Scalable Parallel Simulation of General Electrical Circuits

by

Douglas Paul, B.A.Sc., M.A.Sc.

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Electronics

Faculty of Engineering

Carleton University

Ottawa, Ontario, Canada

© 2014

Douglas Paul

Abstract

As circuit sizes increase, a means to improve the performance of simulations is constantly demanded, without sacrificing the accuracy of the results. Traditionally, improvements were obtainable through the raw increase of computational power of computing platforms, which allow the sequential algorithms used in circuit simulation to perform at a faster rate.

Recently, however, these performance improvements are no longer obtained through the improvement of individual processors, but rather by including more processors in the system. The result is that higher performance is now obtained via exploiting multicore processors, distributed clusters, and cloud platforms. Traditional sequential algorithms cannot take advantage of the improvements promised by these new systems.

Existing parallel algorithms for circuit simulation are based on the domain decomposition approach. However, it has been demonstrated that domain decomposition suffers scalability problems as the number of processors in a system increases. To address this problem, a new parallel circuit simulation algorithm is presented that allows modern multicore and distributed processors to be exploited to realize this performance improvement. These improvements are obtained without sacrificing accuracy or resorting to iterative techniques. The scalability improvements using the proposed algorithm have been demonstrated through the consideration of several industrial examples.

Acknowledgements

I would like to express my gratitude to my supervisors, Professors Michel Nakhla and Ram Achar, for their guidance and timely support throughout the entire process of researching and preparing this thesis. I would also like to thank the entire Analog Simulation team at Mentor Graphics for their valuable discussions and provision of industrial examples, without which it would have been impossible to verify the performance of this work.

I would also like to express my appreciation to my family and friends, most notably my parents, Bonita and Douglas Paul, and my partner, Karine Savard, for their amazing support throughout my entire PhD program. Their support made it possible for this work to reach this conclusion and for it, I am very grateful.

Contents

1	Introduction	1
1.1	Background and Motivation	1
1.2	Contributions	3
1.3	Organization of the Thesis	4
2	Background	6
2.1	Basis of Circuit Simulation	6
2.2	Circuit Partitioning	9
2.2.1	Flat Partitioning	9
2.2.2	Hierarchical Partitioning	12
2.3	Domain Decomposition	14
2.3.1	Flat Domain Decomposition	14
2.3.2	Hierarchical Domain Decomposition	22
2.3.3	Survey of Current Developments	23
2.4	System Architecture Concerns	24
2.4.1	Memory Architecture	25
2.4.2	Implications for Algorithm Cost	28
2.5	Summary	29

3	Simulation of General Circuits with Node Tearing	30
3.1	Node Tearing	30
3.1.1	Partitioning via Node Tearing	31
3.1.2	Partitions with Multiple Links	36
3.2	Companion Form of Circuits	38
3.2.1	Companion Form for Time-Domain Analysis	38
3.2.2	Companion Form for Nonlinear Time-Domain Circuits	40
3.3	Development of the General Partitioned Circuit Simulation Method . . .	43
3.4	Summary	46
4	Parallel Circuit Simulation with Node Tearing	48
4.1	Parallel Implementation of Node Tearing	48
4.1.1	Task 1: System Formulation	49
4.1.2	Task 2: Decoupled Partition Solution	50
4.1.3	Task 3: Determination of Partition Coupling Influence	50
4.1.4	Task 4: Construction of Interconnection Matrix	51
4.1.5	Task 5: Initial Solution of Partition Interconnection	51
4.1.6	Task 6: Final Solution of Partition Interconnection	52
4.1.7	Task 7: Update of Full Partition Solution	52
4.1.8	Summary	52
4.2	Comparison with Domain Decomposition	53
4.2.1	Comparison of Computational Steps	59
4.2.2	Comparison of CPU Cost	61
4.3	Numerical Examples	69
4.3.1	DSP example	70

4.3.2	Dual SRAM example	73
4.3.3	Array Multiplier example	76
4.3.4	Ring Oscillator example	80
4.4	Summary	82
5	Addressing Partitioning Issues in Node Tearing	83
5.1	Extension of the Node Tearing Concept for non-Voltage Variables	84
5.2	Optimized Handling of Voltage Sources	87
5.3	Handling of Singular Subcircuits	92
5.4	Summary	97
6	Optimized Parallel Implementation for Simulation of General Circuits with Node Tearing	98
6.1	Development of the Pipelined Multi-stage Solution of the Interconnection Matrix	98
6.1.1	Construction and Structure of the Interconnection Matrix	99
6.1.2	Blockwise Updating of the Interconnection Matrix and its Pipelined Multi-stage Solution	103
6.1.3	Addressing Block Fill-ins	108
6.2	Development of the Proposed Parallel Task Scheduler	109
6.2.1	Tasks for Parallel Execution	111
6.2.2	Scheduling Algorithm A	112
6.2.3	Scheduling Algorithm B	118
6.3	Platform Specific Optimizations	119
6.4	Numerical Results	126

6.4.1	Pipelined Multi-stage Solution of Interconnection Matrix	126
6.4.2	DSP example	130
6.4.3	Dual SRAM example	132
6.4.4	Array Multiplier example	140
6.4.5	Ring Oscillator example	146
6.5	Summary	148
7	Hierarchical Circuit Formulations Exploiting Node Tearing	149
7.1	Hierarchical Formulation with Node Tearing	149
7.1.1	Recursive Node Tearing	150
7.2	Hierarchical Simulation with Hybrid Implementation of Node and Branch Tearing	156
7.3	Numerical Examples	158
7.3.1	DSP example	161
7.3.2	Dual SRAM example	162
7.3.3	Array Multiplier example	164
7.3.4	Ring Oscillator example	165
7.4	Interpretation and Future Directions	167
7.5	Summary	168
8	Parallel Simulation of Massively Coupled Interconnect Networks	169
8.1	Motivation	169
8.2	Multiconductor Transmission Line Equations and Transverse Partitioning	171
8.3	Development of the Proposed Parallel Simulation Algorithm	175
8.3.1	Parallel Simulation using Physical Partitioning (PS-PP)	175

8.3.2	Parallel Simulation using Physical and Time-Domain Partitioning (PS-PTP)	178
8.3.3	Parallel Task Scheduling for Networks with Varying Lengths and Differing Terminations	186
8.4	Implementation Details and Considerations	189
8.4.1	Additional Partitioning for Uniform Lumped Segmentation	189
8.4.2	Computational Complexity with Uniform Lumped Segmentation .	191
8.4.3	System Architecture Considerations	194
8.5	Computational Results	198
8.5.1	Processor Scalability Example	198
8.5.2	Long Coupled-Line Eye Diagram	201
8.5.3	Network with Varying Line Lengths and Terminations	204
8.6	Summary	209
9	Conclusions and Future Work	210
9.1	Summary	210
9.2	Future Work	212

List of Figures

2.1	An illustrative circuit.	10
2.2	Hypergraph corresponding to the illustrative circuit.	11
2.3	Example partitioning for the illustrative circuit hypergraph.	11
2.4	Example circuit with hierarchy and the resulting partitioning.	13
2.5	Illustration of branch tearing.	15
2.6	Illustration of scalability.	20
2.7	Illustration of domain decomposition border growth.	21
2.8	Illustration of the nested bordered block diagonal form.	22
2.9	Conceptual Uniform Memory Access (UMA) system.	25
2.10	Conceptual Cache-Coherent Non-Uniform Memory Access (ccNUMA) system.	27
3.1	Partitioned circuit.	32
3.2	Illustration of partitioning with node tearing.	33
3.3	Example for tearing analysis.	34
3.4	Open-circuit voltage measurement.	35
3.5	Thévenin equivalent impedance measurement.	35
3.6	Link current measurement.	36

3.7	Companion model for a capacitor.	39
3.8	Companion model for an inductor.	40
3.9	Companion model for a nonlinear resistor.	42
4.1	Partitioning where border size is identical between node and branch tearing.	57
4.2	Partitioning where border size is greater in node tearing than branch tearing.	58
4.3	Topology for the DSP example.	71
4.4	Illustration of the behaviour of the CPU overhead cost for Steps a + c of the proposed node-tearing-based algorithm with an increasing number of links (for $l \ll n$) for the DSP example.	71
4.5	Parallel scalability for the DSP example.	72
4.6	Accuracy comparison for the DSP example, for a sample digital calculation result (X1.OUT6).	72
4.7	Topology for the SRAM example.	74
4.8	Parallel scalability for the dual SRAM example.	74
4.9	Accuracy comparison for the dual SRAM example, for the XRAM0.- WRITE_ENABLE signal.	75
4.10	Accuracy comparison for the dual SRAM example, for the XRAM0.- VALID_COLUMN_ADR signal.	75
4.11	Circuit block for the multiplier example.	77
4.12	Overall topology for the multiplier example.	77
4.13	Parallel scalability for the multiplier example.	78
4.14	Accuracy comparison for the multiplier example, for the final result signal P0.	78

