UNIVERSITY OF
Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

A Fine-grained Approach to Parallelise the Circuit Simulation Process on Highly Parallel Systems

by

Mansour Rasoulzadeh Darabad

A thesis for the degree of
Doctor of Philosophy

in the
Faculty of Physical Sciences and Engineering
Electronics and Computer Science

November 2015

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

Faculty of Physical Sciences and Engineering
ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Mansour Rasoulzadeh Darabad

Over the last decades, there has been a rapid growth in the size and complexity of electronic circuits. Since the clock speed of microprocessors is saturated around 3 GHz, computers are no longer able to keep up with the simulation challenges and new simulation approaches are required. SPICE-like simulation algorithms have many intrinsically sequential elements. With the availability of multi-core systems, several attempts have been made to speed up the circuit simulation process, in terms of parallelising the device evaluation or the matrix solution phase. However, these methods have resulted in limited speed-ups or compromised accuracy. Another existing issue is the barrier between device evaluation and matrix solution phases, which prevents the whole simulation process being parallelised.

Most of the existing attempts on parallelising circuit simulation algorithm are based on the conventional and coarse grained methods. We propose new very fine-grained parallel approaches for the matrix solution and device evaluation phases with the possibility of mixed analysis of the two phases to totally parallelise the simulation process. Instead of the conventional direct matrix solvers we use highly parallel iterative methods. The motivation behind this approach is the availability of new parallel platforms and architectures for highly parallel and distributed simulations. SpiNNaker project is one of the new architectures which aims to model large-scale spiking neural networks on a massively parallel million-core system. The purpose of this work is proposing very fine-grained parallel approaches and preparing the ground work for performing the proposed methods on highly parallel structures.

In this work, the matrix solution part of the circuit simulation process is performed using a Jacobi-type iterative method. The proposed fine-grained parallel method distributes the solution of circuit equations across a large number of light-weight processors by allocating one processor to each circuit equation. The device modelling process is inherently a parallel task since modelling each device can be done independently. For the device modelling phase, we use the Secant method instead of conventional Newton-Raphson iterations to avoid the calculation of partial derivatives at each iteration.

The proposed methods are applied to a number of benchmark matrices and test circuits and are optimised to work best on sparse systems. Simulation results confirm

the functionality of the proposed Jacobi-type iterations in the parallel solution of matrix equations. Compared to a conventional direct matrix solution (LU-factorisation), the proposed fine-grained parallel iterative method performs better as the size of the problem increases with a speed-up of around 2.5x for the largest example matrix. Furthermore, by replacing the computationally intensive Newton-Raphson iterations for the device evaluation phase with the Secant method, which benefits from a much simpler algorithm, the simulation time is improved by a factor of 3 to 4 for the example circuits. Finally, simultaneous evaluation of the proposed parallel iterative method for matrix solution and the Secant method for device evaluation, to replace the conventional direct solution and Newton-Raphson iterations, resulted in a significant improvement in the simulation time of the under test circuits. The simulation results suggest that compared to the conventional sequential algorithm, the proposed fine-grained parallel approach has achieved an overall speed-up of 16x and 22x for the two test circuits.

Contents

Acknowledgements **xv**

Abbreviations **xvii**

List of Symbols **xix**

1 Introduction **1**
- 1.1 Computer Simulation . . . 1
- 1.2 Introduction to Circuit Analysis . . . 2
- 1.3 Multi-core and Parallel Simulation . . . 4
- 1.4 Research Motivation . . . 6
- 1.5 Research Objectives . . . 7
- 1.6 Thesis Contribution . . . 9

2 Literature Review **11**
- 2.1 The SPICE Algorithm . . . 12
- 2.2 Matrix Construction . . . 13
  - 2.2.1 Classical Methods . . . 13
  - 2.2.2 Tableau Formulation . . . 15
  - 2.2.3 Modified Nodal Analysis . . . 15
    - 2.2.3.1 Modelling Nonlinear Elements . . . 18
    - 2.2.3.2 Automatic Equation Formulation . . . 19
    - 2.2.3.3 Properties of Circuit Simulation Matrices . . . 24
- 2.3 Matrix Solution . . . 24
  - 2.3.1 Direct Methods . . . 24
    - 2.3.1.1 Classical Gaussian Elimination . . . 25
    - 2.3.1.2 LU-factorisation . . . 27
  - 2.3.2 Iterative Methods . . . 29
    - 2.3.2.1 Stationary Iterative Methods . . . 30
    - 2.3.2.2 Non-stationary Iterative Methods . . . 32
  - 2.3.3 Hybrid Methods . . . 32
  - 2.3.4 Discussion on Matrix Solution Methods . . . 34
- 2.4 Parallel Circuit Simulation . . . 35
  - 2.4.1 Parallel and Distributed Computing . . . 35
    - 2.4.1.1 Parallelisation Issues . . . 36
    - 2.4.1.2 Parallel Computing Systems . . . 37
    - 2.4.1.3 Speed-up . . . 38
- 2.5 Parallel SPICE . . . 38

2.5.1 Existing Work . . . 39
2.5.2 Parallel SPICE Simulation Challenges . . . 42
2.6 Summary . . . 43

3 Random Jacobi Iterations 45
3.1 Generating Test Matrices for the Preliminary Simulations . . . 46
3.2 Applying Iterative Algorithms to Test Matrices . . . 47
3.2.1 Convergence Comparison . . . 47
3.2.2 Effects of Parallel and Non-deterministic Evaluation . . . 49
3.2.3 Initial Guess Vector . . . 53
3.3 Euclidean Norm Calculation . . . 54
3.4 Summary and Discussion . . . 58

4 Device Evaluation 61
4.1 Device Modelling . . . 61
4.2 Linear Approximation of Nonlinear Elements . . . 63
4.2.1 The Newton-Raphson Method . . . 63
4.2.2 Fixed Slope Approach . . . 67
4.2.3 The Secant Method . . . 68
4.3 Case Studies on NR and Secant Methods . . . 69
4.3.1 Case Study # 1 . . . 69
4.3.2 Case Study # 2 . . . 75
4.4 Summary and Discussion . . . 77

5 Parallel Matrix Solution 79
5.1 Test Matrices . . . 79
5.2 Applying the Random Jacobi Method to Test Matrices . . . 82
5.2.1 Test Matrix Set #1 . . . 82
5.2.2 Test Matrix Set #2 . . . 85
5.2.2.1 Single-core Simulations . . . 86
5.2.2.2 Virtual Many-core Simulations . . . 87
5.2.2.3 Real Many-core Simulations . . . 88
5.2.3 Optimising the Communication Between Processors . . . 92
5.2.4 Time Measurement . . . 97
5.3 Summary and Discussion . . . 99

6 Simultaneous Analysis of the Two Simulation Phases 101
6.1 Properties of the Proposed Methods . . . 101
6.2 Simultaneous Evaluation . . . 102
6.2.1 Circuit Equations and Linear Device Models . . . 103
6.2.2 Highly Parallel and Simultaneous Evaluation . . . 105
6.3 Implementation Considerations . . . 108
6.3.1 Discussion of Implementation on SpiNNaker . . . 109
6.4 Summary and Discussion . . . 112

7 Conclusions 115
7.1 Summary . . . 115
7.2 Novel Contributions . . . 116

7.3 Future Work . . . 119

A Simulations and codes **121**
A.1 Generating Test Matrices by Matlab . . . 121
A.2 Convergence Comparison of the Iterative Methods . . . 124
A.3 Newton-Raphson and Secant Methods Comparison . . . 125
A.4 C++ Code for Iterative Solution Algorithms on Single Core . . . 129

B Message Passing Interface **131**
B.1 Virtual Many-core Simulations using MPI . . . 131
B.2 The Iridis Computer Cluster . . . 132
B.3 Real Many-core Simulations using MPI on Iridis . . . 134

C Paper Presented in DAC 2013 **135**

Bibliography **139**

List of Figures

1.1 35 years of processor trend data [1] 5

2.1 Simplified block diagram of the SPICE simulation process. 14
2.2 (a) A simple network; (b) Its graph [2]. 15
2.3 Tableau formulation for the network in Figure 2.2 [2]. 16
2.4 (a) An example network (b) System matrix set up using MNA [2]. 17
2.5 Stamps for elements of the circuit in Figure 2.4. a) Independent voltage source stamp b) Inductor stamp c) CCVS stamp (the resistor value can be zero) [2]. 18
2.6 a) A conductance b) Stamp for the conductance G c) A current source d) Stamp for the current source I [3] 20
2.7 a) A simplified diode model b) Diode stamp [3] 21
2.8 a) A simplified MOS transistor model b) MOS transistor stamp [3] 22
2.9 a) Capacitor companion model b) Capacitor stamp [3] 22
2.10 Gaussian elimination process for a 4*4 matrix [4]. 25
2.11 Gaussian elimination: a) The original matrix system b) Forward elimination applied c) Backward substitution [5]. 26
2.12 Gaussian-Jordan elimination: a) The original matrix system along with the added Identity matrix b) Matrix of coefficients is converted to a diagonal matrix by the elementary row operations [5]. 27
2.13 LU-factorisation of a square matrix, A [6]. 28
2.14 (a) The Gauss-Jacobi algorithm (b) The Gauss-Seidel algorithm [7]. 31
2.15 (a) SIM method versus modifies SIM using adaptive relaxation (b) BiCGSTAB method versus pre-conditioned BiCDSTAB (c) SIM-AR versus BiCGSTAB-precon [8]. 33
2.16 Parallelisation; distribution of a problem over several CPUs [9]. 36
2.17 Processors communication for (a) shared (b) distributed memory organisation [10]. 38
2.18 Barriers between device evaluation and matrix solution phases [11]. 39

3.1 Comparing the convergence speed of Gauss-Seidel, normal Jacobi, and Random Jacobi iterations for four different test matrices. 48
3.2 The effect of the order of the evaluation of equations on the convergence of Gauss-Seidel and Jacobi methods. For each method, the simulation was repeated for three times. 49
3.3 The effect of failure of some processors on the convergence of the Random Jacobi method. 51
3.4 The effect of improper update of some processors on the convergence of the Random Jacobi method. 52

3.5 Convergence of Random Jacobi iterations when 10% of the processors do not update at each iteration. 53
3.6 Convergence of Random Jacobi iterations when some processors update more frequently. 54
3.7 The effect of the initial guess vector on the convergence of iterative methods. 55
3.8 Distribution of the solution vector elements for the test matrix of Figure 3.7. 55
3.9 Iteration stop criterion: The Euclidean norm between the solution at each iteration and the real solution. 57
3.10 Iteration stop criterion: The Euclidean norm between the solutions of the last two successive iterations. 57

4.1 Circuit analysis flowchart [6] 62
4.2 Assembling a diode and a transistor into a matrix system using their stamps [12] 63
4.3 Finding the root of a function, $f(x)$, using the Newton-Raphson method [13]. 64
4.4 a) A simple diode circuit b) Graphs of the equations. 64
4.5 Newton-Raphson convergence issue caused by a) a local extremum b) a nonconvergent cycle [13]. 66
4.6 Using constant slopes in the G^0 approach 67
4.7 Linear approximation using the Secant method 68
4.8 An example circuit with one MOS transistor 70
4.9 Four different methods to perform device modelling and matrix solution. . 72
4.10 A MOS Differential Pair 75
4.11 A and RHS matrices for the circuit in Figure 4.10 76

5.1 Format of the matrices extracted from Ngspice simulations 81
5.2 Transient analysis of *cir919res* circuit using Ngspice and the output text files containing extracted matrices. 83
5.3 The A and RHS matrices extracted from Ngspice simulation into the *matdump.txt* and *rhsdump.txt* files. 84
5.4 A highly parallel structure: one processor per circuit equation. 86
5.5 A highly parallel structure: one processor per circuit equation. 88
5.6 Communication between processors for the parallel Random Jacobi method. 89
5.7 Parallel Jacobi Matrix Solution Process Using MPI on Iridis. 90
5.8 Execution time comparison for 4 different matrix solution approaches. . . 91
5.9 A 6 by 6 test matrix. 93
5.10 Communication between processors for the parallel Random Jacobi method when using the proposed optimised pattern. 94
5.11 Pre-analysing the circuit matrix structure to obtain a suitable pattern to reduce required communications between processors. 95
5.12 Send and receive buffers for exchanging data between processors. 96
5.13 Execution time comparison for three different matrix solution methods. . 98

6.1 a) Undesired stops of the linearisation phase when using direct matrix solution methods b) Possibility of resolving the issue by using a parallel and iterative matrix solver 103
6.2 Communications and calculations required for cores number 3, 4, and 8 in Figure 4.10 106

6.3 Execution time comparison for the differential pair MOS test circuit in Figure 4.10 107
6.4 Execution time comparison for the single MOS test circuit in Figure 4.8 . 108
6.5 Principal architectural components of a SpiNNaker node [14]. 110
6.6 Spinnaker multiprocessor architecture [15]. 111
6.7 The SpiNNaker machine. [16]. 111

B.1 Implementation of MPI in C++ program. 132
B.2 Compiling and running MPI on virtual many-cores. 133
B.3 An example of scripts used to submit parallel MPI jobs to Iridis. 134

List of Tables

3.1 Test matrices for preliminary simulations. 46
3.2 Iteration stop criterion: Euclidean norm between the solution at each iteration and the real solution. 58
3.3 Iteration stop criterion: Euclidean norm between the solutions of the last two successive iterations. 58

4.1 Number of NR iterations for the diode circuit in Figure 4.4.a 66
4.2 Convergence comparison of the NR and the fixed slope iterations for the diode circuit in Figure 4.4a 67
4.3 Convergence comparison of NR and Secant iterations for the diode circuit in Figure 4.4a with the first initial guess set. 69
4.4 Convergence comparison of NR and Secant iterations for the diode circuit in Figure 4.4a with the second initial guess set. 69
4.5 Solution to the circuit in Figure 4.8 using the NR method along with a direct matrix solver 73
4.6 Solution to the circuit in Figure 4.8 using the NR method along with the Jacobi method as an iterative matrix solver 73
4.7 Starting values for simulation results in Table 4.5 and Table 4.6 73
4.8 Solution to the circuit in Figure 4.8 using the Secant method and a direct matrix solver 74
4.9 Solution to the circuit in Figure 4.8 using the Secant method and the Jacobi method as an iterative matrix solver 74
4.10 Starting values for simulation results in Table 4.5 and Table 4.6 74
4.11 Number of iterations and execution time required to obtain the solution by different methods for the test circuit in Figure 4.8. 75
4.12 Starting values for simulation results of the test circuit in Figure 4.10 . . 76
4.13 Number of iterations and execution time required to obtain the solution by different methods for the test circuit in Figure 4.10. 77

5.1 Test matrices from the Florida Sparse Matrix Collection. 80
5.2 Test matrices from Ngspice Simulations. 81
5.3 The Parallel Random Jacobi method applied to the circuit matrices from different iterations of various time points of a transient analysis. 85
5.4 The Random Jacobi method on a single-core. 86
5.5 The Gauss Seidel method on a single-core. 87
5.6 Execution time comparison of the Gauss Seidel and Random Jacobi iterations on a single processor. † The unit for time measurement is *Seconds*. 87
5.7 Random Jacobi using MPI on a virtual parallel network of processors. † The unit for time measurement is *Seconds*. 88

5.8 Random Jacobi using MPI on a parallel many-core cluster. † The unit for time measurement is *Seconds*. 91
5.9 Send/Receive pattern for reducing the required communication. 94
5.10 The solution vector of the matrix system obtained in different iterations. . 96
5.11 Random Jacobi using MPI on a parallel many-core cluster with optimised communication. † The unit for time measurement is *Seconds*. 97
5.12 Effect of sparsity on matrix solution time using the optimised communication pattern. † The unit for time measurement is *Seconds*. 98

6.1 Number of Secant iterations and execution time required to obtain the solution by a highly parallel evaluation. 107

Acknowledgements

I would like to thank Prof. Mark Zwolinski, my supervisor, for his kind help and valuable comments and support during my PhD.

I acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

Abbreviations

AC	Alternating Current
API	Application Programming Interface
BiCGSTAB	Bi-Conjugate Gradient Stabilized
BiCGSTAB-precon	Pre-conditioned Bi-Conjugate Gradient Stabilized
CAD	Computer Aided Design
CANCER	Computer Analysis of Nonlinear Circuits Excluding Radiation
CCVS	Current Controlled Voltage Source
CPU	Central Processing Unit
DC	Direct Current
FPGA	Field-Programmable Gate Array
GJ	Gauss-Jordan
GPU	Graphics Processing Unit
I	Identity Matrix
IC	Integrated Circuit
KCL	Kirchhoff Current Law
KVL	Kirchhoff Voltage Law
MNA	Modified Nodal Analysis
MPI	Message Passing Interface
NA	Nodal Analysis
NR	Newton-Raphson
OpenMP	Open Multi-Processing
RHS	Right Hand Side
SIM	Sparse Iterative Method
SIM-AR	Sparse Iterative Method with Adaptive Relaxation
SOR	Successive Over Relaxation
SPICE	Simulation Program with Integrated Circuits Emphasis
VLSI	Very Large Scale Integration

List of Symbols

f	Function
I_s	Reverse bias saturation current
λ	A constant equal to the inverse of the thermal voltage (v_T)
D	Diagonal matrix
U	Strictly upper triangular matrix
L	Strictly lower triangular matrix
G	Conductance
I	Current source or Identity matrix
p	Number of processors
f_p	Fraction of work which can be carried out in parallel
S_p	Speed-up
C	Capacitor
h	Time step
v_{th}	Threshold voltage
k'	A constant in MOS transistors current
w	Channel width
l	Channel length

*To my wife, **Pegah**,*
who has always been there for me with her great support.

Chapter 1

Introduction

1.1 Computer Simulation

Before electronic circuits became as large and complicated as they are nowadays, computational methods had very little contribution in analysis and design of electronic networks. Designers could synthesise a network using a simple and routine procedure. They just needed to set up the network on a breadboard, apply test stimuli and measure responses. Then, some modifications were made based on the circuit's behaviour until the system met the desired specifications [17, 18].

When integrated circuits with complicated structures and quite a large number of elements were introduced and access to computers became more pervasive, the situation changed. Experimental approaches were no longer feasible and new approaches were needed to simplify and accelerate the simulation process. The alternative approach was to *simulate* electronic circuits using computer methods. Integrated circuits made the fabrication of faster computers possible and fast computers made the design of integrated circuits easier. In other words, technological progress led to the design of very large networks containing a large number of interconnected transistors on a small chip and such a huge network could not be synthesised by experimental methods. Recently, relatively powerful computers have become accessible which make computational methods more and more important. Nowadays, many of designers treat the simulated circuits as the objective reality and expect the real circuit to emulate the simulation [17, 18, 3].

In addition to the advancements in computer technologies, some major innovations in numerical mathematics such as sparse matrix solution methods, linear methods for the solution of differential equations, sensitivity analysis techniques, optimisation methods, and parallel algorithms and structures have had important impacts on all aspects of *Computer Aided Design (CAD)* process. After the advent of computer based methods, many circuit analysis programs were developed, some of them are still in use today, and

a few of them are widely used in research projects as well as industry to simulate and design electronic circuits [17, 19].

Simulation Program with Integrated Circuits Emphasis (SPICE) from the University of California at Berkeley is one of the standard computer programs which has become dominant in circuit analysis field and is widely used for both academic and industrial purposes [20, 21].

Most of the general purpose circuit simulation programs provide the following capabilities:

- *Direct Current (DC) analysis*: can be used for linear or nonlinear circuits and determines the operating point of the circuit with inductors shorted and capacitors opened.
- *Alternating Current (AC) small signal analysis*: computes the frequency domain response of the circuit. It first determines the operating point of the system, then models all the nonlinear devices with their linear small signal equivalent, and finally analyses the circuit over a specified frequency range.
- *Transient Analysis*: to obtain the time domain response of the circuit. This analysis computes the transient output variables as a function of time over the specified time intervals. The initial conditions are determined by a DC analysis.

In addition, some circuit simulators provide a variety of other functions such as pole/zero analysis, noise analysis, and sensitivity analysis [22].

Circuit simulation is one of the areas in which there have been a considerable amount of research during the last few years to accelerate the simulation process. Over the past decades, designers mostly relied on advancements in computer architecture to speed up circuit simulation applications such as SPICE. However, approaches such as increasing the clock frequency and using new architectures at the expense of area and power have faced some physical limitations such as clock frequency and power [23, 24]. Recently, using parallel algorithms on multi-core and many-core systems have become more of interest for researchers [25, 26, 27, 28] and several works have been done especially to parallelise the circuit simulation algorithms in order to speed up the circuit simulation process [29, 12, 30].

1.2 Introduction to Circuit Analysis

All circuit analysis programs consist of three main parts: the input part, the simulator part, and the output part. The role of the input or network description part is to describe

the circuit, describe the excitations, and also control the analysis. The simulator or network analysis part performs the main computational task which includes different analysis modes. The output or postprocessor part stores the results obtained from analyses. These results can be used for displaying the analysed data and comparing different analyses [17, 19]. In the current work, we are interested in the second part of circuit analysis programs: the simulator.

Analysing a circuit is the act of computing node voltages and branch currents for a specific excitation. The most common method for equation formulation of electronic circuits is using *Kirchhoff Current Law (KCL)* in conjunction with branch constitutive equations. There are specific methods to construct the equations and matrices describing the systems, among which *Nodal Analysis (NA)* and *Modified Nodal Analysis (MNA)* methods are widely used. To automate the formulation process, a stamp for each element is defined using the network element stamp method which in fact shows the contribution of each element to the network description matrix [3, 31, 32]. Matrix construction methods using element stamps are reviewed in Chapter 2 Section 2.2.3.

When the circuit description matrix is formulated, the next step is solving the matrix of nonlinear circuit equations to calculate the unknown node voltages and branch currents. Solving the matrix of nonlinear circuit equations consists of two main phases which are done iteratively. First, nonlinear circuit elements are approximated by a linear model using linearisation techniques such as *Newton-Raphson (NR)* to form a linear matrix system describing the circuit in the corresponding operating point. This phase is called device evaluation or linearisation. The second step is to solve the linearised matrix system in the form of $Ax = b$ where A is the matrix of conductances, x refers to the vector of unknowns, and b, the *Right Hand Side (RHS)* vector, is the excitation vector. This phase is known as matrix solution [17, 3].

Device evaluation is the act of replacing the nonlinear circuit elements with equivalent linear models to convert the nonlinear equations to linear ones. This is done by using techniques such as the *Secant* method or the *Newton-Raphson* method. The Newton-Raphson iterative method starts with an initial guess and approximates the nonlinear curves with a straight line at the operating voltage/current of the element by calculating the partial derivatives at each iteration. Therefore, this method requires one initial point and also direct calculation of the derivative of the function at each iteration. The Secant method is simpler since it does not need the derivative of the function. However, since it calculates the linear equation by using two points instead of the tangent of the line, it is relatively slower and also needs two initial points to start [33]. The advantages and drawbacks of the linearisation techniques will be discussed in Section 4.2 of Chapter 4 in more detail to see which method is more suitable for our specific needs. Later in Chapter 4 Sections 4.2 and 4.3, the implementations of both methods on real circuits will be shown.

The numerical solution methods for linear systems of equations $Ax = b$, are broadly classified into two categories: direct methods such as *Gaussian elimination* and *LU-factorisation* and iterative methods such as *Jacobi* and *Gauss-Seidel* methods. Direct methods, based on matrix decomposition, theoretically obtain the exact solution using a finite number of operations. Because of their robustness and predictable behaviour, in some cases, direct methods are preferred to iterative methods. Unfortunately, this is rarely true in real applications because direct methods are impractical due to rounding errors. The error made in one step, spreads in all the following steps. Indirect methods, based on iterative processes, use successive approximations to obtain solutions for linear systems and of course the accuracy of the solution depends on the number of iterations [34, 35].

In this work, among all the matrix solving methods, the main focus will be mostly on those methods which are iterative and benefit from parallel algorithms wiht the possibility of being implemented in parallel. Different matrix solution algorithms, techniques to improve the efficiency of the solution process, their advantages and drawbacks, and also suitable methods for our specific work are studied in the literature review chapter Section 2.3.

1.3 Multi-core and Parallel Simulation

The circuit analysis process is a computationally intensive task. It involves various numerical methods to be used in different phases of simulation for tasks such as matrix solution, calculating partial derivatives, numerical approximations, etc [3]. During the past few decades, because of the rapid growth in the size and complexity of integrated circuits, faster simulators were designed to handle the simulation process within a reasonable amount of time. This mainly relied on *Central Processing Units (CPUs)* with higher clock speeds. Increasing performance was mainly done by either increasing the clock speed, which means having more cycles, and/or increasing the number of executions per cycle. This trend was constantly increasing until about 10 years ago when the trend slowed down and the CPU clock speed reached around 3 GHz. For the last few years, there has been only a slight increase in the CPU speed and it seems that the increasing trend has reached an end and saturated around 3.5 GHz, while the number of transistors is still steadily increasing [24, 36, 1, 37]. There has always been a growing demand for faster simulation, higher accuracy, and better performance, which motivates the designers to speed up the simulation process. Simulations of large *Integrated Circuit (IC)* designs may take several days, weeks, or even months. Therefore, one of the most important bottlenecks in electronic circuits design and verification flow is *simulation* [38].

The trend of CPU performance, the current situation, and anticipated limits until 2015 are shown in Figure 1.1. There are several limitations for speeding up processors' clock, among which the most important is power consumption (heat dissipation). While the number of transistors is increasing, as predicted by Moore's law, some other factors such as the clock speed of processors are being saturated around a peak value. Therefore, single-core simulation is no longer able to progress with the same speed as the size and complexity of integrated circuits are advancing [24, 39].

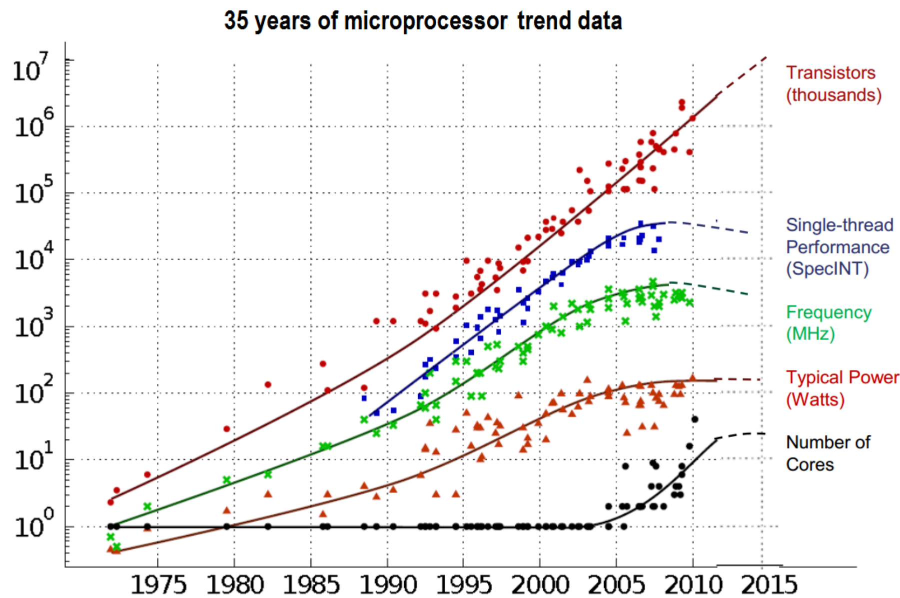


FIGURE 1.1: 35 years of processor trend data [1]

To overcome this issue, designers started to employ multi-threaded and multi-core CPUs to perform multitasking using parallel programs or by running multiple applications concurrently. Although both approaches are based on concurrency, multi-threaded CPUs try to exploit CPU resources at core level by running instructions using multiple threads, while multi-core approaches mostly focus on scalability by increasing the number of cores. Even though multi-threaded and multi-core CPUs achieve better performance compared to single-core processors, there are a number of physical limitations associated with them and as technology scales further, it is not possible to simply increase the number of complex cores to keep up with the trend. The alternative is to go from multi-core to many-core by employing more processors with smaller and less complex structures. Each small processor in a many-core system has lower performance compared to the processors in a multi-core architecture; however, the total performance of a many-core system can be much higher than that of a multi-core one [27, 28, 40, 41].

In recent years, with the availability of multi-core and many-core systems, several attempts have been made to speed up SPICE simulation by taking advantage of parallel processing to parallelise the device evaluation and matrix solution phases on multi-core

CPUs, *Graphics Processing Units (GPUs)*, and *Field-Programmable Gate Arrays (FPGAs)*. However, these methods have resulted in limited speed-ups and there are still existing challenges in parallel simulation which have not been addressed [29, 12, 30, 42]. More importantly, the recent developments mostly use coarse-grained parallel approaches on a small number of processors and do not perform parallelisation on highly parallel and fine-grained systems. Furthermore, exploiting the mentioned multi-core approaches has not solved the problem of inherent sequential properties of conventional SPICE simulations. For example, there is a barrier between NR iterations and the matrix solution phase during the circuit simulation process. These two phases cannot be done simultaneously because each phase depends on the results from the other phase. The matrix solution phase cannot start until the device evaluation is finished and the next device evaluation cannot be performed before the matrix solution process is completed. This limits the amount of possible parallelisation. Most of the existing research try to parallelise either the device evaluation phase or the matrix solution phase of the circuit simulation flow. Despite noticeable results being reported on speeding up either of the simulation phases, totally parallelising the simulation process is still one of the main existing bottlenecks.

These limits will be addressed as part of the current work and new approaches will be proposed to overcome these limits. The main simulation phases will be studied separately for possible parallel implementations that can lead to a new algorithm for highly parallel evaluation of the simulation process. Parallelism on multi-core and many-core architectures and their benefits and challenges will be reviewed in more detail in Chapter 2 Section 2.4 along with a review on parallel circuit simulation literature from its early stages to very recent works.

1.4 Research Motivation

Most of the recent research and existing work on parallel circuit simulation focus on improving the conventional circuit simulation algorithms using their proposed parallel methods. It seems that not enough attention has been paid to new approaches to replace the conventional SPICE algorithm and the existing attempts only try to modify and/or optimise the current conventional SPICE algorithm. The existing parallel approaches are mostly coarse grained and rely on known and well-studied parallel systems such as parallel GPU or FPGA architectures.

The SPICE algorithm is still the standard algorithm for circuit simulation purposes after almost four decades. The original version of SPICE was a program called Computer Analysis of Nonlinear Circuits Excluding Radiation (CANCER) which was refined and renamed later to SPICE and released in 1975 at University of California Berkeley. Soon after, other variations of SPICE were releases such as SPICE2 and SPICE3 which offered

more simulation capabilities such as including *Metal Oxide Semiconductor Field Effect Transistor (MOSFET)* models and the Modified Nodal Analysis method. SPICE is an open source tool and widely used in educational and commercial applications [43, 44]. The advancements in the circuit simulation and design area and SPICE being an open source tool, there have been a number of different tools integrated with SPICE to add more credibilities to SPICE simulations by using new technologies and hardware such as GPUs, FPGAs, cloud computing, and parallel processing [45, 46, 12, 47, 48]. However, to the best of our knowledge, most of the attempts to parallelise the SPICE simulation process are multi-core coarse-grained approaches and no many-core fine-grained work has been done or reported in this area.

This work proposes new parallel approaches to parallelise and hence speed up the circuit simulation algorithms on highly parallel many-core systems. The main motivations behind this research are first the nature of the proposed approach which can be performed in a very fine-grained and highly parallel way and second the recent developments of massively parallel architectures such as the *Spiking Neural Network Architecture (SpiNNaker)* project. The SpiNNaker project is a massively parallel million-core computer inspired by the structure of the human brain and designed for modelling large-scale spiking neural networks [14, 49]. The availability of such a network is in fact the driving force behind this project and in this work we aim to prepare the ground work for parallelising circuit simulation process using our very fine-grained proposed approaches to be implemented on massively parallel systems in future.

1.5 Research Objectives

The current thesis proposes a new method for speeding up the SPICE simulation process. The method is based on many-core fine-grained parallelisation by allocating one processor to each circuit equation. The circuit equations are distributed across a large number of light-weight parallel processors and evaluated in a completely random (non-deterministic) order. Conventional parallel methods for SPICE simulation try to parallelise the direct matrix solution across a limited number of processors (coarse-grained parallelisation). In this research, the aim is to employ a large number of light-weight processors to perform both the device evaluation and matrix solution phases in a massively parallel form. This not only will speed up the simulation phases by exploiting the benefits of a highly parallel network of processors but also prepares the groundwork to tackle some of the current constraints on totally parallelising SPICE.

The device evaluation is conventionally performed using Newton-Raphson iterations [17, 3]. To avoid the above mentioned barrier between NR iterations and matrix solution phases, we use a *Jacobi-type* iterative approach for the matrix solution process and call it *Random Jacobi* method. At each device evaluation iteration, the linear equations of the

circuit are evaluated on a network of parallel processors independently and in a random order. As soon as a new entry of the unknown vector is calculated, it is passed to the linearisation iteration to perform device evaluation without the need for waiting for the completion of the matrix solution phase. Meanwhile, the processor collects the most up to date values from other processors to start performing the next iteration. Then, the device evaluation phase will produce a more precise linear model of the circuit to be used by the matrix solution process.

The device modelling process can be easily performed concurrently according to its inherent parallelism. Each nonlinear element can be evaluated independently and its equivalent linear model is placed in the corresponding location in the matrix system. The Newton-Raphson method uses numerical evaluation of partial derivatives of nonlinear functions for the linearisation process, which needs a large amount of computational effort. To simplify and accelerate this process on a highly parallel network of processors, we use more simple linearisation techniques such as the Secant method for the device evaluation phase. An in-depth review and discussion on the proposed parallel matrix solution and device modelling methods is done in Chapter 4 and Chapter 5.

In this work, a number of simulations on some benchmark circuits are done to perform the Random Jacobi method on a single-core machine, by *Message Passing Interface (MPI)* [50, 51] under Linux on a single-core machine, and by MPI in a highly parallel form on a cluster of processors.

These simulations were performed to evaluate the functionality of Random Jacobi iterations on many-core systems and the possibility of combining linearisation and Random Jacobi iterations. Although MPI might not be the best parallel platform for highly parallel computing because of the problem of communications overhead, we use MPI to implement the proposed approaches in this work. By taking advantage of the sparsity of circuit simulation matrices, a specific communication pattern is proposed to decrease the amount of required communications. Other parallel computing systems such as *Open Multi-Processing (OpenMP)* [52] are also introduced in the literature review chapter Section 2.4 and the reason behind choosing MPI for this work is explained. By simultaneous evaluation of parallel and iterative matrix solution phases using the proposed methods, it is possible to eliminate the existing barrier between the main two phases of the circuit simulation process and thus speed up the simulation process.

As stated in Section 1.3 of this chapter and will be studied in Section 2.4 of the literature review chapter in more detail, interest in parallel circuit simulation has been increasing during the past decade and a number of studies have been done on parallelising SPICE simulation with the aim of speeding up the process. The motivation of the current work is to eliminate some of the existing constraints of totally parallelising SPICE simulation by implementing new methods based on parallel algorithms on highly parallel networks of processors.

All in all, the main objectives of this work can be defined as follows:

1. Perform a number of preliminary simulations to evaluate the feasibility of the Jacobi iterative method when the equations are solved in a non-deterministic order. Chapter 3 will address this objective.
2. Replace the computationally intensive Newton-Raphson method with a simpler linearisation approach for the device evaluation phase. This objective will be tackled in Chapter 4.
3. Evaluate the Random Jacobi iterations on a highly parallel many-core system by allocating one processor to each circuit equation. This evaluation will be performed in Chapter 5.
4. Find a suitable communication method to optimise the communication between the parallel processors for an efficient data exchange. A communication pattern will be introduced and tested in Chapter 5 to fulfil this objective.
5. Assess the overall speed-up improvement, achieved by the proposed methods, by simultaneous evaluation of the two main phases of the circuit simulation process in Chapter 6.

1.6 Thesis Contribution

Later in Chapter 2 Section 2.4.2 it will be shown that a number of recent attempts to parallelise SPICE simulation process with the aim of increasing the simulation speed have resulted in limited speed-ups. In this research, a new approach is proposed to perform conventional SPICE simulation phases using different techniques based on fine-grained parallel methods with the aim of eliminating some of the existing limitations on the amount of possible parallelisation. Unlike the conventional parallel methods, the proposed parallel evaluation of matrix solution process is performed on a very fine-grained network of processors providing massively parallel processing possibilities. Furthermore, a simpler linearisation method is proposed instead of the conventional Newton-Raphson technique and the possibility of combining device evaluation and matrix solution iterations is studied. Overall, the main contribution of the current research is accelerating the circuit simulation process, which seems to be one of the most important challenges according to the discussed limitations on processors' performance with targeting massively parallel architectures such as SpiNNaker for implementing the proposed parallel methods in future.

The remainder of this thesis is organised in the following manner. ***Chapter 2***, literature review, starts with a brief review on SPICE algorithm and then studies matrix construction methods. Afterwards, the main matrix solution methods along with their

specifications are covered and a discussion is given on the methods which are more suitable for the purpose of this research. In the final part of the literature review, the importance of parallel simulation and the related work in this area are highlighted. Then, related work in the field of parallel SPICE simulation is reviewed, the current challenges are addressed, and the necessity of new approaches to overcome the existing constraints are discussed.

Chapter 3 includes the evaluation of the functionality of the proposed Random Jacobi iterations by performing some preliminary simulations by Matlab on randomly generated matrices to confirm the functionality of the method and identify the effective factors on its performance.

Various device evaluation techniques are investigated and simulated in ***Chapter 4*** to compare the results of the conventional methods and the proposed methods. This helps to find out whether or not the proposed methods are appropriate for our work and worth being employed for the purpose of this work. Advantages and drawbacks of each technique are discussed and the best method for our specific use is chosen based on the important criteria for our massively parallel approach.

The proposed iterative method for matrix solution phase is simulated on a single-core machine, a virtual many-core system, and also a real many-core cluster in ***Chapter 5***. Simulation results are compared and discussed based on a number of important factors such as the number of iterations, error margin, and execution time. Moreover, a new communication pattern is proposed to increase the communication efficiency of parallel computing using MPI.

In ***Chapter 6***, various aspects of mixing the two main phases of simulation, as part of the aims and objectives of this work, is studied in order to perform simultaneous evaluation of the two phases, which is currently one of the issues in parallel circuit simulation algorithms. The circuit equations are evaluated on a fine-grained parallel system of processors and the required device models are calculated on the same processors, when they are required, in order to avoid the undesired barriers between the two phases.
Finally, in ***Chapter 7***, which is the conclusion of this work, a summary of the thesis is given. Then the proposed methods, which are used during the work are briefly reviewed, and the simulation results are discussed. The achievements of the work are listed and possible future work and developments are introduced.

Chapter 2

Literature Review

Circuit analysis is concerned with the formulation and solution of the circuit equations to obtain the node voltages and branch currents of a circuit for a specific excitation. Because of the advancements during the last few decades and the advent of relatively large electronic circuits compared to the past, it is necessary to use computer programs to analyse circuits for higher speed, accuracy, and reliability. The main steps in such computer programs are: describing the circuit and excitations, formulating network equations, solving the equations, and finally displaying the analysis results [17, 18].

SPICE, developed by Electronics Research Laboratory of the University of California, Berkeley (1975), is one of the most powerful general purpose circuit simulation tools to simulate and analyse the behaviour of integrated circuits [20, 53]. SPICE and its variants are widely used in electronic circuit simulation and are capable of performing several types of analysis among which the important ones can be named as nonlinear DC analysis, transient analysis, AC analysis, and noise and sensitivity analysis.

The main steps of a SPICE simulation can be listed as follows [20]:

- *Circuit equation formulation*: The simulation process starts with formulating the circuit description equations using nodal analysis techniques.
- *Modelling time varying elements*: For transient analysis, based on the current time point, a model is calculated for each time varying element by numerical integration techniques.
- *Generating a linear model*: The behaviour of the nonlinear elements in circuit equations is modelled by a linear equivalent using linearisation methods to create a linear system of equations.
- *Solving the linear matrix system*: The resultant linear system of equations is solved by matrix solution techniques to obtain the unknown node voltages and branch currents.

This chapter starts with a brief review of the SPICE algorithm. Then, the equation formulation process and the construction of the circuit description matrix is discussed in more detail. A number of methods to construct the matrix system are studied and their advantages and drawbacks are outlined. Then, automatic equation formulation methods is discussed and element stamps are introduced. Furthermore, different matrix solution methods, their advantages, and disadvantages will be reviewed. A brief summary of the methods will be given and a discussion will be made on the suitability of the reviewed methods. Finally, the importance of parallel and distributed computing and the existing challenges in this area are highlighted and attempts to parallelise SPICE simulation process are reviewed and discussed.

In choosing a suitable matrix solution method, some important aspects such as simplicity, accuracy, speed, and efficiency in terms of cost and storage should be taken into account. For example, in a specific case, a slower method with the possibility of being implemented in parallel may be preferred to a relatively faster sequential method [17, 3, 22].

2.1 The SPICE Algorithm

SPICE simulates the behaviour of electronic circuits by solving the nonlinear differential equations describing the circuit. The simulation process starts with the formulation of a set of nonlinear differential equations representing the contribution of circuit elements by evaluation of KCL at different nodes of the circuit. There are a variety of techniques to assemble the circuit equations into a matrix form. SPICE uses the MNA technique, which has some advantages over other conventional methods such as the nodal analysis and tableau formulation, in handling voltage sources and controlled current sources. This is discussed later in the matrix construction section in more detail [54, 55].

The nonlinear elements must be approximated with a linear equivalent to be able to form a linear matrix system. The reason is that because of the presence of nonlinear elements, the equations cannot be solved analytically and linearisation using numerical techniques are required to express the nonlinear system in the form of an equivalent linear system. The linearisation process is done by NR iterations. Using an initial guess for operating point calculations, the NR method provides a linear approximation for nonlinear elements. Then, by solving the linear matrix system in the form of $Ax = b$ for x, a new more precise operating point is obtained. Repeating the solution of matrix and linearisation of equations at each iteration results in the solution vector x. The stop criteria for NR iterations are checked at each iteration by a process known as iteration control to stop iterations if a desired precision is achieved. An iteration control checks whether all the node voltages and branch currents are within a predefined range for two successive iterations [54, 55].

When the solution is obtained for a specific time point, SPICE recalculates the contribution of time-varying components to form a new set of nonlinear differential equations to simulate the behaviour of the circuit for the next time step as is the case in performing a transient analysis [54, 55].

Figure 2.1 shows a simplified block diagram for the SPICE simulation process. The outer block, the dashed box, represents a complete analysis for one time point including formation of nonlinear differential equations, linearisation of nonlinear elements, and the solution of the linear matrix system. This process is repeated for all time points until the required time range is fully covered. The time step is defined based on several factors and can vary depending on the simulation process [22, 54]. The middle block, solid line box, includes NR iterations for lineralisation and the matrix solution phase which is performed several times at each time step to calculate the unknown x vector. At each NR iteration, when the linear matrix system is generated, there are different methods to solve the linear system of equations. This step is shown with the inner block, represented by the dotted box.

