

[DL] J. W. Demmel and X. Li. SuperLU: Sparse Gaussian elimination on high performance computers. `http://www.eecs.berkeley.edu/~demmel/SuperLU.html`.

[DS00] R. W. Dutton and A. J. Strojwas. Perspectives on technology and technology-driven CAD. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1544–1560, 2000.

[DS04] T. A. Davis and K. Stanley. KLU: a “Clark Kent” sparse LU factorization algorithm for circuit matrices. In *2004 SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

[DV07] Z. Drmac and K. Veselic. New fast and accurate Jacobi SVD algorithm. I. *SIAM J. Mat. Anal. Applics.*, 29(4):1322–1342, 2007.

[FB94] K. Fernando and Parlett B. Accurate singular values and differential qd algorithms. *Numer. Math.*, 67:191–229, 1994.

[FM84] R. Fletcher and S. P. J. Matthews. Stable modification of explicit LU factors for simplex updates. *Mathematical Programming*, 20:167–184, 1984.

[FM87] W. Fichtner and M. Morf, editors. *VLSI CAD Tools and Applications.* Kluwer Academic Publishers, 1987.

[Gar09] Gartner. *Gartner Dataquest.* Gartner Inc., 2009.

[GDX$^+$97] M. Ghosn, L. Deng, J. M. Xu, Y. Liu, and F. Moses. *Documentation of Program NONBAN.* National Cooperative Highway Research Program 12-36, Transportation Research Board, National Research Council, U.S.A., 1997.

[GGMS74] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28(126):505–535, 1974.

[GKO95] B. Gustafsson, H. O. Kreiss, and J. Oliger. *Time Dependent Problems and Difference Methods.* A Wiley-Interscience Publication, 1995.

[GL81] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[GM74] P. E. Gill and W. Murray. Newton-type methods for unconstrained and linearly constrained optimization. *Math. Prog.*, 28:311–350, 1974.

[GMH91] P. E. Gill, W. Murray, and Wright M. H. *Numerical Linear Algebra and Optimization.* Addison-Wesley Publishing Company, Redwood City, CA, 1991.

[GMS05] P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Rev.*, 47(1):99–131, 2005.

[God85] G. B. Godfrey, editor. *Multi-Storey Buildings in Steel.* John Hopkins University Press, Collins, London, England, second edition, 1985.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulation. *J. Comput. Phys.*, 73:325–348, 1987.

[GSS93] J. T. Gierlinski, R. J. Sears, and N. K. Shetty. Integrity assessment of fixed offshore structures: A case study using RASOS software. *Proc. 12th Int. Conf., ASME, New York*, pages 399–408, 1993.

[GVL96] G. H. Golub and C. F. Van Loan. *Matrix Computations.* John Hopkins University Press, Baltimore, MD, third edition, 1996.

[Haj80] I. N. Hajj. Sparsity considerations in network solution by tearing. *IEEE Transaction on Circuits and Systems*, 5, 1980.

[Hea02] M. T. Heath. *Scientific Computing*. McGraw Hill, 2002.

[HN87a] E. Hairer and S. P. Norsett. *Solving Ordinary Differential Equations I*. Springer-Verlag, 1987.

[HN87b] E. Hairer and S. P. Norsett. *Solving Ordinary Differential Equations II*. Springer-Verlag, 1987.

[HS91] J. Holnicki-Szulc. *Virtual Distortion Method*. Springer, Berlin, 1991.

[Hut98] J. Hutcheson. Executive advisory: The market for system-on-chip, July 15, 1998 and the market for system-on-chip testing, July 27, 1998. Technical report, VLSI Resarch Inc., 1998.

[HWP84] I. Hirai, B. P. Wang, and W. D. Pilkey. An efficient zooming method for finite element analysis. *International J. for Numerical Methods in Engineering*, 20:1671–1683, 1984.

[HZDS95] C. X. Huang, B. Zhang, A. C. Deng, and B. Swirski. The design and implementation of powermill. In *International Symposium On Low Power Design, April 1995*, 1995.

[IRT] IRTS. 2009 international technology roadmap for semiconductors. `http://www.itrs.net/Links/2008ITRS/Home2008.htm`.