4.15	Accuracy comparison for the multiplier example, for the intermediate result signal B0_A.	79
4.16	Accuracy comparison for the multiplier example, for the intermediate result signal A0_A.	79
4.17	Parallel scalability for the ring oscillator example.	81
5.1	Example circuit for demonstrating current variable tearing.	84
5.2	Corresponding hypergraph for the example circuit in Figure 5.1.	84
5.3	Transformed example circuit.	85
5.4	Example circuit after splitting.	85
5.5	Example circuit with a highly-connected V_E node.	88
5.6	Example circuit for voltage source handling.	90
5.7	Example case illustrating singular partitioning.	93
5.8	Example singular partitioning with compensation.	96
6.1	Process of the creation of the $\mathbf{Z}$ matrix.	105
6.2	Pseudocode for the pipelined factorization/forward substitution process of $\mathbf{Z}$	106
6.3	Pseudocode for the pipelined backward substitution process of $\mathbf{Z}$	107
6.4	Overall Newton-Raphson iteration flow.	110
6.5	Variable definitions for scheduling algorithms.	112
6.6	Pseudocode for support functions for scheduling algorithms (Part 1 of 3).	113
6.7	Pseudocode for support functions for scheduling algorithms (Part 2 of 3).	114
6.8	Pseudocode for support functions for scheduling algorithms (Part 3 of 3).	115
6.9	Pseudocode for Scheduling Algorithm A.	116
6.10	Pseudocode for Scheduling Algorithm B.	116
6.11	Pseudocode for pipeline depth selection.	119

6.12	Thread activity graph with POSIX thread synchronization.	121
6.13	Thread activity graph with hardware thread synchronization.	121
6.14	Pseudocode for worker thread implementation.	122
6.15	Pseudocode for Scheduling Algorithm B, task 4 processing, with scheduling controlled by the master thread.	124
6.16	Lock-based implementation pseudocode for task 4 processing in Scheduling Algorithm B, where the scheduler runs in each thread.	124
6.17	Example circuit for illustration of Z matrix solution pipelining: (a) Before partitioning. (b) After partitioning.	127
6.18	Parallel scalability for the DSP example.	131
6.19	Thread activity graph for one NR iteration of the dual SRAM example, with 32 partitions and 8 processors using Scheduling Algorithm A.	134
6.20	Thread activity graph for one NR iteration of the dual SRAM example, with 32 partitions and 8 processors using Scheduling Algorithm B.	136
6.21	Parallel scalability for the dual SRAM example, partitions = $2 \times$ CPUs.	138
6.22	Parallel scalability for the dual SRAM example.	139
6.23	Thread activity graph for one NR iteration of the multiplier example, with 32 partitions and 8 processors using Scheduling Algorithm A.	141
6.24	Thread activity graph for one NR iteration of the multiplier example, with 32 partitions and 8 processors using Scheduling Algorithm B.	143
6.25	Parallel scalability for the multiplier example, partitions = $2 \times$ CPUs.	144
6.26	Parallel scalability for the multiplier example.	146
6.27	Parallel scalability for the ring oscillator example.	147
7.1	Circuit partitioning example using recursive bisection.	151

7.2	Circuit partitioning example using recursive bisection, tree representation.	152
7.3	Circuit partitioning transformation to hierarchical recursive bisection form.	159
7.4	Circuit partitioning transformation to hybrid node-tearing/branch-tearing form.	160
7.5	Parallel scalability for the DSP example.	161
7.6	Parallel scalability for the dual SRAM example.	163
7.7	Parallel scalability for the multiplier example.	164
7.8	Parallel scalability for the ring oscillator example.	166
8.1	n_L coupled transmission line system.	172
8.2	Partitioned system for waveform relaxation.	175
8.3	Illustration of task flow requirements for the PS-PP approach.	176
8.4	Illustration of task flow requirements for the PS-PTP approach.	180
8.5	Pseudocode for the proposed PS-PTP algorithm.	184
8.6	Pseudocode for the task flow verification for the PS-PTP algorithm. . . .	185
8.7	Pseudocode for the waveform relaxation iteration.	185
8.8	Pseudocode for the proposed PS-PTP algorithm with modified scheduling for networks with varying line lengths and differing terminations.	188
8.9	Uniform lumped segmentation model with relaxation sources, for the m^{th} segment.	190
8.10	Comparison of the computational cost of waveform relaxation iterations versus the number of lines (on a single CPU).	193
8.11	Comparison of the computational cost for a full coupled line simulation and the proposed PS-PTP approach.	194
8.12	Comparison of CPU work scaling for the simulation stage.	196

8.13	Comparison of CPU work scaling for the source update stage.	197
8.14	Comparison of CPU work scaling for the overall simulation (including source update cost).	197
8.15	Example coupled line network.	198
8.16	Simulation results for the far end of aggressor line ($V_{out,1}$).	200
8.17	Simulation results for the near end of victim line ($V_{out,1}$).	200
8.18	Speedup analysis for the proposed parallel algorithms.	201
8.19	Simulation results for the far end of line 1.	203
8.20	Simulation results for the far end of line 17.	203
8.21	Example coupled line network, with varying line lengths and differing terminations.	205
8.22	Speedup analysis for the proposed parallel algorithms for varying line lengths, sources, and loads.	207
8.23	Simulation convergence, for the far end of line 6.	207
8.24	Simulation convergence, for the far end of line 7.	208
8.25	Simulation convergence, for the far end of line 8.	208

List of Tables

3.1	Cost comparison of matrix-matrix multiplication	46
4.1	Cost comparison	69
4.2	Performance results for the DSP example	73
4.3	Performance results for the dual SRAM example	76
4.4	Performance results for the multiplier example	80
4.5	Performance results for the ring oscillator example	81
6.1	Performance results for the DSP example	132
6.2	CPU cost of one NR iteration task for the dual SRAM example, 32 partitions	137
6.3	Parallel scalability for the dual SRAM example, partitions = $2 \times$ CPUs . .	138
6.4	Performance results for the dual SRAM example	140
6.5	CPU cost of one NR iteration task for the multiplier example, 32 partitions	144
6.6	Parallel scalability for the multiplier example, partitions = $2 \times$ CPUs. . .	145
6.7	Performance results for the multiplier example	145
6.8	Performance results for the ring oscillator example	147
7.1	Performance results for the DSP example	162
7.2	Performance results for the dual SRAM example	163

7.3	Performance results for the multiplier example	165
7.4	Performance results for the ring oscillator example.	166
8.1	Execution stages for Illustrative Example 7 demonstrating the PS-PP approach	177
8.2	Execution stages for Illustrative Example 8 demonstrating the PS-PTP approach, $n_L \geq n_P$	180
8.3	Execution stages for Illustrative Example 9 demonstrating the PS-PTP approach, $n_L < n_P$	183
8.4	Execution process for Illustrative Example 10 demonstrating the PS-PTP approach with non-uniform line circuits	187
8.5	Execution process for Illustrative Example 11 demonstrating the PS-PTP approach with varying line circuits and modified scheduling algorithm . .	189
8.6	Performance of the proposed algorithm for Example C	206

List of Symbols

α	Term for complexity of a matrix factorization algorithm: for a matrix of size n , the complexity will be $O(n^\alpha)$.
β	Coefficient reflecting the size of a border in a bordered block diagonal matrix.
γ	Vector containing the currents in an interconnection network for a circuit partitioned with node tearing.
Γ	Matrix containing the partition responses to unit current sources being connected to each port.
ϵ	Error tolerance: a set of Newton-Raphson iterations is considered converged when the error is below ϵ .
ξ	A vector containing connectivity information for a single link between partitions.
Ξ	A matrix containing all of the ξ vectors for a given partitioning.
ω	Angular frequency
A	The left-hand side matrix from a linear system of equations.

$\mathbf{b}$	The right-hand side vector from a linear system of equations.
$\mathbf{b}(t)$	A vector containing the time-domain excitations for a circuit.
$\mathbf{C}$	A matrix containing capacitance and inductance information for an MNA representation of a circuit. For transmission line network parameters, the per-unit-length capacitance matrix.
$e_j(x, t)$	Coupling causing an induced voltage in the j^{th} line in an interconnection network.
f	The fraction of an algorithm that can be executed in parallel.
$\mathbf{f}(\mathbf{x})$	A vector containing equations for the nonlinear elements in an MNA representation of a circuit.
$\mathbf{G}$	A matrix containing conductance information for an MNA representation of a circuit. For transmission line network parameters, the per-unit-length conductance matrix.
h	The step size with which the time domain has been discretized in a transient circuit simulation.
$\mathbf{i}(x, t)$	Vector representing the current in each of the conductors of an interconnection network.
$\tilde{i}(d, t)$	Function for a current relaxation source representing the signals coupled from the other lines.
l	The number of links needed to connect the partitions in a system partitioned with node tearing.

