AN IMPROVED IMPLEMENTATION OF SPARSITY DETECTION OF SPARSE DERIVATIVE MATRICES

TASNUBA JESMIN
Bachelor of Science, Mawlana Bhashani Science and Technology University, 2013

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

AN IMPROVED IMPLEMENTATION OF SPARSITY DETECTION OF SPARSE DERIVATIVE MATRICES

TASNUBA JESMIN

Date of Defence: August 21, 2018

Dedication

dedicated to my parents &

my husband....

Abstract

Optimization is a crucial branch of research with application in numerous domain. Determination of sparsity is a vital stream of optimization research with potentials for improvement. Manual determination of sparsity structure of Jacobian matrix for a large problem is complicated and highly error-prone. The main motivation of this research is to propose an efficient algorithm which can effectively detect and represent sparsity of unknown Jacobian matrices. Automated sparsity detection algorithms find an optimal or near-optimal solution, which reduces time and space complexity for large scale data. Our proposed approach efficiently generates symmetric pattern utilizing band matrix and reduces the number of gradient evaluation. For efficient solution, we integrate our approach with existing pattern detection process. Greedy coloring algorithm is used for column portioning and multilevel algorithm with voting scheme is implemented for detection of sparsity pattern. Finally, parallel computation is used to reduce processing time of the overall approach.

Acknowledgments

I would like to express my gratitude to everyone who encouraged and guided me throughout this thesis work. First and foremost, I would like to offer my sincerest appreciation towards my supervisor, Dr. Shahadat Hossain for his continuous guidance and valuable recommendations throughout my Masters program. His patience and knowledge have always enlightened my research path. This thesis is a testament to his encouragement and without him, it would not have been completed.

I would also like to express sincere appreciations towards Dr. Robert Benkoczi and Dr. Saurya Das for being a part of my supervisory committee and offering their valuable guidance and wisdom.

I must express my gratitude to my parents who have always loved me unconditionally and prayed for me throughout the time of my research. I am really grateful to them for teaching me to work hard for the things that I aspire to achieve.

Finally and most importantly, I owe thanks to a very special person, my husband, Jeeshan for his continued and unfailing love, support and encouragement during the challenges of graduate school. He was there to pull me through the times when things seemed impossible and helped keep things in perspective.

Lazima, Imtiaz, Nabi, Shayla, Priya, Wali and Shoeb thank you for always being there, your constant support was a great source of motivation. You have been more than friends in this journey.

My acknowledgment would be incomplete without expressing my gratitude to God Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research study and to persevere.

Contents

List of Tables

List of Figures

List of Notations

S_0 : **I**nitial Guess pattern

M : **O**riginal Matrix pattern

(.) : **Z**ero Entries in a Matrix

ψ : **S**tructurally Orthogonal Column Partition or Mapping of a Matrix

β : **O**rthogonal Column of Color Group

f^J : **F**laws in J Matrix denoted as J Flaws

f^Y : **F**laws in Y Matrix denoted as Y Flaws

p : **C**ombination of J and Y Flaws

p^T : **T**ranspose matrix of p Matrix

P : **P**ossible flaw matrix

n : **N**umber of Rows / Columns

nnz : **N**umber of Nonzeros

ρ_{max} : **M**aximum number of nonzeroes in any row

η : **P**ositive Threshold

α : **B**andwidth of a Symmetric Pattern

Chapter 1

Introduction

Exploitation of sparsity plays an important role in the computer implementation of algorithms, numerical analysis, and partial differential equations. Prior knowledge of sparsity and the structure of matrices is beneficial for storage and computation. Application of numerical methods (column partitioning and effective data structures) on the sparse matrix can be effective with the known structural pattern of matrices.

This thesis focuses on sparsity calculation of sparse derivative matrices. Designing the proposed approach of this thesis starts with an intuitive formulation of the derivative matrix computation, data structure for storage and computation and sequential algorithm. In a step by step manner, we incrementally develop a Multilevel version of our application and finally arrive at an efficient sequential and multicore parallel implementation. The performance of our solutions is demonstrated in the result section.

1.1 Motivation

Repeated evaluation of Jacobian matrices is a common challenge in numerical analysis and computing algorithms in general. The problem becomes more significant when the matrix is sparse. However, the computation of the Jacobian of a sparse matrix is more challenging that of the dense one. In the computation of sparse matrix, zero elements are not stored explicitly and also their computation is avoided. Matrix size, the pattern of data organization, memory traffic, the organization of cache and also the member of floating point operation affect computation complexity of sparse matrix.

In 1974, Curtis, Powell and Reid observed that the sparsity pattern of a Jacobian matrix can be determined by using a few function evaluations [13]. In the context of minimizing the number of function evaluations for computation of sparse matrix, matrix partitioning problem is raised. Different researcher proposed different partition problem but those modeled as a special graph coloring problem. The graph coloring models provide effective heuristic algorithms for matrix partition.

Most methods for solving non-linear equation require the evaluation or estimation of Jacobian matrices. Sparsity pattern detection for that Jacobian matrices can be tedious and highly error-prone by hand even when the source code for the function evaluation program is available. There are few works on automatically detection pattern process on the available source code. If the source code is unavailable, then there required another approach. To solve this, Carter, Hossain, and Sultana [8] used symmetric information of derivative matrices to detect the missing or wrong entries in the approximate sparsity pattern, for the determination of the true pattern of sparse Jacobian matrices.

The overall objective of this research is to improve their implementation for effective and efficient detection of symmetric, sparse, Jacobian matrix which is unknown. Existing data set were analyzed and we overcome the limitations of existing approaches by utilizing a band matrix instead of the random pattern. In addition, we added a *parallel Multilevel algorithm* for solving systems of nonlinear equations where the Jacobian is known to be sparse. Overall the main target will be reducing the number gradient evaluation and execution time.

1.2 Our Contributions

Since we assume that the Jacobian matrix is available as a black box which is symmetric and sparse, we tried to identify the original pattern by utilizing symmetry. We have implemented an efficient data structure for the sparse matrix to save memory space and time. Our contributions are listed below:

- Instead of the random matrix, we have used tridiagonal band matrix for generating initial guess pattern which reduces gradient evaluation cost by almost half than previous [38].
- For column partitioning, we use the New Exact approach of DSJM [28] which can give better coloring, compared to methods used in [9].
- To detect flaws (missing entries in the approximated pattern), the symmetric matrix is usually expanded to full matrix as done in [9]. We avoid full expansion and deal with non-zero entries outside of the tri-diagonal band, which increase efficiency.
- We used parallelism to improve the efficiency of thesis [38]. The limitations and future works of [38] were identified and implemented accordingly in this thesis.
- We present our numerical results using large-scale data set collected from two verified sources: [2] and [3].
- Table 5.2 and 5.3 shows the efficiency of this work, compared to [38]. Table 5.4 and 5.5 represent execution time comparison between two different computer configuration and parallel implementations.
- Parallel execution using different multicore technique is presented in table5.6 and table 5.7

1.3 Thesis Organization

Including this one, this thesis has six chapters. In this chapter, we have already discussed some basic concept on the Jacobian matrix, the necessity of Jacobian matrix and some literature related to manipulation of sparse Jacobian matrix.

In chapter **2** we introduce some definition and preliminaries. This chapter will conclude with a description of the CPR algorithm.

In chapter **3** efficient data structures for sparse matrix storing and manipulation for our algorithm are described. Some computation process to detect sparsity pattern is also included in this chapter.

In chapter **4** we will discuss the processes to detect missing pattern elements and Multilevel algorithm for large-scale sparsity pattern of Jacobian matrix determination. Concept of parallelism and implementation of it in the Multilevel algorithm will also be described in the last part of this chapter.

In chapter **5** we represent our experimental results that demonstrate the efficiency of our data structure and algorithm. We also show the comparison of efficiency among sequential and parallel computational with results reported in the literature.

We wrap up this thesis with concluding remarks and possible future directions presented in chapter **6**.

Chapter 2

Problem Definition and Background Study

In this chapter, we will introduce some mathematical definition and preliminaries which will be helpful to describe this thesis. We will also discuss the CPR algorithm to discover sparse Jacobian matrix mismatch pattern. We will explain matrix partitioning and motivation of column partitioning for determination of sparse Jacobian matrix. Finally, we will present the data structure for flaw identification.

2.1 Sparse Matrix

A matrix can be denoted as sparse whenever we can utilize the advantages of the large number of zero elements and their locations. The Structural plot of BCSPWR02 [2] sparse matrix containing 49 rows and 49 columns is shown in figure 2.1. The matrix has $49 \times 49 = 2401$ elements, but it only has 108 non-zero elements. Approximately 96% of total elements are zeros in this matrix, which indicates high sparsity of this matrix.

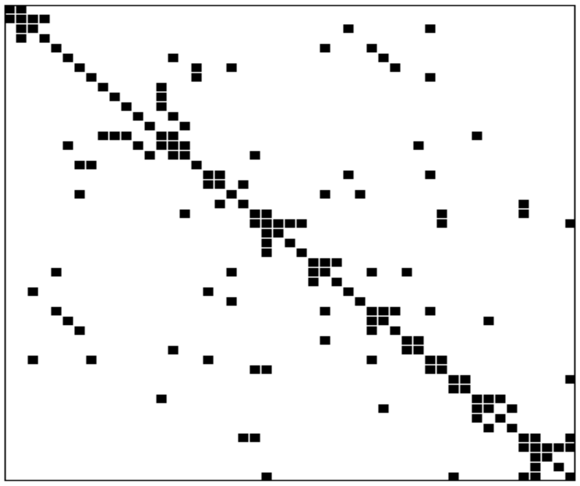

Figure 2.1: Example of Sparse Matrix (Non-zero elements are shown in black) Name:bcspwr_02, size:49×49, 108 non-zero elements, collected [2]

2.2 Band Matrix

Band matrix is a sparse matrix in which the non-zero elements are located in a band surrounding the main diagonal. Band is an area of non-zero elements which are confined to the diagonal region or around specific regions inside the matrix. This implies that non-zero elements will not be scattered all around the matrix. If a matrix has lower bandwidth p then $a_{ij} = 0$ where $j > i$ and $i > j + p$ and for upper bandwidth q, $a_{ij} = 0$ where $i > j$ and $j > i + q$ [19].

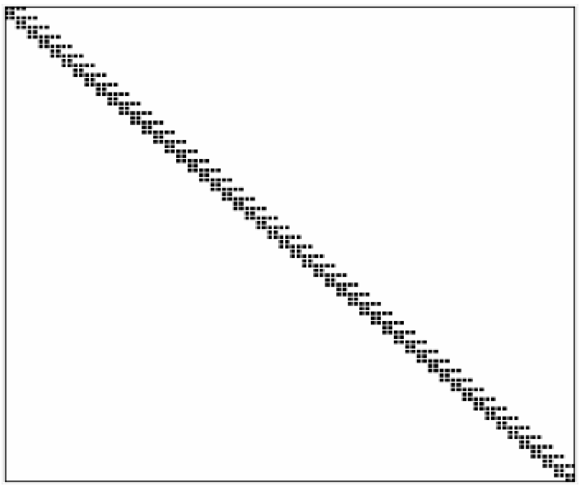

Figure 2.2: Example of Band Matrix (Non-zero elements are shown in black) Name:olm100, size:100×100, 396 non-zero elements, collected [2]

Utilization of band Matrix for matrix computations, especially in a symmetric matrix, is gaining popularity. To improve performance in optimization for sparse computations, the concept of a band matrix can be a great approach. One such case is reported by Melgaard and Sincovec, who have shown that band matrices can be estimated with a few evaluation of Algorithmic Differentiation (AD) [32]. When the non-zero elements of a band matrix located only on the diagonal plus or minus one column that matrix is called Tri-diagonal Band Matrix [36]. In tri-diagonal band matrix k1 = k2 = 1. In this thesis, we will use Tri-diagonal Band Matrix as an approximated symmetric pattern.

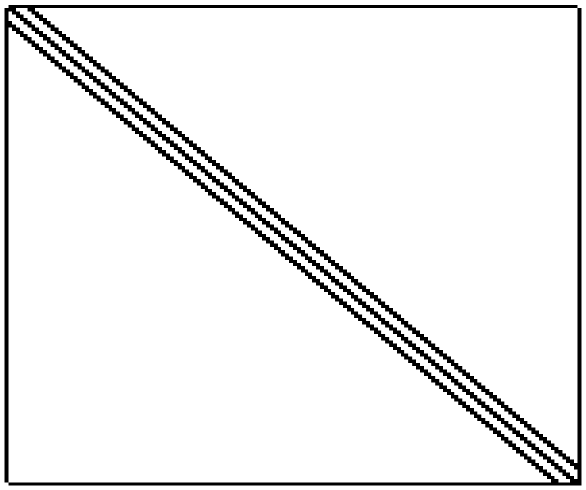

Figure 2.3: Example of Tri-diagonal Band Matrix (Non-zero elements are shown in black) Name:young1c, size:841 $\times$ 841, 4089 non-zero elements, collected [2]

$$\begin{bmatrix}
a_{11} & a_{12} & . & . & . & . & . & . & . \\
a_{21} & a_{22} & a_{23} & . & . & . & . & . & . \\
. & a_{32} & a_{33} & a_{34} & . & . & . & . & . \\
. & . & a_{43} & a_{44} & a_{45} & . & . & . & . \\
. & . & . & a_{54} & a_{55} & a_{56} & . & . & . \\
. & . & . & . & a_{65} & a_{66} & a_{67} & . & . \\
. & . & . & . & . & a_{76} & a_{77} & a_{78} & . \\
. & . & . & . & . & . & a_{87} & a_{88} & a_{89} \\
. & . & . & . & . & . & . & a_{89} & a_{99}
\end{bmatrix}$$

Figure 2.4: Example of Tri-diagonal Band Matrix with size 9×9

2.3 Problem Definition

In numerical analysis getting the exact solution is not always possible and/or doing the whole calculation by hand is error-prone. This is particularly true for optimization problems where the variables correspond to mesh points in scattered position. As an example, if we try to compute $\sqrt{3}$ by using a calculator, then it is easy to solve or get accurate answers. This is a relatively small example which leads to a large scale problem. Now let us consider $\sqrt{3}$ as a solution of the equation $f(x) = x^2 - 3 = 0$. For this equation, it is hard to find out an exact numerical solution. Since it is a nonlinear problem, solving the approximate equation near a given point by its tangent line will be easier. Suppose, somehow we got an initial approximation for this solution which is x_0. This initial approximation is probably not that good and so we would like to find a better approximation.

We know that $f(x) = x^2 - 3$ so that $f'(x) = 2x$. Now, this is where it becomes interesting since we can repeat derivations which will provide even better approximate solutions to the equation. We could generate better approximations for $\sqrt{3}$ at every step and apply linear equation to get better results than before. This is the key concept of Newton's Method [14][27]. But one of the main drawbacks of Newton's method is, it requires derivative information for every iteration, which increases function evaluation cost[27].

2.3.1 Gradient

Let us consider a minimization problem $min f(x)$ where $x \in \mathbb{R}^n$. We assume that this function $f : \mathbb{R}^n \mapsto \mathbb{R}$ continuously differentiable twice. The first derivative of $f(x)$ is denoted by $g(x) : \mathbb{R}^n \mapsto \mathbb{R}^n$, which is the gradient of that function and $g \equiv \bigtriangledown f$. This gradient function of $f(x_1, x_2, x_3, \ldots, x_n)$ can be represented as

$$g(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \\ . \\ . \\ . \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{2.1}$$

whose elements $\frac{\partial f}{\partial x_i}$ are the partial derivative with respect to $x_i | i = 1, \ldots, n$ of given function $f(x)$. Suppose we have a function

$$f(x) = 9x_1^2 + Sinx_1 + 8x_2^3 + Cosx_2 \tag{2.2}$$

The gradient $g(x)$ for this function will be calculated depending on two independent variable x_1 and x_2

$$g(x) = \bigtriangledown f(x_1, x_2) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

$$g(x_1, x_2) = \begin{bmatrix} 18x_1 + Cosx_1 \\ 24x_2^2 - Sinx_2 \end{bmatrix}$$

2.3.2 Jacobian of the Gradient

Now the derivative of this gradient function $g(x)$ is denoted by J, which is independent of the order and symmetric. This Jacobian of function $g : \mathbb{R}^n \mapsto \mathbb{R}^n$ is also called Hessian of function f and $J \equiv \bigtriangledown^2 f$. This Jacobian function $J : \mathbb{R}^n \mapsto \mathbb{R}^{n\times n}$ is symmetric and sparse and it can be represented as:

$$J = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\frac{\partial f}{\partial x_1}) & \ldots & \frac{\partial f}{\partial x_1}(\frac{\partial f}{\partial x_n}) \\ \frac{\partial f}{\partial x_2}(\frac{\partial f}{\partial x_1}) & \ldots & \frac{\partial f}{\partial x_2}(\frac{\partial f}{\partial x_n}) \\ . & . & . \\ . & . & . \\ . & . & . \\ \frac{\partial f}{\partial x_n}(\frac{\partial f}{\partial x_1}) & \ldots & \frac{\partial f}{\partial x_n}(\frac{\partial f}{\partial x_n}) \end{bmatrix} \tag{2.3}$$

If we calculate Jacobian $J(x)$ of gradient $g(x)$ for function $f(x)$ (2.2) using equation (2.3), it will be as follows:

$$J(x) = \bigtriangledown g(x_1, x_2) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\frac{\partial f}{\partial x_1}) & \frac{\partial f}{\partial x_1}(\frac{\partial f}{\partial x_2}) \\ \frac{\partial f}{\partial x_2}(\frac{\partial f}{\partial x_1}) & \frac{\partial f}{\partial x_2}(\frac{\partial f}{\partial x_2}) \end{bmatrix}$$

$$J(x_1, x_2) = \begin{bmatrix} 18 - Sinx_1 & 0 \\ 0 & 48x_2 - Cosx_2 \end{bmatrix}$$

Sparsity structure for matrix $J \in \mathbb{R}^{n\times n}$ is symmetric where the nonzero entries are confined to the main diagonal.