[JHW$^+$01] R.-L. Jiang, J.-W. Hsia, M.-J Wang, C.-H. Wang, and J.-E. Chen. Guardband determination for the detection of off-state and junction leakages in DRAM testing. In *Proceedings of the Asian Test Symposium, 2001*, pages 151–156, 2001.

[KK98] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Computing*, 20(1):359–392, 1998.

[KM95] U. Kirsh and F. Moses. *An Improved Reanalysis Method for Grillage-Type Structures*. Cambridge, England, 1995.

[KR70] U. Kirsh and M. F. Rubinstein. Modification of structural analysis by the solution of a reduced set of equations. Paper Eng-0570, University of California, Los Angeles, CA, 1970.

[KR72] U. Kirsh and M. F. Rubinstein. Reanalysis of limited structural design modifications. *J. Engineering Mechanics Division, ASCE*, 98(1):61–70, 1972.

[Kun95] K. S. Kundert. *The Designer's Guide to Spice and Spectre*. Kluwer Academic Publishers, 1995.

[KWC00] C.-T. Kao, S. Wu, and J. E. Chen. Case study of failure analysis and guardband determination for a 64M-bit DRAM. In *Proceedings of the Asian Test Symposium, 2000*, pages 447–451, 2000.

[KY97] K. J. Kerns and A. T. Yang. Stable and efficient reduction of large, multiport RC networks by pole analysis via congruence transformations. *IEEE Trans. on Computer-Aided Design*, 16(7), 1997.

[LaM96] A. LaMarca. *Caches and Algorithms.* PhD thesis, Computer Science Department, University of Washington, WA, 1996.

[Law85] K. H. Law. Sparse matrix factor modification in structural reanalysis. *International J. Numerical Methods Engrg.*, 21:37–63, 1985.

[Law89] K. H. Law. On updating the structure of sparse matrix factors. *International J. Numerical Methods Engrg.*, 28:2339–2360, 1989.

[LKMS93] S. Lin, E. S. Kuh, and M. Mareck-Sadowska. Stepwise equivalent conductance circuit simulation technique. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 12, 1993.

[LL99] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *J. of Algorithms*, 31:66–104, 1999.

[Mar57] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3, 1957.

[MG79] W. McGuire and R. H. Gallagher. *Matrix Structural Analysis.* Wiley, New York, 1979.

[MMD08] M. W. Mahoney, M. Maggioni, and P. Drineas. Tensor-CUR decompositions for tensor-based data. *SIAM J. on Matrix Analysis and Applications*, 30:957–987, 2008.

[MR99] R. B. Makode, P. V. Corotis and M. R. Ramirez. Nonlinear analysis of frame structures by pseudodistortions. *J. Structural Engineering, ASCE*, 125, 1999.

[MUM] MUMPS: a MUltifrontal Massively Parallel sparse direct Solver. `http://graal.ens-lyon.fr/MUMPS/`.

[MW73] B. Mohraz and R. N. Wright. Solving topologically modified structures. *Computers and Structures*, 3:341–35, 1973.

[Nag75] L. W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* PhD thesis, Electronic Research Laboratory, College of Engineering, University of California at Berkeley, May 1975.

[Nas02] Nassda. Hsim user's guide. Technical report, Nassda, 2002.

[New79] A. R. Newton. Techniques for the simulation of large-scale integrated circuits. *IEEE Trans. CAS*, Sept, 1979.

[NPVS81] A. R. Newton, D. O. Pederson, A. L. S. Vincentelli, and C. H. Sequin. Design aids for VLSI: The Berkeley perspective. *IEEE Transactions on Circuits and Systems*, 7:660–680, 1981.

[OF02] S. J. Osher and R. P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

[PRV95] T. L. Pillage, R. A. Rohrer, and C. Visweswariah. *Electrical Circuit and System Simulation Methods*. McGraw-Hill, Inc., 1995.

[PS82] C. C. Paige and M. A. Saunders. Algorithm 583, LSQR: Sparse linear equations and least-squares problems. *TOMS*, 8(2):195–209, 1982.

[PTVF99] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flammery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1999.

[Rei01] T. R. Reid. *The Chip: How Two Americans Invented the Microchip and Launched a Revolution*. Random Hourse, New York, 2001.

[RH76] N. B. Rabbat and H. Y. Hsieh. A latent macromodular approach to large-scale sparse networks. *IEEE Transations on Circuits and Systems*, 12, 1976.