$\mathbf{L}$	For transmission line network parameters, the per-unit-length inductance matrix.
$\mathbf{M}$	A matrix containing the Jacobian of a vector of nonlinear equations for use in Newton-Raphson iterations.
n	The size of a system.
n_B	Number of time blocks used in time-domain partitioning.
n_L	Number of lines in a coupled interconnection network.
n_P	Number of partitions or number of processors.
p	The number of processors in a system for analysis of a parallel algorithm's speedup.
$q_j(x, t)$	Coupling causing an induced current in the j^{th} line in an interconnection network.
R	The speedup factor of a parallel algorithm.
$\mathbf{R}$	For transmission line network parameters, the per-unit-length resistance matrix.
$\mathbf{v}(x, t)$	Vector representing the voltage seen on each of the conductors of an interconnection network.
$\tilde{v}(d, t)$	Function for a voltage relaxation source representing the signals coupled from the other lines.

- $\mathbf{x}$ The vector of unknowns in a linear system, or the vector of circuit variables in an MNA system.
- $\hat{\mathbf{x}}$ The solved vector of circuit variables for a partition in a partitioned system, before considering the effects of the other partitions.
- $\mathbf{Y}$ A matrix consisting of the combination of $\mathbf{G}$ and $\mathbf{C}$ for either a time-domain or frequency-domain MNA analysis.
- $\mathbf{Z}$ A matrix capturing the terminal characteristics (z-parameters) of the ports of the partitions in a system partitioned with node tearing.

Abbreviations

ALU	<i>Arithmetic and Logic Unit</i> , one of the building blocks of computer processors.
ccNUMA	<i>Cache-Coherent Non-Uniform Memory Access</i> , a system architecture for multi-processor systems where the cost of accessing the same memory location is not the same for each processor.
CPU	<i>Central Processing Unit</i> , a main processor in a computer system.
DD	<i>Domain Decomposition</i> , a method for solving a partitioned linear system.
DSP	<i>Digital Signal Processor</i> , a special type of computer processor designed for the processing of digital signals in real time.
EM	<i>Electromagnetic</i> , in the context of simulations, including interference and other parasitic effects that impact signal integrity.
FB	<i>Forward-Backward Substitutions</i> , a simple matrix solution method used when the system matrix has been decomposed into the product of a lower triangular and an upper triangular matrix.

LU	<i>Lower Triangular/Upper Triangular</i> , a form of matrix decomposition into two triangular matrices, part of a method to solve a system of equations without resorting to matrix inversion.
MNA	<i>Modified Nodal Analysis</i> , a technique of formulating circuit equations based on Kirchoff's Current Law and nodal analysis, with modifications to the handling of voltage sources and inductors.
NR	<i>Newton-Raphson</i> , a well-known method of solving a nonlinear algebraic equation, where, starting from an initial guess of the solution, the solution is iteratively refined by solving the linearized system at the current point and updating the solution until convergence is reached.
PS-PP	<i>Parallel Simulation using Physical Partitioning</i> , a form of parallel simulation of interconnects using waveform relaxation and finding parallelism in the physical structure of the device.
PS-PTP	<i>Parallel Simulation using Physical and Time-Domain Partitioning</i> , similar for PS-PP but with additional partitioning in the time-domain to improve processor utilization.
RF	<i>Radio Frequency</i> , circuit simulations where the operating frequency is significantly higher than the baseband.
SPICE	<i>Simulation Program with Integrated Circuit Emphasis</i> , the standard simulator for time-domain simulation of nonlinear circuits.
SRAM	<i>Static Random-Access Memory</i> , a high-performance form of memory where each memory cell consists of a latch instead of a capacitor and does not

need to be periodically refreshed.

TP *Transverse Partitioning*, a form of interconnect partitioning where the partitioning is done lengthwise, between the coupled lines, instead of cutting across the interconnects.

UMA *Uniform Memory Access*, a system architecture for multi-processor systems where the cost of accessing a particular memory address is the same for each processor in the system.

WR *Waveform Relaxation*, an iterative method for solving electrical circuits.

Chapter 1

Introduction

1.1 Background and Motivation

Circuit simulators are increasingly being tasked with the simulation of larger circuits. These large circuits are appearing for a variety of reasons. One major reason is that modern electronic products have a multitude of functions in one unit, and there is a desire to simulate the entire system together. There are also emerging mixed-domain design needs, with a single design requiring the simulation of various analog, digital, EM, RF and thermal modules at all levels of the system hierarchy.

In addition, as operating frequencies of devices increase as their physical sizes decrease, the circuits must include modelling of previously negligible effects of interconnects and other parasitics, such as ringing, delay, distortion, reflections and crosstalk. Including these effects requires complex distributed models, which in turn can significantly inflate the resulting circuit sizes, leading to excessive simulation time.

It is to be noted that the core operation of a circuit simulation is the Newton-Raphson iteration, which consists of solving a linearized system of equations. It is known that the complexity of this task is $O(n^\alpha)$, where n is the number of system states, and α is a function

of the sparsity of the system matrix. In typical circuits, α varies between 1.1 and 2.4 [1].

Much work has been done to improve the efficiency of the task of solving this system [2–4], which has the effect of reducing α . This was the preferred approach in the past, as in traditional computer architectures, performance improvements are obtained as the raw processing power of the system increases. This improvement of processing ability coupled with algorithmic improvement has been outpacing the growth of the complexity of simulation tasks.

However, recently, the raw processing power of systems has been increasing through the adoption of parallel processing architectures instead of by increasing the pure performance of a single processor. However, traditional algorithms, which generally are sequential in nature, can only exploit the power of a single processor, and are not positioned to take advantage of the emerging parallel platforms.

The combination of these two effects implies that the clear way to improve the performance of solvers in circuit simulators is to find a means to exploit the parallel processing abilities that are becoming ubiquitous in modern computer architectures.

The major existing approach that works to realize this goal is known as Domain Decomposition [5]. This method works by partitioning the overall circuit into several subcircuits, and using the resulting structure to permute the system matrix into a bordered block-diagonal form, which can be solved using parallel techniques. Much work has been presented developing this technique [6, 7]; however, it has been found that this method suffers from scalability problems when the number of processors in a system increases [8, 9].

In order to overcome this major bottleneck, in this thesis, a novel approach to the parallel solution of a general circuit system is proposed. This approach is based on the

techniques of Node Tearing [10, 11], and is based on the fundamental circuit theory of large-scale sensitivities. The new method provides for superior scalability, which makes it more attractive than the Domain Decomposition approach, for general circuits.

For special classes of circuits, it is possible to develop even more scalable parallel algorithms by targeting the specific circuit and the architecture of the system being exploited. To analyze the potential benefits of this approach, the problem of simulating large coupled networks of transmission lines has been additionally considered. A method for simulating these networks exploiting Waveform Relaxation [12] and Transverse Partitioning [13] is proposed, and an implementation tuned to a common parallel architecture has been created.

1.2 Contributions

The focus of the thesis is to develop an efficient method for parallel simulation of general circuits. The specific contributions of the thesis are as follows.

1. An algorithm is proposed for the simulation of a partitioned circuit [14], exploiting the technique of node tearing (which is otherwise limited to linear circuits [11]), for transient simulation of nonlinear circuits.
2. A scalable parallel algorithm for circuit simulation [15], based on the proposed method, and the design of an efficient implementation are developed [16].
3. A detailed analysis of the algorithmic complexity of the proposed algorithm compared to the widely-used existing technique, domain decomposition with branch tearing, is performed [15].
4. A simulator implementation to efficiently handle the case of massively coupled

transmission line networks in a scalable fashion is also proposed [17, 18].

1.3 Organization of the Thesis

The thesis is organized as follows. Chapter 2 gives a literature survey and background on the existing competing methods as well as the techniques that form the foundation for this work. This includes an overview of domain decomposition methods and circuit partitioning.

Chapter 3 describes the formulation of the proposed partitioned simulation algorithm for the case where a flat partitioning method is used. Instead of using a domain-decomposition-based approach, an alternative approach, node tearing, is described. The algorithm is then refined for maximum flexibility and efficiency.

In Chapter 4, the means to implement the proposed algorithm on a parallel computer is studied. The anticipated scalability characteristics compared to Domain Decomposition are developed through a detailed comparison of the two techniques, and a set of example circuits of greatly varying topologies are tested to validate the performance.

Next, Chapter 5 extends the proposed algorithm by improving the flexibility of the partitioning. By reducing the constraints on the partitioning, a more efficient partitioning is easier to find. This, in turn, can improve scalability. In addition, some additional methods to improve efficiency are also proposed.

In Chapter 6, the proposed parallel algorithm is analysed in more detail, this time with regards to optimizing processor utilisation. By more efficiently making use of the processors, even more scalability can be obtained.

Chapter 7 describes a preliminary exploration of the extension of this algorithm to the case of hierarchical partitioning, which exploits the existing structure of the overall system

to improve the quality of the partitioning and to ideally provide additional parallelism. Additional hierarchical modes are further explored, with the examination of exploiting domain decomposition for the solution of the lowest levels of the hierarchy tree.

Next, Chapter 8 demonstrates how additional efficiencies can be found when a specific class of problems is considered and optimizations specific to an architecture are implemented. In this case, the specific problem considered is that of the simulation of massively coupled interconnect networks.

Finally, Chapter 9 provides a brief summary of the thesis, gives some concluding remarks and suggests some directions for future work.