SPICE is still the standard circuit simulation tool for almost 40 years. SPICE uses direct methods for the matrix solution and the NR method for nonlinear device modelling. There have been several attempts to optimise its algorithm, which have accelerated the simulation time. However, the conventional SPICE algorithm has not changed much [22, 54]. The existing attempts for speeding up the SPICE simulation algorithm is reviewed in this chapter and new approaches and algorithms to accelerate the circuit simulation process will be introduced in Chapters 4 to 6.

2.2 Matrix Construction

A typical transient SPICE simulation performs analysis on a number of time steps. Each time step of a transient analysis consists of multiple iterations. Each of these iterations includes two main steps which are evaluating the circuit devices and putting the device models in a matrix form and then solution of the resultant matrix [54]. This section reviews the first step, which is concerned with the construction of the circuit description matrix.

2.2.1 Classical Methods

The simplest methods for formulating the electronic networks are nodal admittance and mesh impedance methods which are based on *Kirchhoff's current Law (KCL)* and *Kirchhoff's Voltage Law (KVL)*, respectively [17, 32]. Nodal analysis was introduced as the topological dual of the mesh analysis but it has become more popular for analysing large electrical systems because of two advantages over the mesh analysis method. Firstly, it

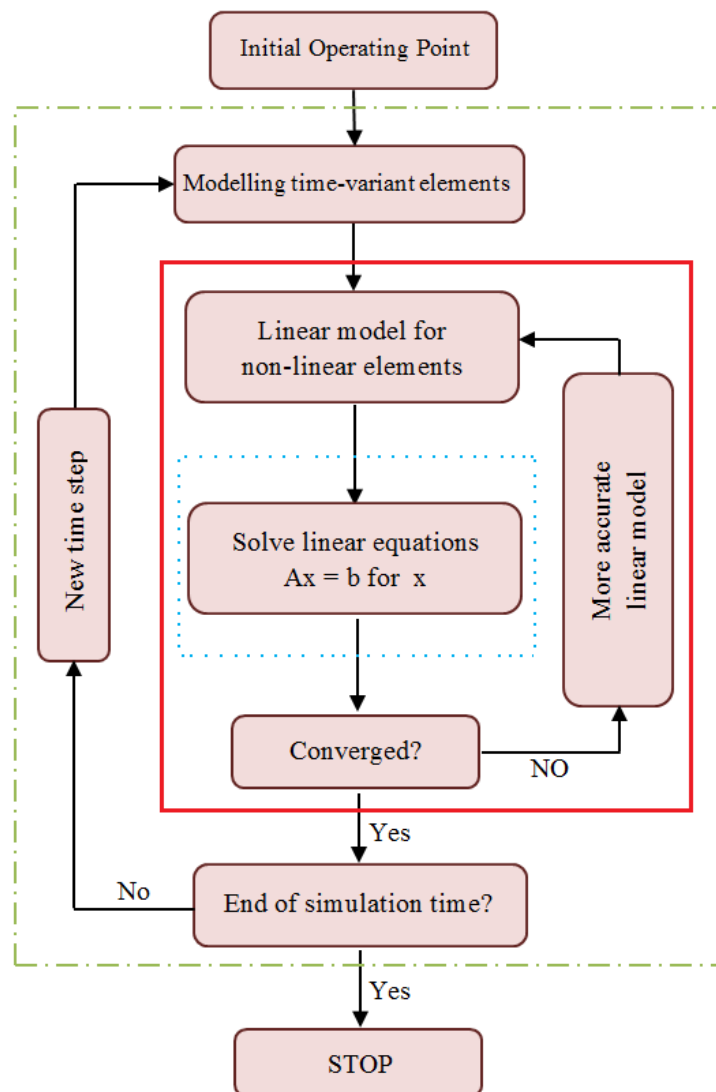


FIGURE 2.1: Simplified block diagram of the SPICE simulation process.

does not have the problem of crossover in non-planar systems. Mesh analysis is only valid for planar systems. A planar network is a network that can be drawn without any element of the network crossing over another element. The algorithms to test planarity of a network and also automatic formulation of meshes are complicated. This is the second reason for the nodal analysis being preferred to the mesh analysis. Although these methods are quite efficient to evaluate simple circuits and have been used in many applications successfully, they are not suitable for some applications and cannot handle all types of elements. For example, classical nodal analysis works based on the sum of currents flowing away from nodes. The problem is that many practical elements such as voltage sources are not expressed in terms of currents. To overcome this problem, it is always possible to use transformations using various theorems such as *Thevenin* and *Norton* transformations or source splitting. However, such transformations are just practical for hand analysis and are not advantageous for computer based analyses. To

avoid these restrictions, general formulation methods such as the tableau formulation and the modified nodal analysis are more of interest [17, 32, 2].

2.2.2 Tableau Formulation

Tableau is the most general formulation method because the solution provides all branch currents, all branch voltages, and all nodal voltages. The problem is that this method leads to a relatively large system of equations and complicated sparse matrix solvers are required to solve it. Tableau formulation needs the concept of graphs and incidence matrices to construct the systems matrix. The size of the system matrix will be twice the number of elements plus the number of ungrounded nodes. For example, for the simple network shown in Figure 2.2, which includes 4 elements and 2 ungrounded nodes, the matrix size will be 10*10 (Figure 2.3). This example points out one of the main difficulties associated with the tableau formulation which leads to a very large system. In nodal formulation, this simple network can be formulated by using just two equations. Another main reason to avoid the tableau formulation in practical applications is the necessity of using graph concepts to generate the system matrix. This makes the formulation process very complicated while there are other methods which offer simpler formulation procedures which are introduced in the next section [2, 56].

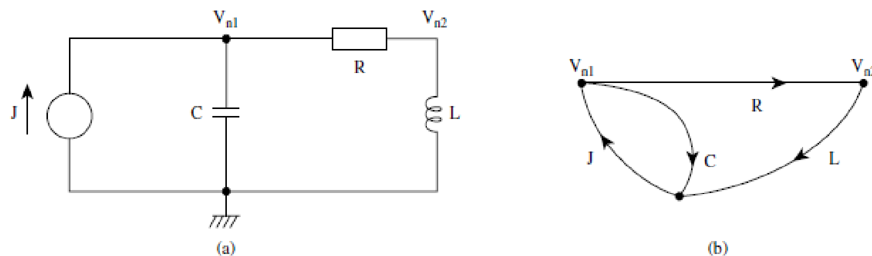


FIGURE 2.2: (a) A simple network; (b) Its graph [2].

2.2.3 Modified Nodal Analysis

The nodal approach for formulating circuit equations meets most of the requirements for an efficient method and therefore has become very popular and is widely used in computer programs. However, in its basic form, it treats some elements such as voltage sources and current dependent elements inefficiently. The modified nodal analysis resolves the mentioned problems in the nodal analysis while preserving its advantages [31].

The first step of formulating the circuit equations for a given network by *MNA* is the same as the basic nodal analysis method. It should be done by applying *KCL* to the ungrounded nodes disregarding the elements that cannot be formulated directly by the

$$
\begin{array}{cccc|cccc|cc}
V_j & V_c & V_R & V_L & I_J & I_C & I_R & I_L & V_{n1} & V_{n2}
\end{array}
$$

$$
\left[\begin{array}{cccc|cccc|cc}
1 & & & & & & & & 1 & \\
 & 1 & & & & & & & -1 & \\
 & & 1 & & & & & & -1 & 1 \\
 & & & 1 & & & & & & -1 \\
\hline
 & & & & 1 & & & & & \\
 & sC & & & & -1 & & & & \\
 & & 1 & & & & R & & & \\
 & & & 1 & & & & sL & & \\
 & & & & -1 & 1 & 1 & & & \\
 & & & & & & -1 & 1 & &
\end{array}\right]
\begin{bmatrix} V_J \\ V_c \\ V_R \\ V_L \\ I_J \\ I_C \\ I_R \\ I_L \\ V_{n1} \\ V_{n2} \end{bmatrix}
=
\begin{bmatrix} \\ \\ \\ \\ \\ J \\ \\ \\ \\ \\ \end{bmatrix}
$$

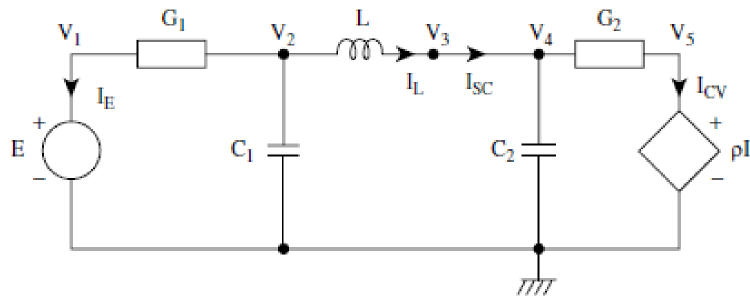
FIGURE 2.3: Tableau formulation for the network in Figure 2.2 [2].

nodal method. Then, for voltage sources and other elements, whose currents are controlling variables, the branch currents are introduced as additional variables and the corresponding branch constitutive relations are considered as additional equations. In this case, these branch currents will be additional output variables. The system matrix for MNA has two parts. The first part is the reduced form of the nodal matrix excluding the contribution of voltage sources, current controlling elements, etc. The second part contains the contribution of those elements which are not included in the first part of the network. For each of these elements, one or more additional rows and columns will be added to the first part (basic nodal matrix). However, it should be emphasised that for most practical circuits, the number of additional variables and equations to introduce voltage sources, inductors, etc. is small compared to the number of nodes. Therefore, the resultant set of variables and equations is large enough to include all the required information and yet small enough to make the formulation efficient [31, 2, 57, 58].

To represent the circuit equations in a computer program, a formulation method is needed to introduce the contribution of each element to the matrix equations one by one. This can be done by using element stamps [17, 3].

An example will show the process of formulating a circuit by MNA and also using element stamps to add the contribution of elements to the system matrix. Consider the network in Figure 2.4. It has no specific application but it is suitable to demonstrate how to set up the modified nodal matrix. An extra node has been added to the network to show the controlling current of the *Current Controlled Voltage Source (CCVS)*.

The network has 5 ungrounded nodes. Thus, the dimension of its nodal part is 5. The voltage source and the inductor will each add another row and column to the system matrix. The CCVS will add two more rows and columns. Altogether, the size of matrix



(a)

$$\begin{bmatrix} G_1 & -G_1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ -G_1 & G_1+sC_1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & sC_2+G_2 & -G_2 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -G_2 & G_2 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & -sL & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -\rho & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ I_E \\ I_L \\ I_{SC} \\ I_{CV} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ E \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(b)

FIGURE 2.4: (a) An example network (b) System matrix set up using MNA [2].

will be 9. At first, the top-left corner 5*5 matrix should be formed disregarding the above-mentioned elements. Then, using the element stamps, the voltage source, the inductor, and the controlled-source are added one by one. The concept of automatic equation formulation using stamps will be reviewed in the next section. To be more clear about the contribution of the elements of this specific example, the stamps for the voltage source, the inductor, and the CCVS are shown in Figure 2.5. This process leads to a 9*9 system matrix while the tableau formulation for the same circuit gives an 18*18 matrix [2].

There are different methods for formulating circuits among which the MNA is preferred. MNA removes all the limitations of the classic nodal method while preserves its advantages. Besides, compared to the tableau formulation, MNA leads to a relatively smaller matrix system with a less complicated approach because unlike the tableau method, it does not need the formation of graphs.

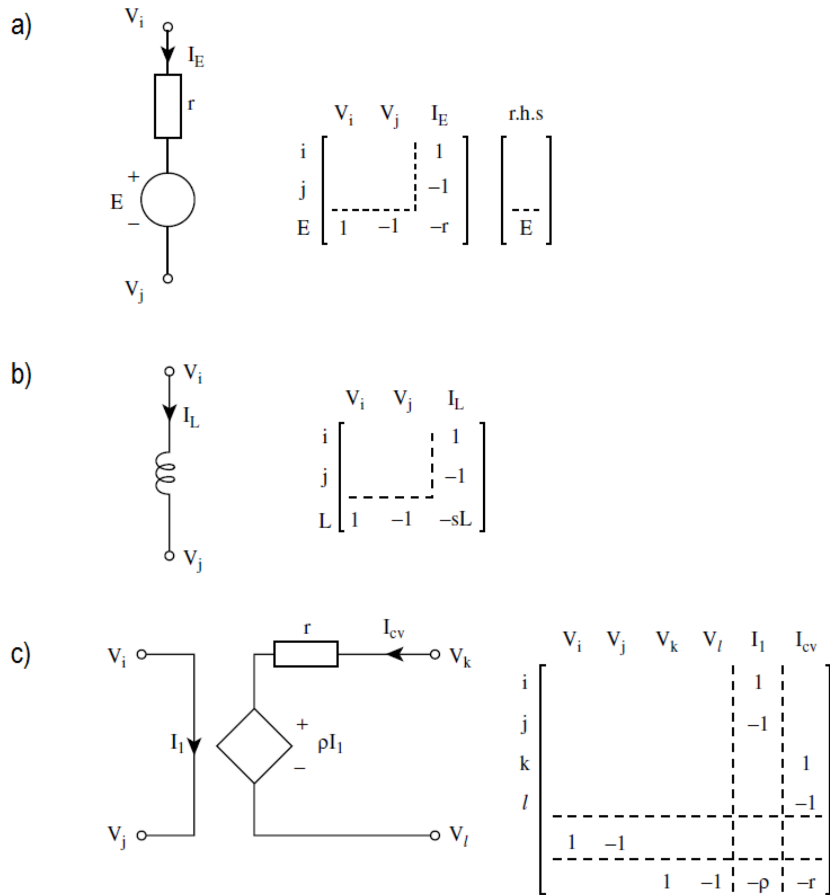


FIGURE 2.5: Stamps for elements of the circuit in Figure 2.4. a) Independent voltage source stamp b) Inductor stamp c) CCVS stamp (the resistor value can be zero) [2].

2.2.3.1 Modelling Nonlinear Elements

For the methods introduced in Section 2.2.3, the Laplace transform is used to explain various formulation methods. For example, for elements such as capacitors and inductors Laplace transforms can be used to express element admittances as $Y_c = sC$ and $Y_L = 1/sL$. When dealing with a linear system and linear elements, it is possible to write the system in a matrix form but in the presence of nonlinear elements, such as diodes and transistors, some restrictions exist. The first issue is that it is not possible to express the characteristics of nonlinear elements in the same symbolic form of linear elements simply by using the Laplace transform. It is possible to write nonlinear equations but it is not possible to cast them into a matrix form. Secondly, a system of nonlinear equations cannot be solved analytically by computer programs. For solving such systems, in most cases, numerical methods are required, which are generally iterative techniques [17, 3].

In general, there are two main techniques to model nonlinear elements: linear approximations and numerical iterative methods. The linear approximation or piecewise linearisation method is normally used when there is no equation for the nonlinear behaviour of

elements and there are only some measured values or the equation is very complicated [17]. This method models the behaviour of nonlinear elements with multiple linear segments. In other words, a set of nodal equations is formed and each equation is linearised separately. The quality of approximation can be improved by increasing the number of segments. However, this complicates the analysis process. The other method is based on numerical iterative techniques such as the Newton-Raphson method. In the Newton-Raphson method, which is most frequently used, the nonlinear system of equations is transformed into a linear system which changes in each iteration. In this method, at first, the Newton-Raphson algorithm is applied to each of the nonlinear elements to produce a linear companion model for each element. Then, equation formulation is performed while nonlinear elements are replaced by their linear companion models. This is done by expanding the nonlinear function by Taylor series and only using the linear terms for the rest of calculations and neglecting the higher orders. This leads to the formation of a Jacobian matrix which includes the first order partial derivatives of the function. This iterative approach is usually referred to as Newton-Raphson iteration and expressed by Equation 2.1, where superscripts denote the iteration number and f and f' stand for the function and its derivative, respectively [17, 3, 33, 59]. Nonlinear device modelling techniques are discussed in more detail along with examples in Chapter 4 Section 4.2.

$$x^{k+1} = x^k + \Delta x^k = -f(x^k)/f'(x^k) \tag{2.1}$$

2.2.3.2 Automatic Equation Formulation

In the first part of this chapter, a review was done on matrix construction techniques such as the Tableau formulation and MNA. The process of MNA is performed by applying KCL to each circuit node by considering node voltages as unknowns and forming the corresponding equations and also writing extra equations for branches which include current dependent elements by considering their currents as extra unknown variables. However, in practice, matrix construction methods such as MNA do not directly form the circuit equations to construct the matrix system. Instead, they start by initialising the A and RHS matrices to zeros and adding the contribution of circuit elements to the matrix one by one. To make the process automated, *element stamps* are introduced. An element stamp is the contribution of a specific element of the circuit to the matrix system which describes the circuit [3, 2, 6].

Assume a conductance between nodes k and j of a circuit as shown in Figure 2.6a. Writing the node equations at nodes k and j leads to Equation 2.2, where Σ_k and Σ_j are the sums of the currents leaving nodes k and j, respectively, other than the conductance current. Therefore, the contribution of conductance G on the matrix system can be formulated as shown in Figure 2.6b. v_k and v_j are the corresponding node voltages. In a similar way, KCL equations for the current source(I_{kj}), which are shown in Figure 2.6c,

are formulated by Equation 2.3. The corresponding stamp for the current source can be seen in Figure 2.6d [3].

$$
\begin{aligned}
\text{node } k : G_{kj}.v_k - G_{kj}.v_j + \Sigma_k = 0 \\
\text{node } j : -G_{kj}.v_k + G_{kj}.v_j + \Sigma_j = 0
\end{aligned}
\tag{2.2}
$$

$$
\begin{aligned}
\text{node } k : I_{kj} + \Sigma_k = 0 \\
\text{node } j : -I_{kj} + \Sigma_j = 0
\end{aligned}
\tag{2.3}
$$

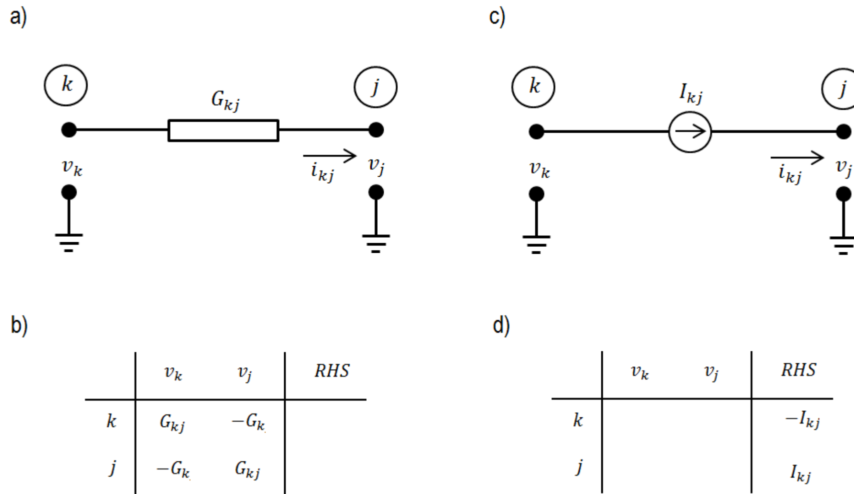


FIGURE 2.6: a) A conductance b) Stamp for the conductance G c) A current source d) Stamp for the current source I [3]

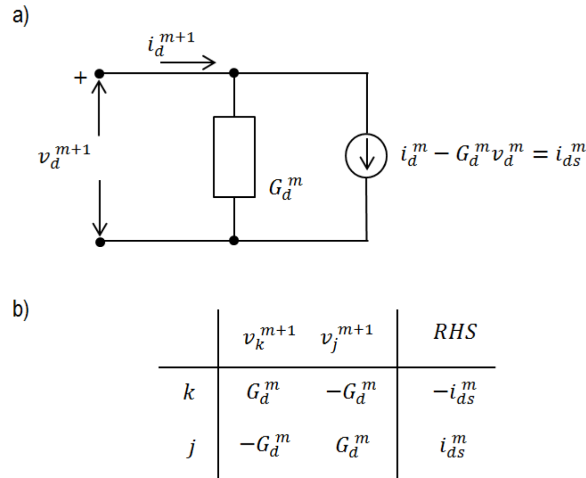
In a similar manner, it is possible to define stamps for other circuit elements e.g. controlled sources, capacitors, inductors, etc. Some circuit elements such as diodes and transistors have more complicated i-v characteristics which makes it more difficult to include their behaviour in circuit equations. In practice, these kinds of elements can be replaced by an equivalent circuit that consists of simple elements such as conductances, capacitors, independent sources, and controlled sources, which is in fact an approximation to the original element. This is called *device modelling* [3].

Diode current can be formulated as shown in Equation 2.4, where i_d is the diode current, I_s is the reverse bias saturation current, v_d is the voltage across the diode, and λ is a constant.

$$
i_d = I_s(e^{\lambda v_d} - 1) \tag{2.4}
$$

It is possible to model the diode by an equivalent circuit that consists of a conductance and a constant current source which will give an approximation of the actual diode, which is assumed to be between nodes k and j of a circuit. A more accurate diode

approximation will also include some capacitors and nonlinear conductances. A simple companion diode model and its corresponding stamp are shown in Figure 2.7. When the element is nonlinear, the equivalent model can be linearised using linearisation techniques such as NR iterations, where the superscript m denotes the iteration number. For example, for a diode, the equivalent linear values, which appear in its stamp, are represented by Equation 2.5 and Equation 2.6 [3]. $\frac{\partial i_d}{\partial v_d}$ is the derivative of the diode current with respect to its voltage.



	v_k^{m+1}	v_j^{m+1}	RHS
k	G_d^m	$-G_d^m$	$-i_{ds}^m$
j	$-G_d^m$	G_d^m	i_{ds}^m

FIGURE 2.7: a) A simplified diode model b) Diode stamp [3]

$$G_d^m = \frac{\partial i_d}{\partial v_d}|_{v_d = v_d^m} \tag{2.5}$$

$$i_{ds}^m = i_d^m - G_d^m v_d^m \tag{2.6}$$

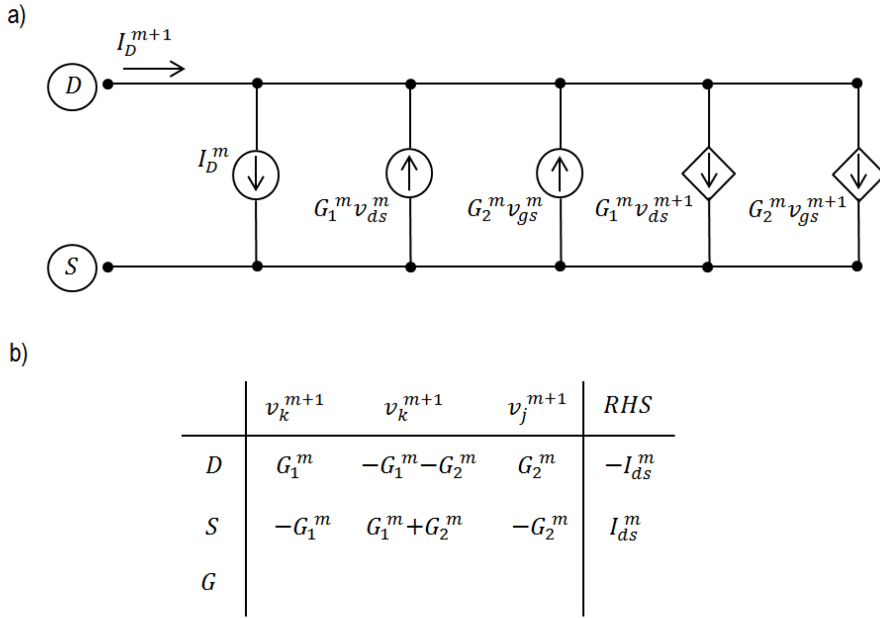
The companion model for a MOS transistor and its stamp are shown in Figure 2.8. The linearised values in the MOS stamp can be calculated by Equation 2.7 to Equation 2.9 [3].

$$G_1^m = \frac{\partial i_d s}{\partial v_{ds}}|_{v_{ds} = v_{ds}^m, v_{gs} = v_{gs}^m} \tag{2.7}$$

$$G_2^m = \frac{\partial i_d s}{\partial v_{gs}}|_{v_{ds} = v_{ds}^m, v_{gs} = v_{gs}^m} \tag{2.8}$$

$$i_{ds}^m = i_d^m - G_1^m v_{ds}^m - G_2^m v_{gs}^m \tag{2.9}$$

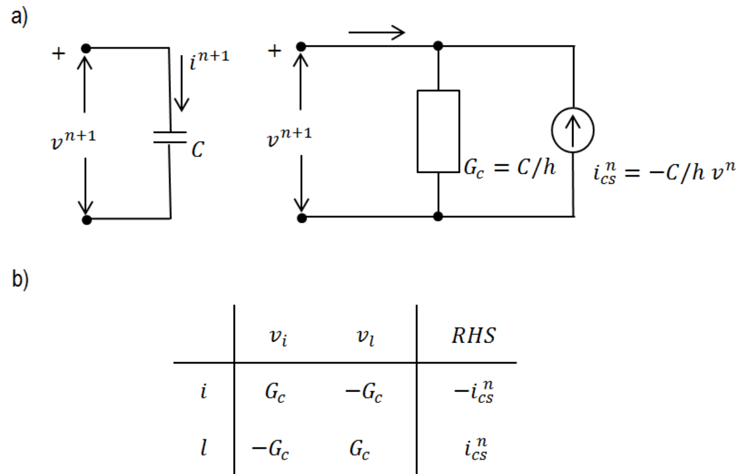
Modelling dynamic elements such as capacitors is slightly different because of the presence of the derivatives of the function in the circuit equations. There are a number of



	v_k^{m+1}	v_k^{m+1}	v_j^{m+1}	RHS
D	G_1^m	$-G_1^m-G_2^m$	G_2^m	$-I_{ds}^m$
S	$-G_1^m$	$G_1^m+G_2^m$	$-G_2^m$	I_{ds}^m
G				

FIGURE 2.8: a) A simplified MOS transistor model b) MOS transistor stamp [3]

integration methods such as the Backward Euler formula to approximate the derivatives. A companion model for a capacitor and its stamp are shown in Figure 2.9. It can be seen that the capacitor can be modelled using a conductance (G_c) and a current source (i_{cs}). The values of the equivalent circuit elements depend on the simulation time point and need to be updated at the beginning of each time point during a transient analysis [3, 6].



	v_i	v_l	RHS
i	G_c	$-G_c$	$-i_{cs}^n$
l	$-G_c$	G_c	i_{cs}^n

FIGURE 2.9: a) Capacitor companion model b) Capacitor stamp [3]

The procedure of dealing with a capacitor (as a time-varying element) in the circuit simulation process can be summarised by the following steps:

- DC operating point of the circuit should be found, excluding the effect of capacitor on the circuit.
- Using integration methods, the differential equation of capacitor should be converted into an algebraic equation at the beginning of each time point during the transient analysis.
- At each time point, the nonlinear system of equations, which also includes the capacitor stamp, should be linearised and solved iteratively until convergence is obtained.
- The results are used to generate a new model for the capacitor to be used in the next time point [17, 3, 6].

Equation 2.10 and Equation 2.11 represent the linear approximation of the differential term using the Backward Euler formula. Depending on the approach, h, which is the time step, can be constant during the simulation or can change adaptively.

$$\dot{x}|_{t=v^{n+1}} = \dot{x}^{n+1} = \frac{x^{n+1} - x^n}{t^{n+1} - t^n} \tag{2.10}$$

$$t^{n+1} - t^n = h \tag{2.11}$$

The procedure of modelling the capacitor by a conductance in parallel with a current source (equivalent elements in companion model) is shown in Equation 2.12. This means that the conductance and current source, which model the capacitor, will have the values stated in Equation 2.13 [3].

$$\begin{aligned} i = C\frac{dv}{dt} \rightarrow \qquad & \dot{v} = \frac{1}{C}i \\ \rightarrow \frac{v^{n+1} - v^n}{h} &= \frac{1}{C}i^{n+1} \\ \rightarrow i^{n+1} &= \frac{C}{h}v^{n+1} - \frac{C}{h}v^n \end{aligned} \tag{2.12}$$

$$G_c = \frac{C}{h}, \quad \text{and} \qquad i_{cs}^m = -\frac{C}{h}v_c^n \tag{2.13}$$

The same procedure applies to modelling other nonlinear and time varying elements in the automatic equation formulation process which is explained and discussed in a number of text books [17, 3, 6].

2.2.3.3 Properties of Circuit Simulation Matrices

When dealing with large electronic circuits one obvious point about the equivalent matrix system is that the resulting matrix of conductances, the A matrix, is very sparse. This is due to the fact that even in very large circuits, each node is only connected to a few adjacent nodes and therefore most of the entries of any row of the conductance matrix is zero [6].

As reviewed in the matrix construction part using the MNA technique, there is an equation for every unknown value (node voltage or branch current) of the circuit. This means that for circuit simulation matrices, the A matrix is also expected to be square. Furthermore, the presence of some elements such as dependent voltage and current sources and elements which produce gain, such as transistors, causes an asymmetric structure for circuit simulation matrices [6]. Therefore, it can generally be said that circuit simulation matrices are expected to be large, square, very sparse, and asymmetric.

In Section 2.2 of this chapter, different matrix construction methods are reviewed, the way of handling nonlinear and time varying elements are introduced, and the step by step procedure of describing an electronic circuit in a matrix form is explained. In the next section, a number of different matrix solution approaches will be reviewed.

2.3 Matrix Solution

When the equations describing a circuit are formulated, they can be written in the matrix form of $Ax = b$ where A is the coefficients matrix, x refers to the vector of unknowns such as node voltages or branch currents, and b (RHS vector) is the excitations vector. There are two main approaches to solve these matrix systems: direct methods and iterative methods. There is a clear distinction between these two methods. Direct methods obtain the exact solution in a finite number of steps while iterative methods use a successive process to approximate the solution over an infinite number of iterations where the exact number of required iterations depends on the required accuracy [17, 3]. In addition, there are some other methods based on combining different matrix solution algorithms in order to use the advantages of methods in different classes, which are called hybrid methods. Later in this chapter, the above mentioned matrix solution methods will be reviewed in more detail.

2.3.1 Direct Methods

This section covers direct matrix solution methods. As it was mentioned in Section 1.2 of the first chapter, direct methods theoretically solve the matrix of equations in a

(predictable) finite number of steps and obtain an exact solution. But, in real applications, this is not always true due to rounding errors. An error made in one step spreads in all the following steps. In other words, even if a unique solution exists, numerical direct methods can fail to obtain the solution if the number of variables is large, because rounding errors can accumulate and lead to a wrong solution. In this case, there will be two alternative ways: using iterative methods (which are discussed in the next section), or using matrix decomposition methods. Decompositions provide a numerically stable method to solve a system of linear equations. In fact, they transform the problems, which are nearly singular, to non-singular ones. Some most frequently used matrix decomposition methods are *Cholesky*, *QR*, *LU*, and *SVD* [60]. Solving a system of equations using decomposition methods can also be classified as a direct method. The other problem associated with direct methods is the solution time. Large scale problems, which include thousands of equations and unknowns, can be very time demanding to solve by standard direct methods. In electronic circuits analysis, the coefficient matrix, A, is usually asymmetric and very sparse. Therefore, suitable sparse matrix strategies are needed to handle them. Otherwise, direct methods may face serious problems when dealing with large sparse systems [34, 60].

2.3.1.1 Classical Gaussian Elimination

Gaussian elimination in its basic form transforms a general system of equations into an upper triangular system. This process is called the forward elimination. Then, the resultant system can be solved by the backward substitution process. For an n^*n matrix, the elimination procedure consists of $n-1$ steps. At the kth step, an appropriate multiple of the kth equation is subtracted from all of the equations below it one by one so that zero elements are introduced below the diagonal element in the kth column. Figure 2.10 shows a schematic diagram for $n = 4$ [4].

FIGURE 2.10: Gaussian elimination process for a 4*4 matrix [4].

As shown in Figure 2.10, the process in each step starts with the diagonal element in the corresponding equation, which is called the pivotal element or pivot. It is clear that the elimination process can be continued unless one of the diagonal elements is zero, in

which case it will break down. According to the elimination procedure, the Gaussian elimination breaks down at the kth step if and only if the leading principal $k{*}k$ minor of A is singular. For square matrices, if the matrix that corresponds to a principal minor is a quadratic upper-left part of the larger matrix, then the principal minor is called a leading principal minor. Hence, the Gaussian elimination will not break down for some special matrices such as row diagonally dominant, column diagonally dominant, and positive-definite matrices. It can be shown that the forward elimination and backward substitution processes require $n^3/3$ and $n^2/2$ operations, respectively. Thus, the Gaussian elimination is an $O(n^3)$ algorithm (the number of required operations is of the order of n^3) [6]. A general form of a matrix system undergoing the Gaussian elimination process in shown in Figure 2.11. One of the disadvantages of this method is the accumulation of errors. Roundoff errors are built up during the successive subtractions and accumulated in X_n. Then, the error will be magnified during the backward substitution process especially if there are small values on the diagonal which cause very large values generated by division.

There is a developed form of the Gaussian elimination, which is called the *Gauss-Jordan (GJ)* elimination. Generally, there are some elementary matrix operations e.g. interchanging rows, multiplying a row by a non-zero number, and adding a linear combination of rows to the others. GJ is a technique that applies some of the above mentioned operations to the matrix of equations. As it is represented in Figure 2.12a, in this method, an *Identity Matrix (I)* of the same size of A matrix is added to the matrix system and will be affected by the same operations which is done on the A matrix. Then, by using the elementary row operations matrix A is transformed into the diagonal form shown in Figure 2.12b. A final step of dividing each row of the system by a'_{ii} will convert the A matrix to an Identity Matrix [6, 5].

a)

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \cdot \\ \cdot \\ c_n \end{pmatrix}$$

b)

$$\begin{pmatrix} 1 & a'_{12} & \cdots & a'_{1n} \\ 0 & 1 & \cdots & a'_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} c'_1 \\ c'_2 \\ \cdot \\ \cdot \\ c'_n \end{pmatrix}$$

c)

$$\left.\begin{aligned} X_n &= c'_n \\ X_i &= c'_i - \sum_{j=i+1}^{n} a'_{ij}\, X_j \end{aligned}\right\}$$

FIGURE 2.11: Gaussian elimination: a) The original matrix system b) Forward elimination applied c) Backward substitution [5].

By the end of this process, the c_i vector will become the solution of the system and the original Identity Matrix, whose elements were shown by b_{ij}, will become the inverse of

a)

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ . & . & & . \\ . & . & & . \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ . \\ . \\ c_n \end{pmatrix} \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ . & . & & . \\ . & . & & . \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

b)

$$\begin{pmatrix} a'_{11} & 0 & \cdots & 0 \\ 0 & a'_{22} & \cdots & 0 \\ . & . & & . \\ . & . & & . \\ 0 & 0 & \cdots & a'_{nn} \end{pmatrix} \begin{pmatrix} c'_1 \\ c'_2 \\ . \\ . \\ c'_n \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ . & . & & . \\ . & . & & . \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

FIGURE 2.12: Gaussian-Jordan elimination: a) The original matrix system along with the added Identity matrix b) Matrix of coefficients is converted to a diagonal matrix by the elementary row operations [5].

the A matrix. Although this method also generates the inverse of the A matrix, which is not really needed for some applications, it requires more operations than the Gaussian elimination. The GJ elimination also suffers from rounding errors especially generated by division when the matrix is close to singular. When the diagonal element is very small, division will generate a very large row element. Subtracting that large element from the remaining rows will lead to a significant roundoff error. Thus, most elimination techniques will include a search and reordering process to find and replace the largest possible elements on the diagonal prior to division to decrease the effect of the roundoff error [4, 5].

The process by which the rows or both rows and columns of a matrix are interchanged in order to put a suitable matrix element in place of the current diagonal element is called *pivoting*. GJ without pivoting does not interchange rows or columns and just multiplies or adds rows. Like the classical Gaussian elimination, the procedure will fail if the pivot element is zero or becomes unstable if the diagonal element is nearly zero. By using pivoting, the process becomes stable. Although there are various techniques to choose a suitable pivot, the largest element of the corresponding row is usually a very good option. The Gaussian elimination can solve systems with multiple RHS vectors but when it is necessary to solve the same system for a new RHS vector, the whole elimination process should be repeated again. Therefore, in practical applications, other direct methods such as LU-factorisation, which do not manipulate the RHS vector, are preferred [4, 5].

2.3.1.2 LU-factorisation

There are a number of different matrix factorisation or decomposition techniques for the solution of linear equations. *LU-factorisation* (sometimes called LU-decomposition) is

a widely used direct method in circuit simulation applications which is reviewed in this section. For the general form of a linear systems of equations, $Ax = b$, by assuming that there is a lower triangular matrix L and an upper triangular matrix U so that $A = LU$, and diagonal entries of L and U are all non-zeros, the system of equations can be written as shown in Figure 2.13 and solved in the following manner. The system can be rewritten in the new form of $Ax = LUx = b$ using L and U matrices. Then, by setting $y = Ux$, y must satisfy $Ly = b$. L is a lower triangular matrix and y can be obtained easily using forward substitution. After finding y, $Ux = y$ can be solved using backward substitution to find the solution [6, 61].

Therefore LU-factorisation method consists of three steps:

- Factorisation (of the order of $n^3/3$ operations)
- Forward substitution (of the order of $n^2/2$ operations)
- Backward-substitution (of the order of $n^2/2$ operations)

and it can be seen that, like Gaussian elimination, *LU-factorisation* is an $O(n^3)$ method [6].

$$A = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix} = LU$$

FIGURE 2.13: LU-factorisation of a square matrix, A [6].

A very important advantage of LU-factorisation compared to Gaussian elimination is that once the coefficient matrix of a system is factorised for the first time, for new RHS vectors, the solution can be obtained only by forward and backward substitutions and there is no need for factorisation. This would be, for example, true for AC analysis in circuit simulation but not valid for some other types of simulations such as transient analysis. It should be noted that there are a number of different approaches for the factorisation part to obtain the L and U matrices and all of them are of the order of $n^3/3$ operations.

Stability of direct methods such as Gaussian elimination (and LU-factorisaion as a variant of it) significantly depends on the form of the A matrix and especially the diagonal values. The general strategy in practical applications is to use pivoting to prevent the matrix entries becoming too large. There are two main options to choose the pivot element a_{kk}: Partial pivoting ($O(n^2)$), which uses the largest element of only the kth column as the pivot and Full pivoting ($O(n^3)$) which searches the remaining matrix from

k to n for the largest absolute value to replace it by the diagonal element as a pivot. Full pivoting is computationally expensive and is not normally required.

Large electronic circuits are very sparse because of the fact that every element is just connected to a few nodes and most of the entries of the A matrix in circuit simulation are zero. Therefore, the simulation process can be accelerated by using *sparse matrix techniques* for storage, pivoting, etc [6, 62]. From the computational point of view, LU-factorisation can be considered as one the most important parts of the conventional circuit simulation process, which is widely used nowadays. When dealing with sparse matrices, the elimination process during the factorisation can create non-zero elements at the positions of originally zero entries. These non-zero entries, known as fill-ins, increase the memory requirements and also the required computational effort. The number of fill-ins during the factorisation process can be reduced using the Markowitz Criterion, which selects suitable pivots to minimise the creation of non-zero entries [63, 64]. The KLU factorisation method is another variation of the LU-factorisation method, which is developed for sparse matrices [65].

2.3.2 Iterative Methods

In Section 2.3.1, direct matrix solution methods are covered. These methods provide a solution to the system of linear equations within a finite number of steps generally of the order of n^3. The term *iterative method* refers to the techniques that approximate the solution of a linear system over a successive process and obtain more accurate results at each iteration. Iterative methods do not guarantee to yield a solution for all systems of equations but when they converge to an answer, it is usually less expensive than direct methods [35]. Comparing iterative methods with direct methods, one may ask why a method that cannot calculate the exact solution is sometimes preferred to one that obtains the exact solution. The answer is that in some applications, iterative methods are easier to implement on high performance computers and more suitable for large-scale problems. The level of accuracy in iterative methods will depend on the number of iterations. However, for iterative methods, the number of required operations per iteration is of the order of n^2. For very large systems of equations, iterative methods can be much faster if they can converge to the solution within a reasonable number of iterations for the required accuracy [35, 5].

Iterative methods are mainly classified into two main groups: stationary and non-stationary methods, but there are some other methods which are a combination of both. Stationary methods are older, easy to understand, and easy to implement but normally not as effective as non-stationary methods, which are a recent development, and relatively harder to implement [35, 66, 67].

The structure of the coefficient matrix e.g. being diagonally dominant has a remarkable impact on the convergence rate of an iterative method. Therefore, in most of the cases, a second matrix called the pre-conditioner matrix, is used to transform the coefficient matrix to the one with more desirable structure. Although the pre-conditioning process includes some extra costs, a good pre-conditioner can effectively improve the convergence rate of an iterative method [35, 68, 69].

2.3.2.1 Stationary Iterative Methods

Iterative methods that can be expressed in the form of Equation 2.14 are called stationary iterative methods. B and c are constants and do not depend on the iteration counter, k. In order to solve $Ax = b$, iterative methods calculate a sequence of approximate solutions $x^{(0)}, x^{(1)}, ..., x^{(k)}$ so that $x^{(k)}$ can be obtained using $x^{(k-1)}$ [35, 7].

$$x^{(k)} = Bx^{(k-1)} + c \tag{2.14}$$

The main stationary methods are *Gauss-Jacobi*, *Gauss-Seidel*, and *Successive Over Relaxation (SOR)* as a variant of the Gauss-Seidel method. Before studying each of the methods individually, it should be mentioned that the starting point for all of these methods is to write the matrix A in the extended form of Equation 2.15 where L and U are strictly lower and upper triangular matrices and D is a diagonal matrix with no zeros on the diagonal. Since A is non-singular, this condition is always achievable by some row reordering. It should be noted that the L and U matrices have zero diagonals and should not to be confused with the L and U matrices of LU-factorisation [35, 7].

$$A = L + D + U \quad \Rightarrow \quad (L + D + U)x = b \tag{2.15}$$

Gauss-Jacobi Method: By rewriting Equation 2.15 in the form of Equation 2.16, the Gauss-Jacobi method can be expressed by Equation 2.17 in which k indicates the iteration counter.

$$x^{(k+1)} = D^{-1}b - D^{-1}(L + U)x^{(k)}, \quad k = 0, 1, \ldots \tag{2.16}$$

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}), \quad i = 0, 1, \ldots, n \tag{2.17}$$

In the Jacobi method, the equations are examined independently and the x vector, which is obtained in the kth, iteration is used in the next iteration. For this reason, the Jacobi

method is known as a method of simultaneous displacement which makes it possible to analyse equations separately in a parallel way.

Gauss-Seidel Method: By rewriting Equation 2.15 in the form of Equation 2.18, it is possible to represent the Gauss-Seidel method by Equation 2.19, in which k indicates the iteration counter [35, 7].

$$Dx^{(k+1)} = b - Lx^{(k+1)} - Ux^{(k)}, \quad k = 0, 1, \dots \tag{2.18}$$

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}), \quad i = 0, 1, \dots, n \tag{2.19}$$

Unlike the Jacobi method for which the equations can be examined simultaneously, in the Gauss-Seidel method, the equations are examined just one at a time and each component of the new iteration depends on all the previously computed components. Therefore, the Gauss-Seidel method is a sequential algorithm. On the other hand, once a new value is obtained, it is immediately used for the next computations. This provides relatively faster convergence for the Gauss-Seidel method compared to the Jacobi method [35, 8, 70].