Definition 2.1. The sparsity structure or sparsity pattern of any matrix $J \in \mathbb{R}^{n\times n}$ denote the nonzero entry specification of J. This thesis defines the sparsity pattern of J as $M = \{(i,k) | m_{i,k} \neq 0, i,k = 1,2,\ldots,n\}$ where $J \in \mathbb{R}^{n\times n}$. Sparse matrix follows this kind of structure where most non-zero entries stayed in main diagonal entries or in a band. The entries of the matrix outside the band allowed to be zero.

Definition 2.2. A sparsity pattern is symmetric if there is a non-zero element at position

(i,k) where i is row index and k is column index for that non-zero element, then there must be a non-zero element at (k,i) position.

In case of dense Jacobian matrix, it is not feasible to compute Hessian matrix. In this situation, according to McCormick [31] approximation to the Hessian can be computed using finite differences of the gradient. If that Jacobian hold sparse structure (i.e., some certain elements are known to be zero), then it is possible to approximate that matrix with a smaller number of gradient evaluations. Gradient evaluation in numerical analysis of a nonlinear function is expensive. In our computation, we tried to keep step size as small as possible. This evaluation process follows differentiation using specially selected sets of vectors as needed.

2.3.3 Finite Difference Approximation

Application of finite difference techniques initiated in the early 1950s [30]. This technique helps to deal with the complex problem of numerical analysis including computation of approximate value. The difference between the numerical solution and the explicit solution is determined by the *flaw*. This flaw is occupied by going from a differential operator to a difference operator. The simple finite difference approximation is :

$$f' \approx \frac{(f(x+h)-f(x)}{h} \tag{2.4}$$

f' denotes the approximate first derivative for function $f(x)$, which means f is differentiable at point x. $h = \varepsilon e_i$ where i=1,2,...,n, e_i is the i^{th} unit vector that is the vector whose elements are all zero except for a 1 in the i^{th} position. Since the ε is very small, so that error will be small and the approximation derivative will be very close to true derivative. In equation 2.4 while $h > 0$ is considered as a forward difference, if $h < 0$ considered as a backward difference. Jacobian of the first derivative of f can be obtained using finite difference. Finite difference approximation comes from Taylor's theorem and using this we can get an approximation of Jacobian–vector product. Even if the gradient is not available, we can use Taylor's theorem for approximation [34]. By exploiting symmetry it is possible to determine unknown pattern of Jacobian matrix as we have already described about the symmetry and sparsity of Jacobian matrix.

2.4 Determination of Sparse Derivative Metrics

Curtis, Powell, and Reid proposed an algorithm to estimate the sparse Jacobian matrix called the CPR algorithm [13] for column partitioning. The CPR technique uses a greedy method to partition columns of a matrix into structurally orthogonal groups. Powell and Toint [35] extended the CPR method. McCormick [31] introduced two new ways of classifying direct methods for pattern detection and graph coloring model for the computation of the Hessian matrix.

Coleman and Moré suggested graph coloring heuristics [11] after analyzing column partitioning problem. They showed that the problem of finding a minimum cardinality of column partitioning is NP-Hard. And this problem is equivalent to a vertex coloring problem of an associated graph.

On the other hand, Goldfarb and Toint [18] studied the finite difference approximations for a partial differential equation which give an optimal estimation for the Jacobian matrix. They developed a uniform graph-theoretic framework for studying the matrix partitioning problems. A complete overview of graph coloring methods for detection of the sparse Jacobian pattern is presented by Gebremedhin and et.al [16]. According to them, to get better efficiency one should take advantage of sparsity and symmetry of the derivative matrices to compute the non-zero entries.

Andreas and Christo [21] also propose a method to detect sparsity pattern of the Jacobian matrix using Algorithmic Differentiation approach. According to Walther [41], sparsity pattern of Jacobian of gradient function, can be determined using forward accumulation. Automatic Differentiation tools such as ADOL-C [41], MAD [15], ADMAT [1] are able to calculate sparsity from the given function evaluation code. If the function evaluation code is unavailable or if the function is provided as black box then alternative efficient techniques are required. One disadvantage of AD application is that it is expensive[40]. In this situation, effective selection of data structure and efficient implementation of the algorithm will help to reduce the number of gradient evaluation for different matrix operations.

2.4.1 CPR Algorithm

Column partitioning consistent with direct determination allows structurally dependent column groups under certain restrictions. Methods in this category usually require fewer gradient evaluations [35, 16]. CPR algorithm is very competitive in finding a minimal number of column group for Jacobian matrix. Implementation of CPR algorithm also requires significantly less execution time for structurally orthogonal column partitioning [39].

Definition 2.3. Structurally Orthogonal Partitioning of the columns of a matrix M is the allocation of columns into groups in such a way that there will be no two columns have the non-zero elements in the same row position in a group. In other words, columns $M(:,j)$ and $M(:,k)$ are structurally orthogonal if there is no such row index i for which $m_{ij} \neq 0$ and $m_{ik} \neq 0$. In this thesis we have denoted structurally orthogonal column partition with ψ.

Definition 2.4. ψ :{1,2,...,n}$\rightarrow$ {1,2,...,p} is a mapping from the set of column indices to groups or colors such that $\psi(j) \neq \psi(l)$ implies columns j and l are structurally dependent.

If a group of columns of matrix M, say column j and l are structurally orthogonal, i.e., no two columns have non-zero entries in the same row position, only one extra function evaluation

$$\frac{\partial F(x+tS)}{\partial t}|_{t=0} = F'(x)S \approx MS = \frac{1}{\varepsilon}[F(x+\varepsilon S) - F(x)] \equiv Y \tag{2.5}$$

is sufficient to read-off the non-zero entries from the product $Y = MS$, where $S = e_j + e_l$ (e_j= column j and e_l= column l)[13]. The value of ε is very small.

Let matrix M be the Jacobian matrix of some function $g(x)$ at x.

$$M = \begin{bmatrix} m_{11} & . & m_{13} & . & . \\ . & m_{22} & . & . & m_{25} \\ m_{31} & . & m_{33} & . & m_{35} \\ . & . & . & m_{44} & . \\ . & m_{52} & m_{53} & . & m_{55} \end{bmatrix}$$

Table 2.1: Structurally orthogonal mapping ψ

1	1	2	1	3

There are three structurally orthogonal column group in matrix M, i.e. $p = 3$. Therefore matrix M can be approximated with three extra function evaluations of form in equation 2.5 for direction set to $e_1 + e_2 + e_4, e_3$ and e_5 in addition to evaluating F at x. The non-zero entries of the matrix can be determined using $Y = MS$ with:

$$S = \begin{bmatrix} e_1 + e_2 + e_4 & e_3 & e_5 \end{bmatrix}$$

and now

$$Y = \begin{bmatrix} m'_{11} & m'_{13} & . \\ m'_{22} & . & m'_{25} \\ m'_{31} & m'_{33} & m'_{35} \\ m'_{44} & . & . \\ m'_{52} & m'_{53} & m'_{55} \end{bmatrix}$$

where $'$ indicates the value which is different than M, result of $M * S$. The value in the matrix Y are computed by using Automatic Differentiation forward mode, thus free of truncation error are obtained at a small constant multiple of the cost of evaluating the function f [8].

Product of M and S gives us compressed matrix Y from which we can see to approximate the matrix M we need extra three function evaluations. Which means that if we apply CPR algorithm, then we just need 4 evaluations of f at $x+\varepsilon$ for vector S. Otherwise we need 6 evaluations of function f. Algorithm 1 is presented the process of the CPR algorithm.

Algorithm 1 Construct Y using CPR

```
Output: Matrix Y
1: for $i = 1$ to $p$ do
2:     Calculate S(:k) {where $S \in \mathbb{R}^{n \times p}$}
3:     Compute $Y(:,k) \leftarrow M * S(:,k)$
4: end for
```

In this thesis, we guess an approximated pattern to determine the pattern of true Jacobian where function code is unavailable. On that approximated pattern we will apply the CPR algorithm. The mismatch (missing non-zero elements relative to original pattern) between the approximated pattern and true Jacobian denoted as the word $flaw$ in this thesis. Detailed description of different types of $flaw$ will be described in 3.

Chapter 3

Efficient Data Structure for Storage and Flaw Computation

In this chapter, we will discuss efficient data structure to store a sparse matrix. Different types of data structure and their memory space requirement for computation will be described in this chapter. Efficient data structure and flaw detection in the guess pattern, will also be discussed in this chapter. At last, the method of flaw detection will be presented.

3.1 Data Structure for Sparse Matrix

In any sparse matrix, most elements are zero, as such, it is a wastage of space to store all matrix elements (zeros and non-zeros). It is naive to use a two-dimensional array for sparse matrix computation, as this is slow and requires large space because of storing zero entries. In order to save memory and time, there are different approaches those use the different data structure for storing the matrix along with the corresponding manipulation. This thesis aims to gain efficiency both in terms of memory utilization and arithmetic operations we can take advantages of the sparse structure.

As an example, let us consider the matrix shown in Figure 3.1 with dimension 87×87. The matrix has been collected form Matrix Market [2]. That matrix has total $87 \times 87 = 7569$ elements but only 541 elements are non-zero.

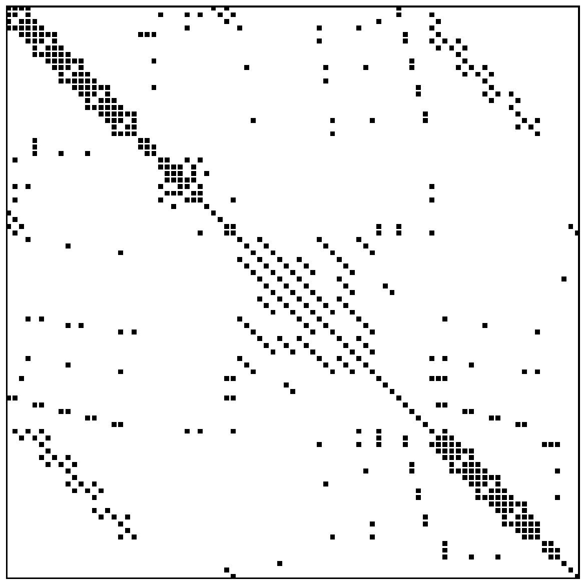

Figure 3.1: Sparse Matrix(Non-zero elements are shown in black) Name:dwt_87, size:87 × 87, 541 non-zero elements, collected [2]

If it can be determined that a matrix consists of few regions of diagonal non-zero elements then it is possible to generate a compressed version by simply storing these diagonal values as vectors and the offsets of each diagonal with respect to the central band. Figure 3.2 represent the compressed version of Matrix dwt_87.

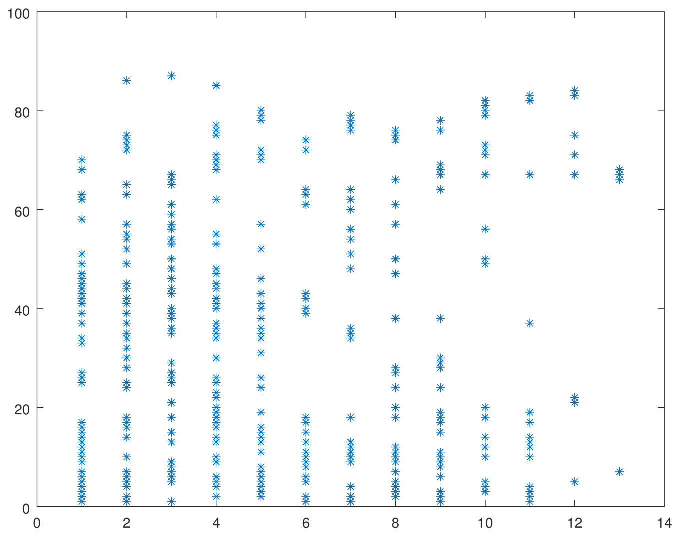

Figure 3.2: Compressed Matrix dwt_87, size 87 × 12

For sparse structure of sparse matrix it is easy to compress based on graph partitioning [10] which require less memory storage for computation. Here we discuss two categories of storage technique which are related to this thesis i.e., Coordinate and Compress storage processes.

3.1.1 Coordinate Storage

Coordinate storage represents any matrix by using three one-dimensional arrays which one is *value*, row index i and column index j of each non-zero elements. In Coordinate storage process, it is not mandatory to follow any kind of specific order for *value* storage. The content of each array can be stored in any order but it maintains order among themselves. Procedure for Coordinate Storage is given in 1.

Procedure 1: Coordinate Storage

```
1: for $k = 1$ to $nnz$ do
2:     Store $v[k] = M[i, j]$
3:     Store $r[k] = i$
4:     Store $c[k] = j$
5: end for
```

We taken one test example matrix $M^{i \times j}$ which have 13 non-zero nnz components to describe this process broadly. Data structure for coordinate storage is represented in figure 3.3:

$$M = \begin{bmatrix} m_{11} & . & . & . & . & m_{16} & . & . \\ . & m_{22} & . & & . & . & . & m_{28} \\ . & . & . & . & m_{35} & . & . & . \\ . & . & . & . & . & . & . & . \\ m_{51} & . & m_{53} & . & . & m_{56} & . & . \\ . & . & . & . & m_{64} & . & . & . \\ . & m_{72} & . & . & . & . & m_{77} & . \\ . & . & m_{83} & . & . & . & m_{88} & \end{bmatrix}$$

value

m_{11}	m_{16}	m_{22}	m_{64}	m_{72}	m_{77}	m_{83}	m_{88}	m_{28}	m_{35}	m_{51}	m_{53}	m_{56}

row

1	1	2	6	7	7	8	8	2	3	5	5	5

col

1	6	2	4	2	7	3	8	8	5	1	3	6

Figure 3.3: Coordinate Storage Data Structure for Sparse Matrix

In Coordinate Storage, size for value storing array is *nnz* and for each non-zero value we require row index in row array and column index in col array. Which means coordinate storage requires $nnz + nnz + nnz = 3nnz$ units memory location to store any matrix. So accessing the elements by row or column requires same cost.

3.1.2 Compressed Storage

Compressed storage can access all non-zero elements directly, making it cost effective and more faster than Coordinate Storage or other traditional processes which utilize two dimensional array storage data structure. In the following sections we discuss two types of Compressed Storage process to store sparse matrix on computer memory.

3.1.3 Compress Sparse Row (CSR) Data Structure

If a matrix is not regularly structured, then one of the most common and efficient storage schemes is Compressed Sparse Row (CSR) scheme. In this scheme all non-zero elements are stored in one dimensional array, while column indices and row pointer are stored in two different arrays. Since this scheme is faster in terms of access and computation, we have chosen CSR for storing and manipulation of matrix data.

In Compress Sparse Row format value of non-zero elements are stored in double type array, column indices are stored in *col_ind* array and row indices are stored in *row_ptr* array by pointing first column index of each row only for non-zero entries. CSR algorithm is shown in procedure 2.

Procedure 2: Compress Sparse Row(CSR)

```
1: for $k = 1$ to $N$ do
2:     for $l = row\_start[k]$ to $row\_start[k+1] - 1$ do
3:         $v[k] = M[k,l]$
4:         $c[k] = col\_ind[l]$
5:         $r[k] = r[k] + nnz[k] * M[l]$
6:     end for
7: end for
```

In figure 3.4 there is one small test example to make CSR data structure more understandable.

$$M = \begin{bmatrix} m_{11} & . & . & . & . & m_{16} & . & . \\ . & m_{22} & . & & . & . & . & m_{28} \\ . & . & . & . & m_{35} & . & . & . \\ . & . & . & . & . & . & . & . \\ m_{51} & . & m_{53} & . & . & m_{56} & . & . \\ . & . & . & . & m_{64} & . & . & . \\ . & m_{72} & . & . & . & . & m_{77} & . \\ . & . & m_{83} & . & . & . & m_{88} & \end{bmatrix}$$

value

m_{11}	m_{16}	m_{22}	m_{28}	m_{35}	m_{51}	m_{53}	m_{56}	m_{64}	m_{72}	m_{77}	m_{83}	m_{88}

col_ind

1	6	2	8	5	1	3	6	4	2	7	3	8

row_ptr

1	3	5	6	6	9	10	12	14

Figure 3.4: CSR Data Structure for Sparse Matrix

If there are m number of rows in any matrix, then size of row_ptr array will be $m+1$. Number of non-zero nnz elements will define the size of both $value$ and col_ind array. Therefore to store data of any sparse matrix using CSR requires $2nnz+m+1$ memory locations. Difference between $col\_ind[row\_ptr[i]]$ and $col\_ind[row\_ptr[i+1]-1]$ indicate column indices of any row i.

3.1.4 Compressed Sparse Column (CSC) Data structure

Compressed Sparse Column also store matrix as a collection of three tuples like CRS. However, CSC stores row index and col_ptr in place of col_ind and row_ptr (in CRS) respectively, the detailed account of CSC is presented in procedure 3.