Chapter 2

Background

Before beginning the development of the proposed algorithms, a review of the necessary preliminaries is presented in this chapter. The first section gives a brief overview of the circuit simulation techniques exploited by SPICE-like simulators. These techniques treat the circuit as a single unit. To develop an algorithm that treats the circuit as a set of subcircuits, a means to divide the circuits into several partitions is needed. This technique is presented in the second section. Next, the existing means to perform a circuit simulation on a set of subcircuits, domain decomposition, is presented in the third section. Finally, the effects of system architecture on parallel algorithm development are considered.

2.1 Basis of Circuit Simulation

The standard for conventional circuit simulation is the SPICE-like simulator, whose basic principles are described in [1]. The overall flow is the following. They first formulate the circuit system as a set of nonlinear ordinary differential equations. To perform a time-domain simulation, an integrating formula is used to discretize the time domain, producing a nonlinear difference equation. The repeated solution of this equation is used to march the

simulation in time from some initial state. Newton-Raphson iterations are used to solve the nonlinear algebraic system of equations.

The nonlinear ordinary differential equations that are formulated by SPICE-like circuit simulators are known as the Modified Nodal Analysis (MNA) formulation [1]. With this formulation, any circuit to be analyzed can be represented with the following system of equations:

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\dot{\mathbf{x}}(t) + \mathbf{f}(\mathbf{x}(t)) = \mathbf{b}(t), \quad (2.1)$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the vector of system states, containing all the node voltage and necessary branch currents, $\dot{\mathbf{x}}(t) \in \mathbb{R}^n$ is its time-domain derivative, n is the number of system states, $\mathbf{G} \in \mathbb{R}^{n \times n}$ is the modified nodal conductance matrix, $\mathbf{C} \in \mathbb{R}^{n \times n}$ is the modified nodal capacitance matrix, $\mathbf{f}(\mathbf{x}(t)) \in \mathbb{R}^n$ captures the behaviour of all the nonlinear devices in the circuit, and $\mathbf{b}(t) \in \mathbb{R}^n$ represents all of the independent sources of the circuit.

To determine the response of this system, it is first converted to a nonlinear difference equation, by discretizing the time domain through the selection of a step size, h , and an appropriate selection of an integrating function. For simplicity, if the Backward Euler [1] method is to be used, as given at time $t = t_i$,

$$\mathbf{x}(t_i + h) = \mathbf{x}(t_i) + h\dot{\mathbf{x}}(t_i + h), \quad (2.2)$$

the resulting difference equation will be:

$$\left[\mathbf{G} + \frac{\mathbf{C}}{h} \right] \mathbf{x}(t_i + h) + \mathbf{f}(\mathbf{x}(t_i + h)) = \frac{\mathbf{C}}{h} \mathbf{x}(t_i) + \mathbf{b}(t_i). \quad (2.3)$$

In practical simulators, different integrating functions can be used, and additionally the value of h does not need to be fixed. For example, h might be smaller when some signals

in the circuit are undergoing sharp transitions, and can be larger when the signals are more steady. The description of such techniques and means for determining the step size are outside of the scope of this work, but a detailed overview can be found in [1].

The solution of (2.3) will give the state of the circuit at time $t = t_i + h$ given the state at time $t = t_i$. As it is a nonlinear algebraic equation, it can be solved using Newton-Raphson iterations, which consists of repeatedly solving the following linear equation:

$$\left[\mathbf{G} + \frac{\mathbf{C}}{h} + \mathbf{M}^{(j)} \right] \mathbf{x}^{(j)} = \mathbf{M}^{(j)} \mathbf{x}^{(j-1)} - \mathbf{f}(\mathbf{x}^{(j-1)}) + \frac{\mathbf{C}}{h} \mathbf{x}(t_i) + \mathbf{b}(t_i + h), \quad (2.4)$$

where j is the iteration number,

$$\mathbf{M}^{(j)} = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}=\mathbf{x}^{(j-1)}}, \quad (2.5)$$

and $\mathbf{x}^{(0)}$ is generally set equal to the final value of $\mathbf{x}$ from the previous time step. There are many techniques for determining convergence, but, in general, criteria similar to the following are used:

- $\|\mathbf{x}^{(j)} - \mathbf{x}^{(j-1)}\| < \epsilon_1$, or that the solution is settling to a certain value, and
- $\left\| \left[\mathbf{G} + \frac{\mathbf{C}}{h} + \mathbf{M}^{(j)} \right] \mathbf{x}^{(j)} - \left[\mathbf{M}^{(j)} \mathbf{x}^{(j-1)} - \mathbf{f}(\mathbf{x}^{(j-1)}) + \frac{\mathbf{C}}{h} \mathbf{x}(t_i) + \mathbf{b}(t_i + h) \right] \right\| < \epsilon_2$, or that $\mathbf{x}^{(j)}$ is a solution to (2.4) within a certain tolerance.

This method solves an entire circuit as a single system of equations, given in (2.4). As discussed previously, the CPU cost, in terms of the computational complexity, of solving this system will be $O(n^\alpha)$, where α varies between 1.1 and 2.4 [1], depending on the circuit and the solution algorithm. As α is larger than 1, the CPU cost will increase faster than the size of the circuit. Hence, in order to solve large systems, it is desirable to split the

circuit into a number of subcircuits. These subcircuits are solved separately and then the coupling among them is considered to find the solution for the overall system. The means to perform this partitioning is considered in the next section.

2.2 Circuit Partitioning

2.2.1 Flat Partitioning

Before a circuit can be analysed using an algorithm that operates on subcircuits, these subcircuits must be defined using partitioning. Partitioning is a technique that is used for many problem domains, so generally available partitioning algorithms are not designed specifically for circuits, but instead partition graphs and hypergraphs. Therefore, to exploit these existing implementations, the circuit topology must be first converted to a hypergraph. A hypergraph is similar to an undirected graph, except hyperedges are used in the place of edges. A hyperedge has the same function as an edge except it can be connected to more than two vertices.

There are many methods by which a circuit can be represented by a hypergraph. For the purposes of illustration, consider a circuit hypergraph where each vertex corresponds to one or more circuit elements, while each hyperedge corresponds to a voltage node. All circuit elements which share a connection to the same node will be connected by a hyperedge corresponding to this node. Each voltage node in the circuit other than the reference node will have a matching hyperedge.

In some cases, different circuit elements may also share current variables. Examples of these include current-controlled sources and inductors with coupling. As it has been defined that each hyperedge corresponds to a voltage node, elements sharing current variables are

Figure 2.1: An illustrative circuit.

coalesced into a single vertex.

To illustrate this method, consider the circuit shown in Figure 2.1. Each element in the circuit is identified by a SPICE-style element name. The corresponding hypergraph is shown in Figure 2.2. Element V3 is a null voltage source used to measure current for the dependent source F1, and these two elements share a current variable and thus become a single vertex. This procedure is also followed for the two inductors and the mutual inductance forming the transformer element.

Once the hypergraph is constructed, a partitioning algorithm is used. This algorithm tries to balance the number of vertices in each partition. It must also minimize the number of hyperedges which are cut in the process. The number of vertices in each partition gives an idea of the total workload present in each partition. Each vertex usually represents a nonlinear device model that must be evaluated, as well as being associated with a certain number of nonzeros in the system matrix. If the number of vertices are similar among the partitions, it can be expected that the work involved in solving these systems will be similar.

Figure 2.2: Hypergraph corresponding to the illustrative circuit.

Figure 2.3: Example partitioning for the illustrative circuit hypergraph.

Each hyperedge that is cut represents a voltage node whose value must be resolved by the overall system solution algorithm. The details of these solution algorithms are given in the following sections; however, it is clear that each cut hyperedge represents a certain amount of work, so the number of cuts must be minimized.

The two optimization criteria can have weights associated with them, so, for example, a certain amount of partition size imbalance can be tolerated in order to obtain a smaller number of hyperedge cuts. In addition, the vertices can be individually weighted depending on the computational cost associated with the class of device it represents. The choice of each of these weights depends greatly on the characteristics of the solution algorithm used and are best chosen following the benchmarking of many example circuits of various types. An example result from the partitioning of the example circuit using a trivial method is shown in Figure 2.3. As can be seen, in this partitioning a single voltage node has been torn between the two partitions.

This task, referred to as *flat partitioning*, is performed by a number of existing hypergraph partitioners, such as hMETIS [19, 20] and Mondriaan [21, 22].

The final result of the partitioning is a set of subgraphs connected by a number of hyperedges. Each of these subgraphs correspond to a subcircuit containing some number of elements. The hyperedges connecting these subgraphs correspond to the hyperedges cut during partitioning. These represent the nodes that are common among the subcircuits.

2.2.2 Hierarchical Partitioning

Some hypergraph partitioners, such as hMETIS, go through a process of graph coarsening to simplify the graph before partitioning. The process attempts to identify a set of subgraphs that are weakly connected, to reduce the problem size, and simplify the process of producing

Figure 2.4: Example circuit with hierarchy and the resulting partitioning.

an optimal partitioning.