The Jacobi and Gauss-Seidel algorithms are shown in Figure 2.14.

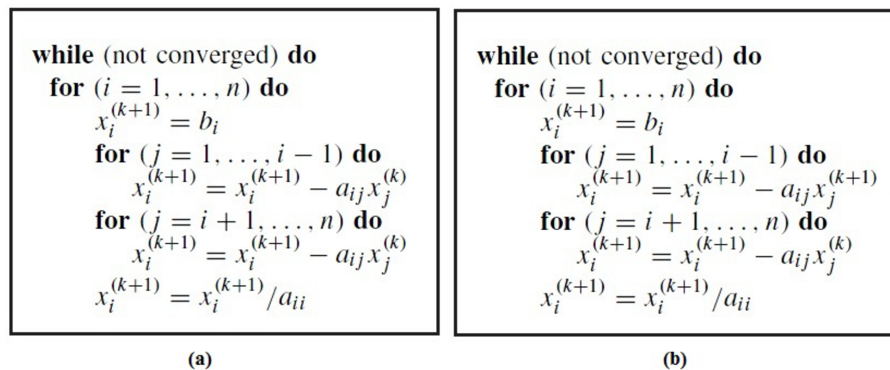


FIGURE 2.14: (a) The Gauss-Jacobi algorithm (b) The Gauss-Seidel algorithm [7].

SOR Method: The *SOR* method is in fact an extended version of the Gauss-Seidel method. It uses a relaxation factor, $\omega \in (0, 2)$, to manipulate the convergence rate. In addition, in cases where the Gauss-Seidel method does not converge, by a proper choice of ω it is possible to make it converge. However, an extra pre-computation to find a suitable ω is required and finding an optimal value for ω is not always possible [7, 70].

2.3.2.2 Non-stationary Iterative Methods

In stationary iterative methods presented by Equation 2.14 in a simple form, the constant coefficients do not depend on iterations while in non-stationary methods, computations involve some information that changes at each iteration. Non-stationary iterative methods are relatively difficult to implement compared to stationary methods. The need for multiple iteration vectors makes it difficult to apply them to large systems. However, they often offer faster convergence. It should also be noted that most of the non-stationary methods are applicable only to symmetric positive definite systems. There are a few developments of them applicable to asymmetric systems but they also try to transform the system to a symmetric problem and solve it [8, 70]. Therefore, these methods are not suitable for the purpose of this work on circuit simulation matrices, which are asymmetric.

2.3.3 Hybrid Methods

Different methods for solving linear equation systems vary in different aspects such as simplicity, ease of implementation, amount of required computations, storage, convergence rate, accuracy, etc. and it is very important to choose a suitable method that works efficiently for a given problem. Sometimes, a combination of methods, called a *hybrid method*, is used for specific problems in order to take advantage of different methods. For example, [8] has introduced a hybrid method which is a combination of two iterative methods. Although it is not related to the circuit simulation field, it clearly shows that how combining two methods can increase the efficiency.

In the above mentioned paper [8], the convergence rate of two different iterative methods are represented and analysed separately. The first method is a stationary iterative method called *Sparse Iterative Method (SIM)* which is a Jacobi-type iterative method. An adaptive relaxation method is used to modify *SIM* to achieve more convergence rate and numerical stability. The second method is a non-stationary iterative method called *Bi-Conjugate Gradient Stabilized (BiCGSTAB)*. The rate of convergence for this method has been improved by pre-conditioning the coefficient matrix and it has been shown that a suitable pre-conditioner for this method is the matrix which is used in the *SIM* method [8].

It is shown that, in general, stationary methods have a faster initial convergence rate which slows down when approaching the accurate solution. On the other hand, the non-stationary technique benefits from a linear convergence rate. A hybrid technique is then introduced which at initial steps uses the fast convergence rate of stationary methods and after a pre-defined number of iterations switches to the non-stationary method. The introduced techniques have been applied to two different systems and the

results have been given [8]. Their simulation results of analysing methods individually and then applying a hybrid method to one of the systems are shown in Figure 2.15 [8].

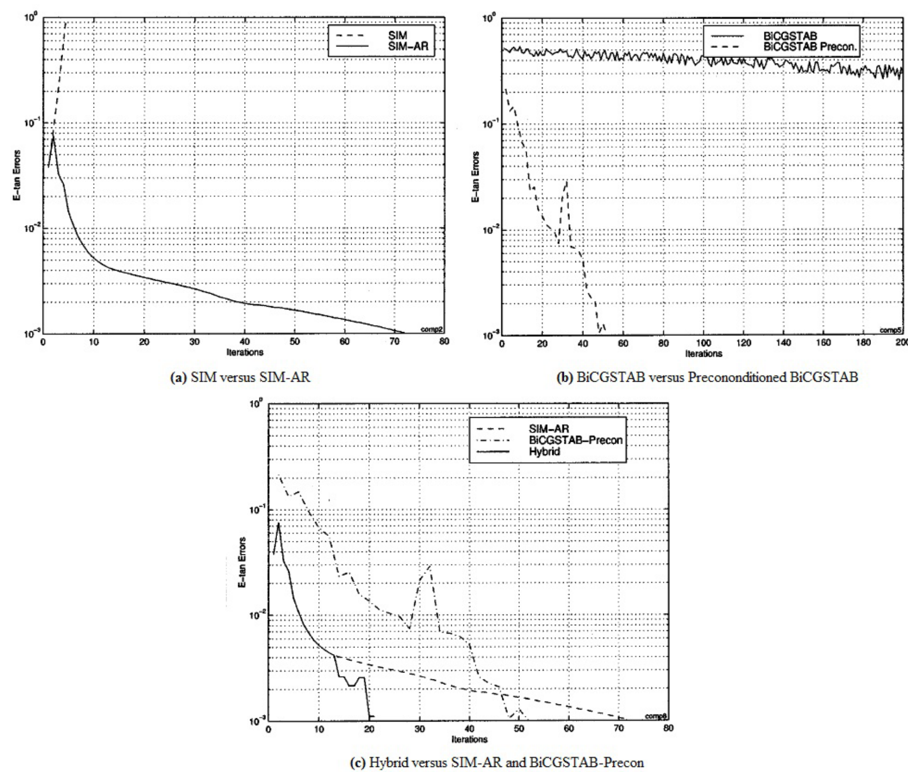


FIGURE 2.15: (a) SIM method versus modifies SIM using adaptive relaxation (b) BiCGSTAB method versus pre-conditioned BiCDSTAB (c) SIM-AR versus BiCGSTAB-precon [8].

Figure 2.15.a shows that the *SIM* method does not converge for this case but applying an adaptive relaxation scheme makes it converge. It can be seen that the convergence rate of the *Sparse Iterative Method with Adaptive Relaxation (SIM-AR)* method is quite fast during the first ten iterations but after that is slows down. In Figure 2.15.b it is represented that using a suitable pre-conditioner matrix in the Pre-conditioned *Bi-Conjugate Gradient Stabilized (BiCGSTAB-precon)* method, leads to a substantial improvement in the convergence rate of the *BiCGSTAB* method. It can also be noticed that except for a few iterations, the convergence rate is almost linear. Figure 2.15.c shows the simulation result of applying the hybrid method to the same system. The hybrid method initially starts with the *SIM-AR* method and after a pre-defined number of iterations, which in this case is around 12-13 iterations, switches to the *BiCGSTAB-precon* method. It is obvious that the hybrid method obtains the result with a considerably faster convergence rate compared to the other two methods [8].

2.3.4 Discussion on Matrix Solution Methods

Matrix solution methods are mainly classified in two categories: direct methods and iterative methods. Each of these categories consists of several different approaches, which have their own advantages and disadvantages. It is not possible to prefer one of the approaches over the other one and different applications may choose either of them as a suitable approach. In general, the pros and cons of direct and iterative methods can be represented as follows [6, 70]. It is assumed that the matrices are non-singular.

Direct Methods:

- Pros
  1. The solution (if it exists) can always be found. When the coefficient matrix is close to singular, the direct solution process may be affected by generation of large values and cannot obtain the correct solution.
  2. There is a fixed number of operations to find the solution and the order of operations is generally of $O(n^3)$.
  3. The obtained solution is exact (neglecting roundoff errors).
- Cons
  1. Accumulation of roundoff errors may cause big errors that can prevent the process to find the solution.
  2. Not suitable for large sparse matrices. The number of operations and growth of roundoff error increases as the size of the problem becomes larger.
  3. The solution process cannot be manipulated by the user. There are a fixed number of operations that need to be completed before getting to the solution.

Iterative Methods:

- Pros
  1. Roundoff error is not as problematic as in direct method cases because generally the iterations are only approximations of the exact solution and roundoff errors only affect the convergence speed not the quality of approximation.
  2. Suitable for large sparse systems. The total required operations per iteration is of the order of $O(n^2)$ and the sparsity of the matrix will simplify the calculations per iteration.
  3. The user can intervene in the solution process. This can be done for example by using a relaxation factor or deciding when to stop the iterations to obtain an approximation of the solution.

- Cons
    1. The solution process may not converge in cases where the coefficient matrix is not strictly diagonally dominant.
    2. The solution is an approximation. To get closer to the exact solution, more iterations are required.
    3. The number of required iterations is unknown and depends on the desired accuracy.

It should be noted that for some methods such as Jacobi-type algorithms, according to the nature of their algorithm, a high degree of parallelisation is possible. Nowadays, the availability of parallel computing machines has given the opportunity to researchers to focus on parallel methods in order to speed up the solution process. In the parallel computing area, there are also different approaches based on different parallelisation methods and parallel machines architectures. Depending on the type of problem and the desired specifications, one can use a synchronous or asynchronous architecture, shared or local memory, a small or large number of processors, etc. Using a suitable parallelisation technique for the problems with a high possibility of being parallelised can possibly lead to a considerable speed-up compared to the sequential systems.

In the next chapter, the concentration will be more on Jacobi-type iterative methods and their parallel evaluations on many-core machines.

2.4 Parallel Circuit Simulation

2.4.1 Parallel and Distributed Computing

Traditional sequential computers are designed to work based on a single CPU. The problem is divided into a series of instructions and only one instruction is executed at a time. By recent technological progress, very large-scale problems have been raised. Thus, faster machines and more efficient methods are required to be able to solve them within a reasonable time [10].

Nowadays, parallel and distributed computing is an area of interest for researchers because of its promising results to speed up the evaluation process for large-scale problems. The trends in the past 20 years for the application and efficiency of parallel computing, shows a bright future for parallelism. Basically, parallel computing methods break the problem into small parts that can be analysed simultaneously using several CPUs working in parallel (Figure 2.16). There are different approaches for parallelising problems such as parallelising the existing serial algorithms or trying to implement the algorithms on multi-core or many-core machines. However, there are some challenges related to parallel computing which do not exist in traditional serial methods and machines [70, 9].

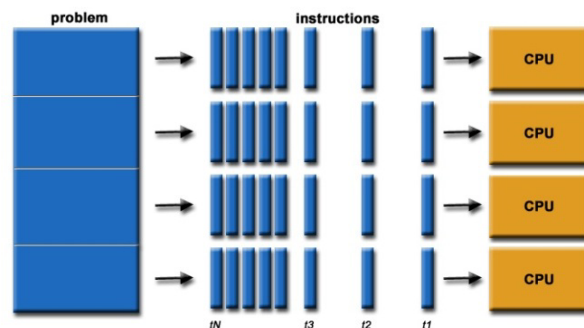


FIGURE 2.16: Parallelisation; distribution of a problem over several CPUs [9].

2.4.1.1 Parallelisation Issues

Although parallelisation can lead to a considerable speed-up in analysing very large-scale problems, there are some issues related to implementation of parallel methods on parallel computers which make the design and analysis process relatively difficult compared to the serial context [70, 10].

- *Task allocation:* the first issue is the process of dividing the bigger problem into smaller tasks and distributing them quite equally (load balancing) over different processors.
- *Communication:* when using several processors in parallel, a processor sometimes requires to send/receive intermediate computational results to/from one or more other processors. This communication between processors needs to be done in a way that does not affect the efficiency of the whole process.
- *Synchronisation:* another issue is synchronisation of the computational results obtained by different processors. There are two main classifications here. In some methods, processors work synchronously which means that there are some predefined time points based on system's clock on which the computations of processors are completed and some intermediate results are available. The problem is that, a processor may need to wait for the arrival of specific data from other processors. This waiting time, can affect the performance of the computation. Some other methods work on an asynchronous basis. In this case, it is not required for a processor to wait at some predetermined time points to get data from other processors. However, implementation of such algorithms is relatively more difficult [70, 10].

2.4.1.2 Parallel Computing Systems

To classify a parallel computer, there are several aspects that should be considered. In this section, some important parameters for describing a parallel computer are reviewed briefly [70, 10, 71].

- *Processors:* type and number of processors are different in different computing systems. Some of these systems have thousands of small processors while in some others there are just a small number of processors (order of 10) which are relatively more powerful.
- *Control:* almost all of parallel computing systems have a sort of central control but with different level of controlling. In some systems, the control mechanism just loads the program and data to processors and then processors are quite independent to work on their tasks. In some other systems, processors are controlled with a high level of details and receive step by step instructions from the central controlling system.
- *Synchronous or asynchronous operation:* in synchronous operations there is a global clock which synchronises the operation of processors but some systems use an asynchronous method independent of system's clock.
- *Interconnection and memory organisation:* an important aspect of a parallel computer is the method a processor uses to communicate with other processors in order to exchange data.

According to these specifications, parallel computers are broadly classified into two categories. In the first category, the system benefits from a shared memory. All processors have access to the shared memory and are able to read from or write to the memory. The problem arises when two or several processors try to read or write at the same time. This can be solved by switching systems which, roughly speaking, define some orders of access to the shared memory for processors (Figure 2.17a). This can lead to a longer access time by increasing the number of processors. A good example for this class can be *Open Multi-Processing (OpenMP)* which is an *Application Programming Interface (API)* that supports multi-platform shared memory multiprocessing programming in *C*, *C++*, and Fortran, on most processor architectures and operating systems [52]. In the second category, which is called distributed memory, processors have their own local memory. In this system, processors have to communicate by sending or receiving messages to other processors in order to have access to their local memories (Figure 2.17b). If the communication is performed based on the system's clock, the method is called systolic. If the processors perform their activities according to the messages they receive disregarding the global clock, the method is called MPI, which is an asynchronous operation [70, 10].

In MPI, data is passed (sent/received) using different function calls and there are several communication topologies such as send, receive, broadcast, scatter, gather, etc. to transfer data between two or more processors [50, 51, 72].

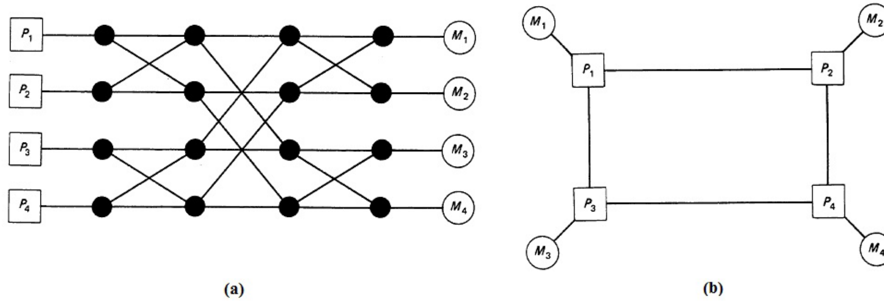


FIGURE 2.17: Processors communication for (a) shared (b) distributed memory organisation [10].

2.4.1.3 Speed-up

One may think that when a problem is distributed for example on p processors, the speed of computation process should increase by a factor of p. It cannot be correct because it is not possible to parallelise the entire problem. Usually a fraction of the whole work can be performed in parallel. Furthermore, as mentioned in Section 2.4.1.1, there are some factors in parallel computing which affect the efficiency of the system.

By using Amdahl's law [73, 74], it is possible to calculate the speed-up from the sequential system to the parallel system. According to this law, the speed-up (S_p) for p processors is shown by Equation 2.20 where f_p is the fraction of work which can be carried out in parallel.

$$S_p = \frac{p}{f_p + (1 - f_p)p} = \frac{1}{(1 - f_p) + \frac{f_p}{p}} \tag{2.20}$$

It can be seen that, if f_p is small, the speed-up is not considerable. While, for bigger values of f_p (close to 1) a better speed-up can be expected [70, 75].

2.5 Parallel SPICE

The SPICE algorithm is intrinsically sequential. By the rapid technology progress in large scale electronic circuits design and very large and complex systems being introduced, circuit simulation process is becoming more and more time demanding. Therefore, speeding up the SPICE algorithm has been a point of interest for researchers during the last decades which has led to new approaches and techniques for parallelising the

SPICE algorithm with the aim of accelerating the simulation process. These works have focused on different parts of the SPICE algorithm and targeted a range of issues related to parallelising the SPICE simulation process. As will be reviewed later in this section, there have been a number of different attempts to parallelise SPICE [12, 30, 38, 76]. Although these works have made improvements and speed-ups in different parts of the simulation process, still there are some constraints on totally parallelising the SPICE algorithm.

In this work, we focus on one of the issues which limits the parallelisation of the SPICE algorithm. One of the problems associated with parallelisation of the circuit simulation algorithm is the presence of nonlinear elements in electronic circuits. When dealing with nonlinear elements, there are two distinctive phases. The first phase is linearising the nonlinear elements and equations and the second phase is solving the matrix equation. Linearisation process is normally done iteratively using the *Newton-Raphson* method. After converting the nonlinear equations to linear ones, the matrix equation describing the system should be solved. Linearisation process (device evaluation phase) for each element can be performed separately in a parallel way. This step must complete before the matrix solving process can start. The next device evaluation phase must start after the previous matrix solving phase is completely done. This creates two barriers between device evaluation and matrix solution phases, which limit the amount of parallelisation as shown in Figure 2.18 [11].

In the next section, there will be a review of the existing work on parallelising the SPICE simulation process, the existing limitaions and issues will be addressed, and the proposed methods to overcome some of the problems will be introduced to be discussed and investigated in more detail in the next chapters.

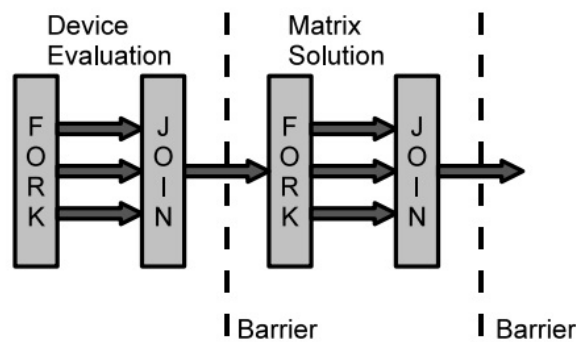


FIGURE 2.18: Barriers between device evaluation and matrix solution phases [11].

2.5.1 Existing Work

Circuit simulation is a computationally intensive process especially when it comes to the design and verification of *Very Large Scale Integration (VLSI)* circuits. Although the

availability of faster processors and development of different simulation tools accelerate the simulation process, there is a continuous demand for higher speed and accuracy. In order to keep up with the trend of increasing size and complexity of electronic circuits and device models, electronic circuit designers try to deliver high performance simulation tools. During the last few years, the clock frequency of CPUs saturated around 3.5 GHz and it seems that single processors and sequential algorithms are no longer able to keep up with the advancements in electronic circuits design. Therefore, parallel simulation has become a point of interest for circuit designers [24, 38].

Although during the recent years the need for parallel circuit simulation has become more obvious, it has been an interesting topic for researchers even a few decades ago. There were early attempts in the 80s and 90s to develop parallel circuit simulation algorithms on a single-core or multiple cores [77, 78, 79, 80]. For example, in 1988, a parallel circuit simulator was implemented which used a multiprocessor computer with shared memory to perform parallel circuit simulation by using direct methods and partitioning the system matrix. They reported around 4 to 7 times speedup for their test circuits compared to single processor simulations [77]. Another example focuses on the model evaluation phase and highlights the fact that device evaluation is a computationally intensive task and can be parallelised to speed up the simulation process. A simple formula is introduced to assess the cost of loading models to the system matrix and balance the load across multiple processors. A speedup of almost $3x$ is reported for model evaluation using their proposed methods [80].

In recent years, several attempts have been reported to parallelise the device evaluation and/or matrix solution processes and a variety of algorithms and hardware designs have been proposed for this purpose on multi-core CPUs, GPUs, and FPGAs [29, 12, 30, 38, 42, 11]. WavePipe is a coarse-grained parallelism approach which simultaneously calculates the circuit solution in multiple adjacent time points using multiple threads. It works based on two proposed methods called backward and forward pipelining. Backward pipelining performs extra calculations by moving backwards along the time points to provide larger future time steps. Forward pipelining performs predictive computing in the direction of forward time steps. Simulation results on some benchmark circuits show around 2 times speed-up [29]. Some other works have focused on parallelising the matrix solution phase on FPGAs by parallelising the KLU matrix solver as a direct method to solve a linear matrix of equations [12]. The device evaluation phase is a computationally intensive part of the SPICE simulation process because each device should be evaluated separately and this needs to be repeated at the beginning of each time step. A parallel approach for device evaluation phase has been proposed by [30] to exploit the natural parallelism of the independent evaluation of each device to accelerate the device evaluation phase using GPUs. Multi-algorithm parallel simulation and multi-thread simulation using OpenMP are among other attempts to parallelise some parts of the SPICE simulation process [42, 11].

There is also more recent work within the last few years on parallel circuit simulation which addresses some new challenges in this area and shows the existing interest on this topic [81, 45, 82, 83]. A development of a parallel circuit simulation algorithm on a GPU has been done in [81]. This work highlights the challenges related to using shared-memory in parallel simulations, proposes a partitioning based approach on distributed-memory, and shows limited speed-ups on multi-core systems [81]. In another study, a very specifically designed simulator for massively repeated small circuits is represented which is suitable for simulating systems with repetitive patterns such as SRAMs [45]. Some very recent works are also introduced on parallel circuit simulation on FPGAs and GPUs [46, 84]. However, most of the recent work also concentrates on speeding up the direct matrix solution methods such as LU-factorisation by either improving the algorithm itself and making it more efficient for sparse matrices or by parallelising the direct matrix solution approaches on multi-core systems such as GPUs and FPGAs [65, 84]. To the best of our knowledge, there is no recent work on parallelising circuit simulation algorithms using new approaches rather than improving the conventional simulator. This can be because of the limited speedups than can be achieved using the current parallel architectures and all the recent attempts are trying to utilise the available multi-core/many-core platforms. As it was pointed out in the Research Motivation section (Section 1.3), the main driving force behind the current work is the availability of massively parallel platforms such as SpiNNaker which motivated us to work on new approaches based on very fine-grained Jacobi-type iterative solution to replace the direct matrix solver which is being claimed to be the bottleneck of the parallel circuit simulation [81, 84].

Although there is an ongoing interest in parallel circuit simulation and there have been a number of very recent attempts in this area, the existing works mainly concentrate on one of the simulation phases, device evaluation or matrix solution. As discussed, since device evaluation is easier to parallelise, most of the current research is on matrix solution and various parallel implementations of direct matrix solutions. In this work, a Jacobi-type iterative method is used for the matrix solution phase in order to use the benefits of parallel matrix solution phase and also iterative solution to overcome some of the existing challenges. The proposed method is based on non-deterministic evaluation of Jacobi iterations on highly parallel systems. Apart from recent attempts on parallel circuit simulation, there are also attempts on speeding up iterative numerical solutions such as Jacobi iterations on highly distributed systems. The most recent work among them reports an implementation of asynchronous Jacobi method on parallel systems using MPI, OpenMP, SHMEM [85]. Although this is not in parallel circuit simulation field and also simplifies the general Jacobi solver by making some assumptions to avoid dealing with an explicit conductances matrix (A), the concept is very close to a part of the work used in this thesis for parallel circuit simulation on highly distributed systems using Jacobi-type iterative approaches for matrix solution phase.

2.5.2 Parallel SPICE Simulation Challenges

As reviewed in the previous section, there are a number of different attempts and research on parallelising circuit simulation process. Although all of them report speed-ups by employing new approaches, there are still limitations on totally parallelising the simulation process by focusing on both of the main simulation phases. Besides some existing issues, such as the barrier between the main simulation phases, have not been solved yet. Circuit simulation is an iterative process which has two main phases: device evaluation to model and linearise nonlinear elements and matrix solution to solve the linear equations describing the circuit.

The model evaluation phase is very straightforward to parallelise due to the inherent parallelism of independent evaluation of each element. However, conventional circuit simulation tools perform this by the Newton-Raphson method which involves calculation of partial derivatives. In order to create linear models for nonlinear devices, at each NR iteration, partial derivatives of nonlinear equations needs to be calculated using numerical integration methods which demands a high amount of computations [12].

Unlike the device evaluation phase, matrix solution is very difficult to parallelise due to the asymmetric and irregular structure of circuit simulation matrices. Circuit simulation tools such as SPICE use direct methods for the matrix solution phase. As introduced earlier in Section 2.4.2.1, a number of attempts have been made to parallelise the matrix solution phase of SPICE simulation by using different approaches such as partitioning, block simulation, parallel multi-algorithms, etc. However, parallelising this stage is one of the main bottlenecks of the simulation process [12].

Apart from the issues related to each simulation phase, there is another important challenge connected to parallel circuit simulation which has remained unsolved and severely limits the total parallelisation of the SPICE algorithm. At each time point of the circuit analysis process, the model evaluation and matrix solution phases need to be performed several times until the solution is obtained with the desired accuracy. Therefore, matrix solution cannot start until device evaluation has finished and the next device evaluation cannot begin until the matrix solution phase has obtained a solution. Although the device evaluation and matrix solution phases have been parallelised individually, the above mentioned gap between the two simulation phases prevents the complete parallelisation of the simulation process [11].

The current work aims to contribute to the solution of these challenges by proposing new approaches to each of the main solution phases and then using the advantages of these approaches to prepare the ground for totally parallelising the SPICE algorithm by removing some of the existing constraints.

2.6 Summary

This literature review chapter started with a brief review of the SPICE algorithm. Matrix construction techniques and automatic equation formulation were studied. There are a number of different methods for matrix construction among which MNA is widely used in circuit simulation tools because it not only leads to a smaller matrix size but also provides solutions for some unsolved issues existing in other matrix construction techniques. Then, the two main approaches for matrix solution (direct solutions and iterative methods) were reviewed and the advantages and drawbacks of each method and their variants were introduced.

In the last section of this chapter, parallel circuit simulation was reviewed. Several attempts at parallelising circuit simulation process were addressed. Although these works present a number of different approaches to parallelise the simulation process, which have led to considerable speed-ups, there are still some unsolved issues which are not addressed in the literature.

The current work targets some of these existing challenges and proposes new approaches for performing the two main simulation phases and also removing some of the existing constraints on totally parallelising the circuit simulation process. The main idea is using a parallel and iterative matrix solution method in conjunction with parallel model evaluation techniques to perform the two phases simultaneously on a highly parallel network of light-weight processors. An in depth study of the proposed methods along with the simulation results on benchmark circuits is done in the next chapters.

Chapter 3

Random Jacobi Iterations

SPICE-like algorithms use direct methods such as LU-factorisation for the matrix solution phase. As discussed in Chapter 2 Section 2.3.4, direct and iterative matrix solution methods have their own advantages and drawbacks. Iterative methods, in some cases, fail to converge to the correct solution. However, for diagonally dominant matrices, iterative methods converge to a solution and total operations required per iteration is of the order of n^2. On the other hand, direct solutions face problems handling large-scale and sparse matrices. Direct methods such as LU-factorisation have order n^3 operations. Therefore, if the number of iterations required for an iterative solution is much less than n, iterative solutions will be computationally less expensive. Besides, for large sparse matrices, the L and U factors can become dense because some of the zeros might be replaced by non-zero entries during the factorisation process. Apart from memory requirements, computations of triangular L and U systems also become costly.

Another issue associated with the direct matrix solution process in SPICE simulations is the undesired barrier between the device evaluation and matrix solving phases. This is because a direct solution generates the solution vector as a whole at the end of the solution process. Therefore, the next linearisation process cannot start until the matrix solution process is completely done. At each NR iteration, the nonlinear circuit model is linearised then the system of linear equations is solved. The matrix solution part also cannot start before the device evaluation process is completed. These gaps limit the amount of parallel work in the simulation process [35, 70, 11].

By using a Jacobi-type iterative method and evaluating the circuit equations in a parallel way and completely random (non-deterministic) order, it is possible to provide the NR iterations with the new entries of the unknown vector as soon as a new entry is calculated. The reason is that in Jacobi-type iterative methods each row of the matrix system can be evaluated independent of the other rows. Therefore, the Jacobi method is very easy to parallelise.

In this chapter, first, a number of test matrices are generated for our simulations. Then, the non-deterministic evaluation of the Jacobi iterative method is reviewed and simulations are performed using Matlab. All Matlab simulations in this work are done using the software version 7.11.0.584($R2010b$). on the test matrices, which are specifically generated for the purpose of our preliminary investigations. *Gauss-Seidel*, normal *Jacobi*, and *Random Jacobi* algorithms are applied to the test matrices. As briefly mentioned in the introduction chapter Section 1.4, we call our proposed iterative matrix solution approach the *Random Jacobi* method because it evaluates the equations in a non-deterministic (random) order independently on a large number of parallel processors. However, in this chapter the parallel Random Jacobi method is simulated on a single-core to investigate its functionality first. Moreover, the effects of randomness of the execution order of Jacobi iterations and some parameters which may affect its efficiency have been studied.

3.1 Generating Test Matrices for the Preliminary Simulations

To investigate the functionality of the Random Jacobi iterations in comparison with the Gauss-Seidel and normal Jacobi iterations, we generated some test matrices specifically for the purpose of our preliminary evaluations. All these test matrices are square, asymmetric, and sparse(as it is expected from a circuit simulation matrix) and are listed in Table 3.1 along with their specifications.

Matrix name	Size	Sparsity
$MTX0020$	20	80%
$MTX0050$	50	90%
$MTX0100$	100	90%
$MTX0500$	500	95%
$MTX1000$	1000	90%

TABLE 3.1: Test matrices for preliminary simulations.

To generate these test matrices, first the required criteria for our matrix is defined such as its size, sparsity, values range, etc. The sparsity rate is equal to the number of zero elements divided by the number of all elements of the matrix. A bigger sparsity percentage means a more sparse matrix.Then, a matrix with the desires size is generated with random values between 0 and 1 for its elements. This is done using the $rand(n)$ function of Matlab which returns an n-by-n matrix of random numbers [86]. The elements of the matrix is shifted by 0.5 unit and multiplied by a pre-defined range coefficient to generate negative values within the desired range. The obtained matrix is multiplied (point to point) into another matrix with the same size which only has 0 and 1 elements with the required sparsity rate. This will generate a sparse matrix within the required range and with the desired sparsity. Finally, the diagonal elements are modified to be bigger

than the other elements of the same row. A final check is done to make sure that the matrix is diagonally dominant, complies with the needed sparsity, and converges to the solution. The relevant Matlab codes are included in Appendix A, Section A.1.

The test matrices are not strictly diagonally dominant but the absolute value of the diagonal element is bigger than all of the other elements in the corresponding row. Although the test matrices are not strictly diagonally dominant, they have been tested to make sure that by applying iterative methods they converge to a solution and do not diverge.

3.2 Applying Iterative Algorithms to Test Matrices

In this section, the functionality of the Random Jacobi iteration method is examined and its efficiency is compared to the Gauss-Seidel and normal Jacobi methods. Simulations are performed using Matlab.

3.2.1 Convergence Comparison

In iterative solutions, as one of the stop criteria, the *Euclidean norm* is compared to a pre-defined threshold at each iteration to stop the iterations as soon as the desired accuracy is obtained. In our preliminary simulations, the simulations are performed for a fixed number of iterations in order to check the convergence but in the main simulations the Euclidean norm will be used as one of the stop criteria. Equation 3.1 shows calculation of the Euclidean norm in which $solVect$ is the solution vector, $xVect$ represents the vector of unknowns calculated at each iteration, and $eNorm$ is the Euclidean norm between these two vectors. The Euclidean norm or Euclidean distance between two vectors shows how close the two vectors are and is obtained by calculating the square root of the sum of the square of the differences between all the corresponding elements of the two vectors. In order to check the convergence of the solution vector to the correct solution, the Euclidean norm is calculated at each iteration as a convergence measure.

$$eNorm = \| solVect - xVect \| = \sqrt{\sum_{i=1}^{n} (solVect_i - xVect_i)^2} \tag{3.1}$$

Figure 3.1 shows the simulation results of applying the iterative methods on 4 test matrices with different sizes ranging from 50 to 1000. The initial guess vector is the same for all of the simulations and equal to a vector of all *ones*. The x axis represents the number of iterations and the y axis stands for the Euclidean norm between the correct solution vector, and the answer which is obtained at each iteration.

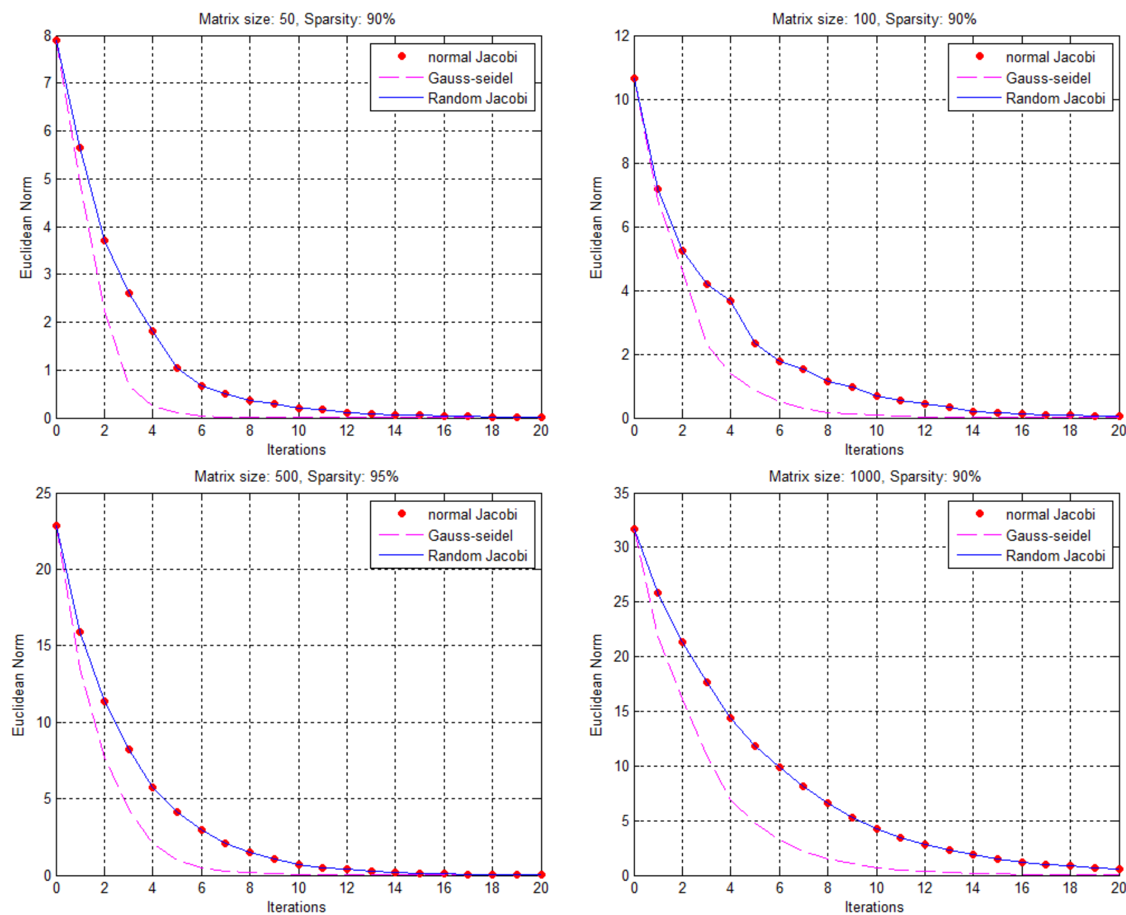


FIGURE 3.1: Comparing the convergence speed of Gauss-Seidel, normal Jacobi, and Random Jacobi iterations for four different test matrices.

For the simulation results in Figure 3.1, the algorithm by which the Random Jacobi method has been performed evaluates the matrix equations in a random order at each iteration so that each equation is calculated only once. At the beginning of the Random Jacobi iterations, a random vector with the size equal to the number of rows of the A matrix is generated. The equations evaluation order is based on this random vector. In fact, in this case, it is the same as normal Jacobi iterations and the only difference is in the order of evaluation of the equations. Since, unlike the Gauss-Seidel method, the solution vector is updated at the end of each iteration in Jacobi-type methods, it is expected that the Random Jacobi and Normal Jacobi methods obtain the same solutions within the same number of iterations. However, the Gauss-Seidel algorithm updates the solution vector right after solving each equation, which leads to a faster convergence. The Matlab code for the algorithm of each method is included in Appendix A, Section A.2.

It can be seen in Figure 3.1 that the Gauss-Seidel method converges faster because of updating the solution vector several times at each iteration (every time an equation is solved). However, this makes Gauss-Seidel a sequential algorithm, which cannot be

implemented in parallel. On the other hand, although the Jacobi method is slower, it can be easily performed in parallel due to the fact that each equation is solved independently and the update is done at the end of each iteration. For the same reason, the order of evaluation is important in the Gauss-Seidel method while it does not affect the solution in the Jacobi-type methods.

Figure 3.2 shows the performance of Gauss-Seidel and Jacobi methods on a test matrix for three separate simulations in which the order of evaluation of the equations is different each time. As represented, the convergence pattern is the same for the Jacobi method for the three different simulations, as the corresponding graphs follow identical patterns, while changing the order of the evaluation of equations affects the convergence rate of the Gauss-Seidel method. Therefore, when using the Gauss-Seidel method, the solution depends on the order in which equations are evaluated and this is because of the existing dependency between the equations.

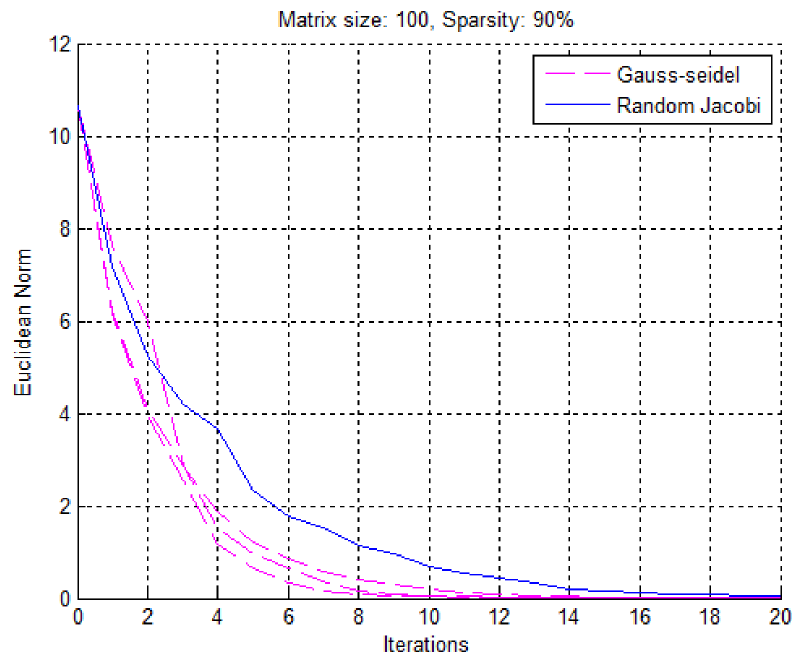


FIGURE 3.2: The effect of the order of the evaluation of equations on the convergence of Gauss-Seidel and Jacobi methods. For each method, the simulation was repeated for three times.

3.2.2 Effects of Parallel and Non-deterministic Evaluation

In Section 3.2.1, it was shown that the normal Jacobi and Random Jacobi methods behave identically when applied to the test matrices. However, that was the case in which a random vector, which includes all the row numbers, was used to determine the evaluation order and each equation was evaluated only once per iteration. When performing Jacobi iterations on many-core systems for solving the matrix equations in

parallel, the situation is quite different. In this section, we simulate the functionality of parallel and non-deterministic evaluation of matrix equations when the equations are solved by Jacobi iterations and investigate the effect of this random evaluation on the convergence of the iterations in a number of different situations.

In the following case studies, the simulation results of different situations which may occur during non-deterministic evaluation of the Jacobi iterations on parallel systems are shown. The graphs for normal Jacobi and Gauss-Seidel methods are the same as previous simulations and are represented only for comparison purposes. These case studies aim to highlight some issues and also benefits concerned with parallel Random Jacobi iterations on a highly parallel network of processors. It should be noted that these issues can occur for any other simulation methods and are not specific to Random Jacobi iterations.

When performing parallel Jacobi iterations on many-core systems, failure of a processor causes the corresponding row of the matrix not to be evaluated. This generates a constant error when calculating the Euclidean norm. As the number of failures increases, the value of the Euclidean norm becomes bigger and bigger due to more equations not being evaluated. This might prevent the system to converge to the correct solution if the stop criteria are not satisfied because of the error developed by the failure of one or more processors. This problem can occur for any parallel system of processors and suggests that appropriate fault detection and correction methods need to be employed. However, this work mostly concentrates on the functionality of the proposed methods in the preliminary simulations.

Figure 3.3 shows the effect of the failure of processors on the convergence of a test matrix of size 100. When all the processors work properly, convergence of the Random Jacobi is the same as normal Jacobi. In Random Jacobi iterations, when some processors fail, a constant error can be seen in the value of the Euclidean norm because the corresponding rows are not calculated and updated with new values for the unknown vector. For this specific example, when one row, two rows, and three rows do not update, the final Euclidean norm values are 0.52, 1.21, and 1.60, respectively while it is 0.01 for normal operation of the Random Jacobi method for a fixed number of iterations equal to 25.