Procedure 3: Compress Sparse Row(CCS)

```
1: for $k = 1$ to $N$ do
2:     for $l = col\_start[k]$ to $col\_start[[k+1]-1]$ do
3:         $v[k] = M[k,l]$
4:         $r[k] = row\_ind[k]$
5:         $c[l] = c[l] + nnz[l] * M[k]$
6:     end for
7: end for
```

$$M = \begin{bmatrix} m_{11} & . & . & . & . & m_{16} & . & . \\ . & m_{22} & . & & . & . & . & m_{28} \\ . & . & . & . & m_{35} & . & . & . \\ . & . & . & . & . & . & . & . \\ m_{51} & . & m_{53} & . & . & m_{56} & . & . \\ . & . & . & . & m_{64} & . & . & . \\ . & m_{72} & . & . & . & . & m_{77} & . \\ . & . & m_{83} & . & . & . & m_{88} & \end{bmatrix}$$

value

m_{11}	m_{51}	m_{22}	m_{72}	m_{53}	m_{83}	m_{35}	m_{64}	m_{16}	m_{56}	m_{77}	m_{88}	m_{28}

row_ind

1	5	2	7	5	8	3	6	1	5	7	8	2

col_ptr

1	3	5	7	7	9	11	13	14

Figure 3.5: CSC Data Structure for Sparse Matrix

For any column j, the $col\_ptr(l)$ indicates the row index of first non-zero entry of that column. Row indices of column l can be calculated in between $row\_ind[col\_ptr[l]]$ and $row\_ind[col\_ptr[l+1]-1]$. It takes same memory space of $2nnz+m+1$, similar to CRS. The figure represent the CSC data structure for $M^{k\times l}$.

3.1.5 Cache Complexity of Sparse Matrix access

The computational complexity of sparse matrix operations depends on memory size, memory traffic and the organization of cache memory. Cache memories minimize the data movement between main memory and Central Processing Unit (CPU) to increase efficiency. To hide the memory-processor speed gap, one approach is using massive multithreaded architectures [7]. However, these architectures have limited availability at present and only two levels of memory are popularly considered in the I/O model. Enormous amounts of data are contained in a memory called *Disks* which are workhorse storage devices [6]. The pattern of data access in the hierarchical memory computing system is referred to as locality of reference. The concept of data locality states that recently accessed (temporal) and sequential data (spatial) are likely to be accessed in near future.

Because of structural pattern and floating point operation in memory accessing, cache performance analysis is very important for sparse matrix computations. In a modern computer, the performance of spatial locality in accessing the matrix is influenced because of cachebased architectures. The performance of cache memory depends on the ratio of miss and hit rate.

Cache memory saves frequently used data in memory according to level. When CPU searched for particular data, cache memory starts searching in a level-wise manner. If the data is found in cache memory a *cache hit* occurs, if not then it is treated as *cache miss*. In case of a miss, the memory block containing the needed data is fetched from the slow memory to fast memory. The I/O complexity of an algorithm can be roughly defined as the number of memory transfers it makes between the *Cache* and *Disk* memories [4].

Bryant et.al [6] described in their book that the better spatial locality reduces cache misses and improves the efficiency of cache memory.

From above discussion and literature review, we understand that if the elements of a sparse matrix are accessed in arbitrary order for each row, then the number of cache misses can be as high as $O(nnz)$. Therefore to achieve full spatial locality in accessing matrix, the elements in each row in CRS (or column in CCS) of a sparse matrix are accessed in the order they are stored in their data structure. As a result, only those misses will occur which are caused by the first reference to a location in empty cache memory.

3.2 Detection of Missing Elements with Initial Guess Pattern S_0

In this section, we will identify missing non-zero elements of Jacobian matrix based on an initial guess pattern where function code is available as a black box. To describe our thesis work we have taken one symmetric initial guess pattern S_0 and the symmetric pattern of all possible $flaw$ have denoted as $flaw\ matrix$.

Here we have listed the overall steps for determination:

- Assume an initial guess pattern S_0 according to the size of our true matrix.
- Compute a structurally orthogonal partition ψ based on guess pattern S_0
- Define the direction matrix S for CPR implementation based on structurally orthogonal partition ψ
- Compute the compressed matrix Y using CPR Algorithm 1
- Detect the possible flaw locations using value symmetry
- Identify the flaw matrix P using pattern symmetry
- Add the flaw matrix P with initial sparsity pattern S_0

3.2.1 Determination of Sparsity Pattern with Good Guess Pattern

We have considered in our thesis two true matrix patterns, M^* and M, to demonstrate the effects of flaws. Both of them are in equal size with different pattern. To describe the process, let us take a small test matrix M^* of size 9×9. M^* is Jacobian matrix which is symmetric and whose pattern is unknown. As described before, we have taken tri-diagonal band matrix as an initial guess pattern S_0 with size 9×9 and our original matrix pattern is denoted by M^*.

$$M^* = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$S_0 = \begin{bmatrix} 1 & 1 & . & . & . & . & . & . & . \\ 1 & 1 & 1 & . & . & . & . & . & . \\ . & 1 & 1 & 1 & . & . & . & . & . \\ . & . & 1 & 1 & 1 & . & . & . & . \\ . & . & . & 1 & 1 & 1 & . & . & . \\ . & . & . & . & 1 & 1 & 1 & . & . \\ . & . & . & . & . & 1 & 1 & 1 & . \\ . & . & . & . & . & . & 1 & 1 & 1 \\ . & . & . & . & . & . & . & 1 & 1 \end{bmatrix}$$

There are at most 3 non-zero elements in any row of our guess pattern. From the definition of structurally orthogonal column partitioning this initial pattern needs to be clustered into at least 3 groups. For this guess pattern, the columns can be easily clustered into three groups. We can keep columns 1,4,7 in color group 1, columns 2,5,8 in color group 2 and columns 3,6,9 in color group 3. In our implementation, we have partitioned S_0 into column groups ψ using graph coloring algorithm implemented in DSJM [28] software.

A structurally orthogonal column partition ψ of guessed pattern S_0 is represented below:

1	2	3	1	2	3	1	2	3

From structurally orthogonal column partition ψ we can define S as follows:

$$S(j,k) = \begin{cases} \beta_j \neq 0, & \text{if } \psi(j) = k, \quad k = 1,2,\ldots,p \\ 0, & \text{otherwise} \end{cases}$$

Now we can represent our matrix S as:

$$S = \begin{bmatrix} \beta 1 & . & . \\ . & \beta 2 & . \\ . & . & \beta 3 \\ \beta 4 & . & . \\ . & \beta 5 & . \\ . & . & \beta 6 \\ \beta 7 & . & . \\ . & \beta 8 & . \\ . & . & \beta 9 \end{bmatrix}$$

From our guess pattern S_0, we can identify the specific array value for col_ind, row_ptr, ψ, group and *count* shown accordingly in table 3.1, 3.2, 3.3, 3.4 and 3.5.

Table 3.1: Col_ind of S_0

1	2	1	2	3	2	3	4	3	4	5	4	5	6	5	6	7	6	7	8	7	8	9	8	9

Table 3.2: Row_ptr of S_0

1	3	6	9	12	15	18	21	24	26

Table 3.3: ψ of S_0

1	2	3	1	2	3	1	2	3

Table 3.4: Color group for S_0

group 1	group 2	group3
1	2	3
4	5	6
7	8	9

Table 3.5: Number of Non-zero elements of each row in S_0

2	3	3	3	3	3	3	3	2

Since we have done column partitioning depending on only non-zero entries in a row, in many cases multiplication with structurally orthogonal group matrix results in a dense matrix. $M^* * S = Y$ in most cases will be compact(most of its entries are not identically zero). In this situation using a two dimensional array for Y is sensible. This will allow locating entry $m_{i,k}$ (or $\psi(k)m_{i,k}$) in Y efficiently.

Let $M, S \in \mathbb{R}^n$ where $m_1, m_2, \ldots, m_n$ columns of matrix M and columns are divided in structurally orthogonal color group as $M = \{m_j : j \in \psi\}$. To generate Y

$$Y = M * S \tag{3.1}$$

$$M * S = \sum_{j \in \psi} m_j * \beta_j \tag{3.2}$$

Since no pair of columns have a non zero entries in the same row, so

$$y_i = m_{i,j} * \beta_j \tag{3.3}$$

Now we will compute Y using algorithm 1. If there is any missing non-zero entry at any row in guess pattern S_0 then there will be overlapping or mismatch in at least one color group. And this perturbation can be observed in matrix Y. If there is no perturbation then we can say, our guess pattern represents the pattern of the true matrix. So pattern of Y matrix represents that how good our initial guess pattern is.

$$\underbrace{\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}}_{M^*} * \underbrace{\begin{bmatrix} \beta 1 & . & . \\ . & \beta 2 & . \\ . & . & \beta 3 \\ \beta 4 & . & . \\ . & \beta 5 & . \\ . & . & \beta 6 \\ \beta 7 & . & . \\ . & \beta 8 & . \\ . & . & \beta 9 \end{bmatrix}}_{S}$$

$$= \underbrace{\begin{bmatrix} \beta 1 & \beta 2 & . \\ \beta 1 & \beta 2 & \beta 3 \\ \beta 4 & \beta 2 & \beta 3 \\ \beta 4 & \beta 5 & \beta 3 \\ \beta 4 & \beta 5 & \beta 6 \\ \beta 7 & \beta 5 & \beta 6 \\ \beta 7 & \beta 8 & \beta 6 \\ \beta 7 & \beta 8 & \beta 9 \\ . & \beta 8 & \beta 9 \end{bmatrix}}_{Y}$$

Figure 3.6: Compressed Matrix Y for Good Guess Pattern

Figure 3.6 represents that there is no perturbation in Y, therefore our guess pattern was good and returns the true pattern.

3.2.2 Determination of Sparsity Pattern with Incorrect Guess S_0

Suppose, our original matrix M is same size as before but has a different pattern. If we take same initial guess pattern then there will be of course some overlapping and/or some entries will be missing.

$$M = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$S_0 = \begin{bmatrix} 1 & 1 & . & . & . & . & . & . & . \\ 1 & 1 & 1 & . & . & . & . & . & . \\ . & 1 & 1 & 1 & . & . & . & . & . \\ . & . & 1 & 1 & 1 & . & . & . & . \\ . & . & . & 1 & 1 & 1 & . & . & . \\ . & . & . & . & 1 & 1 & 1 & . & . \\ . & . & . & . & . & 1 & 1 & 1 & . \\ . & . & . & . & . & . & 1 & 1 & 1 \\ . & . & . & . & . & . & . & 1 & 1 \end{bmatrix}$$

Now we have to follow the same procedure as described before. In this case since dimension of original matrix is equal and guess pattern is same, the directional derivative S is also same as previous. For matrix M calculation of Y is given below

$$\underbrace{\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \alpha \\ 1 & 1 & 1 & 0 & 0 & \alpha & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & \alpha & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & \alpha & 0 & 0 & 1 & 1 & 1 \\ \alpha & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}}_{\text{M}} * \underbrace{\begin{bmatrix} \beta 1 & . & . \\ . & \beta 2 & . \\ . & . & \beta 3 \\ \beta 4 & . & . \\ . & \beta 5 & . \\ . & . & \beta 6 \\ \beta 7 & . & . \\ . & \beta 8 & . \\ . & . & \beta 9 \end{bmatrix}}_{\text{S}}$$

$$= \underbrace{\begin{bmatrix} \beta 1 & \beta 2 & \beta 9 * \alpha \\ \beta 1 & \beta 2 & \beta 3 + \beta 6 * \alpha \\ \beta 4 & \beta 2 & \beta 3 \\ \beta 4 & \beta 5 + \beta 8 * \alpha & \beta 3 \\ \beta 4 & \beta 5 & \beta 6 \\ \beta 7 & \beta 2 * \alpha + \beta 5 & \beta 6 \\ \beta 7 & \beta 8 & \beta 6 \\ \beta 4 * \alpha + \beta 7 & \beta 8 & \beta 9 \\ \beta 1 * \alpha & \beta 8 & \beta 9 \end{bmatrix}}_{\text{Y}}$$

Figure 3.7: Compressed Matrix Y for Incorrect Guess Pattern

Because of misplaced non-zero elements perturbation will take place. Now from this Y pattern we have to identify flaw position in the initial guess pattern $S_0 \neq M$ by using the value symmetry.

3.2.3 Determination of Flaw Location

This section provides a detailed description for identification of flaw location. To detect flaw location we will apply CPR algorithm on sparsity pattern M_0 and follow a set of rules to upgrade that pattern. M_0 is an approximation of M. By *approximation* we imply that few entries in our guess pattern will be wrong (i.e., zero labeled non-zero and non-zero labeled zero). Both types of wrong entries are considered as $flaws$.

Assume that there is a non-zero element at position (i,k), $M(i,k) \neq 0$ which has been mislabelled as 0 in the pattern S_0. As a result, product of M and S obtained via the CPR method will have some wrong entries or perturbation. In this situation two scenarios are possible:

i. $S_0(i,k) \neq 0$ and $\psi(k) = \psi(k')$ where $k \neq k'$ there exist an index

ii. $S_0(i,k) \neq 0$ and $\psi(k) = \psi(k')$ where $k \neq k'$ there does not exist an index.

Matrix Y is the result of the product, M and S, from which we have to identify characteristics of flaw. We select one position as flaw position in sparsity pattern M_0. We treat the following as *flawed* elements:

Remark 3.1. Elements which are missing in approximated pattern

Remark 3.2. Elements that do not satisfy value symmetry

Definition 3.3. Assume that application of CPR algorithm 1 on initial guess pattern S_0 results in structurally orthogonal color group ψ_k and the expansion matrix generated from Y is W . If $M_0(i,k) \neq W_0(i,k)$ then (i,k) and (k,i) both are treated as J flaw and denoted as f^J.

Definition 3.4. Assume that the application of CPR 1 on initial guess pattern S_0 results in matrix Y. If $y(i,l)$ has non-zero element, but $S_0(i,k) = 0$ where $k \in \psi(l)$ then $y(i,l)$ are treated as Y flaw and denoted as f^Y .

Remark 3.5. If $m_0(i,k') \neq m_0(k',i)$ then it violates the value symmetry. Two flaws would be counted at position (i,k) and (k,i) if $|m_0(i,k) - m_0(k,i)| > \eta_1$, where η_1 is a positive threshold.

Remark 3.6. If $m_0(i,k) = 0$ but $Y_0(i,l) \neq 0$ where $\psi(k) = l$. Flaw counted at position (i,l) of matrix Y_0 if $|Y_0(i,l)| > \varepsilon\eta_2$,where η_2 is a positive threshold.

M_0 is approximated guess pattern of M and i is row index for M_0, M, S_0 and Y. k represents column index for matrix M, M_0 and represent S_0 where l is column index for Y. ψ indicates color group and η_1,η_2 are considered as user adjustable tolerance. During floating point operations, η is defined as a positive threshold to account for errors. Now that we have identified the flaw and type (of flaw), the next step would be to identify the flawed entry, i.e, if at position (i,j) there is a flaw, $m_0(i,k) \neq m_0(k,i)$ which implies that there is a flawed entry. Since we cannot recognize which one is actually flawed, we have counted both of them as flawed entries.

Remark 3.7. $m_0(i,k) : m_0(i,l) \neq 0$ is not included in the current sparsity pattern of the symmetric matrix where color column $\psi(k) = \psi(l)$.

Remark 3.8. $m_0(k,i) : m_0(k,l) \neq 0$ is not included in the current sparsity pattern of the symmetric matrix where color column $\psi(i) = \psi(l)$.

Above discussion leads to algorithm 2

Algorithm 2 Flaw Detection of matrix M_0

```
1: for i =1 to n do
2:     $a = row\_ptr(i)$ to $row\_ptr(i+1) - 1$
3:     $m \leftarrow col\_ind(a)$
4:     $k \leftarrow \psi$
5:     $W_{i,j} \equiv Y_{i,k}/\psi$
6: end for
```

Here W is our expansion matrix. Now we can access non-zero elements using algorithm 2. We know that at which position $m(i,k)$ is mapped to positions in matrix $Y(i,\psi(l))$. After mapping for each row we can identify all J flaws and rest of them in matrix Y is marked as Y flaws.

The next step is the identification of flaw location. If there is a flaw at position (i,k) where $k \in \psi_m$ then it is possible that there can be flawed entries at any column of m color group in row i. These concepts lead to following remark:

Remark 3.9. If $m_0(i,k)$ is a J flaw and column k belongs to color group a then it is possible that there are missing one or more elements in row i. So that the set of possible position for missing elements are $(i,q) : q \in \psi_a$.

Remark 3.10. If $Y_0(i,l)$ is an Y flaw and column l belongs to color group a then it is possible that there are missing one or more elements in row i. So that the set of possible position for missing elements are $(i,q) : q \in \psi_a$.

Algorithm 2 gives us this equation 3.4 for determination of missing elements in J Since $Y,S \in \mathbb{R}^n \times k$ where $y_1, y_2, \ldots, y_n$ columns of matrix Y and $y_l : l \in \psi_m$(m= number of color group). Let $w_1, w_2, \ldots, w_n$ columns of matrix W and $w_k : k \in \psi_m$(m= number of color group). Then we can calculate our expanded matrix W using equation3.4

$$w(i,k) = y(i,l)/\psi(m) \tag{3.4}$$

Here $w(i)$ and $w(k)$ define row index and column index respectively in matrix W. And $w(i,k)$ indicate the position of non-zero elements in same matrix. Matrix Y and color group ψ_m are result of CPR algorithm. Application of above equation 3.4 on our computed Y and $\psi(m)$ gives below result:

$$\underset{Y}{\begin{bmatrix} \beta1 & \beta2 & \beta9*\alpha \\ \beta1 & \beta2 & \beta3+\beta6*\alpha \\ \beta4 & \beta2 & \beta3 \\ \beta4 & \beta5+\beta8*\alpha & \beta3 \\ \beta4 & \beta5 & \beta6 \\ \beta7 & \beta2*\alpha+\beta5 & \beta6 \\ \beta7 & \beta8 & \beta6 \\ \beta4*\alpha+\beta7 & \beta8 & \beta9 \\ \beta1*\alpha & \beta8 & \beta9 \end{bmatrix}} \quad \underset{S}{\begin{bmatrix} \beta1 & . & . \\ . & \beta2 & . \\ . & . & \beta3 \\ \beta4 & . & . \\ . & \beta5 & . \\ . & . & \beta6 \\ \beta7 & . & . \\ . & \beta8 & . \\ . & . & \beta9 \end{bmatrix}}$$

$$= \underset{W}{\begin{bmatrix} 1 & 1 & \alpha(\beta9/\beta3) & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1+\alpha(\beta6/\beta3) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1+\alpha(\beta8/\beta5) & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1+\alpha(\beta2/\beta5) & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1+\alpha(\beta4/\beta7) & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \alpha(\beta1/7) & 1 & 1 \end{bmatrix}}$$

Figure 3.8: Expanded Matrix W from the Compressed Matrix Y using $w(i,k) = y(i,l)/\psi(m)$

From the resulting matrix J we can clearly see overlapping in row 2,4,6,8 at column 3,5,5,7 accordingly. And in row 1 and 9 at column 3 and 7 respectively has flaw but there are no overlapping. So we can say that in our matrix *W* there are 4 *J* flaws and 2 *Y* flaws. Here we have represented 2 types flaw matrix and the color group they belong. f^J and f^Y denote *J* flaw and *Y* flaw correspondingly.