In many cases, however, circuits are designed in a hierarchical manner. For example, in a digital design, there is a top-level design, which consists of interconnected modules, which in turn are formed by other modules, down to the level of digital cells. In a SPICE netlist, this is seen by the use of subcircuit and macro elements.

If this design hierarchy is included in the graph generation process, the result is a graph that consists of a set of interconnected subgraphs. This can be considered as a pre-coarsened graph. In the case where the netlist hierarchy corresponds to a design hierarchy, this can ease the burden of the partitioner and result in a high quality partitioning.

A partitioning algorithm that exploits the hierarchy present in the netlist will be referred to as a *hierarchical partitioner*. For this type of implementation, in addition to providing a set of partitions and the nodes which connect them, a hierarchical partitioner also provides a tree representation of the hierarchy of these partitions, as illustrated in Figure 2.4. This tree structure is exploited in the method presented in Section 2.3.2.

2.3 Domain Decomposition

In this section, an overview and literature survey of the current parallel circuit simulation techniques, exploiting Domain Decomposition (DD) [5–7], is given.

2.3.1 Flat Domain Decomposition

As described in Section 2.1, the transient simulation of a circuit consists of the repeated solution of Newton-Raphson iterations, defined by the linear equation (2.4). For the clarity of this discussion, this system can be mapped into the standard linear equation form,

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad (2.6)$$

where

$$\mathbf{A} = \mathbf{G} + \frac{\mathbf{C}}{h} + \mathbf{M}^{(j)} \quad (2.7)$$

$$\mathbf{x} = \mathbf{x}^{(j)}, \quad \text{and} \quad (2.8)$$

$$\mathbf{b} = \mathbf{M}^{(j)}\mathbf{x}^{(j-1)} - \mathbf{f}(\mathbf{x}^{(j-1)}) + \frac{\mathbf{C}}{h}\mathbf{x}(t_i) + \mathbf{b}(t_i + h). \quad (2.9)$$

Before starting the simulation, a partitioning is found using the methods of Section 2.2 to define several sub-circuits. From the results of the partitioning, the $\mathbf{A}$ matrix and $\mathbf{x}$ vector are reordered in the following manner:

- Variables corresponding to the hyperedges within each partition subgraph are collected together, and
- Variables corresponding to the hyperedges which connect the subgraphs are shifted to the end.

Figure 2.5: Illustration of branch tearing.

At a circuit level, this results in the partitioning method illustrated in Figure 2.5. With this method, branches are found at the partition edges. These branches are effectively ‘torn’ between the partition and the inter-partition space, so this form of partitioning is referred to as *branch tearing*.

Mathematically, this can be viewed as applying a permutation to the system of (2.6). If this permutation is represented as the matrix P , the following system is obtained:

$$PAP^T Px = Pb, \quad (2.10)$$

$$(PAP^T)\hat{x} = \hat{b}, \quad (2.11)$$

$$\begin{bmatrix} A_1 & & & F_1 \\ & A_2 & & F_2 \\ & & \ddots & \vdots \\ & & & A_p & F_p \\ E_1 & E_2 & \dots & E_p & A_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \\ x_0 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \\ b_0 \end{bmatrix}. \quad (2.12)$$

The form of (2.12) is known as *bordered block diagonal*. To solve this type of system while exposing parallelism, a special solution procedure is used. First, the system is reorganized to isolate the $\mathbf{x}_0$ variables:

$$\begin{bmatrix} \mathbf{A}_1 & & \\ & \mathbf{A}_2 & \\ & & \ddots \\ & & & \mathbf{A}_p \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_p \end{bmatrix} - \begin{bmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \\ \vdots \\ \mathbf{F}_p \end{bmatrix} \mathbf{x}_0, \quad (2.13)$$

$$\mathbf{A}_0 \mathbf{x}_0 = \mathbf{b}_0 - \begin{bmatrix} \mathbf{E}_1 & \mathbf{E}_2 & \dots & \mathbf{E}_p \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_p \end{bmatrix}^\top \quad (2.14)$$

Next, each $\mathbf{x}_i$ vector is split into two parts, $\hat{\mathbf{x}}_i + \Delta \mathbf{x}_i$, by the following definition:

$$\begin{bmatrix} \mathbf{A}_1 & & \\ & \mathbf{A}_2 & \\ & & \ddots \\ & & & \mathbf{A}_p \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_1 \\ \hat{\mathbf{x}}_2 \\ \vdots \\ \hat{\mathbf{x}}_p \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_p \end{bmatrix}, \quad (2.15)$$

$$\begin{bmatrix} \mathbf{A}_1 & & \\ & \mathbf{A}_2 & \\ & & \ddots \\ & & & \mathbf{A}_p \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_1 \\ \Delta \mathbf{x}_2 \\ \vdots \\ \Delta \mathbf{x}_p \end{bmatrix} = - \begin{bmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \\ \vdots \\ \mathbf{F}_p \end{bmatrix} \mathbf{x}_0. \quad (2.16)$$

Applying this to (2.14), gives:

$$\begin{aligned} \mathbf{A}_0 \mathbf{x}_0 = \mathbf{b}_0 - \begin{bmatrix} \mathbf{E}_1 & \mathbf{E}_2 & \dots & \mathbf{E}_p \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_1 & \hat{\mathbf{x}}_2 & \dots & \hat{\mathbf{x}}_p \end{bmatrix}^\top - \\ \begin{bmatrix} \mathbf{E}_1 & \mathbf{E}_2 & \dots & \mathbf{E}_p \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_1 & \Delta \mathbf{x}_2 & \dots & \Delta \mathbf{x}_p \end{bmatrix}^\top \end{aligned} \quad (2.17)$$

Next, (2.16) is rearranged and substituted into (2.17):

$$\begin{bmatrix} \Delta \mathbf{x}_1 \\ \Delta \mathbf{x}_2 \\ \vdots \\ \Delta \mathbf{x}_p \end{bmatrix} = - \begin{bmatrix} \mathbf{A}_1^{-1} \mathbf{F}_1 \\ \mathbf{A}_2^{-1} \mathbf{F}_2 \\ \vdots \\ \mathbf{A}_p^{-1} \mathbf{F}_p \end{bmatrix} \mathbf{x}_0, \quad (2.18)$$

$$\mathbf{A}_0 \mathbf{x}_0 = \mathbf{b}_0 - \begin{bmatrix} \mathbf{E}_1 & \mathbf{E}_2 & \dots & \mathbf{E}_p \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_1 & \hat{\mathbf{x}}_2 & \dots & \hat{\mathbf{x}}_p \end{bmatrix}^\top + \begin{bmatrix} \mathbf{E}_1 & \mathbf{E}_2 & \dots & \mathbf{E}_p \end{bmatrix} \begin{bmatrix} \mathbf{A}_1^{-1} \mathbf{F}_1 \\ \mathbf{A}_2^{-1} \mathbf{F}_2 \\ \vdots \\ \mathbf{A}_p^{-1} \mathbf{F}_p \end{bmatrix} \mathbf{x}_0. \quad (2.19)$$

After some rearranging, this can be written as

$$\left[\mathbf{A}_0 - \sum_{i=1}^p \mathbf{E}_i \mathbf{A}_i^{-1} \mathbf{F}_i \right] \mathbf{x}_0 = \mathbf{b}_0 - \sum_{i=1}^p \mathbf{E}_i \hat{\mathbf{x}}_i. \quad (2.20)$$

Therefore, the entire system can be solved by first solving (2.15), followed by (2.20). Once $\mathbf{x}_0$ has been found, then the $\Delta \mathbf{x}_i$ vectors can be found using (2.16), and the final solutions are found by

$$\mathbf{x}_i = \hat{\mathbf{x}}_i + \Delta \mathbf{x}_i. \quad (2.21)$$

In a parallel implementation, parallelism can be found by exploiting the block diagonal structure of the matrices in (2.15) and (2.16), as well as by evaluating the parts of the summations in (2.20) simultaneously.

However, one of the major limitations of the existing DD-based parallel circuit simulation algorithms is that they do not scale well with the increasing number of links among

the partitions. To see why this is a problem, consider a hypothetical circuit being simulated on three platforms: one with two processors, one with four processors, and one with eight processors.

Assume that the partitioning being used will produce a number of partitions equal to the number of processors. Assume also that the partitioning being generated will be optimal. As more partitions are created, more hyperedges from the original circuit graph must be cut. As these cut hyperedges correspond directly to the size of the border, this means that the border will be necessarily larger as the number of partitions increase. Therefore, let n be the size of the system, and β_j be the size of the border with j partitions, such that $\beta_2 \leq \beta_4 \leq \beta_8 \leq n$.