Lack of communication between processors can also affect convergence of the Random Jacobi iterations when they are evaluated using distributed memory many-core systems. In this case, one or more processors may not be able to provide a new entry for the unknown vector in some of iterations due to poor communications, delay in calculating new values, etc. Figure 3.4 shows cases in which some of the processors do not update at some of the iterations. In other words, a particular processor may not update a new entry in a specific iteration because of, for example, communication issues but in other iterations it works properly. As can be seen in Figure 3.4, although improper update of some processors decreases the convergence speed, unlike the case of failure in

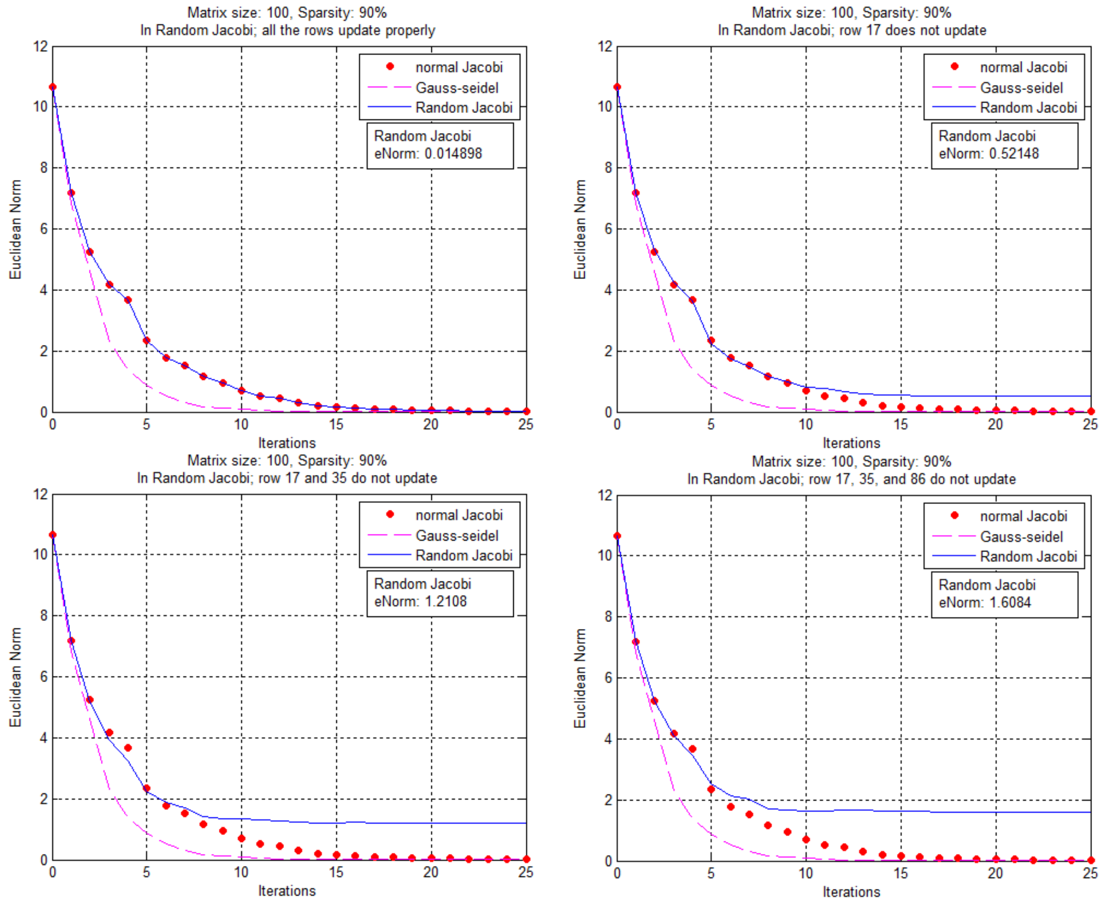


FIGURE 3.3: The effect of failure of some processors on the convergence of the Random Jacobi method.

processors (Figure 3.3), the system converges to the solution without a constant error in the Euclidean norm.

In the simulations shown in Figure 3.4, the processors that do not update are chosen randomly at each iteration and the effect of improper update of two, four, and six processors on the convergence of the Random Jacobi method is represented. Figure 3.5a, shows five different evaluations of Random Jacobi iterations on the 100*100 test matrix. For this particular example, when the stop criterion for the convergence of the methods is defined as *Euclidean norm* < 0.2, the number of iterations for each method to converge is shown in Figure 3.5b. The Gauss-Seidel method converges in eight iterations, the normal Jacobi method converges in 14 iterations, and the number of iterations for the Random Jacobi method has an average of 20 iterations for five different simulations. The number of iterations for the Random Jacobi method in this case varies between 17 and 23 and depends on the rows which are not updated because of the failure. It should be noted that, in this example, the Gauss-Seidel and normal Jacobi iterations operate without any fault in processors but the Random Jacobi method experiences some faults in the parallel working processors at each iteration.

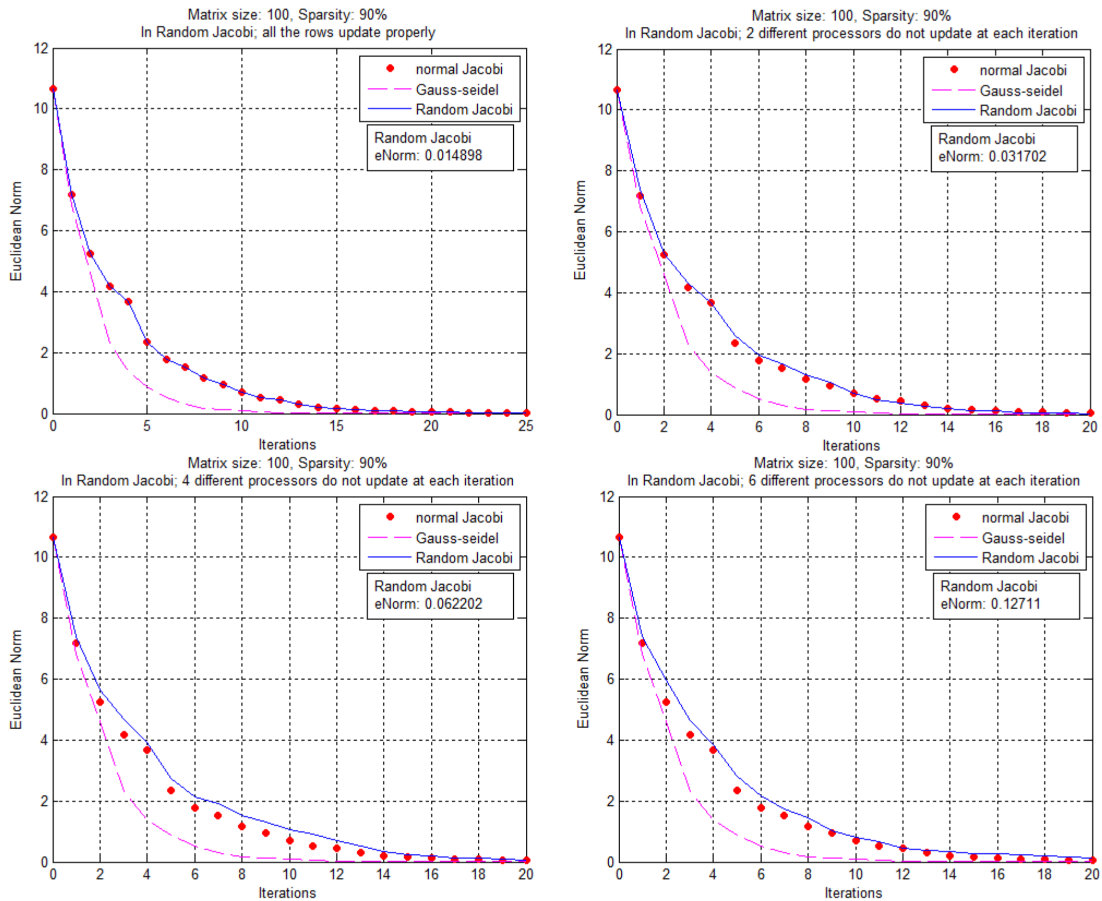


FIGURE 3.4: The effect of improper update of some processors on the convergence of the Random Jacobi method.

The reason for the Gauss-Seidel method being faster than the Jacobi method (converges with fewer iterations) is that it solves equations one by one in a sequential order and updates the solution vector immediately after solving each equation. However, the Jacobi method updates the solution vector after all the equations are evaluated (which can be done in parallel). When performing Jacobi iterations in a parallel and non-deterministic order by independent evaluation of the equations on separate cores, some cores may complete their tasks sooner than the others and have time to update their values more than once in one iteration. This makes it closer to the Gauss-Seidel case and as represented in Figure 3.6, as the number of the cores that update more than once at each iteration increases, the number of iterations required for convergence decreases. For example, the Jacobi method converges to the solution with 17 iterations for the example matrix in Figure 3.6. When 25% of the processors update more frequently, the number of iterations decreases to 16 and for 75% of the processors with more frequent updates, it reaches to twelve iterations, which is very close to ten iterations of the Gauss-Seidel method.

The results show that modelling the parallel evaluation of the Random Jacobi method on many-core parallel systems converges to the solution even if a few number of processors

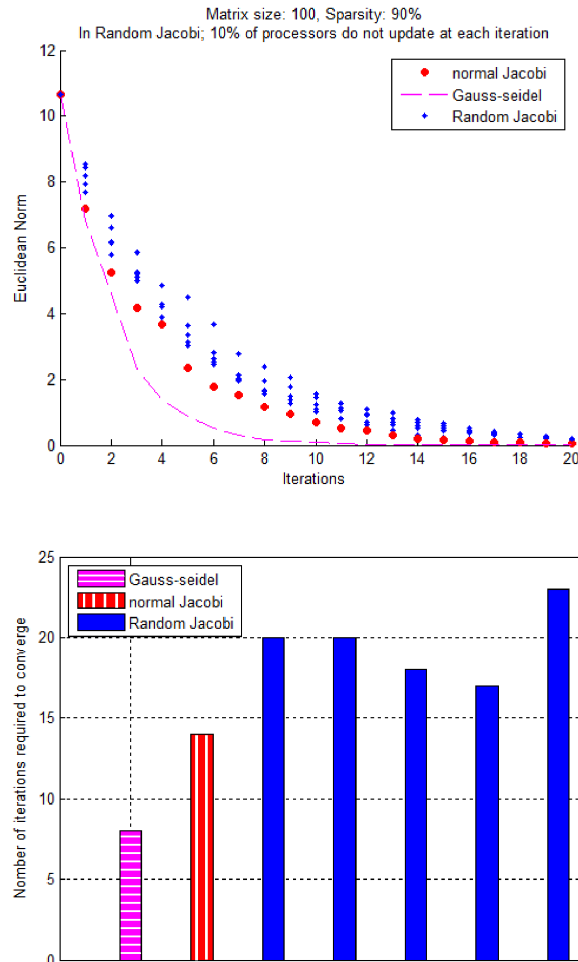


FIGURE 3.5: Convergence of Random Jacobi iterations when 10% of the processors do not update at each iteration.

do not update properly in some iterations. However, failure of one or more processors causes a constant error which might prevent the system converging to the solution. Thus, this case is a more serious issue to be taken into consideration when evaluating Random Jacobi on real many-core systems. On the other hand, parallel evaluation of the Jacobi method can lead to a faster convergence if properly implemented using a suitable communication pattern to update their solution as soon as it is ready.

3.2.3 Initial Guess Vector

In this section, the effect of choosing a suitable initial guess vector has been simulated on the 100*100 test matrix for four different initial guess vectors. Figure 3.7 shows the convergence graphs for the Gauss-Seidel, normal Jacobi, and Random Jacobi for initial guess vectors of all 0s, all 1s, all 4s, and all -4s. It should be noted that for this specific matrix, the values of the solution vector are in the range of -1 and 1. A more precise distribution of the solution vector is represented in Figure 3.8, which shows that most

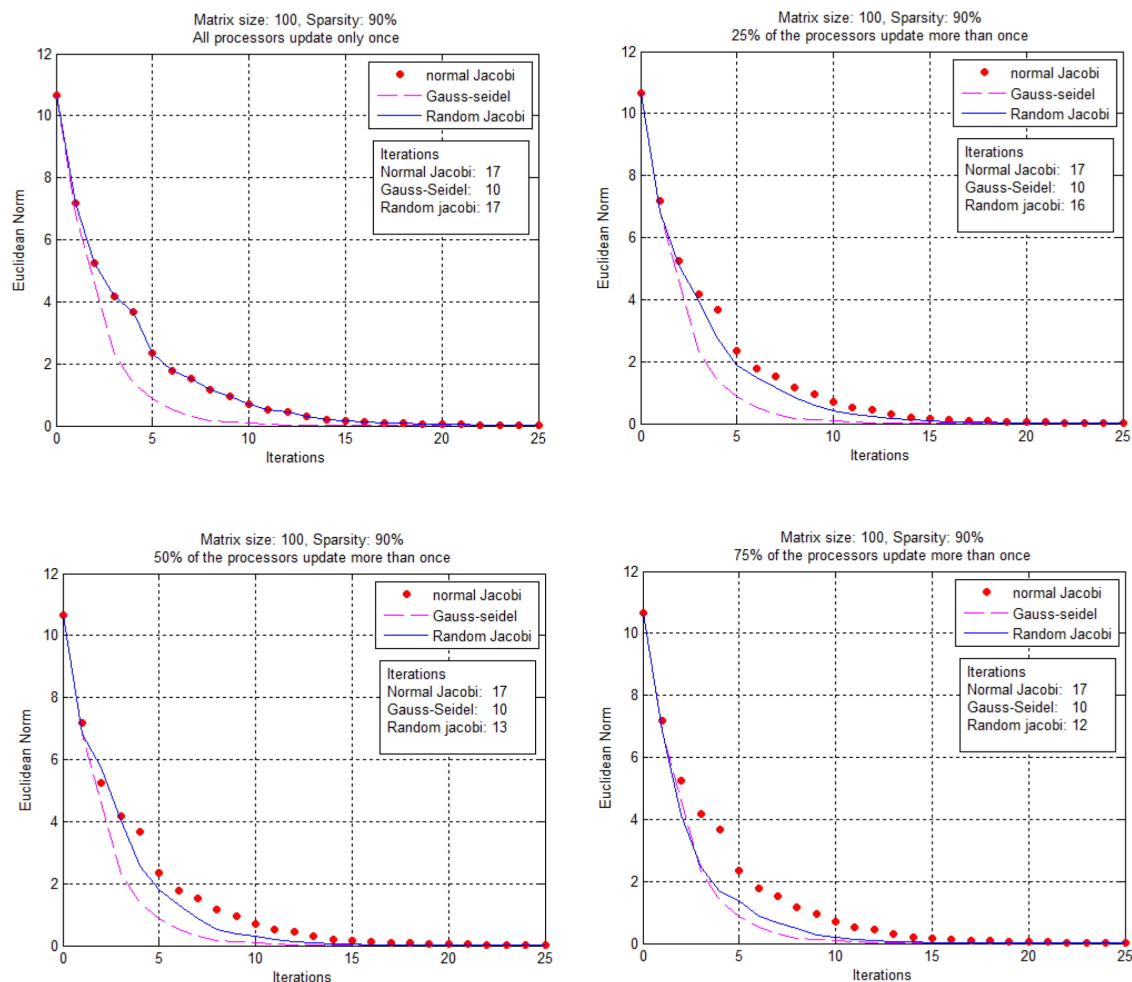


FIGURE 3.6: Convergence of Random Jacobi iterations when some processors update more frequently.

of the solution vector elements are quite close to zero. As expected, for a fixed number of iterations, which is 25 for this simulation, a better initial guess vector, which is all 0s for this case, results in a smaller value for the Euclidean norm of the solution compared to initial guesses such as all 4s or all -4s. However, with a less precise initial guess, although the number of required iterations for convergence increases and bigger values for Euclidean norm is obtained, the system still converges to the correct solution and does not diverge.

3.3 Euclidean Norm Calculation

When using iterative methods, there should be some stop criteria to terminate the iterations at some point during the simulation. These can be based on accuracy, number of iterations, simulation time, or a combination of them. In Section 3.2, the simulations on test matrices were performed only for a fixed number of iterations. In this section, for

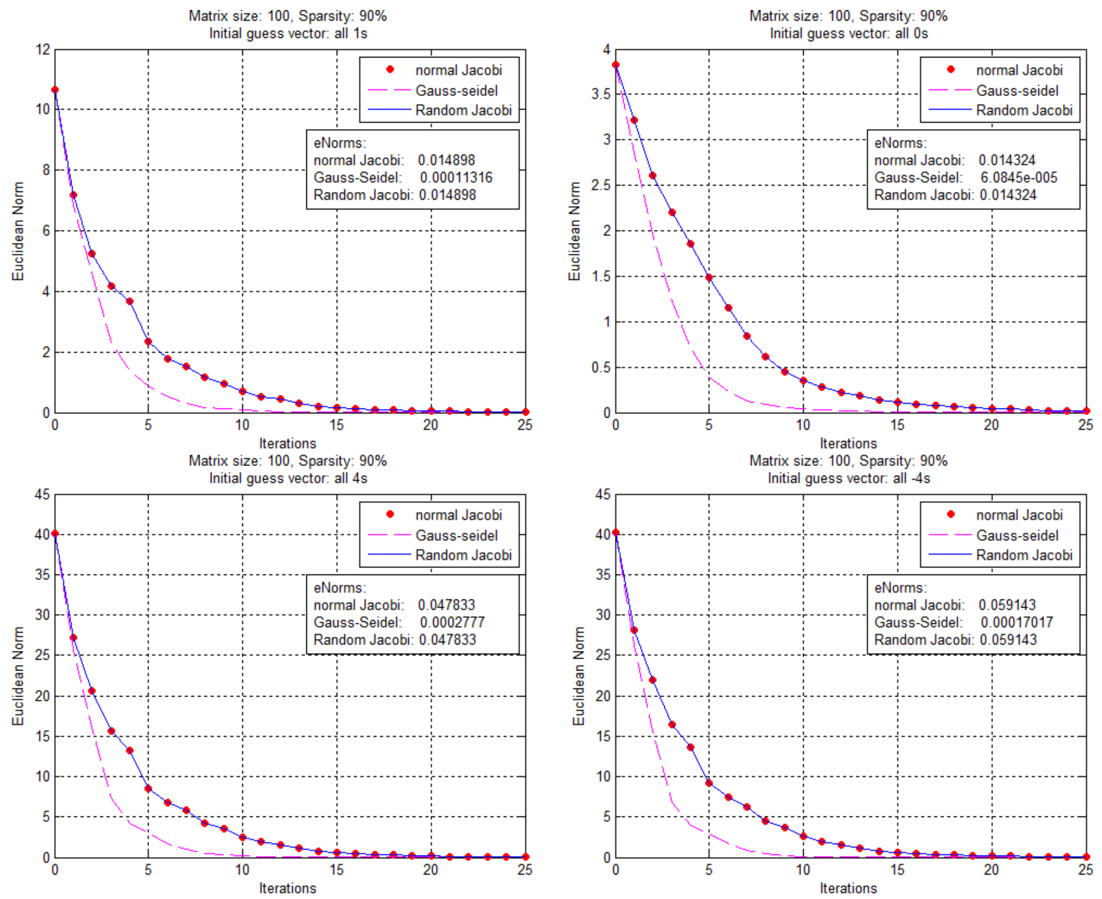


FIGURE 3.7: The effect of the initial guess vector on the convergence of iterative methods.

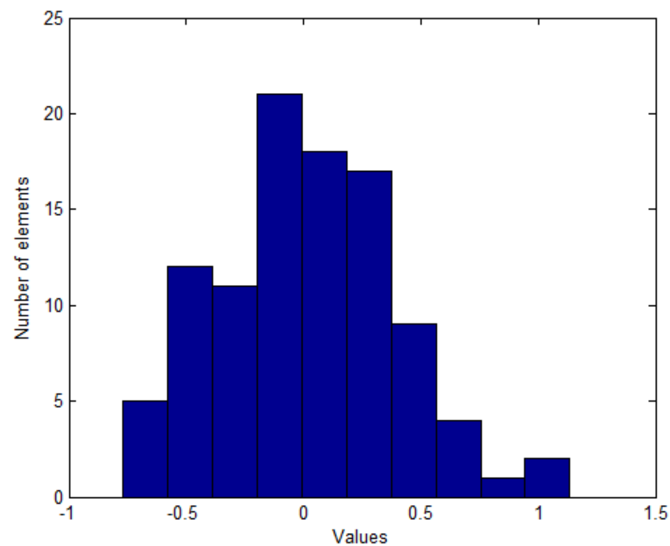


FIGURE 3.8: Distribution of the solution vector elements for the test matrix of Figure 3.7.

our simulations on test matrices, two factors will be used as stop criteria: the maximum iteration number and the Euclidean norm. In fact, the main factor is the Euclidean norm but in case of oscillation, divergence, or very low convergence rates, another criterion is needed to stop the iteration. This is where the maximum iteration number becomes important.

The Euclidean norm can be calculated in two different ways. The first way is by comparing the solution vector at each iteration with the real solution of the system. The iterations stop when the Euclidean norm between those vectors is smaller than a predefined threshold. The second method is by comparing the solutions for the last two consecutive iterations. The first method is more reliable because the result is being compared to the real solution of the system but is applicable only if the solution of the system is known, which is not the case for most of the simulations. If, for example, an iterative method is being used as the matrix solution technique for the simulations, the system solution is not known and the Euclidean norm should be calculated based on the results obtained within the last two iterations. Although in this method there is no need to know the solution to check convergence, there is the danger of converging to a wrong solution.

The two methods are simulated on a test matrix of size 100 for solving the matrix by Gauss Seidel, normal Jacobi, and Random Jacobi iterations. The two stop criteria are the maximum number of iterations, which is set to 50, and the Euclidean norm with the threshold of 10^{-2}.

Figure 3.9 shows the simulation results when the Euclidean norm is calculated by the first method, comparing the results at each iteration with the real solution of the system. In this case, the convergence is obtained in 15 iterations for the Gauss-Seidel method and 26 iterations for the Jacobi method as represented in Figure 3.9. The value at iteration zero stands for the Euclidean norm between the solution and the initial guess. In Figure 3.10, the Euclidean norm is obtained by comparing the solutions in the last two successive iterations. It can be seen that the number of iterations required for convergence using this approach is 14 and 29 for the Gauss Seidel and Jacobi methods, respectively. In this case, the Euclidean norm between the results of two successive iterations needs to be smaller than 10^{-2} in order to stop the iterations. To make sure that the obtained result is the correct solution, the Euclidean norm between the results and the real solutions is calculated which is equal to 0.0148 for the Gauss Seidel method and 0.0049 for the Jacobi iterations. It shows that with the second Euclidean norm calculation method, the Jacobi method has stopped before its Euclidean norm (0.0148) becomes smaller than the threshold (0.01) because the Euclidean norm between the two successive iterations (0.0094) has already become smaller than the threshold. For the same reason, more Jacobi iterations are required in the second approach. It should also be noted that for the second approach, there is no value for iteration zero because at

least the result of the first iteration is needed in order to start calculating its norm with the initial guess.

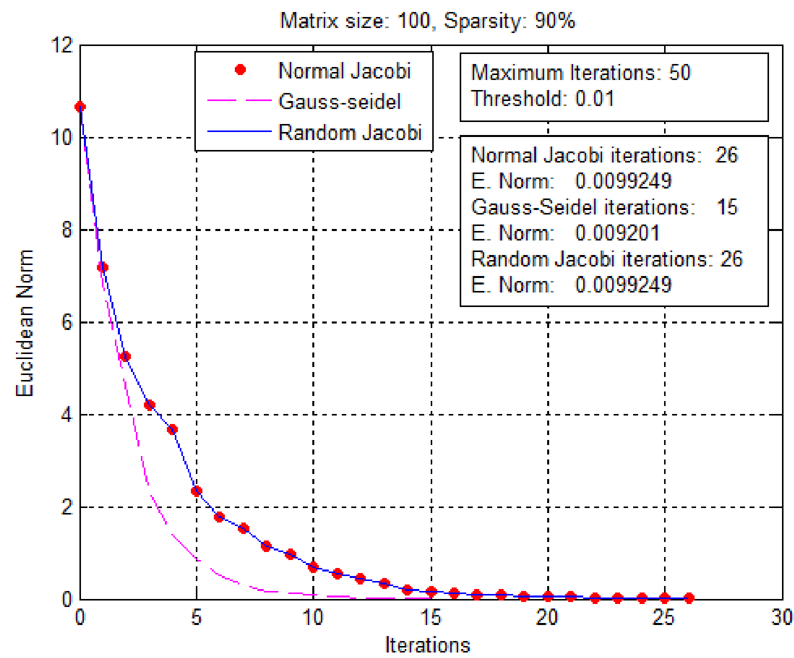


FIGURE 3.9: Iteration stop criterion: The Euclidean norm between the solution at each iteration and the real solution.

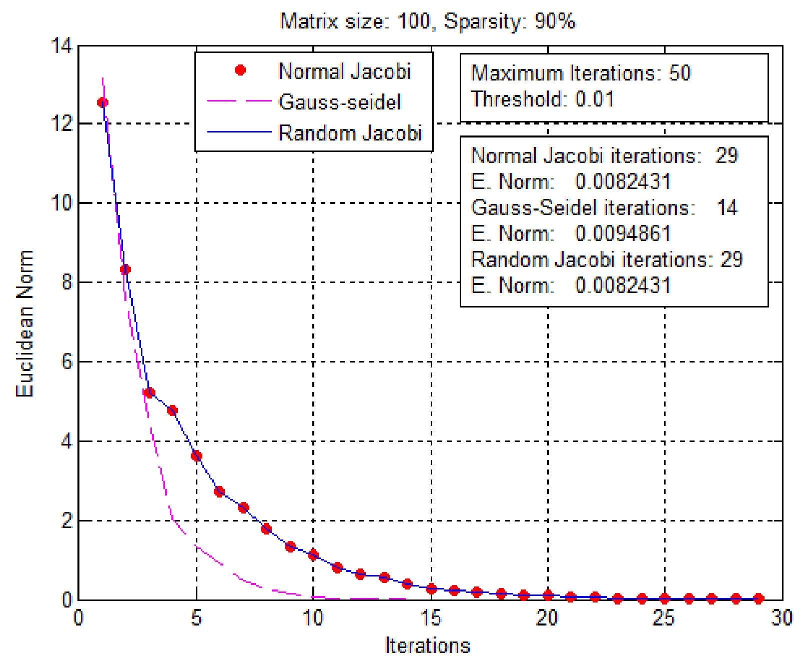


FIGURE 3.10: Iteration stop criterion: The Euclidean norm between the solutions of the last two successive iterations.

Simulation results of Euclidean norm calculation methods for the other test matrices are shown in Table 3.2 for the first method and in Table 3.3 for the second method of Euclidean norm calculation. Same as the previous example for the test matrix size

100, these results also confirm that both methods converge to the solution although with different number of required iterations. Moreover, the number of iterations in the second method is slightly more than the number of iterations in the first method. For the simulations performed in the current work, since the final solution is not known, the second approach for the Euclidean norm calculation will be used.

Size	Jacobi iterations	Jacobi eNorm	Gauss iterations	Gauss eNorm
20	12	0.0088	7	0.0086
50	20	0.0091	8	0.0067
500	23	0.0083	13	0.0059
1000	41	0.0085	22	0.0075

Table 3.2: Iteration stop criterion: Euclidean norm between the solution at each iteration and the real solution.

Size	Jacobi iterations	Jacobi eNorm	Gauss iterations	Gauss eNorm
20	14	0.0064	9	0.0027
50	22	0.0099	9	0.0080
500	25	0.0068	14	0.0071
1000	43	0.0093	23	0.0091

Table 3.3: Iteration stop criterion: Euclidean norm between the solutions of the last two successive iterations.

3.4 Summary and Discussion

When comparing different iterative methods for solving linear systems of equations, one important aspect to consider is the rate of convergence. For a sequence of iterations, which converges to a solution, an iterative method would be preferred if it obtains the correct solution with less error (less Euclidean norm in this work) and with a smaller number of iterations. In this chapter, important factors of the convergence rate of iterative methods are studied. The effect of the equation evaluation order on the Jacobi iterative method is investigated. The aim is to assess the effect of evaluating equations in a completely random order, on the convergence of Jacobi iterations as would be the case in evaluating Jacobi iterations on a highly parallel network of processors working asynchronously and independently.

The Gauss-Seidel, normal Jacobi, and Random Jacobi methods are applied to several test matrices and simulated by Matlab on a sequential machine. Simulation results show that the Random Jacobi method obtains the correct solution within the same number of iterations that normal Jacobi method does. Since the aim of this work is to perform parallel Jacobi iterations on a massively parallel network of processors, the effects of possible faults which can arise in parallel computing applications have also been examined.

When performing Random Jacobi iterations on a highly parallel system by allocating one processor to each circuit equation, although with a slightly more number of iterations, the system converges to the correct solution even if some nodes do not update properly in some of the iterations due to poor communications or delays. However, if one or more processors never update because of a fault, the system may not converge to the solution. Overall, preliminary simulation results suggest that Random Jacobi iterations function properly as an iterative method for solving matrix systems. According to the results obtained in this chapter, evaluation of Random Jacobi iterations on circuit simulation benchmark matrices on parallel platforms will be examined and discussed in Chapter 5.

Conventional matrix solution in circuit simulation algorithms are mostly done using direct solution methods which have very limited capabilities for highly parallel implementations because of the sparse and irregular structures of circuit simulation matrices. The contribution of this chapter to the thesis is to study the Jacobi iterative method with a random and non-deterministic order of evaluation as a matrix solution technique which can be implemented using highly parallel algorithms. It was shown to converge to the solution within a limited number of iterations and with the possibility of all the equations being evaluated independently and massively in parallel and thus is suitable for our proposed highly parallel matrix solution approach which will be introduced in Chapter 5.

Chapter 4

Device Evaluation

Device evaluation is the process of determining the contribution of circuit elements to the system matrix along with linearising the nonlinear elements. At each time step of the circuit analysis process, companion models of linear and nonlinear circuit elements in the form of conductances and current sources need to be calculated and assembled into the conductance matrix and the RHS vector, respectively. In this chapter, first the SPICE algorithm for device modelling is reviewed and the potential parallelisation of this phase is considered. Then, some other algorithms, which benefit from simpler implementations for nonlinear devices, are studied and their advantages and disadvantages are discussed compared to the conventional nonlinear device evaluation, which is done by the Newton-Raphson method in SPICE simulations. Besides, both methods are analysed in conjunction with direct and iterative matrix solution methods to evaluate their functionality and performance. According to the simulation results of some of our test circuits, the proposed approach, which needs less computational effort, functions properly with both direct and iterative solutions although with a higher number of iterations. The device evaluation phase is very straight forward to parallelise due to the possibility of independent evaluation of each device. Therefore, performing the suggested device evaluation method on a highly parallel system along with the parallel evaluation of the proposed iterative matrix solution method will be used as the preliminary basis for parallel evaluation of the SPICE algorithm.

4.1 Device Modelling

At each time step of the circuit simulation process, to calculate the unknown node voltages and branch currents, it is necessary to construct a matrix system in the form of $Ax = b$ and solve it for x. The whole procedure is an iterative process which consists of several steps.

First, the contribution of all circuit elements should be added into the matrix system using the automatic equation formulation techniques described in Chapter 2 Section 2.2.3. The nonlinear behaviour of elements is modelled with linear equivalents and the resultant linear algebraic system of equations is solved. In a transient circuit analysis process, there are two main nested loops, as shown in Figure 4.1. The outer loop computes new values for time-variant elements such as capacitors at the begining of every new time point when doing a transient analysis. The inner loop, which is normally NR iterations, linearises the nonlinear equations and solves the linear matrix of equations. It repeats approximating more accurate models based on the most recent values obtained from the matrix solution phase until the iterations converge to a solution with a pre-defined accuracy [6].

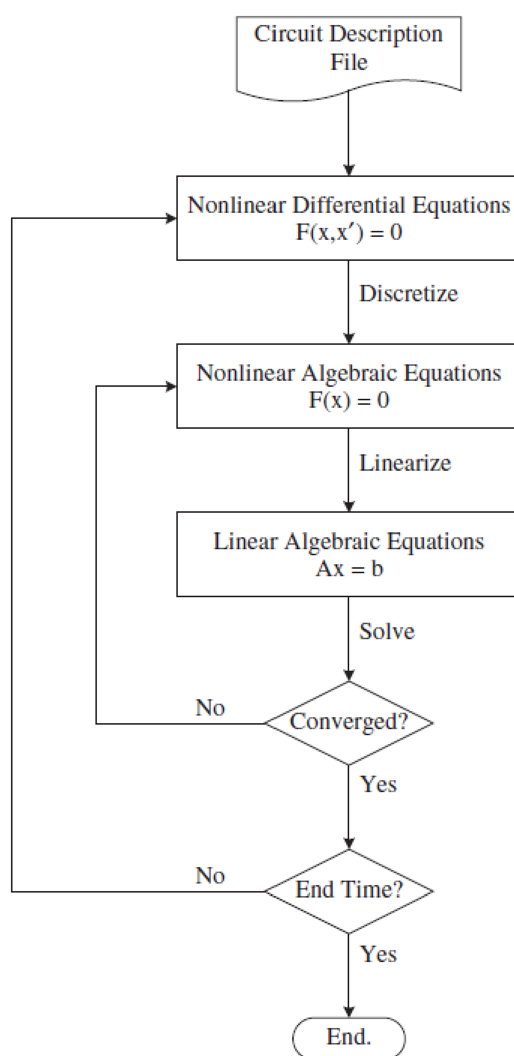


FIGURE 4.1: Circuit analysis flowchart [6]

This chapter focuses on a part of the inner loop which calculates linear models for nonlinear elements. SPICE uses the NR method to model nonlinear elements. Within the inner loop, the contribution of elements such as resistors and constant sources needs

to be calculated only once since their values remain constant throughout the simulation process. Nonlinear element models, however, must be updated at each NR iteration because of the dependency of their values on node voltages. SPICE finds the operating points of nonlinear elements using NR iterations in order to generate a linear model for them. To construct the linear matrix system, stamps of each element are added to the corresponding matrix nodes based on the element terminals and connections. For example, Figure 4.2 shows how the stamps of a two-terminal element (diode) and a three-terminal element (transistor) will sit in the matrix system [12, 54].

As represented in Figure 4.2, for example, the two-terminal nonlinear device will fill four elements of the matrix of conductances, A, and two elements of the RHS vector. However, since there is a nonlinear dependency between the voltage and current of such a device, a linear model needs to be evaluated for it and plugged into the matrix system to have a linear matrix system. This is where numerical methods are required to generate linear approximations for the nonlinear device.

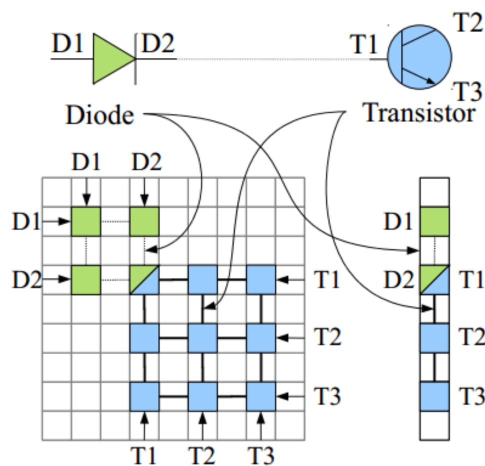


FIGURE 4.2: Assembling a diode and a transistor into a matrix system using their stamps [12]

4.2 Linear Approximation of Nonlinear Elements

4.2.1 The Newton-Raphson Method

Newton-Raphson is a numerical iterative method which is used for nonlinear equation solution and root finding purposes. It works based on linear approximations. Figure 4.3 shows the NR iterations to find the root of the function $f(x)$ [13].

The iterative process starts with an initial guess (point 1) which needs to be fairly close to the solution. Then the nonlinear curve is approximated by a line (the tangent to the curve at point 1). When working with a well-behaved function, this will lead to a

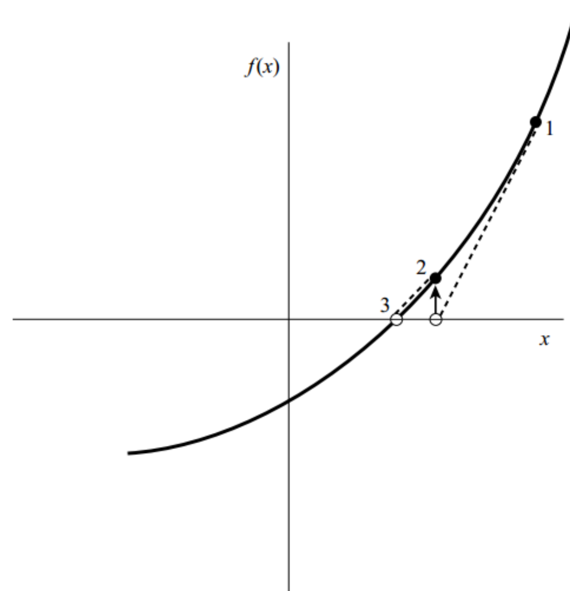


FIGURE 4.3: Finding the root of a function, $f(x)$, using the Newton-Raphson method [13].

new point, 2, which is a more accurate approximation. The tangent of the function is again calculated using the new value and this process is repeated until converging to the solution [87, 13].

Consider the simple circuit in Figure 4.4a, which contains a diode as a nonlinear element.

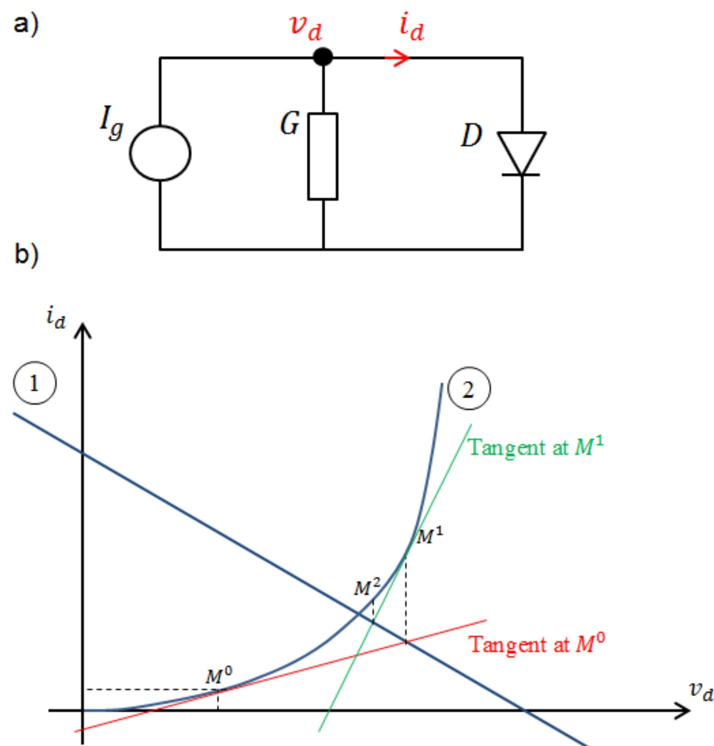


FIGURE 4.4: a) A simple diode circuit b) Graphs of the equations.

By applying KCL to the only node of the circuit and writing the diode current equation in terms of its terminal voltages as a branch current equation, Equation 4.1 and Equation 4.2 represent the system of equations describing the circuit.

$$G.V_d + i_d = I_g \tag{4.1}$$

$$i_d = I_s(e^{\lambda v_d} - 1) \tag{4.2}$$

The first equation is linear and is shown by a line with the slope of $-G$ on the $i - v$ axis in Figure 4.4b. The second equation is nonlinear and shows the nonlinear behaviour of the diode by a curve on Figure 4.4b. The current (i_d) and voltage (v_d) corresponding to the intersection point of these two equations are the solutions of the circuit. In order to find the correct values for the unknown current and voltage, the nonlinear curve is first approximated by a line to convert the system of nonlinear equations to a linear algebraic system of equations. The Newton Raphson method needs a starting point which is shown by M^0 with the coordinates of (v_d^0,i_d^0). The curve is approximated by a line tangent to it at point M^0. The slope of the line is equivalent to the conductance in diode's stamp in its companion model as explained in Chapter 2 Section 2.2.3. This is noted by G_d and can be found by calculating the value for the derivative of the diode current (Equation 4.3) at v_d^0. It is shown by a red line, which intercepts the linear equation at point M^1. This is the first approximation of the solution and can be obtained by solving the linear set of equations.

$$G_d = \frac{\partial i_d}{\partial v_d} = \lambda I_s e^{\lambda v_d} \tag{4.3}$$

For the next NR iteration, M^1 is used as the starting point and in the same way, the next approximation of the solution is calculated. This process repeats until it converges to the solution within a pre-defined accuracy. The number of required iterations depends on a number of factors. For a higher accuracy, more iterations are required and better initial guess values will lead to quicker convergence which means fewer iterations.

The unknown values of the diode circuit in Figure 4.4a are (i_d) and (v_d). Assume that the other elements and parameters have the values: $G = 0.1s$, $I_g = 100mA$, $I_s = 10^{-12}A$, and $\lambda = 40$ and the initial guess for the node voltage is $v_d^0 = 1.0v$ which results in $i_d^0 = 0.0265A$.

Table 4.1 shows the number of NR iterations required to find the solution with two different error margins and initial guess sets. The first column, error margin, is in fact the threshold to stop the iterations and as it gets smaller more iterations are required to converge to the solution. The second column, initial guess, shows the starting value

for the diode voltage which is normally a reasonable guess. The more accurate the initial guess, the fewer the number of iterations required for convergence. Although the iterations converge to the solution with both initial guesses, it can be seen that a closer starting value can notably accelerate the solution process. For example, when the error margin is defined as 10^{-3}, with a starting point of $v_d = 1v$, 18 iterations are required to find the solution ($v_d = 0.6v$) while a starting point of $v_d = 0.7v$, which is much closer to the solution, will only need six NR iterations to converge.

Error margin	Initial guess v_d	Number of iterations	Solution set
10^{-3}	$1.0v$	18	$v_d = 0.6096v$ $i_d = 0.0405A$
10^{-5}	$1.0v$	20	$v_d = 0.6097v$ $i_d = 0.0390A$
10^{-3}	$0.7v$	6	$v_d = 0.6096v$ $i_d = 0.0404A$
10^{-5}	$0.7v$	8	$v_d = 0.6097v$ $i_d = 0.0390A$

TABLE 4.1: Number of NR iterations for the diode circuit in Figure 4.4.a

The NR method is reasonably fast and in most cases converges to the solution but it also has a number of drawbacks. When dealing with well-behaved functions, it does not have convergence issues but in some cases, such as the two cases shown in Figure 4.5, it may face serious convergence issues. Another downside of the method is that the evaluation of the partial derivatives (Equation 4.3) is needed at each NR iteration for linear approximation, which increases the required calculations per iteration. In the next sections of this chapter, some alternative methods to replace the NR methods are investigated. The aim is to employ simpler linearisation methods without the need for calculating partial derivatives at every iteration.

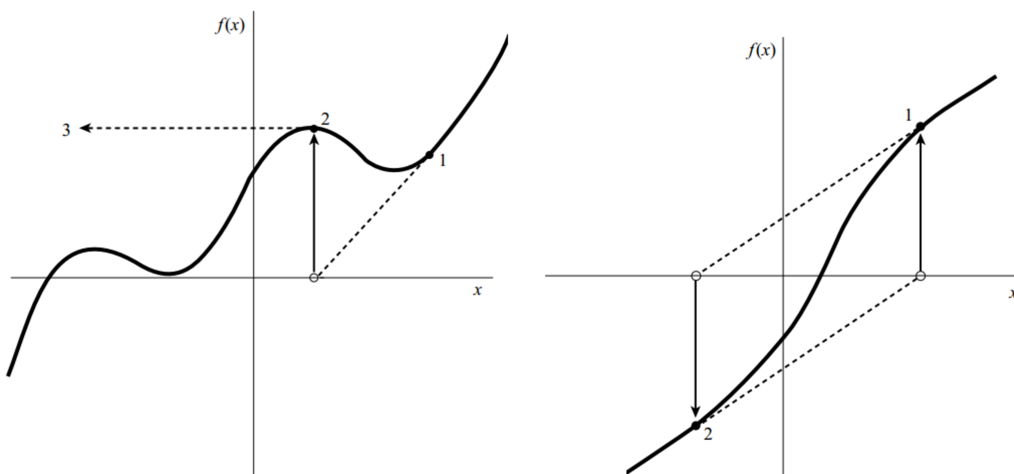


FIGURE 4.5: Newton-Raphson convergence issue caused by a) a local extremum b) a nonconvergent cycle [13].