3.3 Data Structure for Flaws

The efficiency of the algorithm largely depends on the data structure of matrix storage and computation, especially for a sparse matrix. Column partitioning, flaw calculation, intersection and union operation in our implementation use CSR data structure. As we discussed previously that CSR(Compress Sparse Row) allow cache-friendly performance to minimize cache misses. Calculation of flaw and explanation of data structure of flaw for example matrix *M* is given below:

$$f^J = \begin{bmatrix} . & . & . & . & . & . & . & . & . \\ . & . & 3 & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & 2 & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & 2 & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . & . \end{bmatrix}$$

CSR Data Structure to store *J* flaw

Table 3.6: Row of J flaw

2	4	6	8

Table 3.7: Group index of J flaw

3	2	2	1

Table 3.8: Flaw_ptr of J flaw

1	2	3	4

$$f^Y = \begin{bmatrix} . & . & 3 \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \\ 1 & . & . \end{bmatrix}$$

CSR Data Structure to store *Y* flaw

Table 3.9: Row of Y flaw

1	9

Table 3.10: Group index of Y flaw

3	1

Table 3.11: Flaw_ptr of Y flaw

1	2

- Since we discuss previously that if there is a flaw at position (i,k) then we consider both (i,k) and (k,i) as a flawed location since we do not know which of these two element is flawed.

- And we also mentioned in 3.9, 3.10 that, if k is a member of m color group then any element of m group at row i can be flawed.

Considering above conditions we can generate possible flaw matrix by performing the union operation.

Remark 3.11. p=$f^J \bigcup f^Y$

$$p(i,k) = \left\{ \begin{array}{cc} 0 & \text{if } f^J(i,k)\text{=} f^Y(i,k) = 0 \\ & \text{where } i,k = 1,2,\ldots,n \\ 1 & \text{otherwise} \end{array} \right\}$$

$$\begin{bmatrix} . & . & * & . & . & * & . & . & * \\ . & . & * & . & . & * & . & . & * \\ . & . & . & . & . & . & . & . & . \\ . & * & . & . & * & . & . & * & . \\ . & . & . & . & . & . & . & . & . \\ . & * & . & . & * & . & . & * & . \\ . & . & . & . & . & . & . & . & . \\ * & . & . & * & . & . & * & . & . \\ * & . & . & * & . & . & * & . & . \end{bmatrix}$$

p=f$^J \bigcup f^Y$

CSR Data Structure to store flaw matrix

Table 3.12: Column of possible flaw in $p = f^J \bigcup f^Y$

3	6	9	3	6	9	2	5	8	2	5	8	1	4	7	1	4	7

Table 3.13: Flaw_ptr of possible flaw in $p = f^J \bigcup f^Y$

1	4	7	10	13	16	19

From f^J and f^Y we can know that we have only 6 elements missing but in flaw matrix we can see that 18 elements are detected as flawed. So there are lots of extra *flawed* elements. To determine the actual flaw we have to apply algorithm 3.

Algorithm 3 Determination of flaw locations

```
Input: $S_0$,$\psi_m$ and $Y_l$,
Output:Set of flaw locations
1: for each pair of $M_0(i,j) \neq M_0(j,i)$ do
2:     if $j >= i$ then
3:         if $|M_0(i,j)$-$M_0(j,i)| > \gamma_1$ then
4:             if $i \neq j$ then
5:                 add (i,j) and (j,i )to the set of $J$ flaw
6:             end if
7:             if $i = j$ then
8:                 add (i,i)to the set of $J$ flaw
9:             end if
10:        end if
11:    end if
12: end for
13: for each pair of $Y_0(i,k) \neq 0$ do
14:     for each pair of $M_0(i,j) = 0$ do
15:         if $\psi(j) = k$ then
16:             if $|Y_0(i,k)| > \varepsilon * \gamma_2$ then
17:                 add (i,j) to the set of $Y$ flaw
18:             end if
19:         end if
20:     end for
21: end for
```

3.3.1 Asymptotic Analysis of Determination of Flaw Location

In algorithm 3 J flaw calculation is done within line 1 to 12 which executes one time for level 0. So if it required k levels then the time complexity will be $O(nk)$ for each addition, $i = i$ and $i \neq j$. Y flaw calculation is done within line 13 to 20 which requires also $O(nk)$.As a result, the total computational complexity will be (1 to 12) $O(nk)$.

3.3.2 Actual Flaw Identification using Value Symmetry

Since our Jacobian matrix is symmetric, we can take advantage of symmetric properties. By taking intersection between p and p^T we can discard those flawed elements which do not satisfy value symmetry.

$$\underset{p}{\begin{bmatrix} . & . & \ast & . & . & \ast & . & . & \ast \\ . & . & \ast & . & . & \ast & . & . & \ast \\ . & . & . & . & . & . & . & . & . \\ . & \ast & . & . & \ast & . & . & \ast & . \\ . & . & . & . & . & . & . & . & . \\ . & \ast & . & . & \ast & . & . & \ast & . \\ . & . & . & . & . & . & . & . & . \\ \ast & . & . & \ast & . & . & \ast & . & . \\ \ast & . & . & \ast & . & . & \ast & . & . \end{bmatrix}} \longrightarrow \underset{p^T}{\begin{bmatrix} . & . & . & . & . & . & . & \ast & \ast \\ . & . & . & \ast & . & \ast & . & . & . \\ \ast & \ast & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & \ast & \ast \\ . & . & . & \ast & . & \ast & . & . & . \\ \ast & \ast & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & \ast & \ast \\ . & . & . & \ast & . & \ast & . & . & . \\ \ast & \ast & . & . & . & . & . & . & . \end{bmatrix}}$$

Remark 3.12. P=$p \bigcap p^T$

$$p(i,k) = \left\{ \begin{array}{ll} 1 & \text{if } f^J(i,k)\text{=}\ f^Y(i,k)\text{=}1 \\ & \text{where } i,k = 1,2,\ldots,n \\ 0 & \text{otherwise} \end{array} \right\}$$

$$\underset{p}{\begin{bmatrix} . & . & * & . & . & * & . & . & * \\ . & . & * & . & . & * & . & . & * \\ . & . & . & . & . & . & . & . & . \\ . & * & . & . & * & . & . & * & . \\ . & . & . & . & . & . & . & . & . \\ . & * & . & . & * & . & . & * & . \\ . & . & . & . & . & . & . & . & . \\ * & . & . & * & . & . & * & . & . \\ * & . & . & * & . & . & * & . & . \end{bmatrix}} \bigcap \underset{p^T}{\begin{bmatrix} . & . & . & . & . & . & . & * & * \\ . & . & . & * & . & * & . & . & . \\ * & * & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & * & * \\ . & . & . & * & . & * & . & . & . \\ * & * & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & * & * \\ . & . & . & * & . & * & . & . & . \\ * & * & . & . & . & . & . & . & . \end{bmatrix}}$$

$$= \underset{P = p \bigcap p^T}{\begin{bmatrix} . & . & . & . & . & . & . & . & * \\ . & . & . & . & . & * & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & * & . \\ . & . & . & . & . & . & . & . & . \\ . & * & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & * & . & . & . & . & . \\ * & . & . & . & . & . & . & . & . \end{bmatrix}}$$

Figure 3.9: Location of $flaws$ in Approximated Pattern

After calculation $P = p \bigcap p^T$ we have identified three pairs flaw (1,9)(9,1), (2,6)(6,2) and (3,8)(8,3). It is not mandatory that after every calculation it will find all pair of flaws. As there are only 3 pairs flaws here, it is identified easily the first time.

Now add this flaw matrix P with our initial guess pattern S_0 to get a new pattern. Figure 3.10 represents the upgraded sparsity pattern S_1

$$\underset{S_0}{\begin{bmatrix} 1 & 1 & . & . & . & . & . & . & . \\ 1 & 1 & 1 & . & . & . & . & . & . \\ . & 1 & 1 & 1 & . & . & . & . & . \\ . & . & 1 & 1 & 1 & . & . & . & . \\ . & . & . & 1 & 1 & 1 & . & . & . \\ . & . & . & . & 1 & 1 & 1 & . & . \\ . & . & . & . & . & 1 & 1 & 1 & . \\ . & . & . & . & . & . & 1 & 1 & 1 \\ . & . & . & . & . & . & . & 1 & 1 \end{bmatrix}} \bigcup \underset{P}{\begin{bmatrix} . & . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . & . \\ 1 & . & . & . & . & . & . & . & . \end{bmatrix}}$$

$$= \underset{S_1 = S_0 \bigcup P}{\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}}$$

Figure 3.10: Upgraded Sparsity Pattern S_1

We can repeat this whole process until that the number of flaws is zero or our flaw matrix is empty.

3.3.3 Algorithm for Determination of Sparsity Pattern for Good Initial Pattern

Algorithm 4 Determination of sparsity pattern for good initial pattern

```
Input: A symmetric trial pattern $S_0$ and $n_{estimate}$
Output:The final true pattern
 1: A $tri\_Diagonal$ band matrix $S_0$ with size n
 2: Calculate column partitioning $\psi$ using DSJM for $S_0$
 3: Generate matrix $Y_0$ using $S_0, \psi, S$ {using CPR Algorithm 1}
 4: Identify the flaw location $F_0$ and count $n_{estimate} = nnz(P_0)$ {using Algorithm 3}
 5: Compute Flaw Matrix $P = (p \bigcap p^T)$ ;
 6: $S_1 = S_0 \bigcup P$
 7: Calculate column partitioning $\beta_0$ using DSJM for $S_0$
 8: Generate Matrix $Y_0$ using $S_1, \psi, S$ {using CPR Algorithm 1}
 9: Exclude spurious elements from $S_0$ {using Algorithm 4}
10: return The final pattern
```

3.3.4 Limitation of Algorithm 4

Algorithm 4 is able to detect sparsity pattern of a small dataset 80×80 in size, even without good guess. For a big set of matrix-like 600×600 in size, the same algorithm can detect the result for a good guess. By a 'good guess' we mean that very few elements are missing in the initial guess pattern S_0. According to Carter if $nnz(M - S_0)$ is sufficiently modest then it is possible to handle spurious elements. Algorithm 4 detect sparsity pattern if $nnz(M - S_0) << n$ and number of flaws is small.

3.3.5 Elimination of Spurious Elements

Carter described in his paper [9] that most of the time there are spurious element besides exact flaws. In the computation of flaw matrix for a large-scale dataset, more than one flaw in each row is a very common pattern. As a result, spurious elements sometimes take place in flaw matrix. In this section, we will discuss the process of abolition of extra flaw. The process we are going to follow mostly resembles Carter's procedure. To remove spurious elements which should be zero we will follow procedure 4. In this algorithm, ε_0 is a positive threshold to find spurious entities in S_0. In our implementation, we take ε_0 equal to be 10^{-6}. We can replace procedure 4 with 5 by a relative test if desired.

Procedure 4: Exclude spurious entities from S_0

```
Output:The final pattern of J
1:  Set the tolerance $\varepsilon > 0$
2:  Set $S_e = \psi$
3:  for every pair (i,j)$\in S_1$ do
4:      if $j > i$ then
5:          if both $m_0(i,j)$ and $m_0(j,i)$ are effectively zero
            $max(|m_0(i,j), m_0(j,i)|) < \varepsilon_0$ then
6:              Add pair $(i,j)$ and $(j,i)$ in $S_e$
7:          end if
8:      end if
9:  end for
10: Remove all elements in $S_e$ from $S_1$ and $M_0$
11: exit
```

Procedure 5: Another Approach to Eliminate spurious entities from S_0

```
Output:The final pattern of J
1:  Set the tolerance $\varepsilon > 0$
2:  for every pair (i,j)$\in M_0$ do
3:      if $j > i$ then
4:          if both $m_0 i, j > \varepsilon_0$ then
5:              Set $S_0(i,j) = 1, M_0(i,j) = 1$
6:          end if
            else
7:          Set $S_0(i,j) = 0, M(i,j) = 0$
8:      end if
9:  end for
```

Chapter 4

Detection of Jacobian Matrix Sparsity Pattern

In this chapter, we will discuss the heuristic partitioning and ordering algorithm of DSJM. It will be followed by the description of Smallest Last Ordering method and Sequential Coloring algorithm of DSJM. Then detail description of Multilevel Algorithm will be presented. Finally, parallel implementation will be discussed, which was used to speed up execution.

4.1 Description about DSJM ToolKit

We have used DSJM toolkit to partition columns of the sparse matrix into a structurally orthogonal color group. DSJM uses a cache friendly data structure (Compress Row, Compress Column or Bucket Heap Sort) for ordering and coloring algorithm which minimizes cache misses. As a result, DSJM can provide better coloring, faster than existing implementations [22]. Two kinds of partitioning algorithms are used in DSJM for direct determination of sparse Jacobian matrices. In the first kind, columns are scanned in either Largest First Ordering (LFO), or Smallest Last Ordering (SLO) or Indence Degree Ordering (IDO), and then partitioning is computed with a minimal number of colors group. Column ordering and partitioning i.e, Saturation Degree (SD) partitioning and Recursive Largest First (RLF) partitioning are done simultaneously. In our thesis, we will take advantage of Smallest Last Ordering method and Sequential Coloring algorithm of DSJM, for structurally orthogonal column partitioning.

4.1.1 Smallest-Last Ordering

SLO appears to provide better result for preprocessing column [17, 12, 10]. Suppose we have ordered set of vertices $V' = v_n, v_{n-1}, \ldots, v_{i+1}$ of graph G. SLO selects i^th vertex u to put into ordered set of vertices from unordered set such that $deg(u)$ is minimum in $G[V \setminus V']$. Here $G[V \setminus V']$ is a graph which is obtained from G by removing the vertices of set V' from V. Major computational steps of Smallest Last Ordering procedure is given below:

Procedure 6: Smallest Last ordering

```
Input: array $D_n$, to store degree information $1 \rightarrow n$
Output: array of ordered columns $O_n$, $1 \rightarrow n$
1: set $D_n \leftarrow n$
2: set $\tau \leftarrow 1, \ldots, n$
3: while $\tau \neq \phi$ do
4:     Select $i \in \tau$ such that $d_g(k) \in G[V \setminus V']$ is minimum
5:     $O_n(D_n) \leftarrow k$
6:     $D_n \leftarrow D_n - 1$
7:     $\tau \leftarrow \tau \setminus k$
8: end while
9: return array of ordered columns,$O_n$
```

The largest computational cost of this algorithm is column deletion of the minimum degree holder. It requires $O(\log i)$ cost to find the column i with $min\ d_g(i) \in G[V \setminus V']$. So for n columns the computational cost will be $O(\sum_{i=1}^{n} \log i)$ which is $O(n \log n)$.

If there are ρ_i number of non-zero elements in each row i, then computational cost for non-zero entries will be $O(\sum_{i|m_{(i,j)} \neq 0} \rho_i)$, and for all column will be $O(\sum_{i=1}^{n} \sum_{i|m_{(i,j)} \neq 0} \rho_i)$. For ρ_i number of non-zero elements each row will be searched ρ_i^2 times to find minimal number of groups. For N number of rows, this algorithm will take $O(\sum_{i=1}^{n} \sum_{i|m_{(i,j)} \neq 0} \rho_i)$ which is proportional to $O(\sum_{i=1}^{n} \rho_i^2)$. So the total computational complexity is $O(\sum_{i=1}^{n} \rho_i^2)$ [22].

4.1.2 Sequential Coloring Method

According to Coleman and Moré [11] the problem of finding a minimum number of column partitioning is equivalent to a vertex coloring problem of an associated graph and that problem is NP-Hard. According to researchers, the greedy algorithm defines the minimal number of color groups when the columns are in specific order [23, 26, 24, 25]. Sequential coloring method is a greedy coloring algorithm which is a variant of CPR partitioning. Before implementing this algorithm, columns should be ordered in either Largest First Ordering (LFO), Smallest Last Ordering (SLO) or Indence Degree Ordering (IDO). Sequential coloring procedure 5 is given below:

Algorithm 5 Sequential Coloring algorithm

```
Input: Order array $O_n$ for non-zero entries $i \rightarrow n$
Output: Color array $\psi_m$ of color columns $i \rightarrow m$
 1: $\psi \leftarrow 0$
 2: $\tau \leftarrow 1, 2, \ldots, n$
 3: for $i = 1$ to $n$ do
 4:     $k \leftarrow Order(i)$
 5:     let $c_m$= min $\{c | c \in i, \ldots, \psi + 1 \neq C_{grp}(l), l \in C_g(k)\}$
 6:     $C_{grp(k)} \leftarrow c_m$
 7:     if $(c_m > C_{grp})$ then
 8:         $\psi \leftarrow \psi + 1$
 9:     end if
10: end for
11: return  Color array of color columns
```

Input for the sequential algorithm is the sparsity pattern of the matrix, with n number of columns. For scanning, an array will be maintained called *order*. $C_g(k)$ represents the set of neighbors column i which cannot be within the same group as column i stayed. c_m represents the least number of the structurally orthogonal group to which column j can be added. ψ contains the total number of structurally orthogonal groups in resulting partition. It takes $O(n)$ steps to find N columns, which is compared with the neighbour (non-zero entries) to find the optimal number of columns.