The size of the systems of (2.15) and (2.16) will thus be $n - \beta_j$ when j partitions are used. It is known that the cost of solving a circuit-originated sparse system of size n is $O(n^\alpha)$, $1.1 \leq \alpha \leq 2.4$ [1]. Therefore, the work involved in solving these equations will be

$$O\left(j \left(\frac{n - \beta_j}{j}\right)^\alpha\right) \quad (2.22)$$

and, since these can be solved in parallel using j processors, the time necessary to solve them will be

$$O\left(\left(\frac{n - \beta_j}{j}\right)^\alpha\right). \quad (2.23)$$

As β_j increases, the size of A_0 in (2.20) will become larger, and the cost of solving this equation will become dominated by the LU factorization of the left-hand-side matrix. The cost of this, and thus the time required as it is a sequential operation, is given by

$$O(\beta_j^\alpha) \quad (2.24)$$

If the other components of the solution are neglected, (2.22) can be taken as the work that can be done in parallel, and (2.24) the work that must be done sequentially. Consider Amdahl's Law [23]

$$R(f, p) = \frac{P}{f + (1 - f)p}, \quad (2.25)$$

where p is the number of processors, f is the fraction of the algorithm's work which can be executed in parallel, and R is the speedup ratio.

It can be seen from the above discussion that in the case of domain decomposition, as p increases, f will decrease. Consider a hypothetical algorithm where 90% of the work is parallelizable. The speedup curve resulting from this is shown in Figure 2.6 as the solid line. As can be seen, the presence of any non-parallelizable work in an algorithm greatly limits scalability in itself. Consider next a second algorithm which starts with 90% parallelizable work for two processors, decreasing to 80% for eight processors. This is shown as the dashed-dotted line. This sort of behaviour, similar to what is seen with domain decomposition, results in even more limited scalability.

Figure 2.7 demonstrates this effect in another manner. In this figure, the effect of the growing β factor as the number of partitions increases is shown. The growing border moves more and more of the matrix from the block-diagonal portion, which is where the parallelism is found. The more work needed to solve this border, the less work there is to be done in parallel, so scalability suffers.

This situation is aggravated further by the possibility that the individual A_j matrices can be singular after the reordering. One way to deal with this case is to relocate the variables corresponding to the local singularities to the border. This has the side effect of further reducing scalability.

In an attempt to avoid the issue of border growth, a different form of domain decompo-

Figure 2.6: Illustration of scalability.

sition, using a hierarchical formulation, has been proposed in the literature, and is briefly described in the next section.

Figure 2.7: Illustration of domain decomposition border growth.

Figure 2.8: Illustration of the nested bordered block diagonal form.

2.3.2 Hierarchical Domain Decomposition

Hierarchical Domain Decomposition [7] is a special formulation of domain decomposition. Its primary characteristic is that instead of a flat partitioning, hierarchical partitioning, as described in Section 2.2.2, is used.

When this style of partitioning is used, the resulting matrix reordering forms a nested bordered block diagonal form. An example three-level tree and the resulting reordered matrix is shown in Figure 2.8.

While it can be shown that the overall border size will always be identical to the equivalent flat formulation, there are now multiple borders instead of a single large border. In essence, the border has been partitioned, and thus can be solved in parallel to a degree permitted by the hierarchy tree. In the given example, border elements 1 and 2 can be solved independently and in parallel after their respective dependencies have been solved. Border element 0 still must be solved after the rest of the system has been solved.

Despite the parallelism that is gained from this, it will still suffer from the problem of a growing border size. To get the most parallelism from this method, the tree must be as broad as possible. This can result in a top-level border that is too large, negating

the improvements obtained by the hierarchical formulation. Therefore, this method can yield speedup improvements, but its scalability to systems containing a large number of processors is limited.

2.3.3 Survey of Current Developments

Domain decomposition techniques have been applied for quite some time in the field of circuit simulation. In [7], the authors have given an overview of the methods developed for parallel circuit simulation up until the end of the 1980's, including both direct and iterative methods. Direct methods are generally more trusted as they are viewed to give an 'exact' solution. Domain decomposition techniques fall into the category of direct methods.

The subsequent developments in parallel circuit simulation techniques started with a core of domain decomposition with other improvements made to try to reduce the drawbacks that were discussed above. Some work [5] focused on the problem of partitioning, while others [24] proposed techniques that introduced elements of iterative solvers. Others [6] developed techniques to exploit circuit latency [1] or use independent step-size control for the subcircuits. Large projects have been undertaken that attempt to integrate all of these methods to produce a more scalable simulation method [9].

While much work has been done to advance the performance of the domain decomposition techniques for resolving a partitioned circuit, all of these techniques still use the same formulation as described above and therefore are victim to the same scalability problems described. Owing to this, yet more recent research has been performed to handle the scalability problem without using domain decomposition. One approach, proposed for example by the NICSLU solver [25] and its GPU implementation [26], suggests a parallel matrix solver that does not use the bordered block diagonal formulation or circuit

partitioning. The drawback, however, is that it was shown not to apply to all types of circuit matrix structures.

Others try create parallelism in the time domain, such as in the WavePipe method [27], however this can result in convergence difficulties as it is not using a purely implicit integration scheme.

Based on the observation that most of the proposed methods, the MAPS method was proposed in both a flat partitioning [28], and in a hierarchical version [29]. This method can exploit different solution algorithms for different parts of the circuit. Combining multiple algorithms in this way is in the realm of “Fast SPICE” simulation, where convergence can be even more precarious [6].

Owing to these results, finding scalability in a “true SPICE” simulation without sacrificing accuracy or convergence characteristics remains an interesting problem whose solution would be valuable. It is this problem that motivates this work.

In a parallel system, it is not only the algorithm design that can affect performance or scalability, but also the suitability of the implementation to the targeted platform. In the next section, some major platform architectures are considered as well as their impacts on algorithm implementations and the measurement of performance.

2.4 System Architecture Concerns

In this section, two popular multi-processor memory architectures are briefly reviewed. Also, the concept of *CPU work* is briefly discussed, and the impact of the differing memory architectures is discussed.

Figure 2.9: Conceptual Uniform Memory Access (UMA) system.

2.4.1 Memory Architecture

The first category of PC memory architectures is known as Uniform Memory Access (UMA) [30] and is commonly associated with earlier Intel Xeon multi-processor systems. A conceptual representation of this architecture is illustrated in Figure 2.9. In this system, each processor has equal access to the single block of system memory. In addition, any communication between processes running on separate physical processors takes place over the same bus as memory accesses.

The second category of architectures is known as Cache-Coherent Non-Uniform Memory Access (ccNUMA) [30] and is associated with AMD Opteron and newer Intel Xeon

Nehalem multi-processor systems. The concept behind this architecture is illustrated in Figure 2.10. In this system, each physical processor has its own local part of system memory, known as a node. Communication between the physical processors occurs over a separate bus. In this architecture, the operating system is responsible for ensuring that the memory allocated by programs is located in the optimal memory node. For the best performance, individual threads of the program should be bound to specific processors to ensure that their memory is always kept in the local node. When this is done, the memory bandwidth is effectively doubled compared to the UMA system for the same memory technology.

Figure 2.10: Conceptual Cache-Coherent Non-Uniform Memory Access (ccNUMA) system.

2.4.2 Implications for Algorithm Cost

When dealing with multiple processors, measuring CPU cost directly in terms of physical time with seconds is not a proper approach. Instead, the cost is measured with a unit of work called CPU-seconds. This is a unit analogous to man-hours, the work done by a person or a group of people. In some contexts, this is also referred to as the *processor time product*, a direct reflection of the units [23]. For example, if 8 CPUs are performing computations simultaneously over 2 seconds, this results in 16 CPU-seconds of work.

The *CPU work* includes both the time spent performing calculations and time spent waiting for data. This is analogous to the effort done by a taxi driver while actually driving on the road and while waiting on a traffic signal. For each task, it would be considered that he is working.

Since the amount of work can change with the number of processors used, measuring it can show the effect of memory bandwidth. Assuming the amount of computation is constant, any changes in *CPU work* will reflect a change in the amount of time spent waiting for data. Correspondingly, if the *CPU work* increases, a lack of memory bandwidth could be causing the execution to delay. On the other hand, if the *CPU work* decreases, additional memory bandwidth could be available causing a reduction in waiting time. It should be noted that because this is a sum of the work done by the individual CPUs, an increase in work does not imply an increase of run time.

General techniques to reduce the memory access cost include ensuring that workspaces used by threads are allocated on the memory closest to the associated processor. In addition, the data corresponding to various subcircuit can be distributed among the memories in the system. When the scheduler is faced with multiple choices of tasks to perform, it can preferentially choose a task whose data is located closest to that processor. These

types of optimizations can greatly increase the complexity of a task scheduler, but large performance improvements can be seen in some cases.

2.5 Summary

In this chapter, background into the field of standard and parallel circuit simulation was provided. First, the main techniques behind traditional circuit simulation were presented. Next, an overview of circuit partitioning was provided. The main goal of partitioning is to divide a circuit into multiple subcircuits that are weakly connected to ease the burden on a parallel solution algorithm. In the following section, a review of the most popular solution technique, domain decomposition, was given. Finally, in the last section, an overview of system architecture considerations for parallel algorithm development was given.

In the next chapter, a new technique for performing the time-domain simulation of nonlinear circuits without using domain decomposition will be presented. This technique will form the basis of the new method to avoid the disadvantages that were discussed in this section.