4.2.2 Fixed Slope Approach

To avoid the calculation of partial derivatives at each linearisation iteration, it is possible to calculate the slope of the tangent to the nonlinear function, which approximates the curve, only once at the first NR iteration and continue the rest of NR iterations with that fixed value. In other words and as an example for the diode circuit shown in Figure 4.4, the first value of the linear parameter for diode is calculated (G_m^0) in the first NR iteration and and there is no need for the calculation of the derivative of the function in the next NR iterations. This is graphically represented in Figure 4.6 for the example diode circuit. As can be seen, the slope of the line approximating the curve at point M^0 remains constant during all NR iterations. Although this simplifies the calculations, it can considerably affect the convergence rate of the iterations.

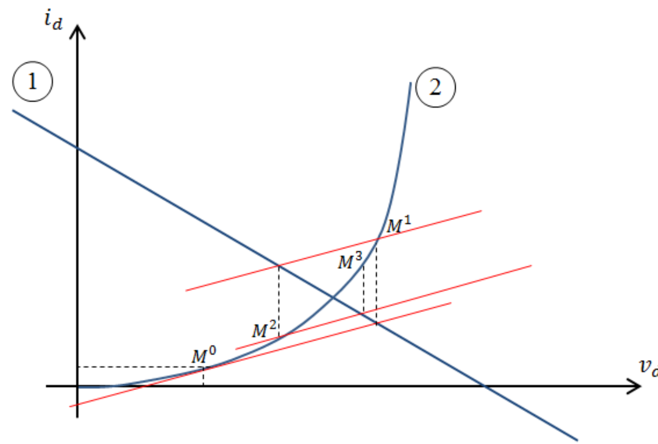


FIGURE 4.6: Using constant slopes in the G^0 approach

Table 4.2 shows a comparison between the number of iterations required to converge to the solution with specific error limits when using the normal NR method and the fixed slope (G^0) approach. The initial guess for the node voltage is $v_d^0 = 0.7v$ in both cases.

Error margin	**number of NR iterations**	**number of G^0 iterations**
10^{-3}	6	18
10^{-5}	8	147

TABLE 4.2: Convergence comparison of the NR and the fixed slope iterations for the diode circuit in Figure 4.4a

Simulation results show that for the error margins of 10^{-3} and 10^{-5}, the fixed slope method requires 18 and 147 iterations, respectively. Compared to the results in Section 4.2.1, which were six and eight iterations for the same cases, it is seen that many more iterations are required in G^0 approach. Although the fixed slope method requires fewer calculations, the results in Table 4.2 suggest that when more accuracy is required, the

number of iterations dramatically increases for this approach which makes it unsuitable for linearisation purposes when high accuracies are required.

4.2.3 The Secant Method

Another linear approximation technique, which also works based on slopes of the function and has simpler calculations compared to the NR method, is the Secant method. Unlike the NR method which requires calculations of function derivatives at each iteration, the Secant method uses 'difference' instead of the actual 'differentiation'. It means that the Secant method uses two adjacent points to approximate the curve by a line passing through these points. Like the NR method, the Secant method does not guarantee convergence but converges for well-behaved functions. The Secant method needs two initial starting points and has slower convergence rate than the NR approach. However, it benefits from simpler calculations since there is no need for calculation of partial derivatives. The procedure of linear approximation by the Secant method is shown in Figure 4.7.

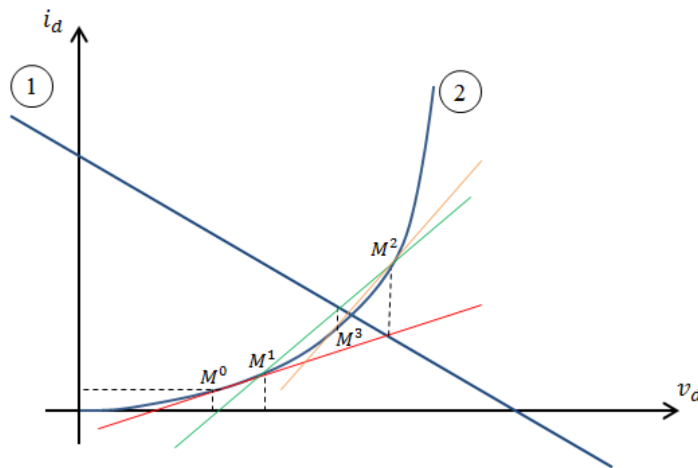


FIGURE 4.7: Linear approximation using the Secant method

At each Secant iteration for the example diode circuit, G_d can be calculated using Equation 4.4. Comparing it to Equation 4.3 shows that evaluation of the derivative of the diode current is not required but two distinct initial points are needed to start the first approximation. Simulation results of applying the Secant method on the example diode circuit using the two starting points of $v_d^0 = 0.8v$ and $v_d^1 = 0.7v$ is represented in Table 4.3 and compared to the NR method ($v_d^0 = 0.7v$).

$$G_d^{n+1} = \frac{i_d^n - i_d^{n-1}}{v_d^n - v_d^{n-1}} \tag{4.4}$$

Error margin	Number of NR iterations	Number of Secant iterations
10^{-3}	6	9
10^{-5}	8	11

TABLE 4.3: Convergence comparison of NR and Secant iterations for the diode circuit in Figure 4.4a with the first initial guess set.

Unlike the fixed slope approach, the Secant method shows a better performance. The number of iterations for the Secant method is quite close to the NR method while none of the two initial guesses were closer to the solution than the NR case. Table 4.4 shows the same simulation using Matlab (All Matlab simulations in this work are done using the software version 7.11.0.584($R2010b$)) for a new set of initial guesses as $v_d^0 = 0.75v$ and $v_d^1 = 0.65v$, which results in the same number of iterations for the Secant compared to the NR method. This was just a simple example for illustration purposes and a few other examples are presented in the next section for a better comparison.

Error margin	Number of NR iterations	Number of Secant iterations
10^{-3}	6	6
10^{-5}	8	8

TABLE 4.4: Convergence comparison of NR and Secant iterations for the diode circuit in Figure 4.4a with the second initial guess set.

4.3 Case Studies on NR and Secant Methods

In this section, the NR and Secant methods are compared using some example circuits and their simulation results, which are discussed in more detail including the circuit diagram, matrix construction, and linearisation process.

4.3.1 Case Study # 1

Consider the circuit in Figure 4.8 which has one MOS transistor with a number of conductances and voltage sources. By using device stamps and the MNA method, the matrix system, which describes the circuit, is constructed and shown in Equation 4.5 and Equation 4.6. There are two zero elements on the diagonal of the conductance matrix because of the presence of voltage sources. Therefore, in the first place, it is necessary to resolve this issue by rearranging the matrix to a form that has non-zeros on the diagonal in order to use iterative matrix solution methods. This is done by exchanging row 1 by row 7 and also row 5 by row 6 for this specific example. In the unknown vector, there

are five voltages for five circuit nodes and two currents which are the currents passing through the voltage sources. G_1, G_2, and i_{mds} are the corresponding terms for the stamp of the MOS transistor and change value at each linearisation iteration according to the terminal voltages and also the current of the MOS transistor. The values of other elements stay fixed for the duration of the simulation at each time point.

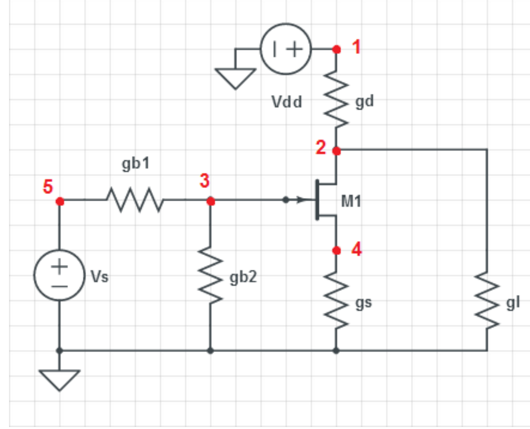


FIGURE 4.8: An example circuit with one MOS transistor

$$A = \begin{bmatrix} g_d & -g_d & 0 & 0 & 0 & 0 & 1 \\ -g_d & g_d + g_l + G_1 & G_2 & -G_1 - G_2 & 0 & 0 & 0 \\ 0 & 0 & g_{b1} + g_{b2} & 0 & -g_{b1} & 0 & 0 \\ 0 & -G_1 & -G_2 & G_1 + G_2 + g_s & 0 & 0 & 0 \\ 0 & 0 & -g_{b1} & 0 & g_{b1} & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.5}$$

$$x = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ i_{vs} \\ i_{vdd} \end{bmatrix}, RHS = \begin{bmatrix} 0 \\ -i_{mds} \\ 0 \\ i_{mds} \\ 0 \\ v_s \\ v_{dd} \end{bmatrix} \tag{4.6}$$

The current of the MOS transistor is given by Equation 4.7 where k' is a constant, w and l are the MOS channel width and length respectively, and v_{th} is the threshold voltage [88]. The MOS stamp values, G_1, G_2, and i_{mds}, can be calculated using Equation 4.8, Equation 4.9, and Equation 4.10, respectively.

$$i_d = \frac{k'}{2}\frac{w}{l}(2(v_{gs} - v_{th})v_{ds} - v_{ds}^2) \tag{4.7}$$

$$G_1 = \frac{\partial i_d}{\partial v_{ds}} = 2k'(v_{gs} - v_{th} - v_{ds}) \tag{4.8}$$

$$G_2 = \frac{\partial i_d}{\partial v_{gs}} = 2k'v_{ds} \tag{4.9}$$

$$i_{mds} = i_d - G_1 v_{ds} - G_2 v_{gs} \tag{4.10}$$

By assuming the following values for circuit elements and constants:

$$\begin{cases} v_{th} = 0.7 & k' = 200e-6 & w/l = 2 \\ v_{dd} = 3 & v_s = 2 & \\ g_{b1} = 5e-4 & g_{b2} = 1e-5 & g_l = 3.33e-5 \\ g_d = 3.33e-5 & g_s = 2e-4 & \end{cases}$$

and then by rearranging the matrix system to have non-zeros on the diagonal of the conductance matrix. The linearised matrix system for the first iteration of calculating the circuit's operating point will be equal to the system shown in Equation 4.11 and Equation 4.12.

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3.33e-5 & 3.06e-4 & 1.2e-4 & -3.6e-4 & 0 & 0 & 0 \\ 0 & 0 & 5.1e-4 & 0 & -5e-4 & 0 & 0 \\ 0 & -2.4e-4 & -1.2e-4 & 5.6e-4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -5e-4 & 0 & 5e-4 & 1 & 0 \\ 3.33e-5 & -3.33e-5 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

$$x = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ i_{vs} \\ i_{vdd} \end{bmatrix}, RHS = \begin{bmatrix} 3 \\ 1.74e-4 \\ 0 \\ -1.74e-4 \\ 2 \\ 0 \\ 0 \end{bmatrix} \tag{4.12}$$

Now, the matrix system of linear equations is ready to be solved to provide new x values for the calculation of more accurate linear models for the MOS transistor in the next linearisation iteration. The matrix solution part can be done using either a direct or an iterative process. For device modelling also it is possible to use either the NR or the

Secant method. The possible combinations of these methods will lead to four different approaches, represented in Figure 4.9, which will be simulated and the results will be compared. It should be noted that the stop criteria based on the accuracy of the solution vector is the same for all simulations in order to have a fair comparison.

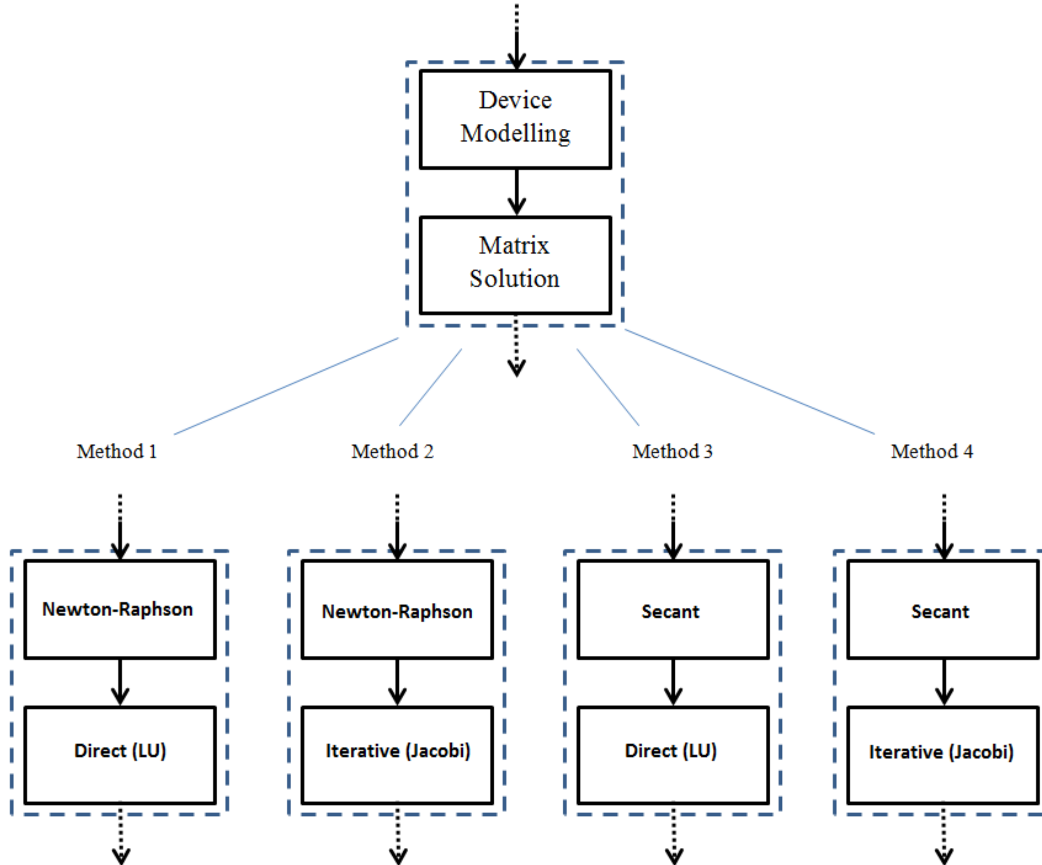


FIGURE 4.9: Four different methods to perform device modelling and matrix solution.

The simulation results of using a direct method for matrix solution and NR for device evaluation, which normally is the case for conventional SPICE simulations, are shown in Table 4.5. The same simulation has also been done using the Jacobi iterative method in conjunction with NR iterations and the simulation results are shown in Table 4.6. In both tables, the top row represents the iteration number. The unknown node voltages and branch currents are listed in the first column and their values at each iteration can be found in the corresponding column to that iteration. The initial guess for NR iterations is the same for both cases and is shown in Table 4.7.

The number of Jacobi iterations used for this simulation is seven and is chosen to be the same as the number of equations. As reviewed in Chapter 2 Section 2.3, direct solutions such as the LU-factorisation have $O(n^3)$ operations and iterative solutions such as the Jacobi method have $O(n^2)$ operations per iteration. So, if the number of iterations is equal to the problem size, both methods will be of the order of n^3. As the size of the

Iteration number	**1**	**2**	**3**	**4**
v_1	3.0000	3.0000	3.0000	3.0000
v_2	0.5126	0.5211	0.5212	0.5212
v_3	1.9608	1.9608	1.9608	1.9608
v_4	0.3291	0.3263	0.3263	0.3263
v_5	2.0000	2.0000	2.0000	2.0000
i_{vs}	-19.60μ	-19.60μ	-19.60μ	-19.60μ
i_{vdd}	-82.91μ	-82.62μ	-82.62μ	-82.62μ

TABLE 4.5: Solution to the circuit in Figure 4.8 using the NR method along with a direct matrix solver

Iteration number	**1**	**2**	**3**	**4**	**5**
v_1	3.0000	3.0000	3.0000	3.0000	3.0000
v_2	0.5013	0.5207	0.5209	0.5212	0.5212
v_3	1.9608	1.9608	1.9608	1.9608	1.9608
v_4	0.3241	0.3247	0.3262	0.3263	0.3263
v_5	2.0000	2.0000	2.0000	2.0000	2.0000
i_{vs}	-19.60μ	-19.60μ	-19.60μ	-19.60μ	-19.60μ
i_{vdd}	-83.30μ	-82.73μ	-82.62μ	-82.62μ	-82.62μ

TABLE 4.6: Solution to the circuit in Figure 4.8 using the NR method along with the Jacobi method as an iterative matrix solver

v_1	v_2	v_3	v_4	v_5	i_{vs}	i_{vdd}
3.0000	0.5000	1.8000	0.2000	2.0000	-30μ	-50μ

TABLE 4.7: Starting values for simulation results in Table 4.5 and Table 4.6

problem increases, if the number of iterations is much smaller than the problem size, the iterative solution will have $O(n^2)$ operations.

By looking at the simulation results, it can be seen that the NR method along with the direct matrix solver (Table 4.5)converges to the solution within three iterations while it takes four iterations for the NR method with the iterative matrix solver (Table 4.6). These two iterations are highlighted in the tables.

It is worth mentioning that the terms v_2, v_4, and v_{dd}, which are the drain voltage, the source voltage, and the current of the MOS transistor, change value at each iteration and the other terms converge to their final value at the first iteration. Therefore, for the next sets of simulation results for this example, only the values of these three variables will be presented.

Table 4.8 and Table 4.9 show the simulation results for using the Secant method for device evaluation in conjunction with direct and iterative matrix solvers, respectively. The number of Jacobi iterations used for this simulation is also 7. As discussed before, the Secant method needs two sets of starting points, which are listed in Table 4.10.

Iteration number	**1**	**4**	**9**	**10**	**11**
v_2	0.5548	0.5269	0.5213	0.5212	0.5212
v_4	0.3151	0.3244	0.3262	0.3263	0.3263
i_{vdd}	-81.50μ	-82.43μ	-82.62μ	-82.62μ	-82.62μ

TABLE 4.8: Solution to the circuit in Figure 4.8 using the Secant method and a direct matrix solver

Iteration number	**1**	**4**	**8**	**9**	**10**
v_2	0.5337	0.5275	0.5212	0.5212	0.5212
v_4	0.2948	0.3207	0.3262	0.3263	0.3263
i_{vdd}	-82.92μ	-82.01μ	-82.62μ	-82.62μ	-82.62μ

TABLE 4.9: Solution to the circuit in Figure 4.8 using the Secant method and the Jacobi method as an iterative matrix solver

	v_1	v_2	v_3	v_4	v_5	i_{vs}	i_{vdd}
Set#1	3.0000	0.4000	1.6000	0.2000	2.0000	-25μ	-40μ
Set#2	3.0000	0.5000	1.8000	0.1500	2.0000	-30μ	-50μ

TABLE 4.10: Starting values for simulation results in Table 4.5 and Table 4.6

The results show that the Secant method with the direct matrix solution converges within ten iterations (Table 4.8) and with the iterative matrix solution, it converges within nine iterations (Table 4.9). It should be recalled that with the NR method instead of the Secant method, three and four iterations for the same circuit were required to converge to the solution. NR obviously converges with fewer iterations but it should also be considered that NR is computationally more complicated than the Secant method because of the need for the numerical evaluation of derivatives. Therefore, execution time measurements are needed to verify which method has been faster. The non-linear device modeling phase for the test circuits using Newton-Raphson and Secant methods had to be done manually in the Matlab code. Therefore, the examples have relatively small sizes. Also, time measurement has been done using the *tic* and *toc* functions of Matlab which measure the elapsed time between tic and toc [86]. The Matlab codes for the four different approaches are included in the Appendix A, Section A.3.

The simulation results of the execution time for each method along with the number of iterations required for convergence is given in Table 4.11. The results show that, according to the number of iterations, the NR method along with both matrix solution approaches converges with fewer iterations. However, by looking at the execution times, it can be seen that the Secant method performs better with a much less simulation time. Another important point to consider is that these simulations are being done sequentially and the Jacobi-type iterative solution performs much better on parallel platforms for large sparse systems. Therefore, even better results for the Secant method along with

the Jacobi matrix solution is expected for parallel simulations. This will be evaluated on real parallel platforms in Chapter 6 Section 6.2.

Methods	Iterations	Execution time (ms)
NR with Direct matrix solution	3	113.2
NR with 7 Jacobi iterations	4	131.4
Secant with Direct matrix solution	10	28.8
Secant with 7 Jacobi iterations	9	64.5

TABLE 4.11: Number of iterations and execution time required to obtain the solution by different methods for the test circuit in Figure 4.8.

4.3.2 Case Study # 2

As the second example, a MOS differential pair with resistive loads will be investigated. Figure 4.10 shows a circuit with two MOS transistors, a few resistors, and voltage and current sources.

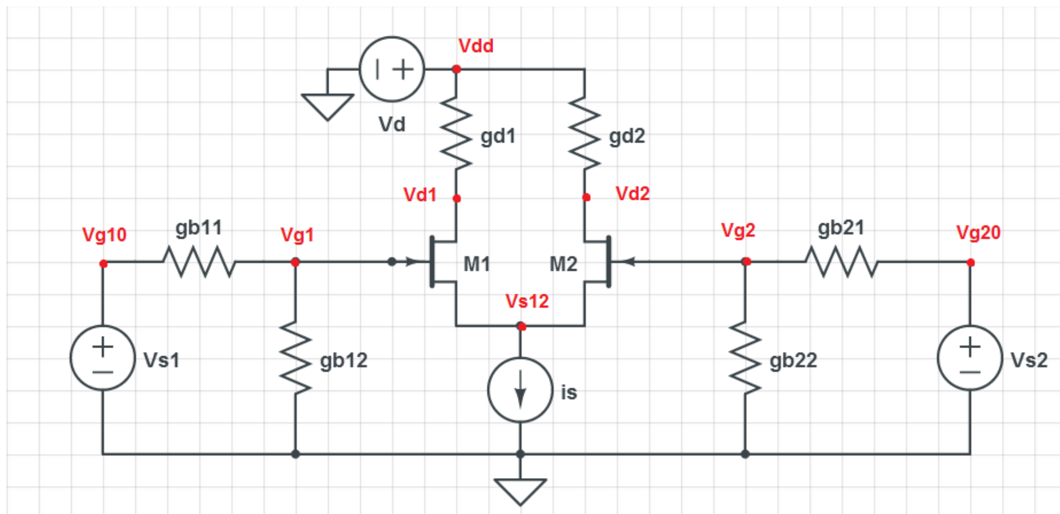


FIGURE 4.10: A MOS Differential Pair

The circuit has eight nodes and three unknown currents through the voltage sources. Therefore, the size of the conductance matrix is expected to be eleven. The values of circuit elements and constants used for these simulations are as follows:

$$\begin{cases} v_{th} = 0.7 & k' = 200e-6 & w/l = 5 & \\ v_d = 5 & v_{s1} = 2.5 & v_{s1} = 2.5 & i_s = 1.50m \\ g_{b11} = 1e-4 & g_{b21} = 1e-4 & g_{b12} = 1e-6 & g_{b22} = 1e-6 \\ g_{d1} = 2e-4 & g_{d2} = 2e-4 & & \end{cases}$$

The matrix of conductances, A, after rearranging and also the RHS vector, b, before the first NR iteration are represented in Figure 4.11. One initial guess vector set for NR iterations and two sets for the Secant method can be found in Table 4.12.

```
A =

    1.0000         0         0         0         0         0         0         0         0         0         0
   -0.0001    0.0001         0         0         0         0         0         0         0         0         0
         0    0.0005    0.0009   -0.0013         0         0         0   -0.0002         0         0         0
         0   -0.0005   -0.0007    0.0025   -0.0007   -0.0005         0         0         0         0         0
         0         0         0   -0.0013    0.0009    0.0005         0   -0.0002         0         0         0
         0         0         0         0         0    0.0001   -0.0001         0         0         0         0
         0         0         0         0         0         0    1.0000         0         0         0         0
         0         0         0         0         0         0         0    1.0000         0         0         0
    0.0001   -0.0001         0         0         0         0         0         0    1.0000         0         0
         0         0         0         0         0   -0.0001    0.0001         0         0    1.0000         0
         0         0   -0.0002         0   -0.0002         0         0    0.0004         0         0    1.0000


b =

    2.5000
         0
    0.0009
   -0.0033
    0.0009
         0
    2.5000
    5.0000
         0
         0
         0
```

FIGURE 4.11: A and RHS matrices for the circuit in Figure 4.10

Unknowns	**NR**	**Secant 1**	**Secant 2**
v_{g10}	2.5	2.5	2.5
v_{g1}	2	1.7	2
v_{d1}	1	0.8	1
v_{s12}	0.5	0.6	0.5
v_{d2}	1	0.8	1
v_{g2}	2	1.7	2
v_{g20}	2.5	2.5	2.5
v_{dd}	5	5	5
i_{vs1}	10μ	20μ	10μ
i_{vs2}	10μ	20μ	10μ
i_{vdd}	$1.20m$	$1.00m$	$1.00m$

TABLE 4.12: Starting values for simulation results of the test circuit in Figure 4.10

The circuit is simulated with NR and Secant methods for device evaluation and direct and iterative methods for matrix solution. The simulation results in terms of the number of iterations required for convergence and execution time for each case are presented in Table 4.13.

The simulation results in Table 4.13 show almost the same behaviour for case number 2 circuit as the circuit in case number 1 with slightly different quantitative results. As for the number of iterations to converge to the solution, the NR method works better than the Secant approach. As it is seen in Table 4.13, the first three rows, which include the results for the NR method along with different matrix solution methods, converge to the solution with fewer number of iterations compared to the last three

Methods	Iterations	Execution time (ms)
NR plus Direct matrix solution	4	150
NR plus 10 Jacobi iterations	7	208
NR plus 50 Jacobi iterations	4	238
Secant plus Direct matrix solution	9	44
Secant plus 10 Jacobi iterations	23	215
Secant plus 50 Jacobi iterations	10	285

Table 4.13: Number of iterations and execution time required to obtain the solution by different methods for the test circuit in Figure 4.10.

rows, which represents the same simulations using the Secant method. However, the Secant approach still has a better overall execution time due to its simpler computations. Based on our measurements, the Jacobi method performs slightly less effective for this example compared to the previous one. The reason is that the Jacobi iterative method is being evaluated on a sequential system, which is not suitable for its parallel algorithm. However, comparing the execution times of the Secant method along with the iterative matrix solution, which is 215 ms, with the NR method along with a direct solution, which is 150 ms, shows that although an inefficient Jacobi method is used, the results are close. A parallel evaluation of the methods will be performed in Chapter 6 Section 6.2.

4.4 Summary and Discussion

To linearise the nonlinear elements in order to form a linear system of equations, there are a number of different numerical approaches, which approximate the nonlinear function with a linear equivalent model by iterative processes. The NR approach is a widely used method for this purpose which uses a starting point and makes approximations by calculating derivatives of the function to model curves with lines at specific operating points. NR is reasonably fast but needs extra computational effort to evaluate partial derivatives at each iteration. The secant method, on the other hand, benefits from simpler calculations since it works based on numerical ‘difference’ by using two initial points instead of calculating partial derivatives at each iteration for linearisation purposes. The Secant method normally converges slower than NR but with fewer computations required per iteration.

As a part of the research objectives for this thesis, in this chapter alternative simpler techniques for device evaluation were investigated. It was shown that the device evaluation phase, which is a parallel task by its nature, can be performed using the Secant method, which benefits from a simpler algorithm compared to the conventional Newton-Raphson ietartions. Newton-Raphson iterations is widely used for device evaluation in circuit simulation algorithms and normally converges much faster to the solution than

other methods such as Secant. However, we believe that using the Secant method, with its much simpler algorithm, in conjunction with our proposed iterative parallel matrix solution approach, which will be reviewed in ***Chapter 5***, can lead to a quicker convergence. Therefore, the contribution of this chapter is to study the possibility of using the Secant method for the device evaluation phase in our proposed method.

To evaluate the functionality and performance of the two methods, two example circuits with MOS transistors as nonlinear elements were simulated in this chapter. Simulation results show that NR converges to the solution in fewer iterations than the Secant method but the Secant method is less time demanding due to its simpler algorithm. The test circuits were simulated by both of the linearisation techniques for the device evaluation phase and in each case the matrix solution phase is done with both direct (LU-factorisation) and iterative (Jacobi) methods on a sequential system.

The NR method along with the direct matrix solution converged to the circuit solution with less than half of the iterations required for the Secant method along with the direct matrix solution to obtain the circuit solution. However, the simulation time for the Secant method was almost four times better than the NR method. The simulation times for the test circuits have decreased from 113 ms and 150 ms to 29 ms and 44 ms, respectively. This shows that by using the Secant method instead of NR, a speed-up of 4x is achieved when a direct matrix solution is used.

Simulations were also performed by using iterative approaches for the matrix solution phase. Although our proposed iterative method functions properly and obtains the solution within a few iterations, its best performance is achieved when it is evaluated on a parallel system. However, when simulated on a sequential system, its simulation times are quite close to the ones from the direct solution. It is expected that by performing simulations on parallel systems, even better results can be obtained for the proposed solution method. This will be investigated in Chapter 6.

Chapter 5

Parallel Matrix Solution

In recent years, parallel evaluation of SPICE to speed up the simulation process has been an interesting topic for researchers which was briefly reviewed in Chapter 2 Section 2.4.2. Although these attempts to parallelise the SPICE algorithm have resulted in limited speed-ups and accelerated matrix solution or device evaluation phases, some inherent sequential issues of the SPICE simulation process have remained unsolved. As discussed before in Chapter 2 Section 2.4.2., the barrier between NR iterations and the matrix solution phase limits the amount of possible parallelism [11].

In this work, a Jacobi-type iterative method (the Random Jacobi) has been proposed which can solve matrix equations on a massively parallel system in a completely non-deterministic order. This not only does parallelise the matrix solution process using a fine-grained approach by allocating one processor to each circuit equation, but also provides the groundwork (as will be seen in Chapter 6) for decreasing some of the parallelisation limitations caused by the barrier between the linearisation and the matrix solution phases.

In this chapter, the proposed Random Jacobi iterative method for the matrix solution phase is applied to a number of test circuits on a highly parallel system. The iterative Jacobi-type methods is evaluated using real circuit simulation matrices on single-core, virtual many-core, and real many-core systems. The proposed pattern for decreasing the amount of required communication is tested and the effect of asynchronous communication is investigated. The results will be presented, compared, and discussed in the final section.

5.1 Test Matrices

To apply matrix solution algorithms, we have used a number of test matrices which are either from the University of Florida Sparse Matrix Collection [89] or extracted from

real SPICE simulations.

The University of Florida Sparse Matrix Collection contains a large and actively growing set of sparse matrices that arise in real applications. It covers a wide spectrum of matrices arising from problems in a variety of fields including circuit simulation which can be accessed and used by a number of different programming languages such as Matlab, Fortran, C/C++, etc. For the specific use of this work, which focuses on iterative solutions, the test matrices were required to have RHS vectors as well. This criterion limited the number of matrices which were useful for our simulations from this collection. We chose a number of matrices with different sizes and sparsities which had RHS vectors. In the collection, the matrices have been stored in a specific format containing only the location of non-zero elements. For the specific use of this work, we read them with Matlab and (if required) reordered each matrix to a diagonal (preferably diagonally dominant) form using Matlab and saved them in separate text files in a plain format. A list of the Florida Sparse Matrix Collection matrices used in this work is provided in Table 5.1. Matrix names are listed in the first column, the second and third columns of the table include the size and the sparsity rate of the matrices, an the fourth column shows that whether or not the matrix is reordered to avoid zeros appearing on the matrix diagonal.

Matrix name	**Size**	**Sparsity**	**Reordered**
$mesh1e1$	48	87%	NO
$pivtol$	102	97%	YES
$Trefethen_150$	150	91%	NO
$Trefethen_200b$	199	93%	NO
$mesh3e1$	289	98%	NO

Table 5.1: Test matrices from the Florida Sparse Matrix Collection.

Another set of matrices used in this thesis is obtained from SPICE simulations. To extract real circuit simulation matrices from SPICE simulations the following steps are taken. First, in order to manipulate the internal processes of the SPICE simulation, we use Ngspice [90] under Linux (Ubuntu). By manipulating some lines in the Ngspice source code, it is possible to save the linear circuit matrix calculated by SPICE at the beginning of each NR iteration. This is the matrix which should be solved at each NR iteration to provide new entries for the next linearisation phase until the desired accuracy is achieved. The matrix is in the specific format shown in Figure 5.1 which only stores the location of non-zero elements.

This should be converted to an easily readable full matrix format. The presence of voltage sources and controlled sources can generate zeros on the diagonal of the matrix when modelling the circuit matrix using the MNA method, which can force the iterative

FIGURE 5.1: Format of the matrices extracted from Ngspice simulations

solution to stop because of the division by zero. To avoid these undesired zeros on the matrix diagonal, after saving the matrix, it is reordered to a diagonal (preferably diagonally dominant) form. Table 5.2 shows the list and specifications of the test matrices extracted from SPICE simulations.

Matrix name	**Size**	**Sparsity**	**Reordered**
$Raz1418$	5	28%	YES
op_design	9	63%	YES
$cir919res$	11	76%	YES
$oplab$	12	68%	YES

TABLE 5.2: Test matrices from Ngspice Simulations.

The test matrices extracted from SPICE simulations which are listed in Table 5.2 are related to just one iteration of the whole simulation process. During a complete circuit simulation process, the circuit matrix is generated and solved several times at each NR iteration and this process itself is repeated for every time step of the transient analysis. Therefore, for a complete simulation, there will be thousands of these matrices generated and solved for each circuit. In the next sections, for some of the test circuits in Table 5.2, a number of matrices at different time points and different NR iterations will be extracted to apply the matrix solution algorithm to them. This will be similar to a transient analysis simulation in which the matrix system needs to be solved several times at each time point. The solution vector for each iteration will be used as the initial guess vector for the iterative matrix solution in the next iteration.

5.2 Applying the Random Jacobi Method to Test Matrices

The functionality of Random Jacobi iterations was studied in Chapter 3 Section 3.2. It was shown that non-deterministic evaluation of matrix equations using the Random Jacobi iterative method results in the same convergence speed and accuracy as the normal Jacobi iterative method. Using Random Jacobi iterations would be the case in which the equations are evaluated asynchronously and independently on a parallel network of processors. In this section, the Random Jacobi method is examined on a number of real circuit matrices extracted from benchmark circuits or SPICE simulations. Parallelising the SPICE simulation process using the Random Jacobi iterative method for solving the system matrix and then combining it with the NR iterations is discussed. Then, the proposed method for solving the system matrix is simulated on single-core and many-core machines and the results are compared and discussed. Simulation on virtual and real multi-core machines have been performed using MPI.

The Random Jacobi iterative method is applied to the test matrices on single-core and many-core systems and the outcomes are compared to see the effect of the parallel evaluation of Jacobi iterations in a non-deterministic order on its convergence rate. When performing iterations, the two stop criteria are a pre-defined maximum iteration number and the Euclidean norm of the solution vectors obtained at each two successive iterations. For the following experiments, the stop threshold is set to be 10^{-5} with a fixed maximum iteration number of 200. If the Euclidean norm becomes smaller than the threshold value or the number of iterations reaches its maximum value, the system will stop iterating. If the first case happens, the final x values from each processor are considered as the solution vector. If the second case happens, the iterative matrix solution process will be considered as *'nonconvergent'* which needs more iterations to converge to the solution (for all of our test matrices, the iterative matrix solution converged in less than 100 iterations).

5.2.1 Test Matrix Set #1

The first set of test matrices are extracted from real circuits by Ngspice simulations as listed in Table 5.2. As explained in the literature review chapter Sections 2.2 and 2.3, during the SPICE transient analysis of a circuit, at each NR iteration, a nonlinear system of equations is constructed and then converted to a linear system in a matrix form of $Ax = b$. Then, the matrix system is solved for x and a new system of equations is constructed based on the most up-to-date version of the x vector and this process is repeated until the solution is obtained with a desired accuracy. Therefore, at each time point, the matrix solution phase needs to be performed several times and then repeated for several time points. Based on the transient analysis time range and also the required accuracy, there might be thousands of matrix solution tasks during each

transient analysis. In the conventional SPICE algorithm, the matrix solution phase is performed by direct methods such as LU-factorisation, which have some drawbacks as previously highlighted in the literature review Section 2.3.4.

The performance of iterative methods increases when a suitable initial guess vector is available. For time dependent and nonlinear problems, circuit simulation algorithms have two processes to generate the linear matrix system, which should be solved in the matrix solution phase. First, time stepping to generate linear models for time varying elements during a transient analysis and secondly, a linearisation process such as Newton-Raphson to deal with nonlinear elements. These are part of an outer loop for which normally a good initial vector is available from previous iterations. Therefore, normally a small number of iterations is required to obtain the approximate solution of the linear matrix system [54, 70].

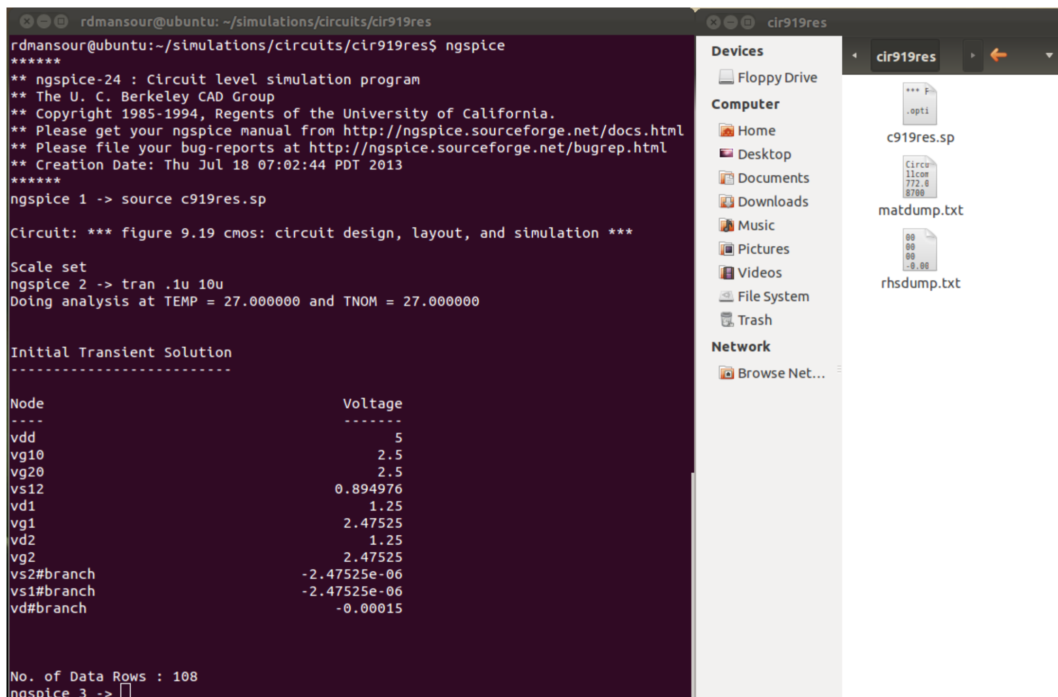


FIGURE 5.2: Transient analysis of *cir919res* circuit using Ngspice and the output text files containing extracted matrices.

In this section, the proposed iterative method will be applied to test matrices extracted from several time points and NR iterations of a transient circuit simulation. To do so, a transient SPICE simulation on the under test circuit is performed and a number of matrix sets at the beginning of the matrix solution phase is extracted from some randomly chosen NR iterations. In order to extract intermediate matrices from Ngspice simulation process it is required to modify the *"cktload.c"* file in the Ngspice source code to dump the A and RHS matrices into an output text file at the beginning of each matrix solution iteration. Then, by performing a transient analysis on our under test circuit, two text files will be generated (*matdump.txt* and *rhsdump.txt*) which contain the A and RHS matrix values. A screen shot of an example simulation on one of our

test circuits (*cir*919*res*) is presented in Figure 5.2. The contents of *matdump.txt* and *rhsdump.txt* files are also shown in Figure 5.3.

matdump.txt (~/simulations/circuits/919res2) - gedit

```
Circuit Matrix
11      real
11      1       1
1       1       4e-05
5       1       -2e-05
7       1       -2e-05
10      2       1
2       2       0.0001
6       2       -0.0001
9       3       1
3       3       0.0001
8       3       -0.0001
1       11      1
2       10      1
3       9       1
2       6       -0.0001
6       6       0.0001001
5       6       0.000382984021459833
4       6       -0.000382984021459833
1       5       -2e-05
6       5       0
5       5       2.00000018621724e-05
4       5       -8.62172415459823e-13
6       4       0
5       4       -0.00045345601435194
4       4       0.000906915431597486
8       4       0
7       4       -0.00045345601435194
3       8       -0.0001
4       8       -0.000382984021459833
8       8       0.0001001
7       8       0.000382984021459833
1       7       -2e-05
4       7       -8.62172415459823e-13
8       7       0
7       7       2.00000018621724e-05
0       0       0.0
```

rhsdump.txt (~/simulations/circuits/919res2) - gedit

```
0
0
0
-0.000950703047761743
0.000455351198891176
0
0.000455351198891176
0
2.5
2.5
5
```

FIGURE 5.3: The A and RHS matrices extracted from Ngspice simulation into the *matdump.txt* and *rhsdump.txt* files.

A number of sample matrices are extracted from different NR iterations at various time points of the transient analysis and solved them using the Random Jacobi iterative method in a highly parallel form using MPI on a many-core cluster[1]. The simulation results are shown in Table 5.3. *Time Point* and *NR Iteration* show which time point of the transient analysis and which Newton-Raphson iteration is used to extract matrices and solve them. The *Iterations* column shows the number of iterations to converge to the

[1]Details of our fine-grained highly parallel simulations using MPI and C++ on a single core machine and also a many-core cluster are provided in Appendix B

solution according to the stop criterion, which is the Euclidean norm becoming smaller than 10^{-5} for these simulations.

Matrix name	**Size**	**Time Point**	**NR iteration**	**Iterations**
op_design	9	1	1	5
		1	2	3
		1	3	1
		5	1	13
		5	2	4
		5	3	1
oplab	12	2	2	30
		2	3	10
		2	4	11
		3	2	11
		4	2	19
cir919res	11	0	3	3
		0	4	3
		0	5	3
		0	13	1
		4	2	3

Table 5.3: The Parallel Random Jacobi method applied to the circuit matrices from different iterations of various time points of a transient analysis.

It can be seen that the proposed iterative method is able to solve the system of equations at each Newton-Raphson iteration after the linearisation phase in a small number of iterations. Besides, by looking at the results more closely, one can see the effect of using the solution of the previous iteration as the starting point of the current iteration in accelerating the iterative process. For example, in the simulation results of the first test matrix, *op_design*, there are three matrices from time point number 1 which are solved in three successive NR iterations. The first matrix requires five iterations, the second three and the third one. This means that, for each time point, as the simulation proceeds, fewer iterations for matrix solution are required.