In each row i if there are ρ_i number of non-zero elements then computational cost for non-zero entries is $O(\sum_{i|m_(i,j) \neq 0} \rho_i)$, and for all columns it will be $O(\sum_{i=1}^{N} \sum_{i|m_{(i,j)} \neq 0} \rho_i)$. For ρ_i number of non-zero elements, each row will be searched ρ_i^2 times in total to find minimal number of groups. So for N number of rows this algorithm will take $O(\sum_{i=1}^{N} \rho_i^2)$ operations which is proportional to $O(\sum_{i=1}^{N} \sum_{i|m_{(i,j)} \neq 0} \rho_i)$. Time complexity for sequential algorithm is $O(\sum_{i=1}^{N} \rho_i^2)$ [22].

4.2 The Multilevel Algorithm

As discussed about the limitation of algorithm 4 (in the previous chapter), the single compilation is not enough for detection of all missing elements, specially for large-scale data set. To overcome this problem, the following steps would be followed:

- Instead of using S_0 for first computation to get augmented pattern $S_0 \bigcup (P \bigcap P^T)$, we can take $S_1 = S_0 + R_1$ as the initial guess pattern.
- For column partition, we can take help from DSJM toolkit.
- Determine the set of flaws for each pattern. Then compute intersection among all such flaws to find the location of missing elements and finally generate the flaw matrix P_1.
- Augment the guess pattern S_0 with the matrix P_1 (which have been computed) to get new pattern S_1.
- Implement the whole process on a serial computer.

In this case, R_1 is one kind of symmetric pattern whose main purpose is making the initial guess pattern a little different than previous, rather than improving it [9].

Since we have taken R_1 randomly so that it has a randomized effect on the location of the spurious element. As a result $P_1^T \bigcap P_1$ is significantly different than $P_0^T \bigcap P_0$. If we consider $P = (P_0^T \bigcap P_0) \bigcap (P_1^T \bigcap P_1) \ldots \bigcap (P_l^T \bigcap P_l)$ then the number of spurious elements will be reduced significantly. In the absence of an initial pattern S_0 random sparsity pattern, R_0

can be used as the guess pattern. In this case, the augmented pattern will be populated. As a result implementation of CPR algorithm will provide no benefit for this heavily populated matrix.

Our main target is minimizing the gradient evaluation cost so that:

- Wise selection of random sparsity pattern is done, where the modest number of symmetric pattern is appointed. If the bandwidth of this pattern is higher than the augmented pattern S_1 will be crowded. On the other hand, if it is low, then it will require more time iterations for flaw calculation. In both cases, the cost of gradient evaluation will increase. Therefore to reduce the number of gradient evaluation, a modest number of the symmetric pattern should be implemented.

- Use threshold values to avoid generating dense patterns. It is a pretty simplistic approach where flaw computation will terminate if the number of possible flaw locations equals or greater than the predetermined threshold. Once terminated, the algorithm will restart with a different set of randomly generated sparsity pattern R_k.

This process will continue as long as a manageable number of possibilities are obtained to generate a new augmented pattern $M_l | l \in \mathbb{N}$. To generate a reduced number of possibilities multiple iterations are required. Each process of this algorithm is called *level* and the repetition of this level is denoted as *Multilevel Algorithm*. For determination of sparsity pattern of the large-scale data, we need to implemented the multilevel algorithm. It is worth mentioning that this technique draws heavily from Carter's approach [9]. Implementation of this multilevel algorithm generates far better results even for a bad or non-existing guess pattern.

However, execution of this algorithm will require large space to store the intermediate patterns in the calculation at level l. Fortunately, it is not needed to store an immediate pattern of flaws P_i nor the pattern for P_l

$$P = (P_0^T \bigcap P_0) \bigcap (P_1^T \bigcap P_1) \ldots \bigcap (P_l^T \bigcap P_l)$$

The calculation of corresponding flaws gives an augmented pattern for addition with initial guess pattern S_0. For any reason, if that augmented flaw matrix becomes overpopulated then the algorithm terminates for that level. Finally, after the removal of spurious flaw entries, true pattern of the Jacobian matrix will be revealed. Application of multilevel algorithm is better for large dimensional matrix rather than smaller ones.

4.2.1 Voting between Levels

We can find flaw locations by comparing individual elements of trial matrix S_l in a level wise manner. In the multilevel algorithm flaw elements can be detected by using voting scheme [9]. We store $Y(i,l)$ where $\psi(i) = l$ for $i = 1,2,\ldots,n$ at every step of the multilevel algorithm.

If any diagonal element at position (i,i) is zero but was considered as non-zero element, in such cases voting scheme can help. The diagonal element at location (i,i) is zero or not, is decided by a distinct tolerance value which acts as a positive threshold. If most of the elements for $S_l(i,i)$, where $l = 0,1,\ldots,k$ hold the same value within that given threshold, then we can speculate that the value is correct for $J(i,i)$. Since voting scheme has no effect on Y flaws, we will calculate only J flaws using this.

This voting method can provide much better result as the required number of levels increase. To obtain the best efficiency, we have followed these condition:

- γ_i = 10^{-12} where $i = 1,2,3$.
- $\beta_i \neq \beta_k$ for all i,k we calculated β_i as β_i=$r_i * \theta_i$ where r_i is randomly selected number and θ_i= 10^{-12}

The process to calculate flaws locations using voting between levels is given in algorithm 7.

In this voting procedure 7, S_l is guess pattern at any level l and array ψ_l represents the column partitioning group of symmetric guess pattern S_l. Y_l is the compressed matrix

Procedure 7: Compute flaw locations using voting between levels

```
Input: $S_l$,$\psi_l$ and $Y_l$,
Output: P { computing J flaw}
1: for $i = 1$ to $n$ do
2:   if $|M_l(i,i) - M_{l-1}(i,i)| < \gamma_3 * (|M_l(i,i) - M_{l-1}(i,i)|)$ where l=0,...,k then
3:     for $l = 0$ to $k$ do
4:       if $|M_l(i,i)$-$M_l(i,i)| > \gamma_1$ then
5:         add (i,i) to the set of flaw J
6:       end if
7:     end forelse
8:     for $l = 0$ to $k$ do
9:       add (i,i) to the set of $J$ flaw
10:    end for
11:  end if
12: end for
13: for each pair of $M_l(i,k) \neq M_l(k,i)$ with $k >= i$ at level l do
14:   if $i \neq k$ then
15:     if $|M_l(i,k) - M_l(k,i)| > \gamma_1$ then
16:       add (i,k) to the set of $J$ flaw
17:     end if
18:   end if
19: end for
20: for each pair of $Y_l(i,k) \neq 0$ at level l do
21:   for each pair of $M_l(i,k) = 0$ do
22:     if $\psi(k) = l$ then
23:       if $|M_l(i,k)| > \gamma_2 * \varepsilon$ then
24:         add (i,k) to the set of $Y$ flaw
25:       end if
26:     end if
27:   end for
28: end for
```

obtained from CPR 1 and P is collection of all flaws locations within a matrix. Here γ_i represents some positive threshold where $i = 1,2,3$.

Two neighbour elements in diagonal position $M_l(i,i)$ and $M_{l-1}(i,i)$ are recognized as equal if and only if $|M_l(i,i)$ - $M_{l-1}(i,i)| < \gamma_3(|M_l(i,i)|$ + $|M_{l-1}(i,i)|)$. However, the following issue may arise. Suppose β_i=β_j for all i, j, then for equal non-zero values, computed location for any flawed element in J will be equal. Then it will be difficult to differentiate and identify. It clearly indicate that, since the value of non-zero and threshold constants affect the algorithm efficiency, we should be careful in selection of those values.

4.2.2 Asymptotic Analysis of Voting Scheme

J flaw calculation is done within line 1 to 12 which executes k times, k is the number of levels. The time complexity of voting at line 2 is $O(k)$ and adding (i,i) flaws from line 3 to 9 is also $O(k)$. As a result, the total computational complexity within line 1 to 12 is $O(nk^2)$.

For $i \neq k$ at line 13 require $O(np)$ time to execute J flaw within line 13 to 18. Since Y flaw has no effect of voting scheme still it required $O(np)$ time as we discussed in algorithm 3. So the complete time complexity for voting algorithm 7 is $O(nk^2) + O(np)$.

4.2.3 Finding all Possible flaws

Let us consider the following example for P_l generation. Suppose we identified flaw at position (i, j), at first we will check if this is within f_l^J i.e., if this is an element with position (i,m), where m is same group of column j. Next it be checked if the flaw is within f_l^Y - is this an element with position (i,k) where k is same group of column j. If any of these conditions are true than we will add this flaw in possible flaw matrix P_l. Same process will be followed for (j,i) position. When we identified one flaw in row position i for column group j, we will add flaws in each column of column group j at row position i, because any one element of that flawed group for this row can be the flaw. For representation and easy calculation we have maintained these arrays like [9], but for sorting elements we have used DSJM [22] toolkit rather than heap sort. The mentioned arrays are as follows:

- A sorted list of array for j flaws, to store(i,m)row and column indices of flaws accordingly.
- A sorted list of array for y flaws, to store(i,k)row and column indices of flaws accordingly.
- Array for row pointer to to point the first flaw of each row

Flaw matrix need not be constructed explicitly and also need not to be stored. We illustrate the full construction here for better understanding. Another task related to flaws calculation is identification of possible source of flaws. Procedure 8 represents this process:

Procedure 8: Compute possible source of flaws

```
Input: $f_l^j$ and $f_l^y$, $k_{max} \geq 2$
Output: Possible P matrix
1: Set $P = \psi$
2: for each (i,j) in $f_l^h$ do
3:     $kc = 0$;
4:     for each (i,k) where $k \in \psi(j)$ do
5:         if $k > i$ then
6:             $kc = kc + 1$;
7:             if $kc > k_{max}$ then
8:                 exceed threshold,Terminate immediately
9:             end ifelse
10:            add (i,k) and (k,i) to flaw matrix
11:        end if
12:    end for
13: end for
14: for each (i,j) in $f_l^y$ do
15:    $kc = 0$
16:    for each (i,k) where $k \in \psi(j)$ do
17:        if $k > i$ then
18:            $kc = kc + 1$
19:            if $kc > k_{max}$ then
20:                exceed threshold,Terminate immediately
21:            end ifelse
22:            add (i,k) and (k,i) to flaw matrix
23:        end if
24:    end for
25: end for
```

For procedure 8 we have set the value for k_{max} greater than or equal to 2. Carter [9] suggested value for k_{max} is 5 in his paper. As we described before, flaws in row i for the column k can be in any column of row i to which the k column belong. Using possible source of flaws and color group array we can compute set all possible flaw matrix. Algorithm 9 describe the member of flaw matrix detection process using structurally orthogonal color group ψ.

Procedure 9: Compute set of all possible flaws

```
Input: $Y_l$,$\psi$ and $P_l$,
Output: The set of possible flaws $P$
1: Set $P = \phi$
2: for each pair of flaw $(i,k)$ do
3:     for each entry of (i,q) where $q \in \psi(k)$ do
4:         if $l > i$ then
5:             add (i,q) and (q.i) to flaw matrix $P$
6:         end if
7:     end for
8: end for
```

4.2.4 Multilevel Algorithm to Determine the Sparsity Structure of Jacobian Matrix

In this section we will present Multilevel Algorithm 6 for detection of sparsity pattern of an unknown Jacobian matrix .This algorithm is a combination of CPR algorithm, DSJM toolkit, voting scheme and flaw computation.

To determine the number of missing elements for initial calculation, we have measured $n_{estimate}$ based on the initial guess pattern. Then we update the value of $n_{estimate}$ by the number of non-zero elements in flaw matrix P. We use a random number generator to produce pairs of integers $(i,j)_k, k = 1,2,\dots,m/2$ (with each integer between 1 to n) to generate a symmetric pattern R_k with m entries. And set R_k will be the union of these pairs, along with the union of $(i,j)_k, k = 1,2,\dots m/2$. The reason for taking random pattern R_k is to generate different guess patterns, so that $S_k = S_{k-1} \bigcup R_k$ at each level of computation. For better performance we should be careful in bandwidth selection for random sparsity pattern.

- The number of non-zero elements in S_k should be at least a small multiple of $n_{estimate}$, i.e. $nnz(S_k) \geq kc_1 * n_{estimate}$
- The number of non-zero elements in R_k should be at least a small multiple of n, i.e. $nnz(R_k) \geq kc_2 * n$

Algorithm 6 Multilevel Algorithm to determine the Sparsity Structure of the Jacobian Matrix

```
Input: Symmetric trial pattern $S_0$ and $n_{estimate}$
Output: The final true pattern
 1: A $tri-diagonal$ band matrix $S_0$ with size n
 2: Set $k = 0$
 3: if $(nnz(S_0) < k_1 * \eta_{estimation})$ then
 4:     Generate a random symmetric pattern $R_0$ where, $nnz(R_0) \geq k * n$
 5:     $S_0 = S_0 + R_0$
 6: end if
 7: Calculate column partitioning $\psi$ using DSJM for $S_0$
 8: Generate Matrix $Y_0$ using $S_0, \psi, S$ {using CPR Algorithm 1}
 9: Identify the flaw location $p$ and count $n_{estimate} = nnz(p)$
10: Compute Flaw Matrix $P_0 = (P_0 \bigcap P_0^T)$ ;
11: while $P_0/\alpha > n_{estimate}$ do
12:     $k = k + 1$
13:     Generate a random symmetric pattern $R_k$ where, $nnz(R_k) \geq k * n$
14:     $S_k = S_{k-1} \bigcup R_k$ where $nnz(R_k) + nnz(S_{k-1}) \geq k_1 * n_{estimate}$
15:     Calculate column partitioning $\psi_k$ using DSJM for $S_k$
16:     Generate Matrix $Y_k$ using $S_k, \psi_k, S$ {using CPR Algorithm 1}
17:     Identify the flaw location $P_k$
18:     Estimate the number of missing elements $n_{estimate} = nnz(P_k)$
19:     if $(k < 2)$ then
20:         go to step 11
21:     end if
22:     Compute the set of flaws $P = (P_0 \bigcap P_0^T \bigcap P_1 \bigcap P_1^T \bigcap \ldots \bigcap P_k \bigcap P_k^T)$
23: end while
24: Set $k = 0$
25: $S_1 = S_0 \bigcup P$
26: Calculate column partitioning $\psi_0$ using DSJM for $S_0$
27: Generate Matrix $Y_0$ using $S_0, \psi_0, S$ {using CPR Algorithm 1}
28: Identify the flaw location $P_0$
29: Estimate the number of missing elements $n_{estimate} = nnz(P_0)$;
30: Get the final pattern of J matrix by eliminating spurious entries from $S_0$ {using procedure 4}
31: if $(n_{estimate} = 0)$ then
32:     exit
33: end if
34: go to step 9;
35: return the final pattern
```

- The while loop should execute until the number of non-zero elements in flaw matrix pattern is significantly small than $n_{estimate}$, i.e. $nnz(p)/\alpha > n_{estimate}$. Where $kc_1 \geq 2$,

$kc_2 \geq 1$ and $\alpha \geq 5$.

- When there exist no initial guess pattern then steps (4-5) execute. This steps also compute if the number of non-zero elements in initial guess pattern is less then multiple of nnz and $n_{estimate}$, i.e. $nnz(S_l) \geq kc_1 * n_{estimate}$

To compute the set of flaws P we do not need to calculate or store the intermediate patterns P_k because P can be directly calculated from the flaws. Through our implementation, we use compressed matrix Y_l and sparsity structure of trial pattern S_l, instead of expanding our Jacobian M_l into full matrix at any level l.

4.2.5 Asymptotic Analysis of Multilevel Algorithm

In the multilevel algorithm, most computationally expensive operations are performed within the while loop iteration. As we discussed before, column partitioning using DSJM toolkit has time complexity $O(\sum_{i=1}^{N} \rho_i^2)$ [22]. CPR algorithm 1 requires $O(nnz)$ to compute matrix Y_k. The time complexity for voting scheme is $O(nk^2) + Onp)$ to compute the flaw locations. The time for eliminating spurious entries from S_0 in line 30 is $O(nnz)$ where nnz is the number of non-zero elements of matrix S_0 in algorithm 4.

From this analysis we can see that Multilevel algorithm improves efficiency in detection of sparsity pattern but it takes longer to execute. To reduce this time consumption we can take advantage of parallel computing techniques.

4.3 Parallel Implementation

Parallel implementation is used extensively in high performance computing environment to enhance performance of sequential code. For detection of sparsity pattern of an unknown Jacobian matrix we have chosen Multilevel algorithm. And to reduce the execution time of that algorithm we have added parallel concept.

Principles of Parallel Algorithm Design A parallel algorithm is a technique to solve a sequential task using multiple processors. To design parallel algorithm the following needs consideration [20]:

- Identify the segment of tasks for which concurrent operations are possible
- Vital mapping of possible concurrent tasks onto multiple processes running in parallel
- Appropriate distribution of input, output and intermediate data
- Careful management of shared data accessing
- Process synchronization during stages of parallel program execution

4.3.1 Multicore Technique

The term *multicore* is used for several core in the entire processor of a single machine [37]. For parallel implementation we have used multicore technique to make our implementation faster. Multicore technique refers to code executing on more than one core of single CPU chip at a time. This technique allows operating systems and applications to schedule multiple *threads* to logical processors of CPU as they perform on multiprocessor systems at a time. Instructions obtained from logical processors are executed simultaneously on shared resources. When multiple threads schedule simultaneously, as the resources are shared for execution,it is important to determine how and when to interleave the execution of the threads. The design of distribution affects *cachemisses* rate [5].