Chapter 3

Simulation of General Circuits with Node Tearing

In the previous chapter, the domain decomposition method for solving a partitioned circuit was presented. Various drawbacks inherent to this approach when applied to a parallel algorithm were discussed. In order to develop a new algorithm to address these deficiencies, a new method for solving a partitioned circuit is proposed. This method is based on circuit analysis techniques, and results in a different formulation. The core of this algorithm, presented in the following sections, is based on the novel combination of two existing techniques. The first is the frequency-domain technique of node tearing. A second technique is exploited to perform a transient simulation of a nonlinear circuit: the companion form of circuits.

3.1 Node Tearing

In this section, another method of solving a circuit's state using a method known as *node tearing* [10, 11] is presented. This method, as presented, handles only the solution of linear circuits in the frequency domain. Any linear circuit can be described in the frequency

domain using the Modified Nodal Analysis (MNA) formulation [1] given by

$$(\mathbf{G} + j\omega\mathbf{C})\mathbf{x}(\omega) = \mathbf{Y}(\omega)\mathbf{x}(\omega) = \mathbf{b}(\omega), \quad (3.1)$$

where $\mathbf{x}(\omega) \in \mathbb{C}^n$ is the vector of system states, containing all the node voltage and necessary branch currents, n is the number of system states, $\mathbf{G} \in \mathbb{R}^{n \times n}$ is the modified nodal conductance matrix, $\mathbf{C} \in \mathbb{R}^{n \times n}$ is the modified nodal capacitance matrix, $\mathbf{Y} \in \mathbb{C}^{n \times n}$ is the combined matrix of $\mathbf{G}$ and $\mathbf{C}$, and $\mathbf{b}(\omega) \in \mathbb{C}^n$ represents all of the independent sources of the circuit.

3.1.1 Partitioning via Node Tearing

Consider a circuit which has been divided into three parts as shown in Figure 3.1 using the techniques of Section 2.2.1. The node tearing technique can be applied to separate these parts into isolated circuits. The major difference between this method and domain decomposition can be seen by comparing Figures 2.5 and 3.1. In the case of node tearing, only nodes are found among the partitions and at their interfaces. Due to this, it is said that the shared nodes are torn among the partitions. In branch tearing, there is always a circuit element at the partition interfaces.

The conceptual derivation of this process is illustrated in Figure 3.2. The first step is to measure the current through each of the nodes that link the partitions, as shown in Figure 3.2(a). In each of these links, a current source of a value equal to the measured currents is inserted, as done in Figure 3.2(b). This has the effect of splitting (or tearing) the node into two, without changing the circuit's response. Finally, the current source itself is split into two, as in Figure 3.2(c), allowing the circuits to be completely isolated.

Figure 3.1: Partitioned circuit.

In this example, the result is four circuits that can be solved independently. The MNA equations for each subcircuit are

$$\begin{aligned}
 Y_1 x_1 &= \hat{b}_1, \\
 Y_2 x_2 &= \hat{b}_2, \\
 Y_3 x_3 &= \hat{b}_3, \quad \text{and} \\
 Y_4 x_4 &= \hat{b}_4,
 \end{aligned} \tag{3.2}$$

where Y_i is the MNA matrix for the i^{th} partition, each x_i corresponds to the MNA variables found in that partition, and $\hat{b}_i$ is formed from combination of the independent sources found in that partition (b_i) and the current sources connected to the ports.

Next, it is required to determine the value of these current sources without requiring the solution of the entire circuit. Consider first an example with two partitions and a single link between them, as shown in Figure 3.3. If this link is between node k in partition 1

Figure 3.2: Illustration of partitioning with node tearing.
 (a) Original partitioned system, with link currents indicated. (b) Partitioned system, with equivalent current sources inserted. (c) Partitioned system with split current sources and isolated partitions.

Figure 3.3: Example for tearing analysis.

and node l in partition 2, the current along this link can be found by taking the Thévenin equivalent of both circuits with respect to these ports, using the following procedure:

1. Construct a binary selector column vector ξ_1 where the k^{th} row is 1 and the rest are zero, with a number of rows matching Y_1 , the MNA matrix for partition 1.
2. Construct a second binary selector column vector ξ_2 where the l^{th} row is -1 and the rest are zero, with a number of rows matching Y_2 , the MNA matrix for partition 2.
3. Find the open-circuit port voltages v_k and v_l (with independent sources enabled) with

$$Y_1 \hat{x}_1 = b_1, \quad (3.3)$$

$$Y_2 \hat{x}_2 = b_2, \quad (3.4)$$

$$v_k = \xi_1^T \hat{x}_1, \quad \text{and} \quad (3.5)$$

$$v_l = -\xi_2^T \hat{x}_2, \quad (3.6)$$

as shown in Figure 3.4, and where $\hat{x}_i$ is the vector of MNA variables for the i^{th} partition with all of the links between the partitions removed.

4. Calculate the Thévenin equivalent impedance for each port by connecting a unit

Figure 3.4: Open-circuit voltage measurement.

Figure 3.5: Thévenin equivalent impedance measurement.

current source to it, while deactivating the circuits' independent sources, using:

$$\mathbf{Y}_1 \mathbf{x}_{z,1} = \boldsymbol{\xi}_1 \quad (3.7)$$

$$\mathbf{Y}_2 \mathbf{x}_{z,2} = \boldsymbol{\xi}_2 \quad (3.8)$$

$$z_{th,1} = \boldsymbol{\xi}_1^T \mathbf{x}_{z,1} \quad (3.9)$$

$$z_{th,2} = \boldsymbol{\xi}_2^T \mathbf{x}_{z,2} \quad (3.10)$$

as shown in Figure 3.5.

5. The two Thévenin equivalent circuits can be connected with a short circuit, as shown in Figure 3.6. The current from partition 2 to partition 1 is given by:

Figure 3.6: Link current measurement.

$$i_{k_1} = \frac{v_l - v_k}{z_{th,2} + z_{th,1}} \quad (3.11)$$

$$= \frac{-\xi_2^T Y_2^{-1} \mathbf{b}_2 - \xi_1^T Y_1^{-1} \mathbf{b}_1}{\xi_2^T Y_2^{-1} \xi_2 + \xi_1^T Y_1^{-1} \xi_1} \quad (3.12)$$

$$i_{k_1} = -\frac{\sum_{i=1}^2 \xi_i^T Y_i^{-1} \mathbf{b}_i}{\sum_{i=1}^2 \xi_i^T Y_i^{-1} \xi_i} \quad (3.13)$$

Therefore, using the formulation given in (3.2), this partitioned system can be solved using

$$\begin{aligned} Y_1 \mathbf{x}_1 &= \hat{\mathbf{b}}_1 = \mathbf{b}_1 + i_{k_1} \xi_1, \quad \text{and} \\ Y_2 \mathbf{x}_2 &= \hat{\mathbf{b}}_2 = \mathbf{b}_2 + i_{k_1} \xi_2. \end{aligned} \quad (3.14)$$

3.1.2 Partitions with Multiple Links

It can be shown that the above derivation can be extended to the case of a circuit divided into p partitions with l links. The following differences will be seen:

- For each partition, the ξ vector will be replaced with a matrix Ξ .

- The dimension of Ξ_i will be $n_i \times l$, where n_i is the dimension of the MNA matrix Y_i .
- Each column of each Ξ_i corresponds to one of the links.
- If the j^{th} link is between partitions a and b , column j of Ξ_a will contain a 1 in the corresponding row, column j of Ξ_b will contain a -1 in the corresponding row, and every other entry in column j for all Ξ matrices will be zero.
- The Thévenin impedance is replaced by an impedance matrix, defined as

$$\mathbf{Z} = - \sum_{j=1}^p \Xi_j^T Y_j^{-1} \Xi_j \quad (3.15)$$

- The currents along each link become a vector, $\boldsymbol{\gamma} \in \mathbb{C}^{l \times 1}$, given by

$$\boldsymbol{\gamma} = -\mathbf{Z}^{-1} \sum_{j=1}^p \Xi_j^T Y_j^{-1} \mathbf{b}_j \quad (3.16)$$

- The full system for the i^{th} partition is thus found with

$$Y_i \mathbf{x}_i = \mathbf{b}_i - \Xi_i \boldsymbol{\gamma} \quad (3.17)$$

The node tearing technique presented in this section was introduced as a frequency-domain technique for linear circuits. However, the most important problem for circuit simulators is the simulation of nonlinear circuits in the time domain. To accomplish this, in the following sections, the *companion form* of a circuit is introduced, which allows for any time-domain simulation task to be cast into a linear DC simulation (or, equivalently, a frequency-domain simulation for $f = 0$). Once a circuit is in this form, linear frequency-domain techniques (such as node tearing techniques) can be applied.

3.2 Companion Form of Circuits

The companion form of circuit elements was introduced as an element-by-element application of time-domain analysis techniques [1]. Using this form, any circuit analysis problem can be converted to a resistive network. First, the linear time-domain analysis problem will be considered.