For the simulations of the first test matrix set, since the matrix sizes are small, it is not possible to have a proper execution time measurement to compare the iterative solution performance with other approaches. In the next section, another set of test matrices with relatively larger sizes will be used for run time comparison.

5.2.2 Test Matrix Set #2

The second set of matrices used for our simulations is some benchmark matrices from the University of Florida Sparse Matrix Collection in Table 5.1, which are sparse asymmetric square matrices with RHS vectors.

5.2.2.1 Single-core Simulations

Table 5.4 shows the simulation results of applying Random Jacobi iterations to the second test matrix set on a single processor with a sequential algorithm written in C++. For each test matrix, its size, the number of iterations required to satisfy convergence criteria, and the value of Euclidean norm between the solutions of the last two iterations are also given in the table. The C++ code of the algorithm which is used for the iterative methods is included in Appendix A, Section A.4. All the matrix solution process is executed on a single processor (Intel CPU, 2.67 GHz, under Windows 7, 64-bit) and equations are solved on this single processor one by one in a sequential manner as represented in Figure 5.4. At the beginning of the algorithm, the A matrix and its RHS vector are read from two text files. In order to simulate the randomness in evaluating equations in Jacobi iterations, at the start of each iteration, a random vector containing numbers from 1 to n (matrix size) is generated and the order in which the equations are evaluated is based on this random vector. This leads to a non-deterministic order of evaluation of the equations.

Matrix name	Size	Iterations	Euclidean norm
$mesh1e1$	48	57	0.00000881
$pivtol$	102	25	0.00000700
$Trefethen_150$	150	95	0.00000905
$Trefethen_200b$	199	28	0.00000795
$mesh3e1$	289	64	0.00000982

TABLE 5.4: The Random Jacobi method on a single-core.

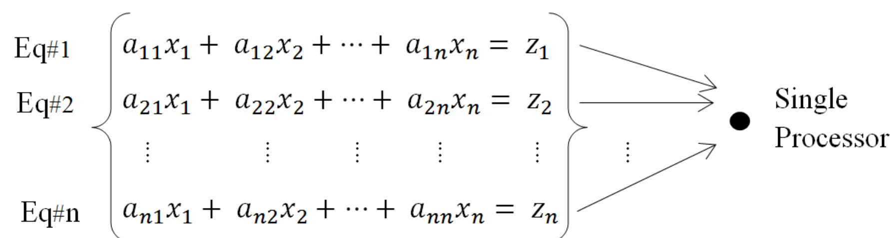


FIGURE 5.4: A highly parallel structure: one processor per circuit equation.

These results will be compared to the results of parallel simulation of the Random Jacobi method on many-core systems. We are practically performing a sequential evaluation of Jacobi iterations and the Jacobi iterative method is best suited for parallel evaluations. Therefore, the simulation results of Gauss Seidel iterations, which is basically a sequential algorithm, on the same single processor are also included to have some extra measures to compare with the parallel simulation results. The simulation results for the Gauss Seidel method on single processor is presented in Table 5.5.

The execution times of the Gauss Seidel and the Random Jacobi methods on a single processor are presented in Table 5.6. Times are shown in seconds. As expected, the

Matrix name	Size	Iterations	Euclidean norm
mesh1e1	48	11	0.00000740
pivtol	102	12	0.00000869
Trefethen_150	150	13	0.00000822
Trefethen_200b	199	9	0.00000105
mesh3e1	289	19	0.00000984

TABLE 5.5: The Gauss Seidel method on a single-core.

Gauss Seidel method with its sequential algorithm performs much faster than the Jacobi method, which is more suitable for parallel implementations. The results will be compared to parallel simulation results in the next sections.

Matrix name	Size	Gauss Seidel Exe. Time†	Random Jacobi Exe. Time†
mesh1e1	48	0.011	0.049
pivtol	102	0.039	0.065
Trefethen_150	150	0.067	0.396
Trefethen_200b	199	0.084	0.219
mesh3e1	289	0.299	0.999

TABLE 5.6: Execution time comparison of the Gauss Seidel and Random Jacobi iterations on a single processor. † The unit for time measurement is *Seconds*.

5.2.2.2 Virtual Many-core Simulations

In this section, the Random Jacobi iteration is evaluated with the same set of data using MPI under Ubuntu on a single processor. Details of using MPI and C++ programming for the simulations of this section can be found in Appendix B, Section B.1. To make the evaluation process virtually parallel, for each test matrix, the number of virtual cores is set to be equal to the number of matrix equations (matrix size). In other words, one virtual processor has been allocated to each circuit equation. As with the single processor case, there is only one processor for the solution of all equations (Figure 5.4). However, in this section the evaluation of equations on the single core is done virtually in parallel. The simulation results are shown in Table 5.7. Using the same algorithm, the number of iterations to converge to the solution is the same as the sequential evaluation on a single processor but the execution time is considerably more. Simulation results show that the virtual parallel system, which is used for running MPI, has serious issues in handling parallel processing. As the size of test matrices increased, for the last two test matrices, the virtual parallel system was unable to obtain a solution and stopped working at some point. Therefore, the virtual parallel system seems not to be a suitable platform to run MPI for highly parallel simulation purposes. Based on the results obtained in this section, MPI on virtual parallel processors will not be used for our parallel simulations

and the main concentration will be on the parallel simulations on real parallel platforms, which is discussed in the next section.

Matrix name	Size	Iterations	Euclidean norm	Exe. Time†
mesh1e1	48	57	0.00000888	40.08
pivtol	102	25	0.00000700	45.32
Trefethen_150	150	95	0.00000905	285.20

TABLE 5.7: Random Jacobi using MPI on a virtual parallel network of processors. † The unit for time measurement is *Seconds*.

5.2.2.3 Real Many-core Simulations

To perform Random Jacobi iterations using real parallel processors, the test matrices have been examined on the Iridis[2] Computer Cluster using MPI. Iridis cluster has 750 compute nodes with dual 2.6 GHz Intel processors and each compute node has 16 CPUs per node with 64 GB of memory. For the details of MPI environment, MPI functions and directives, which are used in these simulations, and how it is incorporated in C/C++ programming environment on the Iridis cluster see Appendix B.

At the beginning of the MPI program, according to the number of matrix rows, which will determine the number of required parallel processors (for example n), the ranks from 0 to $n-1$ is given to the processors. This is shown in Figure 5.5.

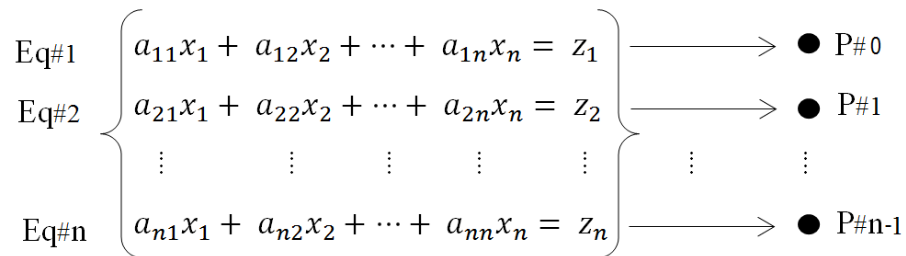


FIGURE 5.5: A highly parallel structure: one processor per circuit equation.

Processors $P_{\#0}$ to $P_{\#n-1}$ are responsible for solving the equations $Eq_{\#1}$ to $Eq_{\#n}$, respectively. Processor $P_{\#0}$ solves the first equation for x_1 as the unknown variable, assuming that other x values are known from the initial condition at the beginning of iterations and later from the solutions obtained by other processors. The same procedure is taken by other processors for their corresponding equations. As a new value is calculated by a processor it needs to be sent to all other processors so that they can use the new data for a more accurate evaluation of the next iteration.

[2] Iridis is the supercomputer of the University of Southampton. The latest generation is called Iridis4. More information can be found in Appendix B, Section B.2

Figure 5.6 shows how each processor communicates with all other processors at each iteration for a network containing 6 parallel processors.

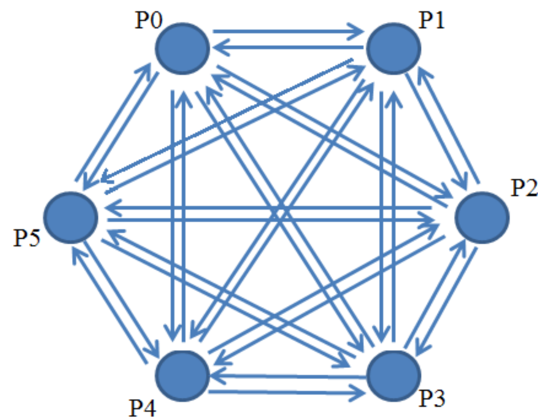


FIGURE 5.6: Communication between processors for the parallel Random Jacobi method.

The matrix solution algorithm is similar to the one used in Section 5.2.2.2 for which some parts of its *C* code including the MPI commands and directives can be found in appendix B, Section B1. The flowchart in Figure 5.7 shows the parallel Jacobi matrix solution process using MPI on Iridis. After the initialisation step, the processor with rank 0, receives the maximum number of iterations and the value for the Euclidean norm as the iteration control criteria. Then using the *MPI_Bcast* function, which is a communication routine in MPI, it broadcasts the number of processors, the number of iterations, and the Euclidean norm to all other processors. The rank 0 processor then reads matrix entries for the A matrix and RHS vector in $Ax = b$ matrix system and scatters each row of the matrix system using the *MPI_Scatter* routine to the corresponding processor according to the row number and the processor rank. At this time, each processor has received a complete equation as a row of the matrix system. Then the Random Jacobi iterations start. Each processor calculates one of the entries of the unknown vector, x, and collects the other entries of the unknown vector from all other processors using the *MPI_Allgather* function, which is a collective routine in MPI communication.

At this step, the convergence criteria should be checked. The Euclidean norm between the obtained solutions for the unknown vector in the last two successive iterations is calculated and compared to the predefined value for stopping the iterations. The second factor to be checked is the maximum number of iterations. The Random Jacobi iterations will be repeated until at least one of the stop criteria is met.

When doing Jacobi iterations using MPI on a real parallel network of processors by having one processor per matrix equation, the order of equations evaluation is completely non-deterministic and there is not any control on the evaluation process and communication between processors. This is a very good example of evaluation of Random Jacobi iterations.

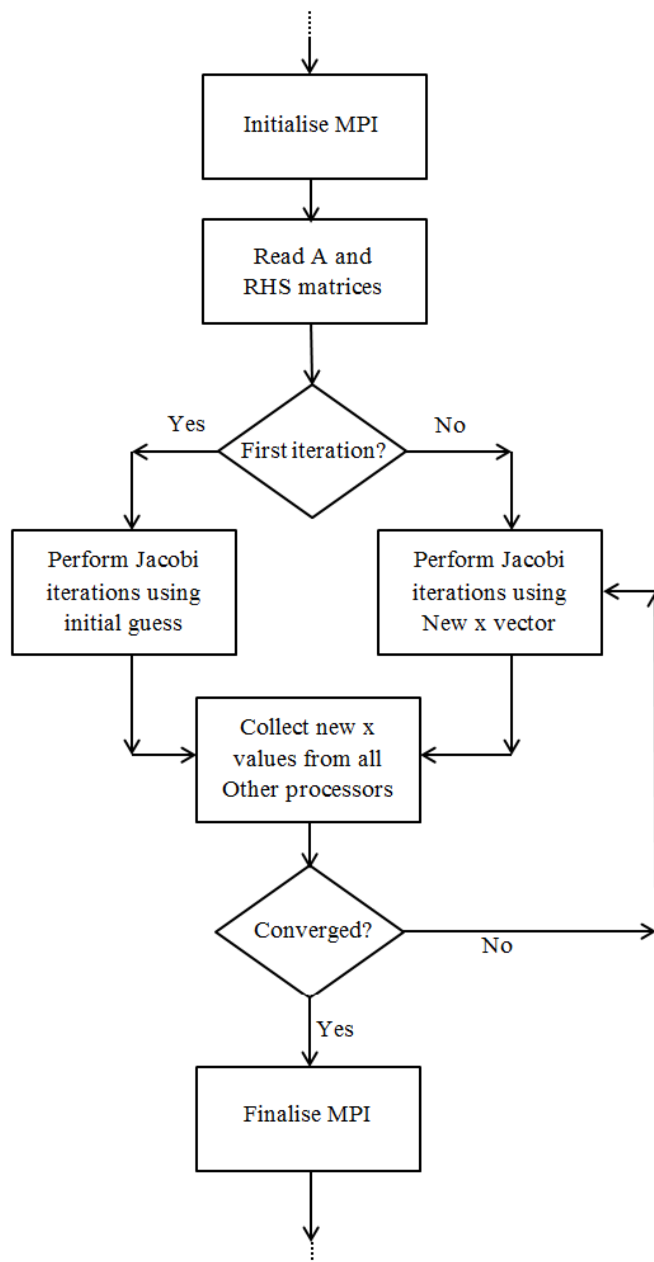


FIGURE 5.7: Parallel Jacobi Matrix Solution Process Using MPI on Iridis.

The simulation results for performing the Random Jacobi method are given in Table 5.8. By comparing the number of iterations with the results in Table 5.4, It can be seen that evaluation of Random Jacobi iterations on single-core and many-core systems results in the same number of iterations for the same stop criteria but with different simulation times, which will be compared in Figure 5.8.

In Section 5.2.2, three iterative approaches are discussed for the matrix solution process. The Gauss Seidel and Jacobi methods on a single-core and the Random Jacobi iterations on a highly parallel many-core system. Figure 5.8 compares the execution times of these

Matrix name	Size	Iterations	Euclidean norm	Exe. Time†
mesh1e1	48	57	0.00000890	0.073
pivtol	102	26	0.00000817	0.148
Trefethen_150	150	95	0.00000911	0.188
Trefethen_200b	199	28	0.00000796	0.253
mesh3e1	289	64	0.00000983	0.330

TABLE 5.8: Random Jacobi using MPI on a parallel many-core cluster. † The unit for time measurement is *Seconds*.

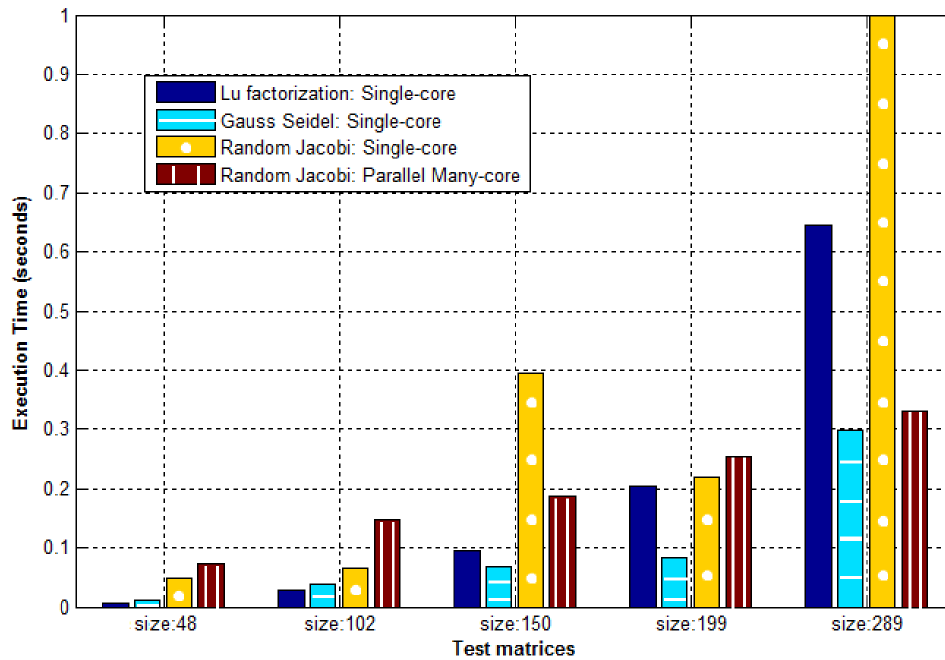


FIGURE 5.8: Execution time comparison for 4 different matrix solution approaches.

methods on the test matrices of this section and also represents the execution time when matrices are solved by a direct method (LU-factorisation) on a single processor.

The direct method, LU-factorisation, performs better on matrices with smaller size but its performance declines as the matrix size increases. The Jacobi method on a single-core, as expected, has the worst performance since it is suitable for parallel applications. The Gauss Seidel method performs better than LU-factorisation when the size of the matrix increases. However, although the Gauss-Seidel approach scales better than LU-factorisation, it seems that it is still not scaling well with the problem size. For example, when the matrix size is increased from 199 to 289, the simulation time of the Gauss-Seidel method is increased by almost a factor of 4. The Parallel Random Jacobi on a many-core system has the worst performance when the problem size is small but it is seen that it performs much better when the matrix size increases. Besides, the parallel

Random Jacobi method scales better than the other three methods with the problem size. There is only a slight increase in its simulation time when the matrix size has changed from 150 to 199 and then 289.

5.2.3 Optimising the Communication Between Processors

The simulation results in Figure 5.8 show parallel evaluation of Jacobi iterations on our many-core cluster by allocating one processor to each circuit equation. These results shows quite a good performance for parallel matrix solution especially when used for large matrices. Instead of evaluating the whole problem by a single processor or a few number of processors, each equation is solved by one processor. Therefore, the time required for solving equations should be much shorter than the single processor case. However, the communication between the processors becomes very important in highly parallel computing. Although the required computational effort is distributed across a large number of processors, communication is a major part of the execution time. At each iteration, every single processor communicates with all other processors to provide them with its new x value and collect the most up-to-date x values from other processors.

The process of iterative solution of the linear equation system can be optimised for a number of factors such as communications and computations. This involves reducing the required communication between the processors and simplifying the computations when possible.

As discussed in the literature review Section 2.2.3.3, because in electronic circuits each node is connected only to a few adjacent nodes, circuit simulation matrices are very sparse and even very large matrices have only a few non-zero elements in each row. This means that for solving a circuit equation, only a few entries from other equations are required based on the structure of the A matrix. Therefore, basically, each processor needs to send/receive updates only to/from a few other processors. This can notably decrease the amount of required communications. Besides, in its primary form, the Euclidean norm as a stop criteria is calculated by each processor for the whole solution vector. However, each processor only needs to calculate the Euclidean norm for its own x value for every two successive iterations to stop iterating when it has converged.

As a simple example to review the optimisation process, consider a matrix system of six equations as shown in Figure 5.9. Each equation will be handled by one processor. Thus, six processors (P_0 to P_5) are allocated to the equations. The initial guess vector and the solution vector are represented by $x_{initial}$ and $x_{solution}$, respectively.

According to the first row of the matrix, which is assigned to P_0, the corresponding x value is calculated by Equation 5.1. Except for the diagonal element in the first row, there is only one non-zero element in this row and other elements of the first row are zero. Hence, as it can be seen in Equation 5.1, the processor with rank 0 only needs

$$\begin{matrix} P_0: \\ P_1: \\ P_2: \\ P_3: \\ P_4: \\ P_5: \end{matrix} \quad \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 & 1 \\ 1 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & -2 & 1 \\ -1 & -1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$x_{initial} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad x_{solution} = \begin{bmatrix} 0.4667 \\ 0.2667 \\ -0.1556 \\ -0.0667 \\ 0.3667 \\ 1.0000 \end{bmatrix}$$

FIGURE 5.9: A 6 by 6 test matrix.

updates of the x_3 element of the x vector, which is calculated by the processor with rank 3 at each iteration. Moreover, the first column of the A matrix shows that processors P_3 and P_5 do not need to receive updates of the x_0 from P_0 because their corresponding element in the first column is equal to zero. Therefore, the processor with rank 0 only needs to receive the x value from P_3 and send its x value to P_1, P_2, and P_4. Equation 5.2 to Equation 5.6 show the required calculations by other processors for obtaining their corresponding x values.

$$P_0 : x_0^{n+1} = \frac{1 + x_3^n}{2} \tag{5.1}$$

$$P_1 : x_1^{n+1} = \frac{x_0^n - x_5^n}{-2} \tag{5.2}$$

$$P_2 : x_2^{n+1} = \frac{-x_0^n}{3} \tag{5.3}$$

$$P_3 : x_3^{n+1} = \frac{2x_4^n - x_5^n}{4} \tag{5.4}$$

$$P_4 : x_4^{n+1} = \frac{x_0^n + x_1^n}{2} \tag{5.5}$$

$$P_5 : x_5^{n+1} = \frac{1}{1} \tag{5.6}$$

Based on the location of the non-zero elements of the matrix, the required send/receive pattern for this matrix system is represented in Table 5.9. As explained, each processor does not need to communicate with all other processors and it exchanges data only with the required processors. This can become even more important when dealing with large matrices. Considering the fact that circuit simulation matrices are very sparse, employing this communication pattern can significantly decrease the required communications. Such a communication pattern is represented in Figure 5.10. Compared to the Figure 5.6, the new pattern needs significantly less communications. The light

dashed arrows stand for the communications which are no longer required due to the new communication pattern obtained based on the sparsity pattern of the test matrix.

Processor	**Send to**	**receive from**
P_0	P_1 and P_2 and P_4	P_3
P_1	P_4	P_0 and P_5
P_2	–	P_0
P_3	P_0	P_4 and P_5
P_4	P_3	P_0 and P_1
P_5	P_1 and P_3	–

Table 5.9: Send/Receive pattern for reducing the required communication.

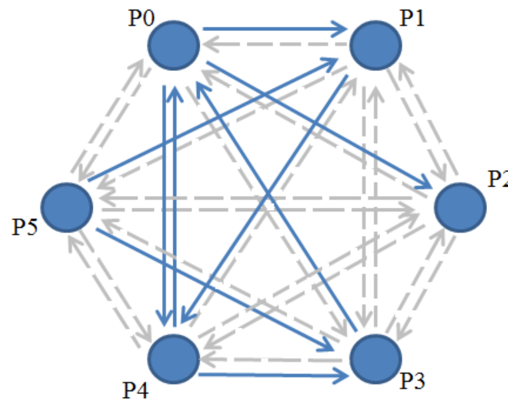


Figure 5.10: Communication between processors for the parallel Random Jacobi method when using the proposed optimised pattern.

A pre-analysis process is required to generate the correct pattern for a more efficient communication. For sparse matrices, the required pre-processing is expected to be negligible compared to the amount of reduction in the communications and thus leads to a better performance. This will be investigated on test matrices. This pre-analysis algorithm can be integrated into the matrix construction and solution process as a communication optimisation stage in our proposed iterative approach. At each time point, once the matrix constructed, its sparsity pattern will remain the same throughout the iterative solution process. Therefore, the pre-analysis process needs to be performed only once in order to obtain the best pattern to decrease the communication between processors as shown in Figure 5.11. The optimisation stage will have the conductance matrix as its input and based on the locations of the non-zero elements, will generate two sets of data as its output, called send and receive buffers which will determine each processor needs to communicate with which other processors.

For previous simulations, since all the processors needed to communicate with each other, all-to-all MPI communication routines such as *MPI_Allgather* were used. For the introduced optimised communication, *MPI_Send* and *MPI_Receive* routines are used as one-to-one communication directives. According to the number of non-zero elements

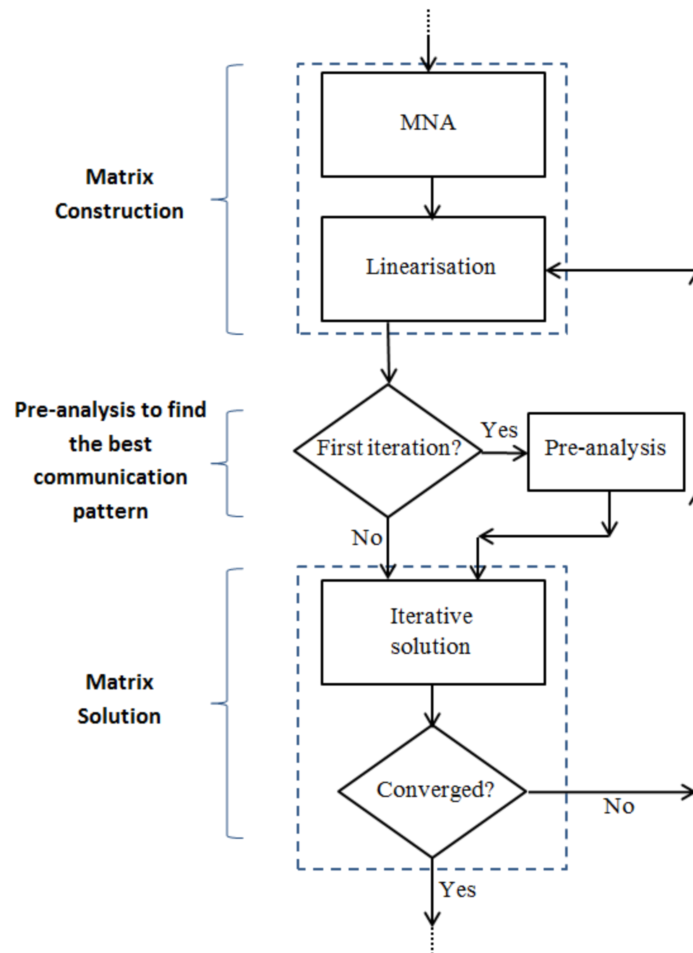


FIGURE 5.11: Pre-analysing the circuit matrix structure to obtain a suitable pattern to reduce required communications between processors.

in each row and column, the number of sends and receives are determined. Then, based on the locations of the non-zero matrix entries and their values, two vectors as send and receive buffers are created containing the rank of the processors involved in the communication and the value which is being sent or received. Since at each iteration, each processor only sends its own x value to the required destinations, the send buffer has only two elements: sender's rank and the x value. The length of the receive buffer can vary depending on the non-zero pattern of the matrix.

Although all-to-all communications, which are completely performed and controlled by the system, might have better performance than one-to-one communications, which are controlled by the user, it is expected that by reducing the number of communications per iteration, a better performance can be obtained overall. However, there will be a tradeoff between the sparsity of the matrix and the required pre-processing.

Coming back to the above example, the x values in the first iteration will be calculated using the initial guess vector without the need to communicate with other processors. For this example, the first set of x values is shown by Equation 5.7.

$$x_0^1 = 1.00,\ x_1^1 = 0.00,\ x_2^1 = -0.33,\ x_3^1 = 0.25,\ x_4^1 = 1.00,\ x_5^1 = 1.00 \qquad (5.7)$$

Now, to proceed with the second iteration, these values should be exchanged between processors that need them based on the send/receive pattern in Table 5.9. Send and receive buffers for each processor in the end of the first iteration are presented in Figure 5.12. The processor with rank 3, for example, needs to send its x value of 0.25 to the processor with rank 0 and receive the x value of 1.00 from the processor with rank 4 and also the x value of 1.00 from the processor with rank 5. Table 5.10 shows the simulation results at some iterations of the matrix solution process.

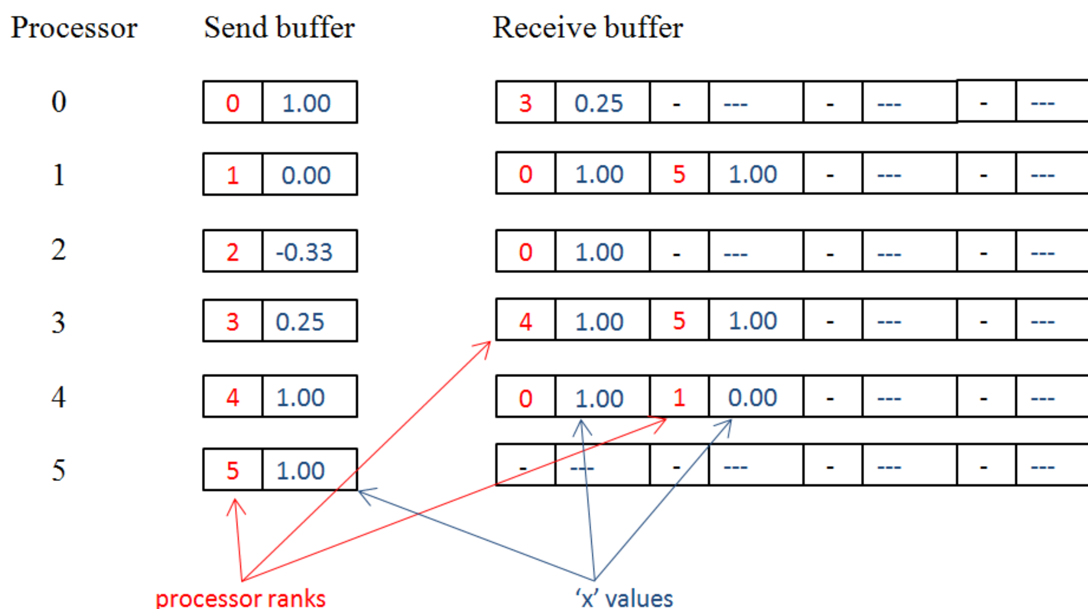


FIGURE 5.12: Send and receive buffers for exchanging data between processors.

	Initial value	Iteration 1	Iteration 3	Iteration 6	Iteration 10
x_0	1	1.00	0.62	0.47	0.46
x_1	1	0.00	0.18	0.27	0.26
x_2	1	-0.33	-0.20	-0.15	-0.15
x_3	1	0.25	0.00	-0.07	-0.06
x_4	1	1.00	0.31	0.35	0.36
x_5	1	1.00	1.00	1.00	1.00

TABLE 5.10: The solution vector of the matrix system obtained in different iterations.

The simulation results of performing the matrix solution phase by the parallel Random Jacobi method using the optimised communication pattern are shown in Table 5.11. It can be seen that for three of the test matrices there is an improvement in the solution time while for the other two matrices the execution time has become worse.

Matrix name	Size	Iterations	Exe. Time†
$mesh1e1$	48	57	0.067
$pivtol$	102	26	0.052
$Trefethen_150$	150	95	0.385
$Trefethen_200b$	199	28	0.450
$mesh3e1$	289	64	0.252

TABLE 5.11: Random Jacobi using MPI on a parallel many-core cluster with optimised communication. † The unit for time measurement is *Seconds*.

Figure 5.13 shows the comparison between execution times of the Random Jacobi iteration with normal and optimised communications. For a better comparison, the runtime of the LU-factorisation method is also included in the graph. An interesting observation is that the runtime improvement has happened notably in the test matrices named $pivtol$, $mesh1e1$, and $mesh3e1$ and according to the test matrices properties table (Table 5.1), which is given in Section 5.1, these two matrices are very sparse. Therefore, based on the suggested communication pattern, the required communication between processors for these two matrices are very low. Furthermore, comparing the results of two different approaches for the Random Jacobi method with the LU-factorisaion shows that the results of the iterative solution when the size of the matrix becomes large are comparable to the direct method and even become better for the largest test matrix.

To investigate the effect of sparsity of the test matrices on the amount of communications required for our proposed optimised communication pattern, a number of test matrices were needed with specific properties so that their size and also the number of iterations need to be the same to eliminate the effect of size and iteration number on the execution time. For this purpose, four test matrices are generated all with the same size equal to 250*250, all converge to the solution within 19 iterations, and the difference is in their sparsity rates which are set to be 95%, 75%, 55%, and 35%. Simulations results are shown in Table 5.12. The execution time of the test matrix $M250_95$, which is the most sparse matrix among all with the sparsity rate of 95%, is the shortest execution time equal to $0.96s$. On the other hand, the test matrix $M250_35$, which is the least sparse matrix among all with the sparsity rate of 35%, has the longest execution time equal to $5.5s$. This confirms the effect of sparsity on the execution time because of its effect on the amount of required communication between the processors in the proposed communication pattern.

5.2.4 Time Measurement

In the current work, several different sequential and parallel algorithms are used on different single-core and multi-core systems. In order to compare the performance of the tested methods, it is required to have a time measurement method which measures the simulation time in the same way for all the algorithms on different platforms. We used

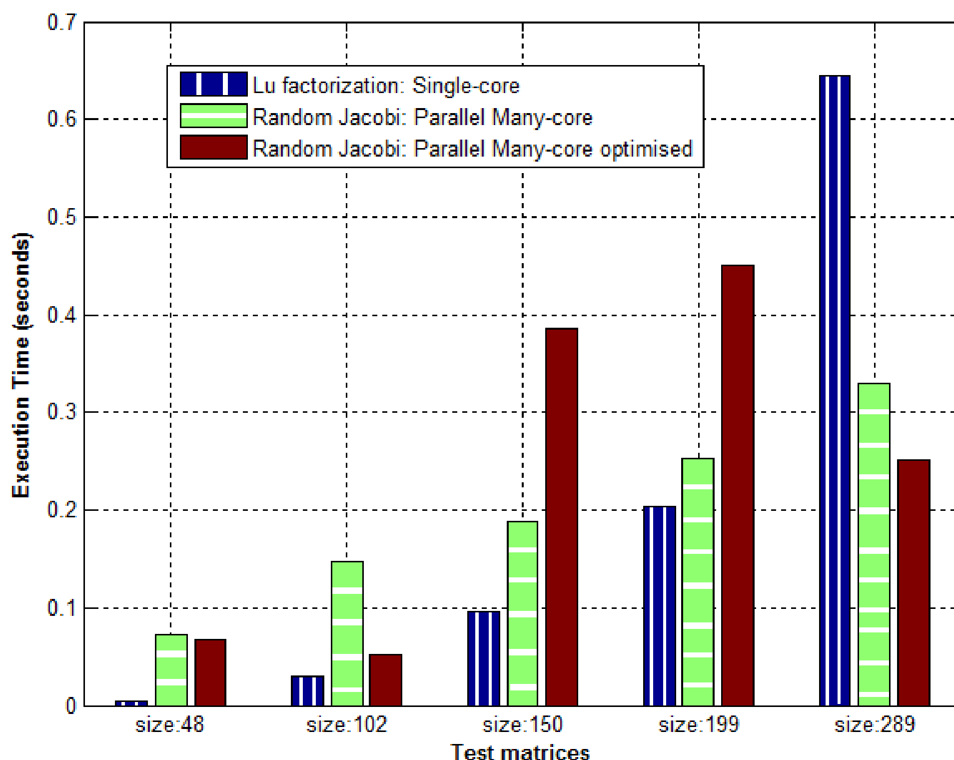


FIGURE 5.13: Execution time comparison for three different matrix solution methods.

Matrix name	Size	Iterations	Sparsity	Exe. Time†
$M250_95$	250	19	95%	0.96
$M250_75$	250	19	75%	2.83
$M250_55$	250	19	55%	4.25
$M250_35$	250	19	35%	5.50

TABLE 5.12: Effect of sparsity on matrix solution time using the optimised communication pattern. † The unit for time measurement is *Seconds*.

sequential simulations by Matlab, sequential and parallel simulations by C/C++, and parallel simulations using MPI on a cluster. Therefore, real wall times (elapsed times) are used for the simulation time of different algorithms.

In Matlab, there is a stopwatch timer to measure the performance. Its uses *tic* and *toc* functions to measure the elapsed time between two points of the algorithm. In other words, it returns the real execution time between the two points. In C/C++, the *clock* function is used to measure the processor time consumed by the program. To make the performance measurement similar to the one used in Matlab, the value returned by the *clock* function should be compared to a previously stored value as the start time. This means that the *clock* function should be called at the beginning and the end of the

algorithm and the difference between the two time values should be calculated as the real execution time. In MPI, there is no global time measurement due to the fact that there is no control on the parallel processors running the parallel algorithm. However, it is possible to use a function called *MPI_Barrier* to block all the processes that have reached this point and then all the processes will start at the same time as the last process reaches the point. Then, the wall time of each process can be measured using another MPI directive called *MPI_Wtime* which returns the elapsed time on the calling processor. As in the previous cases, in order to measure the real execution time between the two points of the algorithm, *MPI_Wtime* should be called twice (at the beginning and end of the algorithm). Examples of the time measurement methods can be found in the codes provided in Appendix A.

5.3 Summary and Discussion

In this chapter, Random Jacobi iterations were tested on different test matrices using a single-core machine, a virtual many-core system, and a real many-core cluster. Simulations show that performing Random Jacobi iterations on a massively parallel network of processors by allocating one processor per circuit node results in the same number of iterations and almost the same Euclidean norm as performing it on single-core or virtual multi-core machines. The virtual parallel system is found not to be a suitable platform for highly parallel simulations as the number of required processors increases. Therefore, most of the simulations, comparisons, and discussion are made on the results from the single-core and real many-core evaluations.

A number of simulations are performed on the execution time of the direct and iterative matrix solutions. When the size of the test matrices are small, the direct solution, LU-factorisation, on a single-core performs better than the iterative solutions. However, it does not scale with the problem size. The Gauss-Seidel method performs better than the Jacobi method on a single-core as expected due to the parallel nature of the Jacobi method. The Gauss-Seidel method also performs much better than the LU-factorisation as the size of matrices increase. Our proposed parallel Random Jacobi iterations on a many-core system obtains better results for bigger circuits. For example, when the size of the matrix is 150, the execution time of the parallel Random Jacobi method is twice the execution time of the LU-factorisation method but for the matrix size of 289, it performs two times faster than the LU-factorisation method.

Furthermore, by exploiting the sparsity pattern of the test matrices, it is shown that an optimised communication pattern can be used for evaluating sparse matrices on highly parallel systems to reduce the amount of required communications between processors and hence reduce the solution time. An extra reduction in the simulation time is achieved for the matrices with a higher rate of sparsity.

To the best of our knowledge, there is no work reported in the circuit simulation area with this level of fine-grained parallelisation for the matrix solution phase. The conventional matrix solution techniques for circuit simulation use direct methods which are very difficult to paralleise. Our proposed iterative method can be performed in a highly parallel way by solving each of the matrix equations on a separate processor independently and thus providing a very fine-grained algorithm for the matrix solution phase.

Chapter 6

Simultaneous Analysis of the Two Simulation Phases

In the last two chapters, the two main phases of the circuit simulation process, device evaluation and matrix solution, were discussed and new approaches were proposed for parallel and distributed evaluation of circuit equations on many-core systems. As reviewed in Chapter 2 Section 2.4.2, one of the factors which limits the possibility of totally parallelising SPICE-like algorithms is the barrier between these two phases. It was shown that each phase on its own can be simulated in parallel in order to increase the simulation speed. However, there are some limitations on evaluating the two phases simultaneously. In the conventional circuit simulation algorithm, at each NR iteration, the device evaluation phase needs to be completely finished before the matrix solution phase starts. Also the next NR iteration cannot start until the matrix solution phase is done and new values for the unknown vector are available. This is the main bottleneck to totally parallelising the circuit simulation process. In this chapter, the specific properties of the proposed methods are used to mix the matrix solution and device evaluation phases and perform a simultaneous evaluation of the two phases to speed up the simulation process and investigate the possibilities for eliminating the above mentioned barrier. The proposed algorithm is applied on test circuits and simulation results are compared and discussed in the final section of this chapter.

6.1 Properties of the Proposed Methods

By looking at NR iterations more closely, it can be seen that the main reason for a barrier between the device modelling and matrix solution phases is the nature of direct matrix solvers implemented in the SPICE algorithm. Although direct matrix solutions are widely used in circuit simulation algorithms, Jacobi type iterative approaches for matrix solution purposes can be performed totally in parallel on a many-core system. Therefore,

by evaluating each of the circuit equations on a distinct processor, the unknown values of the circuit can be evaluated independently and calculated asynchronously across a large number of parallel cores. The advantage of such a parallel and asynchronous computation is that there is no need for the total completion of the matrix solution phase to pass the obtained values to the device evaluation process. As soon as a value for one of the unknowns of the circuit is obtained, it can be used by the device modelling iterations to produce a more accurate model. Then, as a new linear model is generated, the processors which are at the beginning of their calculations can use the more up to date values of linear models. Figure 6.1a illustrates this difference. One can see that there is an undesired stop during the matrix solution process in NR iterations due to the unavailability of new values for the next device modelling iteration. Although device evaluation and matrix solution phases in conventional methods can be performed in parallel separately, this barrier between the two phases limits the amount of possible parallelisation. It can be seen in Figure 6.1b that this issue can be resolved by employing a parallel and iterative matrix solver which regularly exchanges data with the linearisation phase without forcing it to stop its calculations. By mixing the two phases of the simulation, it is possible to accelerate the overall simulation process by eliminating the undesired stops of the NR iterations.

The conventional SPICE algorithm uses NR iterations for linearisation purposes. In our simulations for mixing the two phases of circuit analysis process, the Secant method is used for linearisation along with our iterative matrix solution approach. Parallel processors will start solving their corresponding equations independently and where it is required, the device models will also be calculated by the same processors. Therefore, simultaneous evaluation of the two phases will be possible.

6.2 Simultaneous Evaluation

It needs to be recalled that we are using a many-core system for the matrix solution phase to evaluate a parallel and asynchronous Jacobi-type solution on a highly distributed network of processors. Each circuit equation is allocated to one distinct processor in the many-core parallel system. Therefore, the process of finding the unknown elements of the circuit is based on simultaneous solution of the n circuit equations rather than a regular matrix solving procedure. There is no global matrix and the rows of the matrix system (circuit equations) are distributed across a large number of parallel processors and solved locally. Each core is meant to continuously calculate the value of one of the unknown variables and update its value by gathering the values of other required variables from the corresponding cores. The device modelling phase can also be continuously performed on the same cores based on the new values.