4.3.2 *OpenMP*: a standard for directive based parallel programming

For our parallel implementation, we have used *OpenMP*. *OpenMP* is an API for writing multithreaded code in C and C++ [20]. Concurrency, synchronization and data handling are provided by *OpenMP* directives. It is a standard for directive based parallel programming [29]. One of the advantage is that *OpenMP* can execute sequentially until directions for parallelism is provided. In C++ it regulates based on pragma compiler directives. The $omp\_get\_num\_threads()$ function returns the number of threads and $omp\_get\_thread\_num()$ function returns the identification number of each thread. *OpenMP* directives generates number of threads for parallel processing. Variables in an *OpenMP* parallel region is called *shared* if there exists one instance of this variable which is shared among all threads. And *private* variable in an *OpenMP* parallel region is local variable for each *thread*.We can assign *private* and *shared* variable in the directive for data handling.

4.3.3 Parallel Multilevel Algorithm

In this section we will present parallel implementation in Multilevel Algorithm 6. To design a model for parallel algorithm we have to apply mapping technique and appropriate strategy to minimize interaction. Since in parallel implementation, tasks are executed concurrently, it is expected that the execution speed will be doubled. Unfortunately this does not happen in practice due to inter-process communication, idling and excess computation [29]. Good serial algorithms need not be well suited for parallel implementation. In most parallel implementations, processes need to switch or swap data with other processes. This swapping affects the efficiency of parallel algorithm by introducing interaction delay. Moreover, good coding structure for serial implementation is not always perfect for parallel computing. To get best performance from parallel implementation, it is important to analyze algorithm, hardware platform and overhead. *Overhead* in parallel implementation indicates the total time required for processing elements over and above required for same sequential implementation. If sequential implementation requires T_s time and for the parallel if it is T_p then the overhead $T_o = T_p - T_s$. Speed up measures the ratio of the time for solving the same problem in sequential to parallel implementation. Efficiency of parallel implementation represents a fraction of time, in which processing elements get ready to be employed.

In parallel implementation we have followed the process outlined for multilevel algorithm, using the same set of constants and tolerance values. In addition, to improve efficiency few parallel constant have been applied in multilevel algorithm 6, which are given below:

- Parallel concept is applied only within inner loop of multilevel algorithm 6
- *private* and *shared* variables are defined at the declaration of *pragma*
- Used multiple *threads* for concurrent task execution

Parallel multilevel algorithm is presented in 7. In that algorithm, total number of *thread* is denoted as *nthread* and thread number is set using *tid* variable in algorithm 7. Symmetric trial pattern R_k is calculated as *private* variable in each *thread*. Number of iteration, initial guess pattern and flaw matrix are denoted respectively as k, S_0, and P, those are *shared* variables.

Algorithm 7 Parallel Multilevel Algorithm to determine the Sparsity Structure of Jacobian Matrix

```
Input: Symmetric trial pattern $S_0$ and $n_{estimate}$
Output: The final true pattern
 1: A $tri-diagonal$ band matrix $S_0$ with size n
 2: Set $k = 0$
 3: if $(nnz(S_0) < k_1 * \eta_{estimation})$ then
 4:     Generate a random symmetric pattern $R_0$ where, $nnz(R_0) \geq k * n$
 5:     $S_1 = S_0 + R_0$
 6: end if
 7: Calculate column partitioning $\psi$ using DSJM for $S_0$
 8: Generate Matrix $Y_0$ using $S_0, \psi, S$ {using CPR Algorithm 1}
 9: Identify the flaw location $F_0$ and count $n_{estimate} = nnz(P_0)$
10: Compute Flaw Matrix $P_0 = (P_0 \bigcap P_0^T)$ ;
11: while $P_0/\alpha > n_{estimate}$ do
12:     #pragma omp parallel private(nthreads, tid,$R_k$) shared(k,S,P)
13:     tid = omp_get_thread_num() { obtain thread number}
14:     nthreads = omp_get_num_threads() { only master thread does this }
15:     $k = k + 1$
16:     Generate a random symmetric pattern $R_k$ where, $nnz(R_k) \geq k * n$
17:     $S_k = S_{k-1} \bigcup R_k$ where $nnz(R_k) + nnz(S_{k-1}) \geq k_1 * n_{estimate}$
18:     Calculate column partitioning $\psi_k$ using DSJM for $S_k$
19:     Generate Matrix $Y_k$ using $S_k, \psi_k, S$ {using CPR Algorithm 1}
20:     Identify the flaw location $P_k$
21:     Estimate the number of missing elements $n_{estimate} = nnz(P_k)$
22:     if $(k < 2)$ then
23:         go to step 11
24:     end if
25:     Compute the set of flaws $P = (P_0 \bigcap P_0^T \bigcap P_1 \bigcap P_1^T \bigcap \ldots \bigcap P_k \bigcap P_k^T)$
26: end while
27: Set $k = 0$
28: $S_1 = S_0 \bigcup P$
29: Calculate column partitioning $\psi_0$ using DSJM for $S_0$
30: Generate Matrix $Y_0$ using $S_0, \psi_0, S$ {using CPR Algorithm 1}
31: Identify the flaw location $P_0$
32: Estimate the number of missing elements $n_{estimate} = nnz(P_0)$
33: Get the final pattern of J matrix by eliminating spurious entries from $S_0$ {using proce-
    dure 4}
34: if $(n_{estimate} = 0)$ then
35:     exit
36: end if
37: go to step 9;
38: return the final pattern
```

Chapter 5

Numerical Experiments

In this chapter, we provide numerical results of proposed algorithms and their application on some test instances. Detail description of test data set for our experiment is given in Section 5.1. Numerical environment to apply our algorithm for the large sparse matrix is described in section 5.2 and finally in section 5.3 the findings of our methodology is presented.

5.1 Test Data Sets

To evaluate our proposed approach we have collected and applied the proposed method on some test data sets. We have collected our data set from two verified sources. First data set is gathered from Matrix Market Collection [2] and second data set is obtained from University of Florida Matrix Collection [3], shown in table 5.2 and 5.3. In all tables, column labeled *Matrix* represent name of the matrix which we have used for our test procedure. The number of columns is equal to number of rows of a matrix as these are all square matrix, which is shown in columns labeled *n*. The column *nnz* (Number of Non-zero) is used to represent the total number of non-zero elements in that matrix.

5.2 Test Environment

We have done all our numerical experiments with test data sets in Table 5.2 and Table 5.3, varying the number of cores and using different computer configuration.

Table 5.1: Description of the machine for implementation

Processor	**Operating System**	**Cache (L2)**	**RAM**	**No. of Core**
Intel®Core^{TM}i5 6360 CPU @ 2.00GHz	macOS (High Seirra)	256 KB	8 GB	2 (Virtual - 4)
Intel®Core^{TM}i7 4770 CPU @ 3.40GHz	64 bit Linux	256KB	8 GB	4

5.3 Test Results

To increase performance we have chosen band matrices for initial guess pattern and used an efficient sparse data structure to handle sparse matrices for our purposed method. In this case, our proposed data structure can save space by at least fifty percent. Carter [9] used heap sort for most of the sorting. But we found that SLO (Smallest Last Ordering) sorting algorithm of DSJM [22] is more efficient for sorting. So in both sequential and parallel implementation we have utilized the benefits of DSJM [22] for sorting. For parallel implementation we have used multicore technique by using *thread* in *pragma* compiler directives. Parallel section can execute multiple tasks at the same time, which reduces the total execution time.

5.4 Numerical Experiments

We have represented our numerical results in table 5.2 by using data sets obtained from Matrix market collection [2]. For table 5.3 we have utilized data sets gathered from University of Florida Matrix Collection [3]. For both table 5.2, 5.3, ρ_{max} denotes the maximum number of non-zero elements in a row of that matrix. In our experiment, range of ρ_{max} is 3-218. The number of color groups in the structurally orthogonal partition of true Jacobian is listed under column *ncolor*. The column *ngeval* represent the total number of gradient evaluations required to detect the sparsity pattern for each test problem. Gradient evaluations from Sultana's thesis is presented in the column $nfeval$. Comparison of these two columns highlights the improvement attained by our implementation. For easier performance comparison, we have represented *ngeval* and *nfeval* column values in a graph. Graph 5.1 has been generated from table 5.2 and graph 5.2 from table 5.3 respectively. We have implemented our sequential implementation in two different environment which we have already described in Test Environment section in table 5.1. In table 5.4 and 5.5 execution time for sparsity detection of each matrix using Intel corei5 processor is listed in the column $corei5(execution\ time)$ and Intel corei7 processor is listed in the column labeled $corei7(execution\ time)$ respectively. Parallel implementation was done on the Intel corei7 machine and execution time using two *threads* is presented in the column *Parallel*. We have also examine the efficiency by using multiple processor in Intel corei7 computer. That experimental result for data set 1 is presented in table 5.6 and data set 2 is in table 5.7

Table 5.2: Matrix data set 1 - computational cost for sparsity detection and comparison with previous work

Matrix	**n**	**nnz**	ρ_{max}	**ncolor**	**ncpr**	**ngeval**	**nfeval**
bcspwr05	443	1,623	10	10	6	150	154
nos6	675	3,255	5	5	7	172	151
young1c	841	4,089	5	5	8	147	162
rdb1250	1,250	7,300	6	8	7	226	297
bcsstm12	1,473	19,659	22	25	7	304	713
lshp1561	1,561	10,681	7	8	7	209	368
plat1919	1,919	32,199	19	24	10	419	780

Continued on next page

Table 5.2 – continued from previous page

Matrix	n	nnz	ρ_{max}	ncolor	ncpr	ngeval	nfeval
rdb2048	2,048	12,032	6	10	7	232	273
orsreg_1	2,205	14,133	7	11	9	239	239
zenios	2,873	27,191	14	48	9	932	1,095
bcsstk21	3,600	26,600	9	14	9	354	535
e20r0000	4,241	131,556	62	69	14	1,613	2,516
fidapm09	4,683	95,053	37	45	12	806	1,567
mhd4800b	4,800	27,520	10	10	10	277	497
bcspwr10	5,300	21,842	14	14	11	267	329
s1rmt3m1	5,489	219,521	48	50	12	1,638	2,864
fidap018	5,773	69,335	18	22	14	792	1,277
fidap015	6,867	96,421	18	22	14	811	1,297
e30r1000	9,661	306,356	62	70	11	1,678	2,998
bcsstk17	10,974	428,650	150	150	10	2,774	3,580
bcsstk18	11,948	149,090	49	49	10	1,065	1,680

Continued on next page

Table 5.2 – continued from previous page

Matrix	**n**	**nnz**	ρ_{max}	**ncolor**	**ncpr**	**ngeval**	**nfeval**
bcsstk29	13,992	619,488	71	72	12	2,611	5,069
bcsstk25	15,439	252,241	59	59	16	1,145	2,950
e40r5000	17,281	553,956	62	70	12	1,925	3,380
bcsstk30	28,924	2,043,492	218	219	12	6,077	9,131

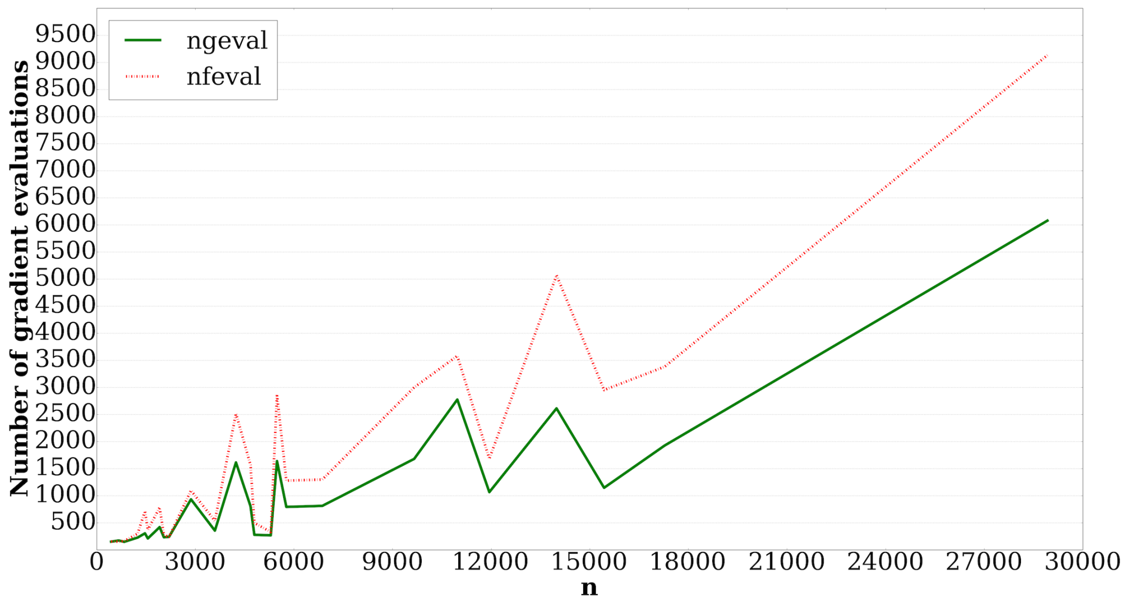

Figure 5.1: Graphical representation of comparison between *ngeval* and *nfeval* for Dataset 1

Table 5.3: Matrix data set 2 - Computational cost for sparsity detection and comparison with previous work

Matrix	**n**	**nnz**	ρ_{max}	**ncolor**	**ncpr**	**ngeval**	**nfeval**
662_bus	662	2,474	10	10	5	106	214
dwt_758	758	5,994	11	12	6	209	342
rdb800l	800	4,640	6	9	6	220	286
olm1000	1,000	4,994	6	6	6	112	241
jagmesh3	1,089	7,361	7	7	7	194	284
jagmesh5	1,180	7,750	7	8	8	197	246
jagmesh8	1,141	7,465	7	10	8	199	324
dwt_1242	1,242	10,426	12	15	10	247	409
lshp1270	1,270	8,668	7	8	7	204	256
jagmesh9	1,349	9,101	7	9	8	200	251
jagmesh4	1,440	9,504	7	9	8	197	263
bcspwr06	1,454	5,300	13	13	7	179	293
bcspwr08	1,624	6,050	14	14	14	178	451
filter2D	1,668	10,750	9	11	11	202	589

Continued on next page

Table 5.3 – continued from previous page

Matrix	**n**	**nnz**	ρ_{max}	**ncolor**	**ncpr**	**ngeval**	**nfeval**
ex_33	1,733	22,189	18	21	14	299	825
watt_1	1,856	11,488	7	12	5	234	259
G26	2,000	39,980	40	82	9	582	1,504
t2dal_a	4,257	37,465	9	12	12	255	894
nasa4704	4,704	104,756	42	47	15	629	1,805
EX5	6,545	295,680	120	192	16	1,015	3,205
Kuu	7,102	340,200	96	108	14	1,801	4,744
G65	8,000	32,000	19	20	12	179	318
delaunay_n13	8,192	49,049	12	14	10	269	578
aft01	8,205	125,567	21	25	11	817	1,367
nemeth02	9,506	394,808	52	52	13	1,664	3,739
wing_nodal	10,937	150,976	28	29	9	778	1,520
linverse	11,999	95,977	9	11	15	456	764
stokes64	12,546	140,034	11	20	12	582	1,083

Continued on next page

Table 5.3 – continued from previous page

Matrix	n	nnz	ρ_{max}	ncolor	ncpr	ngeval	nfeval
barth5	15,606	107,362	11	11	13	789	1,235
gyro_m	17,361	340,431	120	120	15	1,978	4,287
trefethen_20000b	19,999	554,435	84	84	16	3,345	4,110
t3dl_e	20,360	20,360	1	3	2	3	3
tube1	21,498	897,056	48	48	11	3,059	5,251

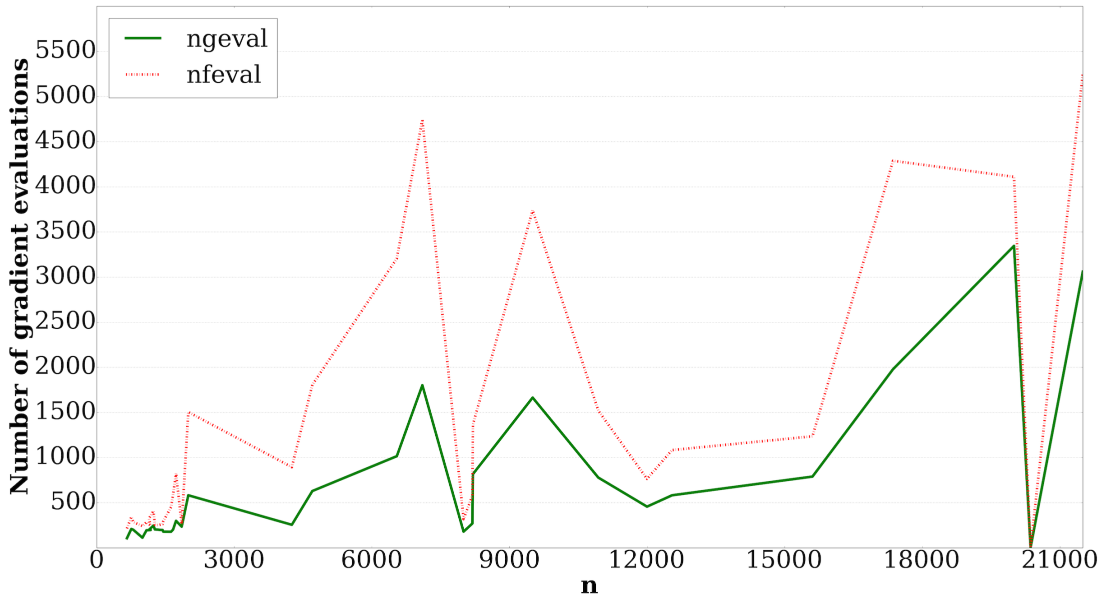

Figure 5.2: Graphical representation of comparison between *ngeval* and *nfeval* for Dataset 2

Table 5.4: Execution time for sparsity detection of data set 1 using corei5 and corei7 processors, with sequential and parallel implementation on corei7.