[ROTH94] V. B. Rao, D. V. Overhauser, T. N. Trick, and L. N. Hajj. *Switch-Level Timing Simulation of MOS VLSI Circuits*. Kluwer Academic Publishers, 1994.

[RSVH79] N. B. C. Rabbat, A. L. Sangiovanni-Vincentelli, and H. Y. Hsieh. A multilevel newton algorithm with macromodeling and latency for the analysis of large-scale nonlinear circuits in the time domain. *IEEE Transactions on Circuits and Systems*, 9, 1979.

[Sau09] M. A. Saunders. LUSOL: Sparse LU for $Ax = b$. `http://www.stanford.edu/group/SOL/software/lusol.html`, 2009.

[SCH67] R. L. Sack, W. C. Carpenter, and G. L. Hatch. Modification of elements in the displacement method. *AIAA J.*, 5(9):1708–1710, 1967.

[SH03] V. Stojanovic and M. Horowitz. Modeling and analysis of high-speed links. In *Custom Integrated Circuits Conference, Sep. 2003*, 2003.

[She99] B. N. Sheehan. TICER: Realizable reduction of extracted RC circuits. In *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conferences on*, pages 200–203, 1999.

[SJN94] R. Saleh, S. J. Jou, and A. R. Newton. *Mixed-Mode Sumulation and Analog Multilevel Simulation*. Kluwer Academic Publishers, 1994.

[SK99] D. Sylvester and K. Keutzer. Rethinking deep-submicron circuit design. *IEEE Computer*, Nov:25–33, 1999.

[SM49] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *Annals of Mathematical Statistics*, 20:621–621, 1949.

[spi] The SPICE Page. `http://bwrc.eecs.berkeley.edu/Classes/icbook/SPICE/`.

[ST75] A. Szygenda and E. W. Thompson. Digital logic simulation in a time-based, table-driven environment. Part I, Design verification. *IEEE Computer Magazine*, March:24–36, 1975.

[Syn03] Synopsis. Nanosim user's guide. Technical report, Synopsis, 2003.

[Ter83] C. J. Terman. *Simulation Tools for Digital LSI Design*. PhD thesis, MIT, 1983.

[TH00] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, second edition, 2000.

[TS03] S. X. D. Tan and C. J. R. Shi. Balanced multi-level multi-way partitioning of analog integrated circuits for hierarchical symbolic analysis. *Integration, the VLSI Journal*, 34:65–58, 2003.

[Wik05] WTC floor picture. `http://en.wikipedia.org/wiki/Image:World_Trade_Center_Building_Design_with_Floor_and_Elevator_Arrangment.jpg`, 2005.

[WP80] B. P. Wang and W. D. Pilkey. Efficient reanalysis of locally modified structures. Technical report, Department of Mechanical and Aerospace Engineering, Virginia University, Charlottesville, VA, 1980.

[WP81] B. P. Wang and W. D. Pilkey. Parameterization in finite element analysis. In *Proceedings of International Symposium on Optimum Structural Design, Tuscon, AZ*, pages 7.1–7.7, 1981.

[WPP83] B. P. Wang, W. D. Pilkey, and A. R. Palazzola. Reanalysis, modal synthesis and dynamic design. In A. K. Noor and W. D. Pilkey, editors, *State-of-the-art Surveys on Finite Element Technology*, pages 225–295. American Society of Mechanical Engineers, 1983.

[Wu76] F. F. Wu. Solution of large scale networks by tearing. *IEEE Transaction on Circuits and Systems*, 12, 1976.

[YZP08] Z. Ye, Z. Zhu, and J. R. Philips. Generalized Krylov recycling methods for multiple related linear equation systems in electromagnetic analysis. In *DAC 2008, June 8–13, Anaheim, CA*. DAC, 2008.

[ZM97] A. Znidaric and F. Moses. Structural safety of existing road bridges. In *7th International Conference on Structural Safety and Reliability (ICOSSAR), Kyoto*, pages 1843–1850, 1997.

[ZM98] A. Znidaric and F. Moses. Resistance of deteriorated post-tensioned concrete beams: An ongoing research. In *8th IFIP WG 7.5 Working Conference on Reliability and Optimization of Structural Systems, Krakow*, pages 339–346, 1998.

[ZT05] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method for Solid and Structural Mechanics*. Elsevier Butierworth Heinemann, sixth edition, 2005.