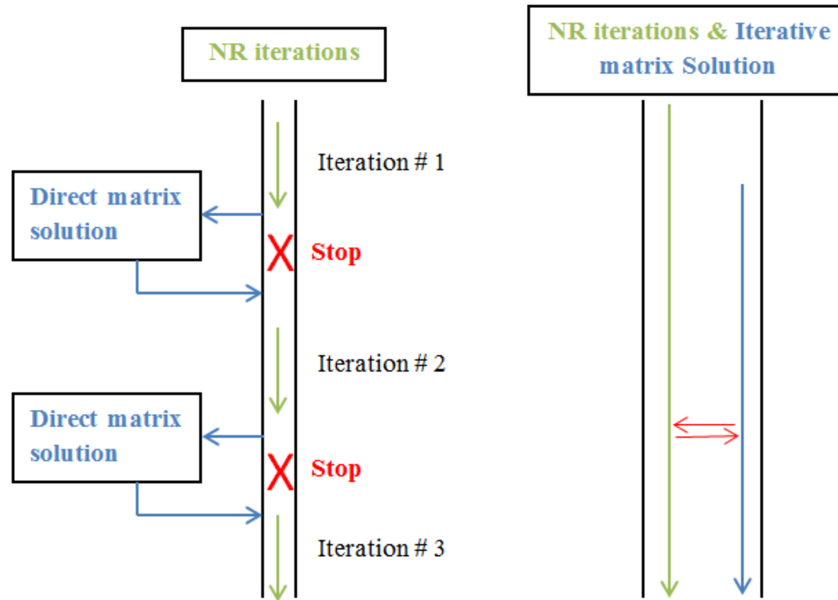


FIGURE 6.1: a) Undesired stops of the linearisation phase when using direct matrix solution methods b) Possibility of resolving the issue by using a parallel and iterative matrix solver

6.2.1 Circuit Equations and Linear Device Models

To have a better understanding of the proposed method, it is applied to one of the test circuits from Chapter 4 Figure 4.10. The circuit has eleven unknown variables (eight node voltages and three branch currents). Since some of the nodes or branches are connected to constant sources, calculation of their values are straight forward. To distribute the solution across parallel cores using the proposed mixed method, each of the circuit equations should be assigned to one core to be evaluated continuously. Here are the required calculations, which need to be done by each core, to obtain the solution of their corresponding unknown element:

core #1:
Equation 6.1 shows the calculations that core #1 needs to continuously perform in order to calculate the value of i_{vs1} as one of the circuit unknowns. gb_{11} has a fixed value (conductance) and remains constant for the duration of the simulation. At the beginning of the process, v_{g1} and v_{g10} are equal to values of the initial guess vector and for the rest of the process will be obtained from the cores which are allocated to calculate them. Therefore, as soon as new values are ready for v_{g1} and v_{g10} from the corresponding cores, they will be passed to the core #1 and a new more accurate value will be computed for i_{vs1} based on these most recent values. It should be noted that there is no need for linearisation computations on this core since its equation does not include any elements that need nonlinear device modelling.

$$i_{vs1} = gb_{11}(v_{g1} - v_{g10}) \tag{6.1}$$

core #2:

The node voltage v_{g1} is calculated by core #2 using Equation 6.2. gb_{11} and gb_{12} have constant values and the value of v_{g1} only depends on changes of v_{g10}, which will be provided by its corresponding core. Again, there is no need for device modelling.

$$v_{g1} = (gb_{11}v_{g10})/(gb_{11} + gb_{12}) \tag{6.2}$$

core #3:

The drain voltage v_{d1} of transistor M_1 is calculated by Equation 6.3. All the G and i_{ds} values stand for the linear models of nonlinear elements. According to the proposed method, in order to calculate an unknown variable on one of the cores, it is expected to have the node voltage and branch currents provided by the other nodes. Therefore, to be able to perform device modelling at the same time as equation solution, these terms need to be expressed in terms of constants and known voltages and/or currents which is quite straight forward. For example, according to element stamps and companion models, G_{21} can be expressed by Equation 6.4, which is in terms of some constants and node voltages from other cores. In Equation 6.4, w, l, and k' are constants, v_{d1} is available from the same core, and v_{s12} should be obtained from the core #4. The other two transistor models which should be calculated by core #3 are shown in Equation 6.5 and Equation 6.6.

$$v_{d1} = \frac{(G_{11} + G_{21})v_{s12} + g_{d1}v_{dd} + (-G_{21})v_{g1} - i_{ds1}}{g_{d1} + G_{11}} \tag{6.3}$$

$$G_{21} = \frac{w}{l}k'(v_{d1} - v_{s12}) \tag{6.4}$$

$$G_{11} = \frac{w}{l}k'((v_{g1} - v_{s12}) - v_{th} - (v_{d1} - v_{s12})) \tag{6.5}$$

$$i_{ds1} = i_d - G_{11}(v_{d1} - v_{s12}) - G_{21}(v_{g1} - v_{s12}) \tag{6.6}$$

In the same manner, nodes 4 to 11 will be evaluating Equation 6.7 to Equation 6.14.

core #4:

$$v_{s12} = \frac{i_{ds} + i_{ds2} - i_s + G_{21}v_{g1} + G_{11}v_{d1} + G_{12}v_{d2} + G_{22}v_{g2}}{G_{11} + G_{21} + G_{12} + G_{22}} \tag{6.7}$$

core #5:

$$v_{d2} = \frac{(G_{12} + G_{22})v_{s1} + g_{d2}v_{dd} + (-G_{22})v_{g2} - i_{ds2}}{g_{d2} + G_{12}} \tag{6.8}$$

core #6:

$$v_{g2} = (gb_{21}v_{g20})/(gb_{21} + gb_{22}) \tag{6.9}$$

core #7:

$$i_{vs2} = gb_{21}(v_{g2} - v_{g20}) \tag{6.10}$$

core #8:

$$i_{vdd} = g_{d1}v_d + g_{d2}v_{d2} - (g_{d1} + g_{d2})v_{dd} \tag{6.11}$$

core #9:

$$v_{g10} = 2.5 \tag{6.12}$$

core #10:

$$v_{g20} = 2.5 \tag{6.13}$$

core #11:

$$v_{dd} = 5 \tag{6.14}$$

All the cores will keep calculating their allocated variables and collect the required updates from other cores as they are available until the iteration stop criteria are achieved.

6.2.2 Highly Parallel and Simultaneous Evaluation

In this section, the process of simultaneous evaluation of the device modelling and matrix solution phases on a highly parallel system is reviewed. The device evaluation can be performed on the same set of cores which are doing the matrix solution process. The number of required processors is still equal to the number of circuit equations. Each node solves its corresponding equation and also needs to calculate the required linear models for nonlinear devices. There might be repeating calculations of the same device model on a number of different cores but there is no need for extra communications to obtain the required device models. In other words, there is not a separate device evaluation phase. The equations are solved in parallel and independently on the parallel cores and each core calculates its required device models as needed.

For example, in the test circuit of Figure 4.10, eleven cores are required in total to perform the computations of the eleven circuit equations. As explained, apart from solving their corresponding rows, some cores will also calculate the required device models. As

an example, the required calculations of cores number 3, 4, and 8 are represented in detail in Figure 6.2. The arrows stand for communicating with other cores and the entries inside the boxes show the required device models for the cores. Core number 3 solves the third row of the circuit matrix to calculate v_{d1}. To do so, it needs to continuously obtain the new values for v_{s12}, v_{dd}, and v_{g1} from cores 4, 11, and 2 respectively. Besides, core number 3 needs to calculate G_{11}, G_{21}, and i_{ds1} as linear device models of transistor M_1. Core 4, requires relatively more work communicating with four other cores to obtain their x values and also calculating the device models for transistors M_1 and M_2. Core 8, however, only needs to obtain three x values from other cores and does not need device evaluation.

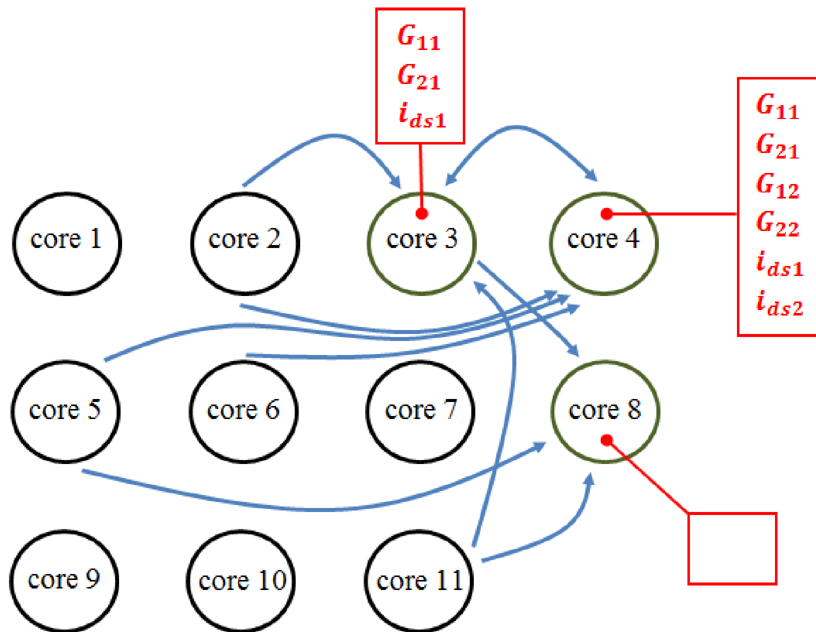


FIGURE 6.2: Communications and calculations required for cores number 3, 4, and 8 in Figure 4.10

The simulations of the proposed approach are performed on a cluster of parallel cores (the Iridis cluster, Appendix B, Section B.2) by allocating one processor to each circuit equation as discussed and presented in Figure 6.2. The device evaluation phase is performed using the Secant method and the matrix solution process is done using the proposed parallel Random Jacobi iterations. Like the simulations in Chapter 5 for device evaluation, 10 and 50 Jacobi iterations are used for the two sets of simulations for the under test differential pair circuit.

Simulation results show that highly parallel evaluation of the circuit equations using the proposed methods obtains the solution with the same number of iterations performed on single-core but with considerably less execution time, as presented in Table 6.1.

Methods used	Iterations	Execution time (ms)
Secant plus 10 Jacobi iterations (parallel)	23	9
Secant plus 50 Jacobi iterations (parallel)	10	26

TABLE 6.1: Number of Secant iterations and execution time required to obtain the solution by a highly parallel evaluation.

In order to compare the results with single-core simulations in Table 4.13 of Chapter 4, all the results are shown in Figure 6.3. As it is seen, the proposed method on a parallel system performs considerably better than single-core simulations. For the case in which the Secant method along with ten Random Jacobi iterations are used, the execution time is only 9 ms which is much faster than the the simulations of the same methods on a single-core with the execution time of above 200 ms. Furthermore, compared to the fastest method on a single-core, which is the Secant method along with the direct matrix solution with the execution time of 44 ms, our result shows a speed-up of almost 5x for this specific test circuit.

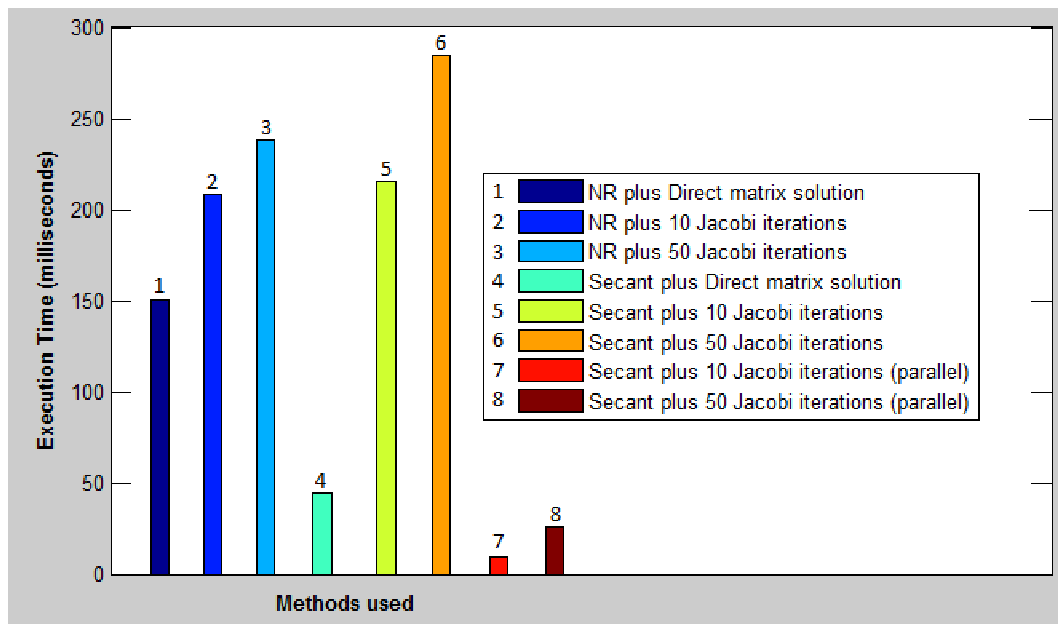


FIGURE 6.3: Execution time comparison for the differential pair MOS test circuit in Figure 4.10

This approach is also applied to the other test circuit (the simple single MOS circuit in Figure 4.8). Compared to the results represented in Chapter 4 (Table 4.11), which includes the results for evaluation of the proposed iterative approach on a single-core, a considerable decrease in the simulation time is also achieved for this test circuit. Figure 6.3 shows the simulation results. Among the first four execution times, which are the results from Chapter 4, simulations which are done with the Secant methods instead of NR, have better results in terms of the simulation times. Also, the secant method with the direct matrix solution (3) has the best execution time (28.2 ms). It can bee seen that evaluating our proposed method on a real parallel system of processors (5)

has resulted in 5 ms for the whole simulation time which is almost six times better than the best result in Chapter 5 on a single-core. Compared to the execution time of our proposed method on a single-core system (64.5 ms), its parallel evaluation shows more than a 12x speed-up.

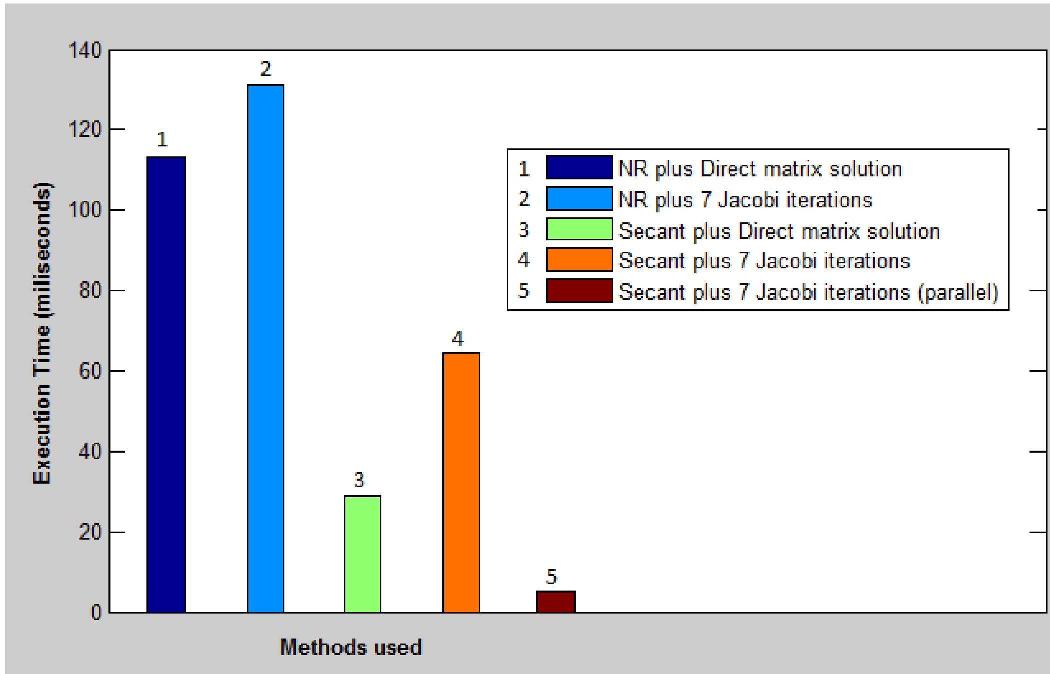


FIGURE 6.4: Execution time comparison for the single MOS test circuit in Figure 4.8

6.3 Implementation Considerations

The sizes of the test circuits used in this chapter for simultaneous evaluation of the two simulation phases are quite small compared to the real world examples. Although in Chapter 5 Section 5.2.2, relatively bigger matrices for the simulations of the matrix solution phase were used, we had a number of limitations for performing our simulations on bigger test matrices. Most of the circuit simulation benchmarks only include the conductance matrix (A) and do not have an RHS vector. This is due to the fact that conventional circuit simulation algorithms use direct matrix solutions, which do not need the manipulation of the RHS vector, while iterative solutions need the RHS vector to be constantly used as a part of the solution process. Therefore, for the matrix solution simulations, we had to limit our selection of benchmark circuits to the ones with an RHS vector. For the simultaneous evaluation approach in this chapter, we replaced the Newton-Raphson iterations with the Secant method and the direct matrix solution approach with an iterative approach. Consequently, the device models had to be manually calculated and therefore, simple non-linear device models and small circuits that could be handled by hand calculations are used. These small examples are used to demonstrate the applicability of the proposed methods on parallel systems. For bigger

circuits and examples of the possible scalability, automatic formulation of our method and its implementation in the SPICE algorithm is required which is beyond the scope of the current work. However, with relatively bigger example sizes only for the matrix solution part, in Chapter 5 Section 5.2.2 we showed that our fine-grained solution scales better with the size of the problem compared to the single-core direct approaches.

The motivation behind our work on highly parallel fine-grained systems is the availability of massively parallel platforms during the last few years. When performing simulations on massively parallel systems, there are a number of implementation and applicability related points to consider. In Chapter 3 Section 3.2.2, it was shown that our Jacobi-type matrix solution algorithm converges to the solution (with a few more iterations) even if some processors do not function properly in some iterations. It was also shown that if a processor fails completely, the system may or may not converge to the solution depending on the convergence criteria. Therefore, a fault tolerant system is required for implementing parallel algorithms on many-core systems. SpiNNaker, as our target architecture and the driving force behind this research, has a processor disposition which allocates some processors as reserve for fault situations. Each SpiNNAker core has 18 processors. One processor is assigned an operating system support role, 16 processors are given application support roles, and the last processor is used as the fault-tolerant spare processor [14, 49]. The implementation costs of performing massively parallel algorithms to accelerate SPICE simulation is also an important point to consider. The costs involved in such a project are accessibility of a massively parallel architecture, the programming efforts required for the proposed algorithms, and implementation of the algorithms in the SPICE simulation process. The SpiNNaker architecture is the target for the implementation of the current work for a massively parallel approach for speeding up the circuit simulation process. The proposed algorithms for the device evaluation and matrix solution phases and also the optimised communication which uses a pre-analysis process are already formulated in this work using C/C++ and simulated on the Iridis supercomputer. However, a great amount of engineering work is required to implement these methods on the SpiNNaker architecture and then include them in the SPICE simulation process which we believe can be studied further in future projects.

6.3.1 Discussion of Implementation on SpiNNaker

In this section the many-core architecture of SpiNNaker is reviewed and the specifications which we believe make SpiNNaker a suitable candidate as a target system for this research are highlighted.

The SpiNNaker project is primarily designed to provide a massively parallel million-core platform which can be used to model the human brain. The main novelty in the SpiNNaker architecture is its specifically designed communication infrastructure which allows a very large number of communications for sending and receiving very small

packets. The SpiNNaker machine consists of a large number of nodes and each node contains a SpiNNaker chip multiprocessor. The architecture of a SpiNNaker node is shown in Figure 6.5. Each node incorporates 18 ARM processors, 96 kB of local memory, and 128 MB of shared memory, a packet router, and peripherals. The nodes are arranged in a two dimensional mesh topology in which every node is connected to 6 other nodes as shown in Figure 6.6. Figure 6.7 shows a possible SpiNNaker machine topology in which the mesh is folded to form its specific architecture [14, 15, 16, 91].

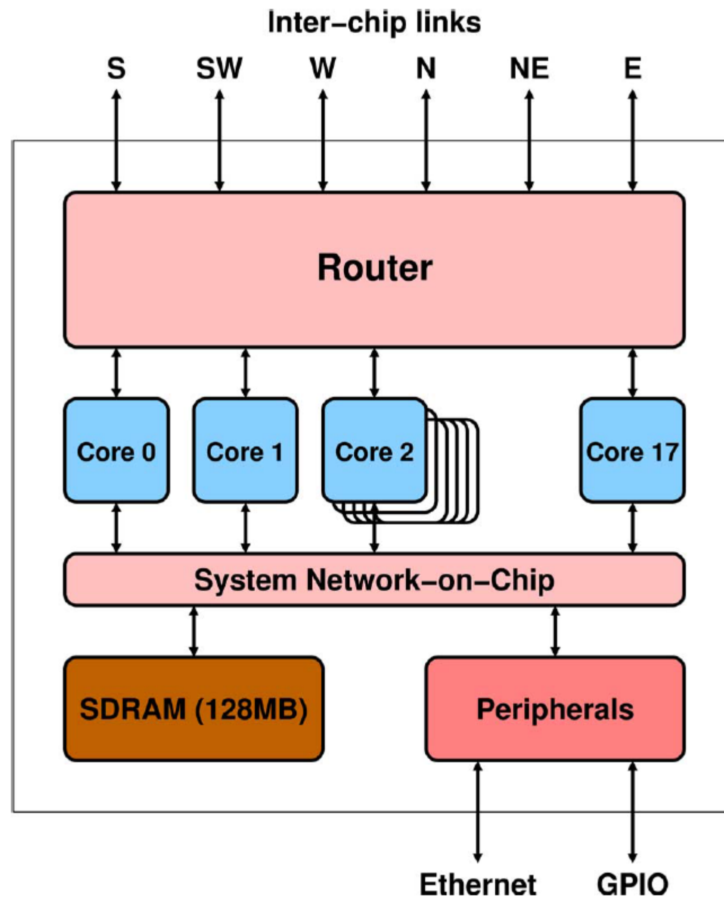


FIGURE 6.5: Principal architectural components of a SpiNNaker node [14].

The optimised communication of the SpiNNaker machine, which uses hardware and software controlled routines, allows very fast communication with low lecel error control possibilities. The role of the packet router is to inspect each packet to check its source and then route it to any local processor or any of other 6 neighbour cores using bidirectional links. One of the 18 processors in the SpiNNaker chip is always reserved to be used for fault tolerant purposes. Furthermore, if any of the links fails, there are emergency routing possibilities in hardware and software to control the error. The hardware can create a new path to the destination using alternative routes. This is possible because every link in the SpiNNAker two dimensional topology can be replaced by an alternative path consisting of two other links. Then the software will monitor it. If the failure repeats, new routes will be established and some of the communications load will be

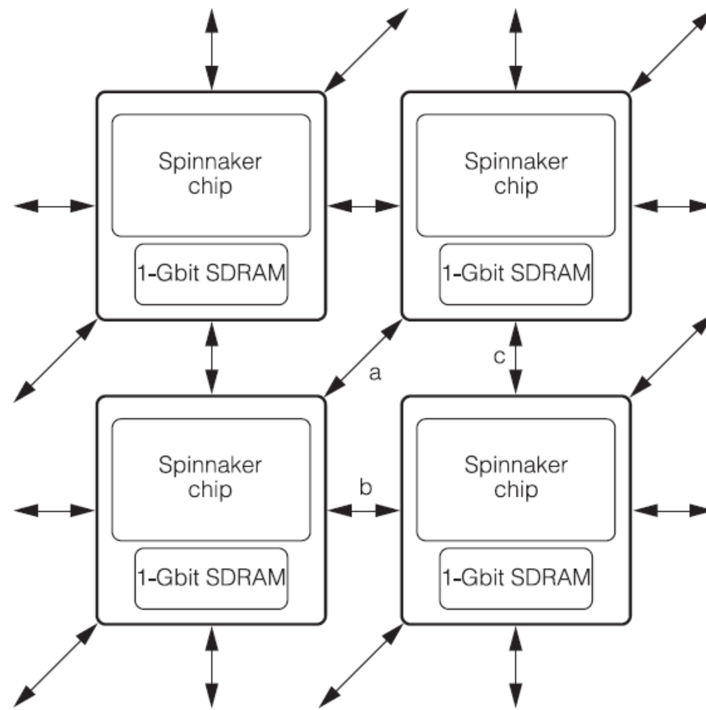


FIGURE 6.6: Spinnaker multiprocessor architecture [15].

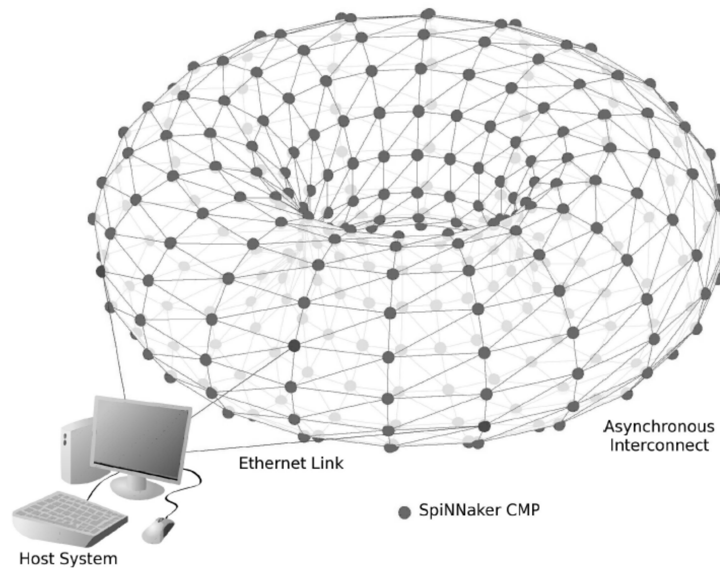


FIGURE 6.7: The SpiNNaker machine. [16].

handled by the alternative routes. If the failure becomes permanent, then all the traffic will be handled by the established routes [16, 91].

The following properties of the SpiNNaker architecture highlight the specifications needed for evaluation of the proposed highly parallel circuit simulation algorithm. In the proposed algorithm, a many-core architecture is required with efficient communications

between the processors. Each processor solves one of the equations and exchanges data (which is a very small packet) with a few other processors based on the sparsity pattern of the system matrix. Moreover, when using a large number of parallel processors, fault-tolerance and error handling are very important to keep the process stable. The SpiNNaker machine also offers a fault tolerant system suitable for our highly parallel algorithms.

6.4 Summary and Discussion

The proposed methods for matrix solution and device evaluation phases were simulated simultaneously on two test circuits using a highly parallel network of processors. Based on the proposed fine-grained parallel approach, the number of required parallel processors is equal to the number of circuit equations. Each processor is responsible for solving its corresponding row and if needed it, calculates the required device models. For both test circuits we observed more than 10x speed-up for parallel evaluation of our proposed method compared to its single-core simulation. Moreover, compared to the fastest method of Chapter 5, which was the Secant method along with the direct matrix solution, the simulation time is improved by a factor of more than 5x. Finally, comparing our results with the conventional NR iterations along with the direct matrix solution on a single-core suggests that the simulation times for the two test circuits have improved by factors of 16x and 22x.

There are a number of important points about parallel and simultaneous evaluation of the two circuit simulation phases which should be highlighted:

- When using the parallel Jacobi iterative method for matrix solution, although by increasing the number of matrix solution iterations, the number of circuit solution iterations decreases, it does not necessarily mean that the execution time will also be improved by using more Jacobi iterations. Matrix solution iterations are performed in each iteration of circuit simulation iterations which can be NR or Secant based on the simulation method. For example in Table 6.1, it can be seen that the Secant method with 10 Jacobi iterations has obtained the circuit solution within 23 iterations while with 50 Jacobi iterations it has obtained the solution in 10 iterations. However, the simulation time of the first case is 9 ms while for the second case it is 26 ms. This suggests that the values obtained by 10 Jacobi iterations are accurate enough to proceed to the next circuit simulation iteration and there is no need for a high number of Jacobi iterations. This can become more important as the size of the problem increases. This is due to the fact that the order of computations for the Jacobi method is N^2. If the required iterations to converge to the solution is equal to N, then the overall computations becomes of

order of N^3. When the number of Jacobi iterations is negligible compared to the problem size, the whole matrix solution problem becomes of the order of N^2.

- Avoiding the solution of a global matrix system in the form of $Ax = b$ and instead evaluating each equation independently on a separate processor using our proposed methods makes it possible for each processor to calculate its required linear models when required. Therefore, there is no need for a separate device evaluation phase and hence there are no undesired stops in the simulation process. Recalling the fact that circuit simulation matrices are very sparse and each node is only connected to a few elements, each processor only performs a limited amount of extra calculations for device evaluation, which has now even become more simplified by using the Secant methods instead of NR.

We used the specific properties of our proposed approach based on random Jaconi method for matrix solution purposes. An iterative matrix solution approach is proposed which can be combined with the device evaluation phase to decrease the amount of undesired stops between the two simulation phases, which happen in conventional circuit simulation algorithms. To the best of our knowledge, there is no similar work with this level of fine-grain parallelisation and more importantly with simultaneous evaluation of the device evaluation and the matrix solution phases. The current work suggests a new approach to possibly replace the conventional approaches and provide the ground work for future works on targeting massively parallel platforms, such as SpiNNaker, to speed up the circuit simulation process.

Chapter 7

Conclusions

This thesis provides new approaches for the main phases of the circuit simulation process and shows that the conventional circuit simulation algorithm can be performed using new alternatives which benefit from ease of implementation on highly parallel systems with simpler algorithms in order to speed up the simulation process. Conventional direct matrix solution is replaced with a parallel and iterative approach and also the computationally intensive linearisation process using NR iterations is replaced with simpler approaches which can be efficiently performed on fine-grained parallel systems. In the first part of this chapter, a summary of the thesis is given. The second part covers the main ideas which shaped the building blocks of this research and represents the main results, achievements, and contributions. Finally, in the last part, a number of suggestions are made as the future work and possible developments of the current research.

7.1 Summary

The current work reviews the existing work and literature in circuit simulation area. Different matrix solution methods are studied and parallel systems and approaches are discussed. For the purpose of this specific work, iterative matrix solution methods are preferred over the direct ones.

Two main iterative methods (Gauss-Seidel and Jacobi) and our proposed iterative approach called Random Jacobi, which is based on non-deterministic evaluation of normal Jacobi iterations, are studied. A number of preliminary simulations are performed to compare the functionality of Random Jacobi, normal Jacobi, and Gauss-Seidel iterations. Simulation results show that the Random Jacobi method obtains exactly the same result as the normal Jacobi approach. The Gauss-Seidel method, as expected, converges fasted than either of them. However, according to the nature of this work based on parallel evaluation of matrix equations, Jacobi-type algorithms are preferred according to their inherent parallelism.

This work performs independent and distributed evaluation of KCL at each circuit node rather than solving the conventional circuit description matrix system. This would be a case in which circuit equations should be evaluated independently and in a non-deterministic order. Therefore, the simulations are performed on a number of benchmark matrices on a highly parallel network of processors by allocating one processor per circuit equation. The results are compared to the ones from the single-core and also the virtual many-core evaluations. Simulation results show that parallel evaluation of the Random Jacobi method for the matrix solution phase results in the same number of iterations required for the convergence compared to the single-core solution. Furthermore, as the size of matrices increases, the proposed parallel and iterative solution performs better than both the single-core iterative solution and also the LU-factorisation method as the conventional direct solution approach.

For the device evaluation phase, it is proposed to use simpler linearisation techniques compared to the conventional NR iterations to avoid the calculation of partial derivatives at each iteration. The proposed Secant method has a much simpler algorithm compared to the NR method. Although the Secant method obtains the linear model in more iterations, simulation results show that its overall execution time is less than NR iterations.

The parallel Random Jacobi iterations for the matrix solution phase and the Secant method for the device evaluation phase are used simultaneously on two test circuits to evaluate the functionality of mixed simulation of the two proposed methods. This is conventionally done using LU-factorisation and NR iterations. Simulation results represent a significant improvement in the execution time of the simulations by using the proposed parallel iterative method. The simulation time has improved by a factor of more than 15x when comparing the LU-factorisation along with the NR method on a single-core with the Random Jacobi iterations along with the Secant method on a parallel many-core system.

7.2 Novel Contributions

Following the objectives mentioned in the introduction chapter, the main achievements and novel contributions of this thesis can be highlighted as follows:

- **To fulfil objectives 1 and 3: Non-deterministic evaluation of the Jacobi iterative method on highly parallel many-core systems to replace the conventional direct matrix solution methods such as LU-factorisation.** Direct matrix solution methods are widely used in conventional circuit simulation algorithms. However, in this work it was shown that using direct matrix solutions causes undesired stops between the two main simulation phases. The possibility

of replacing direct matrix solution methods with a fine-grained parallel iterative solution was assessed to distribute the system solution across a large number of parallel processors. A non-deterministic evaluation of the Jacobi iterative method was used for this purpose. This was initially tested in Chapter 3, as a part of the first objective of this research, to confirm the functionality of the proposed non-deterministic evaluation of the Jacobi iterative method. Then, according to the simulation results on a number of test matrices we observed that the proposed iterative solution functions properly when evaluated on a highly parallel system and obtains the solution within the same number of iterations as the single-core case. This was shown in Table 5.4 and Table 5.8 in Chapter 5. However, the execution time has improved significantly compared to the single-core simulations. As the size of test matrices increases, the parallel iterative solution performs better and for the largest test matrix obtains the solution 2 times and 3 times faster than the direct method and single-core iterative method, respectively. The overall results of the simulations were presented in Section 5.2.2.3 of Chapter 5 in Table 5.8 and Figure 5.8 to fulfil the third objective stated in the introduction chapter. Furthermore, compared to the conventional direct solution method, LU-factorisation, it was shown that the proposed Jacobi type iterative solutions performs faster than the direct solution as the size of problem increases. To the best of our knowledge, there is no work in the circuit simulation area with this level of fine-grained parallelism for the matrix solution phase. We believe that, with the availability of massively parallel platforms such as SpiNNaker, novel approaches based on very fine-grained parallel methods will become of more interest in speeding up the circuit simulation process.

- **To address objective 2: Use of simple device evaluation techniques in circuit simulation algorithms in conjunction with an iterative matrix solution to replace the computationally intensive conventional Newton-Raphson method.**
  SPICE-like circuit simulation algorithms use NR iterations to model the behaviour of the nonlinear elements. Although it is a widely used method with a high convergence rate, it suffers from the high amount of calculations required for numerically solving the partial derivatives. In this thesis in Chapter 4, a much simpler approach was used based on the Secant method for linearisation purposes and was integrated with the proposed iterative matrix solution. This was highlighted in the introduction chapter as the second objective of this work. The simulation results, which were given in Chapter 4 Table 4.11 and Table 4.13 showed that the Secant method converges to the solution with more iterations compared to the NR method (almost 3 times more iterations). However, due to the Secant method's simple algorithm without the need for calculating partial derivatives at each iteration, its total execution time is less than or equal to the NR method's execution time with

a factor of (4x to 1x). It should be reminded that for these simulations, the iterative matrix solution is performed sequentially, while it works best on parallel systems. Therefore, as claimed in Chapter 4, and achieved in Chapter 6, which will be shown as a part of objective 5, even better results obtained by parallel and simultaneous evaluation of the two phases. In the proposed highly parallel circuit simulation approach, the aim is to break problems into a large number of simple tasks to be evaluated on a highly parallel system. It was shown that, the Secant method can be a suitable candidate for the proposed fine-grained method when used in conjunction with our highly parallel iterative matrix solution approach.

- **To accomplish objective 4: Introduction of an optimised communication pattern for parallel MPI simulations suitable for sparse matrices to decrease the required communication between processors.**
  The proposed parallel many-core simulation which was used in this thesis requires extensive data exchange between processors. Since the circuit simulation matrices are very sparse, an optimised pattern for the proposed iterative matrix solution method was introduced and used in Chapter 5 Section 5.3 which works best for sparse matrices. The simulation results, which were shown in Table 5.11 and Figure 5.13 of Chapter 5, confirmed that by using the proposed pattern, we can improve the solution time for sparse matrices. These simulation results cover the fourth objective of this thesis. For example, for the largest test matrix the execution time, when the optimised communication pattern was used, was improved 23% and 61% compared to the non-optimised parallel matrix solution and direct matrix solution, respectively. However, for dense matrices, the amount of pre-calculations required for extracting the correct pattern dominates the improvement in the solution time. Therefore, the proposed optimisation pattern is only useful for sparse matrix systems. Simulations results in Table 5.12 of Chapter 5 show that for the same matrix size and solution iterations, when the sparsity rate of matrices changed from 35% to 95%, a 5x improvement is achieved in the matrix solution time using the proposed communication pattern. The communication optimisation can be integrated as an intermediate stage between matrix construction and matrix solution phases to generate a communication pattern between processors to minimise the required amount of communications.

- **To fulfil objective 5: Simultaneous evaluation of the two main simulation phases to overcome the existing issue of parallelising the whole circuit simulation process caused by the barrier between the two phases.**
  According to the circuit simulation literature, all circuit simulation algorithms evaluate the two main circuit simulation phases (device evaluation and matrix solution) separately. This is because of the barrier between the device evaluation

and matrix solution phases. The use of an iterative matrix solution method in this work made it possible to perform a simultaneous evaluation of the two proposed methods on a highly parallel system, which was covered in the Chapter 6 of this thesis. The Random Jacobi method was used instead of direct methods for the matrix solution phase and the Secant method instead of NR iterations for the device evaluation phase. Combining the two methods and performing the simulation on a very fine-grained system of parallel processors by allocating one processor per each circuit equation, as the fifth objective of this work, led to even more improvement in simulation times compared to the sperate evaluations of each of the methods. Simulation results were presented in Figure 6.3 and Figure 6.4 of Chapter 6. For example, for one of the test circuits which was used for these simulations, conventional NR iterations with the direct matrix solution on a single-core resulted in 150 ms for the execution time. The simulation time was decreased to 9 ms when simultaneous evaluation was performed using the proposed methods. This shows more than a 15x speed-up for parallel simulation of the under test circuit. Also, for the other test matrix, an speed-up of 22x is achieved. It was shown that this is a suitable parallel approach to eliminate the barrier between the device evaluation and matrix solution phases, which is one of the unsolved issues in this area. It is caused by the nature of the conventional direct matrix solution approaches that needs completion of each phase in order to start the next phase. To the best of our knowledge, this work is the first attempt to propose a novel approach for massively parallel evaluation of the circuit simulation algorithm using iterative matrix solution methods and can be the ground work for future developments of highly parallel circuit simulators. By employing more efficient massively parallel platforms in terms of communication between processors even better results can be achieved.

7.3 Future Work

The current work introduces new approaches for evaluating the main phases of the circuit simulation algorithm. This thesis mainly discusses the preliminary evaluations to confirm the functionality of the proposed methods and measures their performance. Based on the areas studied in this work, further research can be done from both the hardware and software points of view to improve the algorithms and develop the proposed methods on other parallel platforms.

In this thesis, MPI was used for highly parallel evaluation of the circuit equations on a cluster of processors by allocating one processor to each circuit equation. We believe that for the preliminary research purposes, using MPI is an appropriate approach to manage the communications between the parallel processors. However, to decrease the communications overhead and perform the communication between the processors in a

more efficient way, it is possible to use other platforms for parallel processing such as FPGAs or other massively parallel computing architectures such as SpiNNaker.

This research is in fact a proof of concept for performing the SPICE-type circuit simulation algorithms, which have been almost unchanged for four decades, using new approaches to accelerate the process of circuit simulation. It was shown that it is possible to replace the conventional direct matrix solution with a very fine-grained and highly parallel Jacobi-type iterative solution and then combine it with the simultaneous evaluation of the device modelling phase which led to some promising results for speeding up the circuit simulation process. However, simple test circuits and benchmark matrices are used in simulations. A substantial amount of work is required to evaluate the proposed methods for large test circuits using, for example, a whole BSIM model for MOS transistors. Therefore, a massively parallel system with specific properties is required to implement the proposed parallel circuit simulation algorithms with more accurate models for very large test circuits. The availability of a more efficient way of communication also has the benefit of performing the equation solutions asynchronously and calculating a local convergence instead of a global convergence until the required accuracy is achieved. SpiNNaker, a novel massively parallel computing platform inspired by the working of the human brain, can be a suitable candidate as a massively parallel platform for further developments of the current work.

Appendix A

Simulations and codes

This appendix includes the diagrams, codes, and extra details about the simulations.

A.1 Generating Test Matrices by Matlab

Matlab code for generating test matrices with desired specifications:

```
%%%%%%%%%%%%%%% Random Sparse Matrix Generator %%%%%%%%%%%%%%%
% 1. a square random matrix with desired size is generated   %
%    with the element values between 0 and 1                 %
% 2. by a shift of half a unit and then timing the matrix by %
%    10, the values range will be between -5:5               %
% 3. another square matrix with the same size is generated   %
%    with a desired sparsity rate of 'spc'                 %
% 4. by multiplying two matrices, a sparce matrix is obtained%
% 5. the matrix is made diagonally dominant                  %
% 6. a random RHS vector is generated                        %
% 7. the solution is calculated using direct matlab methods  %
% 8. the matrices are saved.                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

matSize = 1000;     % matrix size
spc = 0.90;         % sparsity coefficient*100 = (percent of zero elements)
rhsspc = 0.90;      % rhs vector sparsity
acoef = 05;         % range coefficient of matrix A
rhscoef = 10;       % range coefficient of RHS vector
sumCoef = 0.15;         % coefficient of diagonal element
infostr = 'size=1000; sparsity=90%; sumCoef = 0.15, A:(-5:5); Z:(-10:10)';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inimat = rand(matSize);         % initial matrix range:(0:1)
inimat = (inimat-0.5).*acoef;   % random matrix range:(-0.5:0.5)*acoeff

% generating a random sparse matrix
zerone=rand(matSize);
for i=1:matSize
    for j=1:matSize
        if i==j
```

```
33              zerone(i,i)=1;
34          elseif zerone(i,j)>spc
35              zerone(i,j)=1;
36          else
37              zerone(i,j)=0;
38          end
39      end
40  end
41  mndd = inimat.*zerone           % matrix not diagonally dominant
42  mnddt = mndd';                  % transpose of mndd to be used for "max"
43  [maxval,maxind]=max(abs(mnddt)) % returns max value and max indice
44                                  % of each column
45
46  % finding suitable additional value for diagonal elements
47  nzre = zeros(1,matSize);        % number of none-zero row elements
48  for m=1:matSize
49      for n=1:matSize
50          if m~=n
51              if mndd(m,n) ~= 0
52                  nzre(1,m)=nzre(1,m)+1;
53              end
54          end
55      end
56  end
57  % sum >> the added value to make the diagonal elemnt dominant
58  sum = nzre.*sumCoef;
59
60  % making the matrix diagonally dominant
61  for i=1:matSize
62      if maxind(i)~=i
63          if mndd(i,maxind(i))>= 0
64              tempval=mndd(i,maxind(i))+sum(1,i);
65          else
66              tempval=mndd(i,maxind(i))-sum(1,i);
67          end
68          mndd(i,maxind(i))=mndd(i,i);
69          mndd(i,i)=tempval;
70      else
71          if mndd(i,i)>= 0
72              mndd(i,i)=mndd(i,i)+sum(1,i);
73          else
74              mndd(i,i)=mndd(i,i)-sum(1,i);
75          end
76      end
77  end
78  A_matrix = mndd % diagonally dominant matrix
79
80  % Generating RHS vector (excitations)
81  b_matrix = rand(matSize,1);
82  b_matrix = (b_matrix-0.5).*rhscoef;
83  rhssp = rand(matSize,1); % rand vectro for sparsity of RHS
84  for i=1:matSize
85      if abs(rhssp(i,1))<rhsspc
86          b_matrix(i,1)=0;
87      end
88  end
89  solution = (A_matrix^-1)*b_matrix   % the exact X vector
90  largest = max(solution)
91
92  % check if the matrix is diagonally dominant
93  ddcount = 0;
94  elcount = 0;
```

```
95  for p=1:matSize
96      ddsum = 0;
97      for q=1:matSize
98          if (p ~= q)
99              ddsum = ddsum + abs(A_matrix(p,q));
100             if (abs(A_matrix(p,p)) < abs(A_matrix(p,q)))
101                 elcount = elcount+1;
102             end
103         end
104     end
105     if (abs(A_matrix(p,p)) < ddsum)
106         ddcount = ddcount+1;
107     end
108 end
109 elcount
110 ddcount
111
112 %save('MTX0020_s80_r05_sc0.50_n01.mat', 'A_matrix', 'b_matrix',...
113 %         'solution', 'matSize', 'spc', 'infostr')
114 % s: size, r: range, sc: sumCoef, n: number
115
116
```

and checking the convergence of the test matrices:

```
1   %%%%%%%%%%%%%%% Convergence Check - Jacobi %%%%%%%%%%%%%%%
2
3   load MTX0020_s80_r05_sc0.30_n01.mat
4   a = A_matrix;
5   z = b_matrix;
6   sol = solution';
7
8   %%%%%%%%%%%%%%%%%%%%% applying methods %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
9   mat_size = size(a,1);                       % matrix size
10  rep = 50;                                   % number of iterations
11  th = 1e-2;
12
13  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Jacobi
14  x=[];                                     % the solution
15  x_init=ones(1,mat_size);                  % initial value
16  i=0;
17  norm_x = 10;
18  while ((i < rep) && (norm_x > th))        % iterations
19      i = i + 1;
20      display(['-- iteration ' num2str(i) '--']);
21      for j = 1:mat_size                    % calculations for each row
22          sum=0;
23          for k = 1:mat_size
24              if j~=k
25              sum=sum+a(j,k)*x_init(k);
26              end
27          end
28          x(j)=(z(j)-sum)/a(j,j);           % new values
29      end
30
31      dif_x = x - x_init;
32      norm_x=sqrt(dif_x*dif_x');
33      norm_xp(i)=norm_x;
34      x_init=x;                             % updating outside loop
35  end
```

```
36
37 x;
38 grid on;
39 plot(norm_xp, 'or','MarkerFaceColor','r','MarkerSize',5);
40 display('*** Jacobi done ***');
```

A.2 Convergence Comparison of the Iterative Methods

Matlab code for convergence comparison of three iterative matrix solution methods: Jacobi, Gauss-Seidel, and Random Jacobi.