Matrix	**n**	**nnz**	**corei5(*execution time*)**	**corei7(*Execution time*)**	
				Sequential	Parallel
bcspwr05	443	1,623	0.34055	0.21	0.16
nos6	675	3,255	0.84878	0.6	0.41
young1c	8,41	4,089	1.65833	1.19	0.89
rdb1250	1,250	7,300	2.85592	1.57	1.4
bcsstm12	1,473	19,659	5.09057	2.42	2.01
lshp1561	1,561	10,681	5.39875	2.12	1.97
plat1919	1,919	32,199	9.38063	6.24	4.64
rdb2048	2,048	12,032	7.56246	4.217	1.21
orsreg_1	2,205	14,133	7.56246	4.68	3.02
zenios	2,873	27,191	10.1719	9.41	5.53
bcsstk21	3,600	26,600	71.1954	16.22	14.81
e20r0000	4,241	13,1556	127.171	83.59	54.14
fidapm09	4,683	95,053	98.822	69.38	43.68

Continued on next page

Table 5.4 – continued from previous page

Matrix	**n**	**nnz**	**corei5(*execution time*)**	**corei7(*Execution time*)**	
				Sequential	Parallel
mhd4800b	4,800	27,520	45.9701	22.55	15.34
bcspwr10	5,300	21,842	47.9037	22.38	12.04
s1rmt3m1	5,489	219,521	115.593	106.062	88.05
fidap018	5,773	69,335	146.796	94.6542	71.98
fidap015	6,867	96,421	153.124	146.167	78.37
e30r1000	9,661	306,356	429.054	381.537	189.75
bcsstk17	10,974	428,650	591.236	459.973	356.61
bcsstk18	11,948	149,090	548.3762	349.632	278.429
bcsstk29	13,992	619,488	694.377	374.191	301.375
bcsstk25	15,439	252,241	558.1949	398.42	261.45
e40r5000	17,281	553,956	894.1562	542.311	332.128
bcsstk30	28,924	204,3492	919.238	702.14	503.01

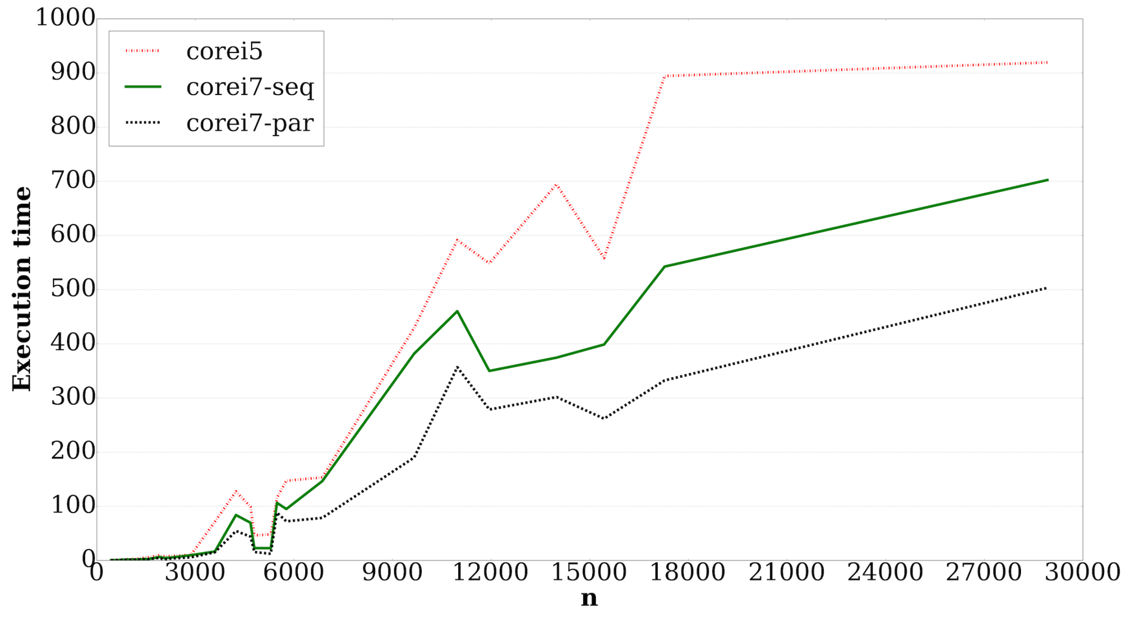

Figure 5.3: Execution time (for sparsity detection of matrices listed in data set 1) of sequential implementation on Intel corei5 and Intel corei7 processor and parallel implementation on Intel corei7 are presented respectively

Table 5.5: Execution time for sparsity detection of data set 2 using corei5 and corei7 processors, with sequential and parallel implementation on corei7.

Matrix	**n**	**nnz**	**corei5(*Execution time*)**	**corei7(*Execution time*)**	
				Sequential	Parallel
662_bus	662	2,474	0.73897	0.21	0.12
dwt_758	758	5,994	1.33236	1.28	0.76
rdb800l	800	4,640	1.11063	0.7	0.45

Continued on next page

Table 5.5 – continued from previous page

Matrix	**n**	**nnz**	**corei5(*execution time*)**	**corei7(*execution time*)**	
				Sequential	Parallel
olm1000	1,000	4,994	1.59536	1.15	0.93
jagmesh3	1,089	7,361	2.97602	2.09	1.56
jagmesh5	1,180	7,750	3.3905	2.09	1.87
jagmesh8	1,141	7,465	3.4183	2.73	1.86
dwt_1242	1,242	10,426	4.63796	2.44	1.65
lshp1270	1,270	8,668	4.25776	3.89	2.52
jagmesh9	1,349	9,101	4.89385	3.50	2.961
jagmesh4	1,440	9,504	4.73182	3.89	3.00
bcspwr06	1,454	5,300	2.63179	1.97	1.59
bcspwr08	1,624	6,050	3.25584	2.38	2.16
filter2D	1,668	10,750	7.58624	6.80	4.32
ex_33	1,733	22,189	9.01422	6.53	4.88
watt_1	1,856	11,488	10.41707	6.13	3.56

Continued on next page

Table 5.5 – continued from previous page

Matrix	**n**	**nnz**	**corei5**(*execution time*)	**corei7**(*execution time*)	
				Sequential	Parallel
G26	2,000	39,980	17.0781	11.57	9.14
t2dal_a	4,257	37,465	61.4602	46.1	27.56
nasa4704	4,704	10,4756	92.0456	69.24	51.0289
EX5	6,545	295,680	212.403	159.276	98.16
Kuu	7,102	340,200	249.929	197.556	140.15
G65	8,000	32,000	132.671	72.29	55.72
delaunay_n13	8,192	49,049	146.346	86.1	57.35
aft01	8,205	125,567	252.821	175.48	123.58
nemeth02	9,506	394,808	432.201	346.836	211.59
wing_nodal	10,937	150,976	541.256	351.11	269.86
linverse	11,999	95,977	517.349	317.33	252.29
stokes64	12,546	140,034	565.293	314.58	264.4
barth5	15,606	107,362	664.53	426.19	309.7

Continued on next page

Table 5.5 – continued from previous page

Matrix	n	nnz	corei5(*execution time*)	corei7(*execution time*)	
				Sequential	Parallel
gyro_m	17,361	340,431	745.450	575.18	306.37
trefethen_20000b	19,999	554,435	744.834	567.38	478.16
t3dl_e	20,360	20,360	584.4398	367.65	254.31
tube1	21,498	897,056	990.236	783.19	502.47

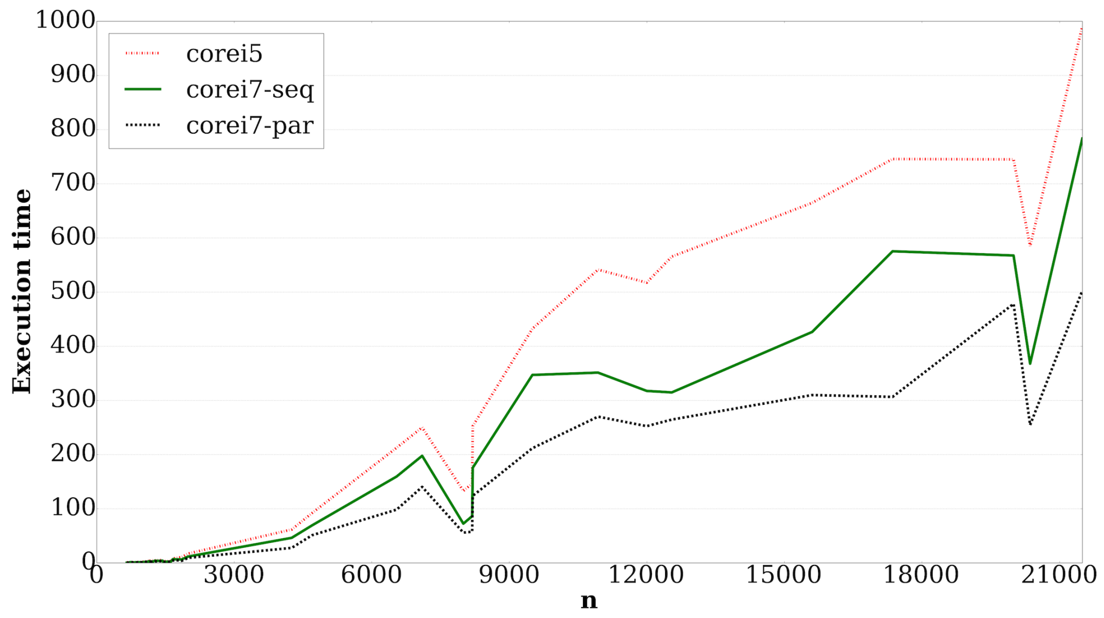

Figure 5.4: Execution time (for sparsity detection of matrices listed in data set 2) of sequential implementation on Intel corei5 and Intel corei7 processor and parallel implementation on Intel corei7 are presented respectively

Table 5.6: Comparison between *thread* $= 2$ and *thread* $= 4$, for sparsity pattern detection in data set 1 based on gradient evaluation and execution time.

Matrix	**n**	**nnz**	**ngeval**		**execution time**	
			thread=2	thread=4	thread=2	thread=4
bcspwr05	443	1623	150	102	0.16	0.19
nos6	675	3,255	172	172	0.41	0.57
young1c	1,138	4,054	147	146	0.89	0.96
rdb1250	1,250	7,300	226	194	1.4	1.55
bcsstm12	1,473	19,659	304	304	2.01	2.38
lshp1561	1,561	10,681	209	201	1.97	2.06
plat1919	1,919	32,199	419	400	4.64	5.48
rdb2048	2,048	12,032	232	207	1.21	2.56
orsreg_1	2,205	14,133	239	238	3.02	3.72
zenios	2,873	27,191	932	912	5.53	6.77
bcsstk21	3,600	26,600	354	275	14.81	14.90
e20r0000	4,241	131,556	1613	1567	54.14	76.1
fidapm09	4,683	95,053	806	805	43.68	68.81

Continued on next page

Table 5.6 – continued from previous page

Matrix	**n**	**nnz**	**ngeval**		**execution time**	
			thread=2	thread=4	thread=2	thread=4
mhd4800b	4,800	27,520	277	271	15.34	19.73
bcspwr10	5,300	21,842	267	268	12.04	16.09
s1rmt3m1	5,489	219,521	1,638	1,636	88.05	94.5
fidap018	5,773	69,335	792	506	71.98	75.23
fidap015	6,867	96,421	811	765	78.37	104.04
e30r1000	9,661	306,356	1,678	1,650	189.75	257.49
bcsstk17	10,974	428,650	2,774	2,709	356.61	400.25
bcsstk18	11,948	149,090	865	864	278.429	315.16
bcsstk29	13,992	619,488	2,611	2,545	301.375	375.80
bcsstk25	15,439	252,241	1,145	1,089	261.45	303.49
e40r5000	17,281	553,956	1,925	1,873	332.128	514.6
bcsstk30	28,924	204,3492	6,077	5,089	503.01	683.75

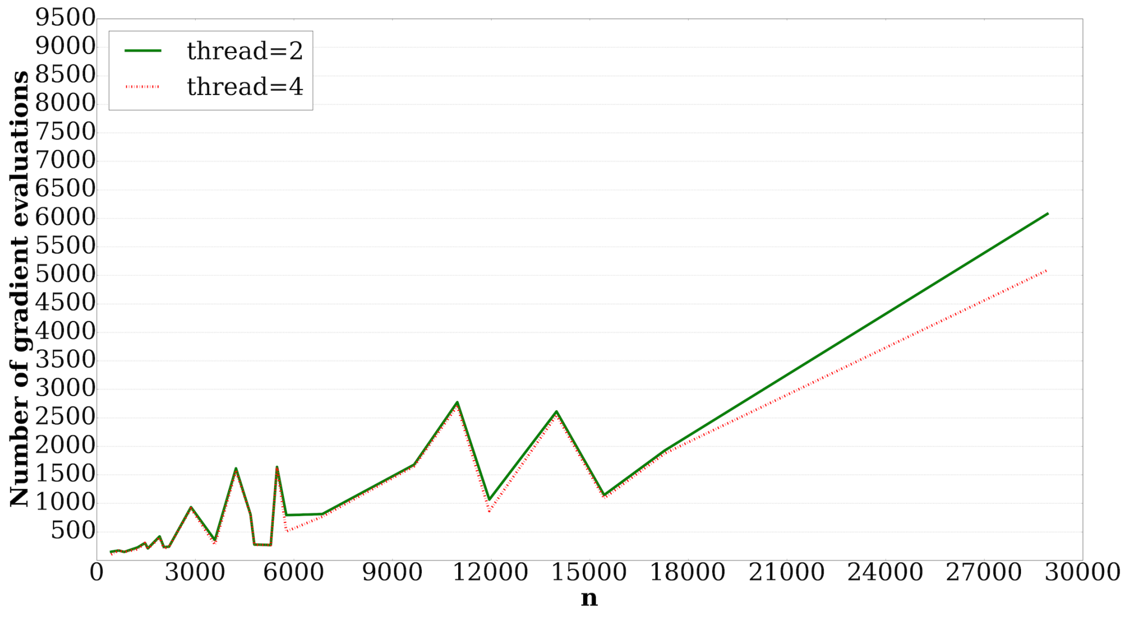

Figure 5.5: The number of gradient evaluation required for data set 1 in implementation *thread* = 2 and *thread* = 4

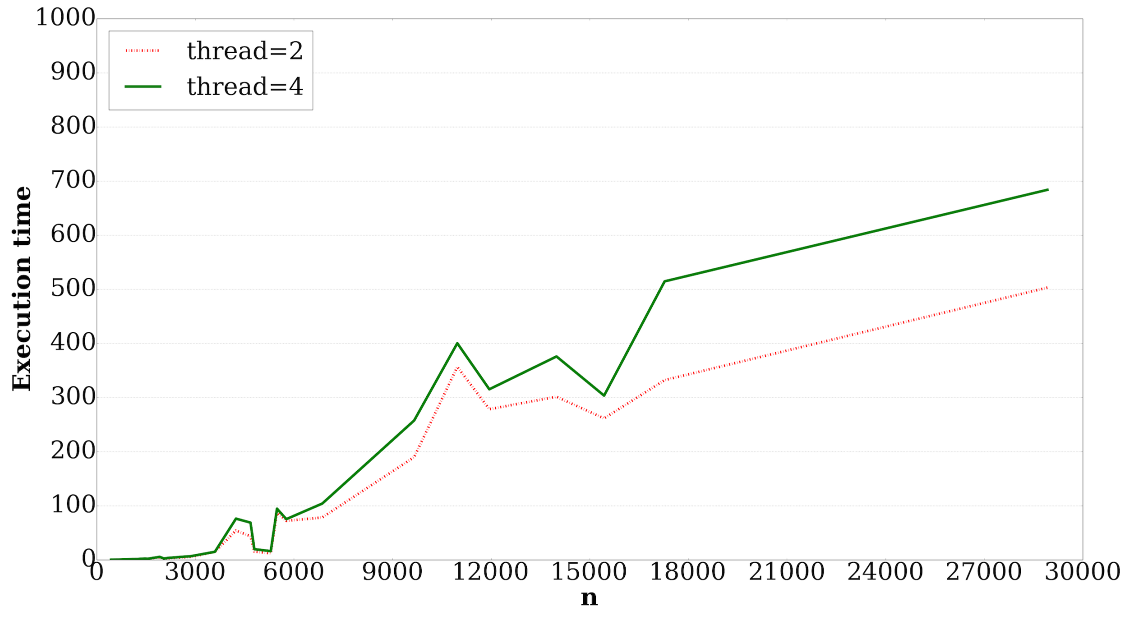

Figure 5.6: Execution time required for data set 1 in implementation *thread* = 2 and *thread* = 4

Table 5.7: Comparison between *thread* = 2 and *thread* = 4, for sparsity pattern detection in data set 2 based on gradient evaluation and execution time.