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Jacobi
2  x=[];                                   % the solution
3  x_init=ones(1,mat_size);                % initial value
4  dif_x0=x_init-sol;
5  norm_x0=sqrt(dif_x0*dif_x0');
6  norm_xp(1)=norm_x0;
7  for i = 1:rep                           % iterations
8      display(['-- iteration ' num2str(i) '--']);
9      for j = 1:mat_size                  % calculations for each row
10         sum=0;
11         for k = 1:mat_size
12             if j~=k
13             sum=sum+a(j,k)*x_init(k);
14             end
15         end
16         x(j)=(z(j)-sum)/a(j,j);         % new values
17     end
18     x_init=x;                           % updating outside loop
19     dif_x=x-sol;
20     norm_x=sqrt(dif_x*dif_x');
21     norm_xp(i+1)=norm_x;
22 end
23 x;
24 grid on;
25 plot(plt, norm_xp, 'or','MarkerFaceColor','r','MarkerSize',5);
26 display('*** Jacobi done ***');
27
28 %%%%%%%%%%%%%%%%%%%%%%%%% Gauss-Seidel
29 y=[];                                   % the solution
30 y_init=ones(1,mat_size);                % initial value
31 dif_y0=y_init-sol;
32 norm_y0=sqrt(dif_y0*dif_y0');
33 norm_yp(1)=norm_y0;
34 for m = 1:rep                           % iterations
35     display(['-- iteration ' num2str(m) '--']);
36     for n = 1:mat_size                  % calculations for each row
37         sum=0;
38         for p = 1:mat_size
39             if n~=p
40             sum=sum+a(n,p)*y_init(p);
41             end
42         end
43         y(n)=(z(n)-sum)/a(n,n);         % new values
44         y_init(n)=y(n);                 % updating inside loop
45     end
46     dif_y=y-sol;
```

```
47      norm_y=sqrt(dif_y*dif_y');
48      norm_yp(m+1)=norm_y;
49  end
50  y;
51  hold on
52  grid on
53  plot(plt, norm_yp, '--m');
54  display('*** Gauss-Seidel done ***');
55
56  %%%%%%%%%%%%%%%%%%%%%%%%%%%%% Random Jacobi
57  r=[];                                    % the solution
58  r_init=ones(1,mat_size);                 % initial value
59  dif_r0=r_init-sol;
60  norm_r0=sqrt(dif_r0*dif_r0');
61  norm_rp(1)=norm_r0;
62  ran_vector=randperm(mat_size);            % creating non-deterministic order
63                                            % of equation evaluation
64  for d = 1:rep                             % iterations
65      display(['-- iteration ' num2str(d) '--']);
66      for e = 1:mat_size                % calculations for each row
67          g=ran_vector(1,e);
68          sum=0;
69          for f = 1:mat_size
70              if g~=f
71              sum=sum+a(g,f)*r_init(f);
72              end
73          end
74          r(g)=(z(g)-sum)/a(g,g);           % new values
75      end
76      r_init=r;                             % updating outside loop
77      dif_r=r-sol;
78      norm_r=sqrt(dif_r*dif_r');
79      norm_rp(d+1)=norm_r;
80  end
81  r;
82  hold on;
83  plot(plt, norm_rp);
84  display('*** Random Jacobi done ***');
85
86  title(['Matrix size: ' num2str(matSize) ', Sparsity: ' num2str(spc*100) '%']);
87  legend('normal Jacobi','Gauss-seidel','Random Jacobi');
88  xlabel('Iterations');
89  ylabel('Euclidean Norm');
90  hold off
```

A.3 Newton-Raphson and Secant Methods Comparison

This section included the Matlab codes we used for performing our simulations on the device evaluation and matrix solution phases to compare the functionality and performance of Newton-Raphson and Secant methods when are used along with a direct matrix solution (LU) and an iterative matrix solution (Jacobi) (Figure 4.9).
Two of the Matlab codes are included in this section which contain the algorithms used for the four mentioned methods.
Newton-Raphson with a direct method:

```
1  % Newton-Raphson for device modelling
2  % and
3  % Direct method (LU) for matrix solution
4
5  A = zeros(7,7);                        % to keep the A matrix
6  b = zeros(7,1);                        % to keep the RHS matrix
7
8  % Linear elements (conductances and independent voltage source)
9  gd = 3.3333e-5; gl = 3.3333e-5; gs = 2e-4;
10 gb1 = 5e-4; gb2 = 1e-5;
11 vdd = 3;
12
13 % initial guess (for unknown node voltages and branch currents)
14 v1 = 3; v2 = 0.5; v3 = 1.8;
15 v4 = 0.2; v5 = 2;
16 ivs = 30e-6; ivdd = 50e-6;
17
18 % constants
19 vth = 0.7; kp = 200e-6; wl = 2;        % NMOS
20
21 % calculating the operating point
22 % rep = 7;                              % number of jacobi iterations
23 mat_size = size(A,1);
24 x=[];                                   % the solution in jacobi iterations
25 st = tic;
26 for titer=1:10
27     % device modeling and linearization (Newton_Raphson) -----------
28     % mos values
29     vgs = v3 - v4; vds = v2 - v4;
30     i_m = 0.5*kp*wl*(2*(vgs - vth)*vds - vds^2);
31 %     G1 = 2*kp*(vgs - vth - vds);
32     [G1,err] = derivest(@(vds) 0.5.*kp.*wl.*(2.*(vgs - vth).*vds - vds.^2),vds)
33 %     G2 = 2*kp*vds;
34     [G2,err] = derivest(@(vgs) 0.5.*kp.*wl.*(2.*(vgs - vth).*vds - vds.^2),vgs)
35     imds = i_m - G1*vds - G2*vgs;
36
37     % Forming the A and b matrices
38     % A matrix with exchanged rows 6,5 and 1,7.
39     A = zeros(7,7);
40     A(7,1) = gd;            A(7,2) = -gd;          A(7,7) = 1;
41     A(2,1) = -gd;       A(2,2) = gd+gl+G1;  A(2,3) = G2;        A(2,4) = -G1-G2;
42     A(3,3) = gb1+gb2;   A(3,5) = -gb1;
43     A(4,2) = -G1;       A(4,3) = -G2;       A(4,4) = G1+G2+gs;
44     A(6,3) = -gb1;      A(6,5) = gb1;       A(6,6) = 1;
45     A(5,5) = 1;
46     A(1,1) = 1;
47     % rhs matrix with exchanged row 6 and 7
48     b = zeros(7,1);
49     b(2,1)= -imds; b(4,1) = imds; b(5,1) = 2; b(1,1) = 3;
50
51 % Direct matrix solution (LU) -----------------------------
52
53 y = splv(A,b);
54 yvect(:,titer)=y;
55
56
57 % updating new results -----------------------------------------
58 v1 = y(1,1); v2 = y(2,1); v3 = y(3,1);
59 v4 = y(4,1); v5 = y(5,1); v6 = y(6,1);
60 iv = y(7,1);
61
62 tm(titer) = toc(st);
```

```
63 end
64 tm = toc(st)
65
66
```

```
1  function x = splv(A, b)
2
3  % splv  The solution to a square, invertible system.
4  % x = splv(A, b) uses the PA = LU factorization
5  % computed by splu to solve Ax = b.
6
7  [P, L, U] = splu(A);
8  [n, n] = size(A);
9
10 % Permute the right hand side.
11 b = P*b;
12
13 % Forward elimination to solve L*c = b.
14 c = zeros(n, 1);
15 for k = 1:n
16   s = 0;
17   for j = 1:k-1
18     s = s + L(k, j)*c(j);
19   end
20   c(k) = b(k) - s;
21 end
22
23 % Back substitution to solve U*x = c.
24 x = zeros(n, 1);
25 for k = n:-1:1
26   t = 0;
27   for j = k+1:n
28     t = t + U(k, j)*x(j);
29   end
30   x(k) = (c(k) - t) / U(k, k);
31 end
```

and Secant with an iterative method:

```
1  clear all
2  clc
3
4  A = zeros(7,7);                        % to keep the A matrix
5  b = zeros(7,1);                        % to keep the RHS matrix
6
7  % Linear elements (conductances and independent voltage source)
8  gd = 3.3333e-5; gl = 3.3333e-5; gs = 2e-4;
9  gb1 = 5e-4; gb2 = 1e-5;
10 vdd = 3;
11
12 % initial guess set 1
13 v1a = 3; v2a = 0.4; v3a = 1.6;
14 v4a = 0.2; v5a = 2;
15 ivsa = 25e-6; ivdda = 40e-6;
16
17 % initial guess set 2
18 v1b = 3; v2b = 0.5; v3b = 1.8;
19 v4b = 0.15; v5b = 2;
20 ivsb = 30e-6; ivddb = 50e-6;
```

```
21
22 % constants
23 vth = 0.7; kp = 200e-6; wl = 2;          % NMOS
24
25 % calculating the operating point
26 rep = 7;                               % number of jacobi iterations
27 mat_size = size(A,1);
28 x=[];                                  % the solution in jacobi iterations
29 st = tic;
30 for titer=1:15
31     % device modeling and linearization (Secant) -----------
32     % mos values
33     vgsa = v3a - v4a; vdsa = v2a - v4a;
34     vgsb = v3b - v4b; vdsb = v2b - v4b;
35     i_ma = 0.5*kp*wl*(2*(vgsa - vth)*vdsa - vdsa^2);
36     i_mb = 0.5*kp*wl*(2*(vgsb - vth)*vdsb - vdsb^2);
37     G1 = (i_mb - i_ma)/(vdsb-vdsa);
38     G2 = (i_mb - i_ma)/(vgsb-vgsa);
39     imds = i_mb -G1*vdsb - G2*vgsb;
40
41     % Forming the A and b matrices
42     % A matrix with exchanged rows 6,5 and 1,7.
43     A = zeros(7,7);
44     A(7,1) = gd;        A(7,2) = -gd;        A(7,7) = 1;
45     A(2,1) = -gd;       A(2,2) = gd+gl+G1;   A(2,3) = G2;        A(2,4) = -G1-G2;
46     A(3,3) = gb1+gb2;   A(3,5) = -gb1;
47     A(4,2) = -G1;       A(4,3) = -G2;       A(4,4) = G1+G2+gs;
48     A(6,3) = -gb1;      A(6,5) = gb1;     A(6,6) = 1;
49     A(5,5) = 1;
50     A(1,1) = 1;
51     % rhs matrix with exchanged row 6 and 7
52     b = zeros(7,1);
53     b(2,1)= -imds; b(4,1) = imds; b(5,1) = 2; b(1,1) = 3;
54
55     % iterative matrix solution (Jacobi) -----------------------------
56
57     % initial value
58     % use initial guess just in the first iteration
59     % in other iterations, the last obtained x vector is used as initial guess
60     % to use the initial guess for NR as starting point of Jacobi
61     if (titer == 1)
62         %x_init = ones(1,mat_size);
63         x_init = [3;0.5;1.8;0.2;2;30e-6;50e-6];
64     end
65
66     i=0;
67     while (i < rep)          % iterations
68     i = i + 1;
69     %display(['-- iteration ' num2str(i) '--']);
70     for j = 1:mat_size                    % calculations for each row
71         sum=0;
72         for k = 1:mat_size
73             if j~=k
74             sum = sum + A(j,k)*x_init(k);
75             end
76         end
77         x(j,1) =(b(j) - sum) /A(j,j);           % new values
78     end
79     x_init = x;                           % updating outside loop
80     jacvect(:,i,titer) = x;
81     end
82     xvect(:,titer)=x;
```

```
83
84 %%% Updating new results
85
86 v1a = v1b; v2a = v2b; v3a = v3b;
87 v4a = v4b; v5a = v5b; ivsa = ivsb;
88 ivdda = ivddb;
89
90 v1b = x(1,1); v2b = x(2,1); v3b = x(3,1);
91 v4b = x(4,1); v5b = x(5,1); ivsb = x(6,1);
92 ivddb = x(7,1);
93
94
95 end
96 tm = toc(st)
```

A.4 C++ Code for Iterative Solution Algorithms on Single Core

Part of the C++ code we used in our simulations to perform normal Jacobi, Random Jacobi, and Gauss-Seidel algorithms is included in this section.

```
if (SJ_Switch == 0) cout<< "\n*** Jacobi Iterations ***\n\n\n";
if (SJ_Switch == 1) cout<< "\n*** Gauss Seidel Iterations ***\n\n\n";
if (SJ_Switch == 2) cout<< "\n*** Random Iterations ***\n\n\n";

double start_t = clock();
double exeTime, exeTimeh;

srand(2);                              // seeding the random function
tempSol = initGuess;
i = -1;
eNormTemp = 10;

for (int rv=0; rv<matSize; ++rv) randVector.push_back(rv);
// using built-in random generator:
random_shuffle ( randVector.begin(), randVector.end() );
// iteration loop
while (i<numIteration && eNormTemp>th){
        i++;
        normCheck = tempSol;

        for (j=0; j<matSize; j++){                // row loop
                rn = randVector[j];
                if (SJ_Switch == 0 || SJ_Switch == 1) rn = j;
                sum = 0;
                for (k=0; k<matSize; k++){
                        if (rn!=k)
                                sum = sum + Matrix[rn][k]*tempSol[k];
                }
```

```
                // calculating new values for the elements of row "j"
                solution[rn] = (RHS[rn]-sum)/Matrix[rn][rn];
                // updating the solution vector for Seidel
                if (SJ_Switch == 1) tempSol = solution;
        }

        // updating the solution vector for Jacobi
        if (SJ_Switch == 0 || SJ_Switch == 2) tempSol = solution;
        // calculating ENorm
        difAddress = subMat(solution, normCheck);
        for (m=0; m<matSize; m++)        difMat[m] = difAddress[m];
        delete [] difAddress;
        eNormTemp = eucNorm(difMat);
        // Adding new ENorm to ENorm vector
        eNorm.push_back(eNormTemp);
}
reqIter = i+1;

exeTime = (clock() - start_t)/CLOCKS_PER_SEC;
exeTimeh = (clock() - start_h)/CLOCKS_PER_SEC;
```

Appendix B

Message Passing Interface

Message Passing Interface or MPI is a library of functions and macros for highly parallel distributed memory environments which can be used with a number of different programming languages such as C/C++, Fortran, and Java. [50]. In this work we used MPI for our many-core simulations on both virtual many-core and real many-core systems and used C++ programming for its implementation. In this appendix, we will review how MPI is incorporated in our C++ code for our highly parallel simulations on Iridis cluster and provide examples and screen shots of the processes and simulations.

B.1 Virtual Many-core Simulations using MPI

In order to incorporate MPI in a C++ program a number of steps are required to make use of MPI library. The MPI header file needs to be included in the beginning of the C++ code. Then MPI should be initialised and communication environment including the number of parallel processors and the rank of each processor should be defined. Then, MPI directives can be used for message passing between processors, and in the end, MPI should be finalised in order to clean up all MPI state. Figure B.1 shows parts of our parallel Jacobi iterative matrix solution code using MPI to solve one of our test matrices (size 102). The MPI parts of the code are highlighted. We ran these simulations under Ubuntu (version 11.10). Because we have used MPI library in our code we need to execute it with MPI commands as shown in Figure B.2. The *mpicc* command is used to compile our C code (*jacobi_mpi.c*, which contains MPI functions, and generate the executive file. Then it is run using the *mpirun* command with the desired number of virtual parallel processors (102 for this example).

```
*jacobi_mpi (~/simulations/mpi/algs2) - gedit
Open  Save  Undo
matdump.txt  jac_normal.c  *jacobi_mpi

/*      ----------------------------------
        Parallel Jacobi Iterations with MPI
        >>>  global convergence check   <<<
        ----------------------------------     */

#include <stdio.h>
#include <time.h>
#include "mpi.h"
#include <math.h>

#define matSize 102
#define maxIter 500
#define th 1e-5

typedef double Gen_Mat[matSize][matSize];

void read_mat(Gen_Mat A_mat, int my_rank, int p);          // function reading matrix A from a file
void read_rhs(  double* b_mat, int my_rank, int p);        // function reading RHS vector from a file
double enorm(  double x[],   double y[], int n);           // Euclidian norm of vectors x and y

main(int argc, char* argv[])
{
    int                 p;                                 // number of processes provided by user when executing the code
    int                 my_rank;                           // rank of each processor
    Gen_Mat             A_mat;                             // the square matrix A
    double              x_mat[matSize];                    // vector of unknowns
    double              b_mat[matSize];                    // rhs vector
    int                 n;                                 // number of processors and processes should be the same for our specific use
    int                 iter;                              // maximum number of iteration provided by use
    double              err;                               // error margin provided by user
    double              conv;                              // for convergence check
    double              start_t1, start_t2, start_w1, start_w2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        n = matSize;
        iter = maxIter;
C  Tab Width: 8  Ln 1, Col 1  INS
                              ⋮
                }
                x_mat[0] = (b_mat[0] - sum) / A_mat[0][my_rank];

                MPI_Allgather(x_mat, 1, MPI_DOUBLE, newx, 1, MPI_DOUBLE, MPI_COMM_WORLD);

                conv = enorm(newx, oldx, matSize);
                if(my_rank == 0){
                              ⋮
                }
        }
        MPI_Abort(MPI_COMM_WORLD, 23);

        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
        return 0;
}
/* ----------- functions ------------------------ */

/* reading matrix A and scattering it across processors*/
void read_mat(Gen_Mat A_mat, int my_rank, int p)
{
                              ⋮
```

FIGURE B.1: Implementation of MPI in C++ program.

B.2 The Iridis Computer Cluster

This appendix highlights the technical aspects of the Iridis computer cluster. The current version, Iridis 4, is the fourth generation cluster of the University of Southampton, one of the largest computational facilities in the UK. In November 2013, Iridis 4 was ranked 179th in the world in the TOP500 list. Iridis provides High Performance Computing facilities in a professionally managed service environment and is available to the University's entire research community.

Iridis is primarily designed as a batch service for users who need to run either distributed memory parallel jobs, or multiple resource-intensive sequential jobs. The main source of

```
rdmansour@ubuntu: ~/simulations/mpi/algs2
rdmansour@ubuntu:~/simulations/mpi/algs2$ clear

rdmansour@ubuntu:~/simulations/mpi/algs2$ mpicc jacobi_mpi.c -o jacmpiexe -lm
rdmansour@ubuntu:~/simulations/mpi/algs2$ mpirun -np 102 ./jacmpiexe

reading matrix A
reading completed

reading RHS vector
reading completed

-------------------- iteration 1 -------------------

Error:  129.825552580178595

Time : 3.4521739483
-------------------- iteration 2 -------------------
```

FIGURE B.2: Compiling and running MPI on virtual many-cores.

information on the Iridis 4 service is User Support wiki accessible to registered users. As well as documentation on how to access and use the service, it has information on training courses, background information on the facility, user forums and links to sources of further information. System Status page reports current status of the system, scheduled system maintenance and any ongoing problems/accidents.
IRIDIS 4 Components:

- 750 compute nodes with dual 2.6 GHz Intel Sandybridge processors;
- Each compute node has 16 CPUs per node with 64 GB of memory;
- 4 high-memory nodes with two 32 cores and 256 GB of RAM;
- 24 Intel Xeon Phi Accelerators;
- 3 login nodes with 16 cores and 125 GB of memory;
- In total 12320 processor-cores providing 250 TFlops peak;
- 1.04 PB of raw storage with Parallel File System;
- InfiniBand network for interprocess communication;
- Moab HPC Suite - advanced workload management system from Adaptive Computing;

MPI is the most popular environment for running parallel jobs, particularly for multi-node jobs. OpenMPI is the recommended implementation and versions for both the Intel and gcc compilers are available by selecting the appropriate environment module, with Intel as the default. Intel MPI is also available as an alternative to OpenMPI.

Support for shared-memory parallelism using OpenMP is available within both Intel and GNU Compilers, and as Iridis 4 has 16 cores per node may offer an attractive option for parallel execution of critical sections of a code.

B.3 Real Many-core Simulations using MPI on Iridis

Iridis uses Linux operating system. To execute our parallel MPI algorithm to Iridis, we need to submit a job script along with our C code. The commands to be run need to be placed in a script file. An example of the script file we used to submit a parallel job to run out algorithm on 102 processors is shown in Figure B.3 which contains information such as number of nodes/processors required, maximum wall time, path to the directory containing the code, and commands to compile and run the C code. When the simulation is completed, the results is saved to an output text file.

```
#!/bin/bash

#PBS -l nodes=102:ppn=1
#PBS -l walltime=00:05:00

#Change to directory from which job was submitted
cd /home/mrd1g10/allsims/4by4pattern

# load openmpi module so that we find the mpirun command
module load openmpi

# Run matmul executable in parallel over number of
node/processors requested
# by job (2*8 by default in #PBS line above), output messages go
to output_file

mpicc pivCode.c -o pivExe

mpirun pivExe
```

FIGURE B.3: An example of scripts used to submit parallel MPI jobs to Iridis.

Appendix C

Paper Presented in DAC 2013

This appendix includes the paper submitted to DAC (Design and Automation Conference) 2013, and accepted to be presented in poster session as a short paper.

Non-Deterministic Evaluation of SPICE-like Simulation Algorithms on Distributed Systems

Mansour R. Darabad
School of Electronics and Computer Sciences
University of Southampton
Southampton SO17 1BJ, UK
mrd1g10@soton.ac.uk

Mark Zwolinski
School of Electronics and Computer Sciences
University of Southampton
Southampton SO17 1BJ, UK
mz@ecs.soton.ac.uk

ABSTRACT

SPICE-like simulation algorithms have many intrinsically sequential elements. Attempts to parallelize these algorithms have resulted in limited speed-ups on multi-cores. We propose to distribute the solution of circuit equations across a large number of light-weight parallel processors. The aim is to avoid the construction of a global network matrix and thus localize all communications by distributed evaluation of Kirchhoff's Current Law at each circuit node. The processors work asynchronously, performing the calculations for each node in a random (non-deterministic) order. Although this results in redundant calculations and slower convergence per node, results suggest that overall a speed-up can be obtained.

Categories and Subject Descriptors

EDA11 [**Electronic Design Automation (EDA)**]: Wild and Crazy Ideas (WACI)

General Terms

Theory, Algorithms, Experimentation

Keywords

Circuit Simulation, Highly Distributed, Parallel, SPICE

1. INTRODUCTION

A SPICE time-domain circuit simulation process has two main phases: device evaluation and matrix solution. Circuit equations are formulated using equation formulation techniques such as MNA (Modified Nodal Analysis). The nonlinear circuit model is solved iteratively by linearization methods such as NR (Newton Raphson). At each NR iterations step a set of simultaneous linear equations in the form of $Ax = b$ is solved, where A is the circuit Jacobian matrix, b (RHS vector) is a vector of excitations, and vector x contains the unknown node voltages and branch currents. The process is repeated until the convergence of the NR iterations. SPICE-type algorithms use direct matrix solving methods like LU factorization to solve $Ax = b$ at each iteration.

Different attempts to speed up the SPICE simulation process, in terms of parallelizing device evaluation and/or matrix solution phase, using multi-core CPUs, GPUs, and FPGAs have resulted in limited speedups or compromised accuracy [2, 3, 4, 5]. More importantly, those attempts mostly use coarse-grained parallel approaches on a small number of processors and do not perform parallelization on large distributed networks.

Moreover, at each NR iteration, the matrix solution process cannot start until the device evaluation phase is done; similarly, the next NR iteration cannot begin until the matrix solution is completed. Therefore, these two barriers between the device evaluation and matrix solution phases limit the amount of possible parallelization [6].

In this paper, we propose a fine-grained parallelization, in which we allocate one processor to each circuit node. We use an iterative equation solution method. On a many-core system, the order of evaluation is non-deterministic, but that the solution and the evaluation time are broadly predictable.

2. PROPOSED METHOD

Because model evaluation of each device can be performed independent of other devices, parallelization is trivial. There have been a number of studies on parallel solvers for the matrix solution phase of the simulation process. However, the efficiency of the parallel solvers can be reduced as the size of the matrix increases. Due to the sparsity of the A matrix and the irregular fine-grained operations required for calculations, conventional multi-core systems are not efficient enough to parallelize matrix solution [2, 4]. Furthermore, exploiting direct methods to solve linear matrix equations causes a barrier at each NR iteration waiting for the new values of x vector to be calculated by the direct solver.

We propose a Jacobi-type iterative method for solving linear matrix equations at each NR iteration. Each row of the A matrix is evaluated on a single processor which is part of a large network of parallel light-weight processors. These processors work asynchronously and calculate the entries of the unknown vector in a completely non-deterministic order. As soon as a new value for any element of the unknown vector (which can be a node voltage or branch current) is calculated, it is passed to the NR iteration to update the non-

Table 1: Simulation results: C/C++ on single core.

Circuit	Matrix size	Iterations	Norm
resistive	16	171	0.00000981
mesh1e1	48	57	0.00000881
pivtol	102	25	0.00000700
Trefethen_150	150	95	0.00000905
Trefethen_200b	199	28	0.00000795
mesh3e1	289	64	0.00000982

Table 2: Simulation results: MPI on Iridis; one processor per node.

Circuit	Matrix size	Iterations	Norm
resistive	16	171	0.00000919
mesh1e1	48	57	0.00000890
pivtol	102	26	0.00000817
Trefethen_150	150	95	0.00000911
Trefethen_200b	199	28	0.00000796
mesh3e1	289	64	0.00000983

linear model of the corresponding component in the matrix. This will lead to a mixed NR and matrix solution iteration.

For the matrix solving part, the Gauss-Seidel method normally converges faster than the Jacobi method but because of the inherent parallelism of Jacobi type iterations, it is possible to distribute it across a large number of parallel processors. Our simulations on benchmark circuits (Table 1) show that performing the Jacobi method in a parallel and *non-deterministic* way so that the equations are solved in a completely random order, which we call *Random Jacobi*, results in the same solution and the same number of iterations for convergence as does the normal Jacobi method.

To investigate the solution of matrix equations on a very fine-grained distributed network of processors (one processor per circuit node), we simulated a number of benchmark circuits from the University of Florida Sparse Matrix Collection [1] and some circuits extracted from Ngspice by dumping the matrix out of the penultimate NR iteration. Simulations are performed using C/C++ on a single core machine, MPI under Linux, and also on a multi-core supercomputer. When running the Jacobi method on the supercomputer, we have no control on the order of execution of equations – equations are solved asynchronously in a non-deterministic order. Experimental results in Table 1 show the evaluation of normal Jacobi on a single core machine. The results for simulating our Random Jacobi method on a real fine-grained network are shown in Table 2. It can be seen that the number of iterations to converge to the solution and the accuracy of calculations (measured by Euclidian norm of two successive solutions for x vector) are very similar.

As the new values for the unknown vector elements are available for parallel non-deterministic calculations, the NR iteration does not need to wait for the completion of matrix solving phase to update the non-linear models with the new values. 1(a) shows the barrier preventing proper parallelization when using direct methods for matrix solution while, 1(b) represents how using Random Jacobi iterations makes it possible to have mixed NR and Jacobi iterations.

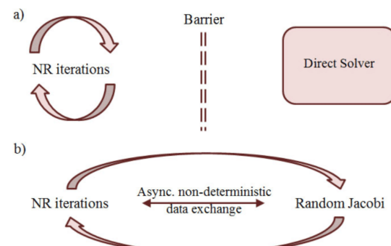


Figure 1: a) NR iterations with direct solver b) NR iterations with Random Jacobi solver.

Our future work will investigate how both NR updating and Random Jacobi iterations can be distributed across a fine-grained network of processors.

3. CONCLUSIONS

A non-deterministic parallel iterative approach is proposed to perform the matrix solving phase of SPICE-like circuit simulation algorithms. This provides new entries for NR iterations as they are available and hence reduces the barrier between device evaluation and matrix solution processes. Simulation results confirm functionality of the proposed Random Jacobi iterations in solving the matrix equations distributed across a large number of light-weight processors.

4. ACKNOWLEDGMENTS

The authors acknowledge the use of the IRIDIS High Performance Computing Facility at the University of Southampton, in this work.

5. REFERENCES

[1] T. A. Davis. The university of florida sparse matrix collection. *NA DIGEST*, 92, 1994.

[2] W. Dong, P. Li, and X. Ye. Wavepipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines. In *Proceedings of the 45th annual Design Automation Conference*, pages 238–243. ACM, June 2008.

[3] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry. Fast circuit simulation on graphics processing units. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 403–408. IEEE Press, January 2009.

[4] N. Kapre and A. DeHon. Parallelizing sparse matrix solve for spice circuit simulation using fpgas. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 190–198, December 2009.

[5] X. Ye, W. Dong, P. Li, and S. Nassif. Maps: multi-algorithm parallel circuit simulation. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 73–78. IEEE Press, November 2008.

[6] M. Zwolinski. Multi-threaded circuit simulation using openmp. In *LASCAS 2010: IEEE Latin American Symposium on Circuits and Systems*, Februery 2010.

Bibliography

[1] C. Moore, "Data processing in exascale-class computer systems," in *Proceedings of the Salishan Conference on High Speed Computing*, ACM, April 2011.

[2] J. Vlach, *Fundamentals of Circuits and Filters, Chapter 23, Tableau and Modified Nodal Formulations.* Circuits and Filters Handbook, 3rd Edition, CRC Press, 2009.

[3] V. Litovski and M. Zwolinski, *VLSI Circuit Simulation and Optimization.* Springer, 1996.

[4] G. Moore, *Computational Linear Algebra; Direct Solution of Linear Systems.* avaialble at *http : //www2.imperial.ac.uk/ gmoore/*, 2009.

[5] G. W. Collins, *Fundamental Numerical Methods and Data Analysis.* eBook, Harvard University Press, 2003.

[6] F. N. Najm, *Circuit Simulation.* Wiley-IEEE Press, 2010.

[7] D. O'Connor, *An Introduction to Numerical Algorithms.* University College, Dublin, available at *http : //www.derekroconnor.net/NA/Notes/NA − −Chaps − −6 − 7.pdf*, 2006.

[8] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Software, Environments, Tools, Society for Industrial and Applied Mathematics, 1987.

[9] B. Barney, *Numerical Algorithms.* Lawrence Livermore National Laboratory, available at: *https : //computing.llnl.gov/tutorials/parallel_comp/*, 2012.

[10] D. Bertsekas and J. Tsitsiklis, *Parallel and distributed computation: numerical methods.* Prentice Hall, 1989.

[11] M. Zwolinski, "Multi-threaded circuit simulation using OpenMP," in *LASCAS 2010: IEEE Latin American Symposium on Circuits and Systems*, February 2010. 24-26 February 2010.

[12] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 190–198, December 2009.

[13] W. Press, *Numerical Recipes in C: The Art of Scientific Computing.* No. v. 4, Cambridge University Press, 1992.

[14] S. Furber, F. Galluppi, S. Temple, and L. Plana, "The SpiNNaker Project," *Proceedings of the IEEE*, vol. 102, pp. 652–665, May 2014.

[15] L. Plana, S. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A gals infrastructure for a massively parallel multiprocessor," *Design Test of Computers, IEEE*, vol. 24, pp. 454–463, Sept 2007.

[16] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown, "Overview of the spinnaker system architecture," *Computers, IEEE Transactions on*, vol. 62, pp. 2454–2467, Dec 2013.

[17] J. Vlach, *Computer Methods for Circuit Analysis and Design.* Van Nostrand Reinhold Electrical/Computer Science and Engine, Springer, 1983.

[18] L. Pillage, R. Rohrer, and C. Visweswariah, *Electronic Circuit and System Simulation Methods.* McGraw-Hill, 1995.

[19] D. O. Pederson, "A historical review of circuit simulation," *Circuits and Systems, IEEE Transactions on*, vol. 31, pp. 103–111, Jan 1984.

[20] L. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* Memorandum, Electronics Research Laboratory, College of Engineering, University of California, 1975.

[21] T. R. Kuphaldt, *Lessons in Electric Circuits, Volume 5 Reference.* Fourth Edition, eBook, 2007.

[22] K. Kundert, *The Designer's Guide to SPICE and Spectre.* The Designer's Guide Book Series, Springer, 1995.

[23] D. Frank, "Power-constrained CMOS scaling limits," *IBM Journal of Research and Development*, vol. 46, pp. 235–244, March 2002.

[24] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, pp. 26–29, Sept. 2005.

[25] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[26] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, M. J. Demmel, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from berkeley," tech. rep., Technical Report, UC Berkeley, 2006.

[27] S. Borkar, "Thousand core chips: A technology perspective," in *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, (New York, NY, USA), pp. 746–749, ACM, 2007.

[28] A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore CPUs," *Computer*, vol. 43, pp. 24–32, March 2010.

[29] W. Dong, P. Li, and X. Ye, "WavePipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *Proceedings of the 45th annual Design Automation Conference*, pp. 238–243, ACM, June 2008.

[30] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pp. 403–408, IEEE Press, January 2009.

[31] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *Circuits and Systems, IEEE Transactions on*, vol. 22, pp. 504 – 509, June 1975.

[32] C. Desoer and E. Kuh, *Basic Circuit Theory*. International student edition, McGraw-Hill, 1984.

[33] J. Mathews and K. Fink, *Numerical methods using MATLAB*. Pearson Prentice Hall, 2004.

[34] T. Davis, *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms, Society for Industrial and Applied Mathematics, 2006.

[35] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[36] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, March 2005.

[37] T. Chen and Y.-K. Chen, "Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pp. 613–616, April 2009.

[38] P. Li, "Parallel circuit simulation: A historical perspective and recent developments," *Foundations and Trends in Electronic Design Automation*, vol. 5, no. 4, pp. 211–318, 2012.

[39] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.

[40] S. Borkar, "Design challenges of technology scaling," *Micro, IEEE*, vol. 19, pp. 23–29, Jul 1999.

[41] J. Held, J. Bautista, and S. Koehl, *From a Few Cores to Many: A Tera-scale Computing Research Overview.* Intel White Paper, Technical Report available at $http://www.intel.co.uk/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-tera-scale-research-paper.pdf$, 2006.

[42] X. Ye, W. Dong, P. Li, and S. Nassif, "MAPS: multi-algorithm parallel circuit simulation," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pp. 73–78, IEEE Press, November 2008.

[43] G. L. Arsov, "The 40-th Anniversary of the Simulation Program with Integrated Circuit Emphasis - SPICE," *IX Symposium Industrial Electronics INDEL*, Nov 2012.

[44] L. Nagel and C. McAndrew, "Is SPICE good enough for tomorrow's analog?," pp. 106–112, Oct 2010.

[45] L. Han, X. Zhao, and Z. Feng, "TinySPICE: A parallel SPICE simulator on GPU for massively repeated small circuit simulations," in *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pp. 1–8, May 2013.

[46] T. Nechma and M. Zwolinski, "Parallel sparse matrix solution for circuit simulation on FPGAs," *Computers, IEEE Transactions on*, vol. 64, pp. 1090–1103, April 2015.

[47] Circuit Lab, "Effortless Schematics Powerful Simulation," 2015. https://www.circuitlab.com/ [Online; accessed 12-July-2015].

[48] TINA Cloud, "Web-based Circuit Design and Analysis," 2015. [http://www.tina.com/English/tinacloud/ Online; accessed 12-July-2015].

[49] Advance Processor Technology Research Group, "SpiNNaker Project," 2015. http://apt.cs.manchester.ac.uk/projects/SpiNNaker/project/ [Online; accessed 12-July-2015].

[50] P. S. Pacheco, "A user's guide to MPI," *Department of Mathematics, University of San Francisco, CA*, March 1998.

[51] A. Scheinine, "Message passing interface tutorial," *Center of Computational Technology and Information Technology Services, Luisiana State University*, September 2008.

[52] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, pp. 46–55, Jan 1998.

[53] K. G. Nichols, T. J. Kazmierski, M. Zwolinski, and A. Brown, "Overview of SPICE-like circuit simulation algorithms," *Circuits, Devices and Systems, IEE Proceedings -*, vol. 141, pp. 242–250, Aug 1994.

[54] A. Vladimirescu, *The SPICE book.* J. Wiley, 1994.

[55] eCircuitCenter, *SPICE ALGORITHM OVERVIEW.* available at *http : //www.ecircuitcenter.com/SpiceTopics/Overview/Overview.htm*, 2003.

[56] K. Singhal and J. Vlach, "Symbolic analysis of analog and digital circuits," *Circuits and Systems, IEEE Transactions on*, vol. 24, pp. 598–609, Nov 1977.

[57] L. Wedepohl and L. Jackson, "Modified nodal analysis: an essential addition to electrical circuit theory and analysis," *Engineering Science and Education Journal*, vol. 11, pp. 84 –92, June 2002.

[58] L. Mandache, D. Topan, and I.-G. Sirbu, "Improved modified nodal analysis of nonlinear analog circuits in the time domain," in *Proceedings of the World Congress on Engineering*, vol. 2, June 2010.

[59] T. J. Ypma, "Historical development of the Newton-Raphson method," *SIAM*, vol. 37, pp. 531–551, Dec. 1995.

[60] J. E. Gentle, W. Hrdle, and Y. Mori, eds., *Handbook of computational statistics : concepts and methods.* New York: Springer-Verlag Berlin Heidelberg, 2004.

[61] I. Gladwell, J. Nagy, and W. Ferguson, *Introduction to Scientific Computing Using Matlab.* Lulu.com, 2011.

[62] I. Duff, A. Erisman, and J. Reid, *Direct methods for sparse matrices.* Monographs on numerical analysis, Clarendon Press, 1986.

[63] C. Ballarin and M. Kauers, "Solving parametric linear systems: an experiment with constraint algebraic programming," in *SIGSAM Bull*, pp. 101–114, 2002.

[64] J. Dobes, "A modified Markowitz criterion for the fast modes of the LU factorization," in *Circuits and Systems, 2005. 48th Midwest Symposium on*, pp. 955–959 Vol. 2, Aug 2005.

[65] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, Sept. 2010.

[66] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations.* No. 16 in Frontiers in Applied Mathematics, SIAM, 1995.

[67] R. Varga, *Matrix Iterative Analysis.* Springer Series in Computational Mathematics, Springer, 2009.

[68] K. Chen, *Matrix Preconditioning Techniques and Applications.* Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press, 2005.

[69] N. Higham, *Accuracy and Stability of Numerical Algorithms: Second Edition.* Society for Industrial and Applied Mathematics, 2002.

[70] J. Burgerscentrum and C. Vuik, *Iterative solution methods.* Delft University of Technology, Delft Institute of Applied Mathematics, available at *http : //ta.twi.tudelft.nl/users/vuik/burgers/lin_notes.pdf*, 2006.

[71] P. K. Suri and S. Mittal, "A comparative study of various computing processing environments: A review," *International Journal of Computer Science and Information Technologies*, vol. 3, pp. 5215–5218, 2012.

[72] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using MPI and OpenMP on multi-core parallel systems," *Parallel Comput.*, vol. 37, pp. 562–575, Sept. 2011.

[73] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[74] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, pp. 33 –38, July 2008.

[75] D. L. Eager, J. Zahorjan, and E. D. Lozowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. Comput.*, vol. 38, pp. 408–423, Mar. 1989.

[76] R. Daniels, H. Von Sosen, and H. Elhak, "Accelerating analog simulation with HSPICE precision parallel technology," *Synopsys, White Paper*, September 2010.

[77] M.-C. Chang and I. Hajj, "iPRIDE: a parallel integrated circuit simulator using direct method," in *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pp. 304–307, Nov 1988.

[78] R. Saleh, K. Gallivan, M.-C. Chang, I. Hajj, D. Smart, and T. N. Trick, "Parallel circuit simulation on supercomputers," *Proceedings of the IEEE*, vol. 77, pp. 1915–1931, Dec 1989.

[79] P. Cox, R. Burch, D. Hocevar, B. Ping Yang, and B. Epler, "Direct circuit simulation algorithms for parallel processing [VLSI]," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 10, pp. 714–725, Jun 1991.

[80] P. Agrawal, S. Goil, S. Liu, and J. Trotter, "Parallel model evaluation for circuit simulation on the PACE multiprocessor," in *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pp. 45–48, Jan 1994.

[81] X. Chen, Y. Wang, and H. Yang, "Parallel circuit simulation on multi/many-core systems," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 2530–2533, May 2012.

[82] D. Paul, M. Nakhla, R. Achar, and N. Nakhla, "Parallel circuit simulation via binary link formulations (PvB)," *Components, Packaging and Manufacturing Technology, IEEE Transactions on*, vol. 3, pp. 768–782, May 2013.

[83] D. Paul, R. Achar, M. Nakhla, and N. Nakhla, "Addressing partitioning issues in parallel circuit simulation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 22, pp. 2713–2723, Dec 2014.

[84] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-accelerated sparse LU factorization for circuit simulation with performance modeling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, pp. 786–795, March 2015.

[85] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *Int. J. High Perform. Comput. Appl.*, vol. 28, pp. 97–111, Feb. 2014.

[86] MATLAB, *version 7.11.0.584 R2010b*. Natick, Massachusetts: The MathWorks Inc., 2010.

[87] C. Woodford and C. Phillips, *Numerical Methods with Worked Examples*. Springer, 1997.

[88] R. Baker, I. of Electrical, E. Engineers, and I. S.-S. C. Society, *CMOS: Circuit Design, Layout, and Simulation*. IEEE Press Series on Microelectronic Systems, Wiley, 2008.

[89] T. A. Davis, "The University of Florida sparse matrix collection," *NA DIGEST*, vol. 92, 1994.

[90] Ngspice, *Ngspice Circuit Simulator*. available at: *http://www.ngspice.org*.

[91] J. Navaridas, M. Luján, J. Miguel-Alonso, L. A. Plana, and S. Furber, "Understanding the interconnection network of spinnaker," pp. 286–295, 2009.