Matrix	**n**	**nnz**	**ngeval**		**execution time**	
			thread=2	thread=4	thread=2	thread= 4
662_bus	662	2,474	106	98	0.12	0.19
dwt_758	758	5,994	209	200	0.76	0.9
rdb800l	800	4,640	220	218	0.45	0.67
olm1000	1,000	4,994	178	178	0.93	0.94
jagmesh3	1,089	7,361	194	193	1.56	1.59
jagmesh5	1,180	7,750	197	190	1.87	1.9
jagmesh8	1,141	7,465	199	192	1.86	1.9
dwt_1242	1,242	10,426	247	241	1.65	1.96
lshp1270	1,270	8,668	204	194	2.52	2.94
jagmesh9	1,349	9,101	200	176	2.961	3.32
jagmesh4	1,440	9,504	197	197	3.0	3.32
bcspwr06	1,454	5,300	179	120	1.59	1.72

Continued on next page

Table 5.7 – continued from previous page

Matrix	**n**	**nnz**	**ngeval**		**execution time**	
			thread=2	thread=4	thread=2	thread=4
bcspwr08	1,624	6,050	178	178	2.16	2.20
filter2D	1,668	10,750	202	202	4.32	5.16
ex_33	1,733	22,189	299	150	4.88	6.47
watt_1	1,856	11,488	234	234	3.56	5.82
G26	2,000	39,980	582	471	9.14	10.41
t2dal_a	4,257	37,465	255	255	27.56	42.00
nasa4704	4,704	104,756	629	453	51.0289	63.71
EX5	6,545	295,680	1,015	1,013	98.16	112.40
Kuu	7,102	340,200	1,801	1,737	140.15	175.59
G65	8,000	32,000	179	160	55.72	63.38
delaunay_n13	8,192	49,049	2,696	2,685	57.35	65.08
aft01	8,205	125,567	417	817	123.58	146.34
nemeth02	9,506	394,808	1,664	1,403	211.59	323.21

Continued on next page

Table 5.7 – continued from previous page

Matrix	**n**	**nnz**	**ngeval**		**execution time**	
			thread=2	thread=4	thread=2	thread=4
wing_nodal	10,937	150,976	778	757	269.86	293.08
linverse	11,999	95,977	456	378	252.29	269.49
stokes64	12,546	140,034	582	58	264.4	285.6
barth5	15,606	107,362	789	783	309.7	349.63
gyro_m	17,361	340,431	1,978	1,690	306.37	429.04
trefethen_20000b	19,999	554,435	3,345	3,330	478.16	559.97
t3dl_e	20,360	20,360	3	3	254.31	259.1
tube1	21,498	897,056	3,059	3,041	502.47	694.77

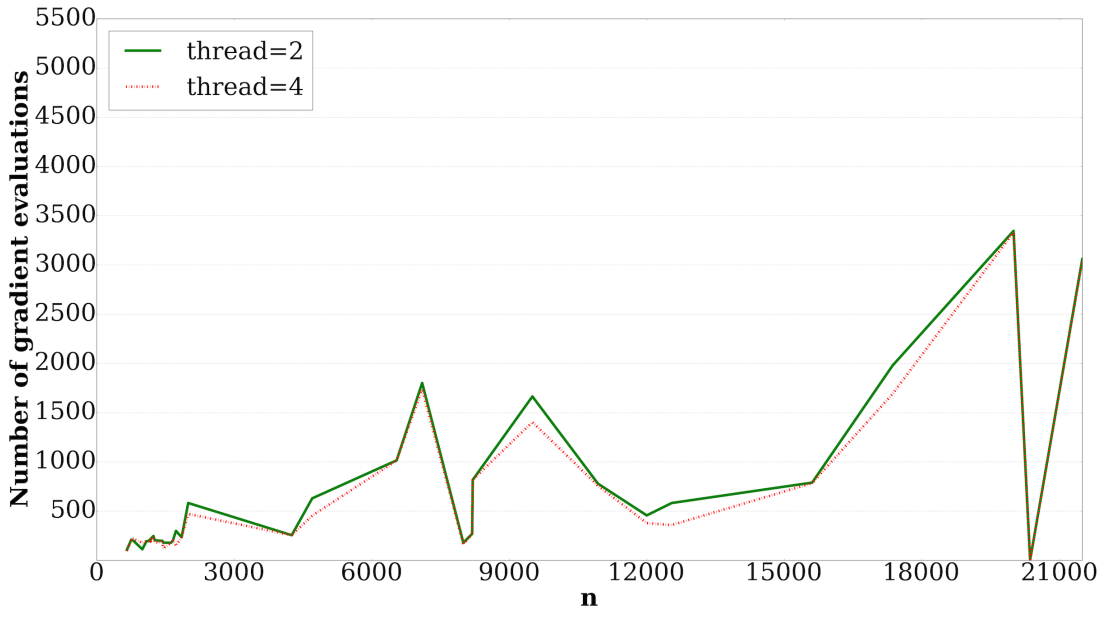

Figure 5.7: Number of gradient evaluations required for data set 2 in implementation *thread* $= 2$ and *thread* $= 4$

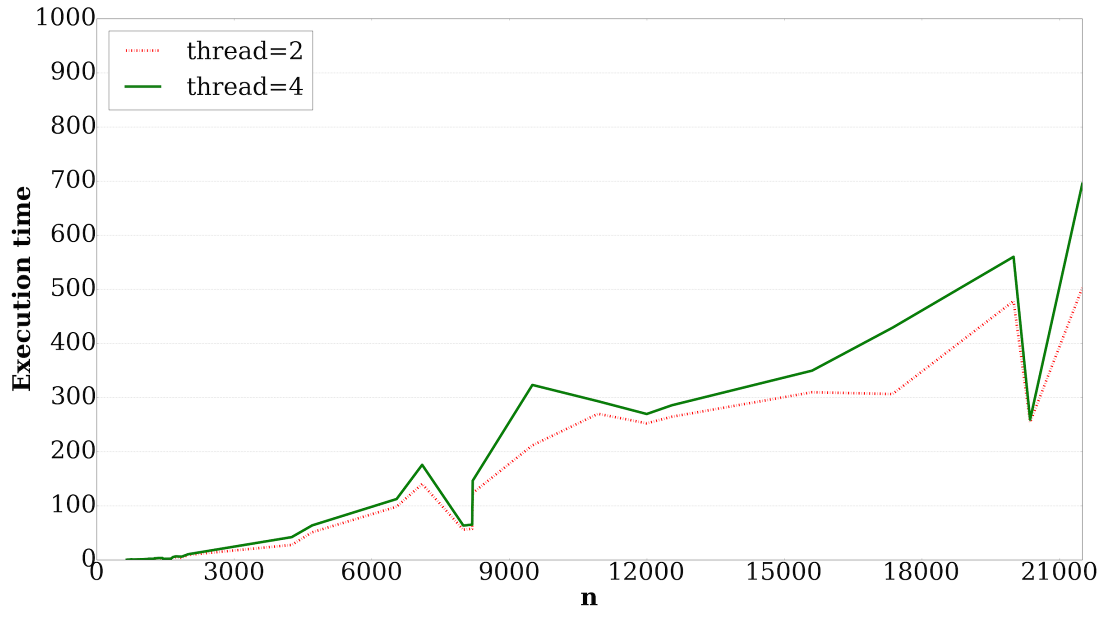

Figure 5.8: Execution time required for data set 2 in implementation *thread* $= 2$ and *thread* $= 4$

Number of Iterations

The number of iteration, inside the while loop, to obtain the true sparsity pattern will be discussed here. From table 5.2 and 5.3 we observed the inner loop iterate 1 to 2 times for the small number of non-zero entries. However, for large number of non-zero elements this sometimes requires 4 to 5 iterations. Actually, the number of iteration within inner loop depends on the randomly generated sparsity pattern at each level. The number of iteration depends on how much good that randomly generated sparsity pattern is. The same holds true for outer loop iteration. For a small number of non-zero entries the number of outer iterations for the multilevel algorithm is 1 to 2 but for a large number of non-zero elements, number of iteration is between 1 and 4. In case of parallel implementation, sometimes this requires less number of iteration not only for parallelism but also for good guess of random sparsity pattern. For parallel implementation, number of iteration happens within the thread which saves time and sometimes minimizes gradient evaluation cost.

Parallel Implementation

In case of parallel implementation iteration within while loop accomplished at the same time in the different core. We have used *OpenmMP* for parallel execution and *thread* to compute iteration process. It is expected to half the execution time for the inner while loop, but does not happen in reality due to many overheads. But required time reduced in parallel execution, keeping evaluation cost same as serial computing in most of cases.

Number of CPR calls

We need CPR algorithm for flaw identification. After every addition of R_k CPR algorithm is required to identify missing elements. If we set R for the first time and add R at each level after calculation and addition of flaw, then we do not need to execute CPR algorithm repeatedly. But this poses one disadvantage, if our initial guess pattern is not good enough, then the number of iteration will increase which will also affect number of gradient evaluation.

Number of Gradient Evaluations

Depending on non-zero value positions and the maximum number of non-zero entries in a row, gradient evaluation differs for approximately same size matrix. Each directional derivative costs one extra evaluation and the sum of total function evaluation is calculated as the gradient evaluation. This gradient evaluation increase inside the while loop iteration because of randomly generated sparsity pattern R_k (step 11 to step 18) in the multilevel algorithm and decreases outside the while loop. In our implementation, we noticed that the number of gradient evaluation is almost half of the previous implementation [38]. We have reported the gradient evaluation values from Sultana's thesis for comparison. Comparison between two columns, $ngeavl$ and $nfeavl$, presented in figure 5.1 and 5.2 clearly identify the improvement of our implementation. Usage of band matrix, appropriate implementation of R_k and efficient algorithm reduce the number of gradient evaluation for sparsity detection.

5.4.1 Important Aspects

In this section, we will present some key facts and observations.

- Initial guess pattern is not always mandatory in our implementation. But to increase the efficiency of our algorithm and reduce gradient evaluation we have assumed an initial band matrix of bandwidth three for pattern detection of an unknown sparse Jacobian matrix. That matrix S_0 is symmetric and sparse with size $n \times n$
- $n_{estimate}$ = nnz - (3 $*$ number of rows). Here nnz denotes number of non-zero entries in the true pattern $M \in R^{n \times n}$
- For compilation with different guess patterns, we have calculated band matrix R of bandwidth five which is generated randomly. This band matrix will be added with our traditional guess pattern within condition $nnz(S0) < k1 * n_{estimate}$

- In multilevel operation, we have chosen the value for $k < 2$. And we have noticed that if we increase the value of k, then the total number of iteration decreases but gradient evaluation increases. The same happened when we increase the bandwidth of symmetric random pattern

- One advantage of voting scheme is, through the whole implementation we need not store flaw matrix after adding it with the symmetric pattern. This saves more space and makes process faster

5.4.2 Summary of Numerical Experiment

Listed below are the key findings obtained obtained from analyzing the facts and figures reported primarily in tables 5.2, 5.3, 5.4 and 5.5.

- There is required at least ρ_{max} number of color group for each matrix. In most cases, the number of color groups is equal to ρ_{max}. From which we can state that the value of $ncolor$ is optimal.

- *ncpr* for test metrics is quite modest and independent of the matrix dimension.

- The partitioning algorithm is very efficient for sparse patterns.

- On average parallel implementation decreases time almost half for inner loop iteration. This affects total execution time and give better performance.

- A key observation is that efficient structure of coding in parallel implementation can reduce overhead and give better performance.

Chapter 6

Conclusion

In this thesis, we have improved multilevel algorithm by integrating band matrices for detection of the unknown pattern of a sparse Jacobian matrix, specially for the large-scale matrices. Our proposed model reduces gradient evaluation, memory storage and execution time. We have also provided a detailed description of how to take advantage of the structural pattern of a sparse matrix. In addition, we have formulated and implemented a parallel version of the multilevel algorithm, where the usage of multicore techniques for parallelism makes our approach faster.

In our results section, we have presented tables which shows the performance of our proposed approach on test data sets collected from Matrix Market and University of Florida. Performance figures show that our proposed approach works better compared to existing techniques. In particular, we have been able to reduce the gradient evaluation and execution time for sparsity pattern detection. The results segment also shows that the proposed approach performs efficiently for the different processor, operating system, and single/parallel implementations.

6.1 Future Works

There are many scopes to extend this thesis, but it will require further analysis of additional data structure, DSJM toolkit implementation and knowledge about parallel implementation. Listed below are some possible future directions for research.

- Increasing the number of processes for parallel programming on a good structure of coding can reduce gradient evaluation and execution time [20]. This experiment is being implemented.

- For parallel implementation CUDA C programming language with NVCC compiler on Linux environment [33] can be implemented for fast processing. But without the proper understanding of our model, it would be difficult for anyone.

- Generating symmetric trial pattern at each level increases gradient evaluation and consumes more time. Instead of generating the symmetric pattern at each level, using the previous one might prove useful for saving time and partitioning which might be an interesting future direction.

Bibliography

[1] Admat. `http://www.cayugaresearch.com/admat.html`accessed 02-05-2018.

[2] The matrix market project. `https://math.nist.gov/MatrixMarket/`accessed 02-05-2018.

[3] The university of florida sparse matrix collection. `https://sparse.tamu.edu/`accessed 02-05-2018.

[4] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[5] Shameem Akhter and Jason Roberts. *Multi-core programming*, volume 33. 2006.

[6] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[7] Aydın Buluç, John Gilbert, and Viral B Shah. Implementing sparse matrices for graph algorithms. In *Graph Algorithms in the Language of Linear Algebra*, pages 287–313. SIAM, 2011.

[8] Richard G Carter, Shahadat Hossain, and Marzia Sultana. Efficient detection of hessian matrix sparsity pattern. *ACM Communications in Computer Algebra*, 50(4):151–154, 2017.

[9] Richard Geoffrey Carter. *Fast numerical determination of symmetric sparsity patterns*. Citeseer, 1992.

[10] Thomas F Coleman, Burton S Garbow, and Jorge J Moré. Software for estimating sparse jacobian matrices. *ACM Transactions on Mathematical Software (TOMS)*, 10(3):329–345, 1984.

[11] Thomas F Coleman and Jorge J Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[12] Thomas F Coleman and Arun Verma. The efficient computation of sparse jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 19(4):1210–1233, 1998.

[13] AR Curtis, Michael JD Powell, and John K Reid. On the estimation of sparse jacobian matrices. *IMA Journal of Applied Mathematics*, 13(1):117–119, 1974.

[14] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*, volume 16. Siam, 1996.

[15] Shaun A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in matlab. *ACM Trans. Math. Softw.*, 32(2):195–222, June 2006.

[16] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM review*, 47(4):629–705, 2005.

[17] Assefaw Hadish Gebremedhin, Arijit Tarafdar, Fredrik Manne, and Alex Pothen. New acyclic and star coloring algorithms with application to computing hessians. *SIAM J. Scientific Computing*, 29(3):1042–1072, 2007.

[18] D Goldfarb and Ph L Toint. Optimal estimation of jacobian and hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, 43(167):69–88, 1984.

[19] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[20] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[21] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[22] Mahmudul Hasan, Shahadat Hossain, Ahamad Imtiaz Khan, Nasrin Hakim Mithila, and Ashraful Huq Suny. Dsjm: a software toolkit for direct determination of sparse jacobian matrices. In *International Congress on Mathematical Software*, pages 275–283. Springer, 2016.

[23] AKM Shahadat Hossain and Trond Steihaug. Computing a sparse jacobian matrix by rows and columns. *Optimization Methods and Software*, 10(1):33–48, 1998.

[24] Shahadat Hossain and Trond Steihaug. Graph models and their efficient implementation for sparse jacobian matrix determination. *Discrete Applied Mathematics*, 161(12):1747–1754, 2013.

[25] Shahadat Hossain and Trond Steihaug. Optimal direct determination of sparse jacobian matrices. *Optimization Methods and Software*, 28(6):1218–1232, 2013.

[26] David Juedes and Jeffrey Jones. Coloring jacobians revisited: a new algorithm for star and˜ acyclic bicoloring. *Optimization Methods and Software*, 27(2):295–309, 2012.

[27] CT Kelley. Iterative methods for linear and nonlinear equations. *Frontiers in applied mathematics*, 16:575–601, 1995.

[28] Ahamad Imtiaz Khan et al. Improved implementation of some coloring algorithms for the determination of large and sparse jacobian matrices. Master's thesis, Lethbridge, Alta.: Universtiy of Lethbridge, Department of Mathematics and Computer Science, 2017.

[29] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[30] Rainald Löhner, K Morgan, and Olgierd C Zienkiewicz. The solution of non-linear hyperbolic equation systems by the finite element method. *International Journal for Numerical Methods in Fluids*, 4(11):1043–1063, 1984.

[31] S Thomas McCormick. Optimal approximation of sparse hessians and its equivalence to a graph coloring problem. *Mathematical Programming*, 26(2):153–171, 1983.

[32] David K. Melgaard and Richard F. Sincovec. General software for two-dimensional nonlinear partial differential equations. *ACM Trans. Math. Softw.*, 7(1):106–125, March 1981.

[33] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[34] Jorge Nocedal and Stephen Wright. Numerical optimization: Springer science & business media. *New York*, 2006.

[35] MJD Powell and Ph L Toint. On the estimation of sparse hessian matrices. *SIAM Journal on Numerical Analysis*, 16(6):1060–1074, 1979.

[36] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[37] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013.

[38] Marzia Sultana et al. On the efficient determination of hessian matrix sparsity pattern: algorithms and data structures. Master's thesis, Lethbridge, Alta: University of Lethbridge, Dept. of Mathematics and Computer Science, 2016.

[39] ET Sun and MA Stadtherr. On sparse finite-difference schemes applied to chemical process engineering problems. *Computers & chemical engineering*, 12(8):849–851, 1988.

[40] Ala Taftaf. *Extensions of algorithmic differentiation by source transformation inspired by modern scientific computing*. Theses, Université Côte d'Azur, January 2017.

[41] Andrea Walther. Getting started with adol-c. In Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo, editors, *Combinatorial Scientific Computing*, number 09061 in Dagstuhl Seminar Proceedings, pages 181–202, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.