3.2.1 Companion Form for Time-Domain Analysis

The linear circuit described by (3.1) can be represented in the time domain as [1]

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\dot{\mathbf{x}}(t) = \mathbf{b}(t), \quad (3.18)$$

where $\mathbf{G}$ and $\mathbf{C}$ remain the same from (3.1), $\mathbf{x}(t)$ contains the states of the circuit as a function of time, $\dot{\mathbf{x}}(t)$ is the time derivative of these states, and $\mathbf{b}(t)$ specifies the voltage and current sources in the circuit.

The standard approach for finding the time-domain response of the circuit based on the above equation is to apply an integrating function to obtain a difference equation. For example, the well-known Backward Euler integrating function is given by

$$x(t_0 + h) = x(t_0) + h \left. \frac{dx}{dt} \right|_{t=t_0+h}, \quad (3.19)$$

where t_0 is the previous time point at which the state is already known, and h is the time step size. If this function is applied to (3.18), the corresponding difference equation is

$$\left[\mathbf{G} + \frac{\mathbf{C}}{h} \right] \mathbf{x}(t_0 + h) = \frac{\mathbf{C}}{h} \mathbf{x}(t_0) + \mathbf{b}(t_0 + h). \quad (3.20)$$

Figure 3.7: Companion model for a capacitor.

Instead of applying the integration formula directly to the MNA equation, with the companion form, it is applied to each circuit element individually. For example, for a capacitor, whose behaviour is given by

$$i_C(t) = C\dot{v}_C(t), \quad (3.21)$$

the application of the Backward Euler integration function gives the following difference equation:

$$i_C(t_0 + h) = \frac{C}{h} (v_C(t_0 + h) - v_C(t_0)). \quad (3.22)$$

This equation can be represented with an equivalent circuit consisting of a parallel combination of a resistor and independent DC current source as shown in Figure 3.7. A similar derivation can be performed for inductors, resulting in the companion circuit of Figure 3.8.

It is apparent that any linear circuit can be transformed into a corresponding companion circuit that has all of its energy storage elements replaced. In other words, the companion

Figure 3.8: Companion model for an inductor.

circuit consists solely of resistors, and independent and dependent sources. The corresponding companion circuit can be solved to obtain the time-domain response of the original circuit at the time point under consideration.

It is also observed that since the companion circuit is also linear, it can be treated as a frequency-domain analysis. Therefore, we can exploit frequency-domain techniques, such as node tearing, to perform a linear time-domain simulation when the companion form is used.

Most practical circuits, however, are not purely linear. To handle the case where nonlinear elements are present, a companion form applicable to a nonlinear simulation is presented in the next section.

3.2.2 Companion Form for Nonlinear Time-Domain Circuits

A general nonlinear circuit can have its behaviour described by the MNA equation [1]

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\dot{\mathbf{x}}(t) + \mathbf{f}(\mathbf{x}(t)) = \mathbf{b}(t), \quad (3.23)$$

which is the same as (3.18) but with an added term on the left-hand side, $f(\mathbf{x}(t))$ that includes the effect of nonlinear elements.

In the standard solution procedure, an integration formula is applied to convert this into a nonlinear difference equation, and Newton-Raphson iterations are applied to the difference equation to find a solution for each time point.

Using the same technique as in the previous section, a companion circuit can be found for a nonlinear circuit. Instead of applying Newton-Raphson iterations to the entire circuit, the effect on a single element is considered. For example, a nonlinear resistor has a behaviour defined by

$$i_R(t) = f(v_R(t)). \quad (3.24)$$

The Newton-Raphson method can be applied directly to this equation, giving the following set of equations:

$$v_R^{(0)}(t_0 + h) = v_R(t_0) \quad (3.25)$$

$$\Phi(\hat{v}_R, t_0) = f(\hat{v}_R) - i_R(t_0 + h) \quad (3.26)$$

$$\Delta v_R^{(i+1)}(t_0 + h) = - \left(\frac{\partial \Phi(\hat{v}_R, t_0)}{\partial \hat{v}_R} \bigg|_{\hat{v}_R = v_R^{(i)}(t_0 + h)} \right)^{-1} \cdot \Phi(v_R^{(i)}(t_0 + h), t_0) \quad (3.27)$$

$$v_R^{(i+1)}(t_0 + h) = v_R^{(i)}(t_0 + h) + \Delta v_R^{(i+1)}(t_0 + h), \quad (3.28)$$

where $v_R^{(0)}(t_0)$ is the initial guess of the voltage across the resistor at time t_0 , and $v_R^{(i)}(t_0)$ is the corresponding value after the i^{th} iteration.

This can be represented with the equivalent circuit shown in Figure 3.9. The solution of this equivalent circuit gives the same result as performing a single iteration of the Newton-Raphson method.

Figure 3.9: Companion model for a nonlinear resistor.

The methods described above can be applied to all types of circuit elements, for linear or nonlinear time-domain simulations, and for any integrating function. The types of elements handled include, but are not limited to, inductors, transformers, transmission lines, nonlinear capacitors, diodes, and transistors.

It is to be noted that the companion circuit, for any given time point, has the following characteristics:

1. All of the energy-storage elements in the circuit are eliminated, with the resulting companion circuit consisting solely of resistors and sources, and
2. The time-domain response of the circuit at a particular time point is obtained by simply solving the corresponding companion circuit using standard linear DC solution techniques, while exploiting some of the merits of frequency-domain techniques (such as node tearing, as outlined in Section 3.1).

3.3 Development of the General Partitioned Circuit

Simulation Method

The development of the proposed algorithm begins with the consideration of the equations (3.2) to (3.17). The solution process of these equations is examined in stages in order to minimize calculations and to obtain maximum parallelism. The steps are outlined below.

Step a) The first step would be to solve the original MNA system for each subcircuit's companion form with its independent sources enabled, exploiting standard sparse LU factorization techniques:

$$\mathbf{Y}_j \hat{\mathbf{x}}_j = \mathbf{b}_j. \quad (3.29)$$

Step b) Next, the first stage of (3.15) is calculated for each partition with

$$\mathbf{Y}_j \mathbf{\Gamma}_j = \mathbf{\Xi}_j. \quad (3.30)$$

This operation will re-use the LU factorization computed to solve (3.29), so will simply consist of a number of forward-backward substitutions involving binary vectors. Note that $\mathbf{\Xi}_j$ will consist of a number of zero columns and also a number of columns that differ only by sign. The forward-backward substitutions clearly do not have to be performed for these cases. Let $\hat{\mathbf{\Xi}}_j$ be the resulting matrix if the extra columns of $\mathbf{\Xi}$ are removed. Next, define a permutation matrix, $\mathbf{P}_j$ is defined, such that

$$\mathbf{\Xi}_j = \mathbf{P}_j \hat{\mathbf{\Xi}}_j. \quad (3.31)$$

The actual system that needs to be solved for this step is thus

$$\mathbf{Y}_j \hat{\mathbf{\Gamma}}_j = \hat{\mathbf{\Xi}}_j, \quad (3.32)$$

where

$$\mathbf{\Gamma}_j = \mathbf{P}_j \hat{\mathbf{\Gamma}}_j. \quad (3.33)$$

Step c) Once these have been found, the $\mathbf{Z}$ matrix for (3.15) is constructed, with

$$\mathbf{Z} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \mathbf{P}_j \hat{\mathbf{\Gamma}}_j. \quad (3.34)$$

It is noted that due to the structure of the $\mathbf{\Xi}$ and $\mathbf{P}$ matrices, this operation can be implemented as a permutation. Since each row of $\mathbf{\Xi}^T \mathbf{P}$ contains at most one non-zero, ± 1 , the corresponding product simply selects and permutes the rows of $\hat{\mathbf{\Gamma}}$, possibly changing the sign of the entries. No multiplication operations are involved, however.

Step d) Next, the $\boldsymbol{\gamma}$ vector in (3.16) is found, using this rearranged form:

$$\mathbf{Z} \boldsymbol{\gamma} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \hat{\mathbf{x}}_j \quad (3.35)$$

Once again, the right-hand-side vector is constructed using permutations.

Step e) Finally, the solution vectors for each subcircuit are found using

$$\mathbf{Y}_j \Delta \mathbf{x}_j = \mathbf{\Xi}_j \boldsymbol{\gamma} \quad (3.36)$$

$$\mathbf{x}_j = \hat{\mathbf{x}}_j - \Delta \mathbf{x}_j \quad (3.37)$$

The numerical advantage of the ability to replace general sparse matrix-matrix multiplications with permutation operations in steps c and d of the proposed algorithm is demonstrated with the following illustrative example.

Illustrative Example 1. Consider the evaluation of the following expression:

$$\mathbf{AB}, \quad (3.38)$$

where $\mathbf{A}$ is a square permutation matrix of dimension n , and $\mathbf{B}$ is a general sparse matrix in $\mathbb{R}^{n \times n}$ with, on average, 4 nonzero elements per row. The operation was performed for different values of n with the following two implementations:

- General sparse matrix-matrix multiplication, and
- Direct permutation,

on two platforms of differing generations:

1. An AMD Opteron 2344HE system (1.7 GHz), and
2. An AMD Opteron 6172 system (2.1 GHz),

and the results are shown in Table 3.1 *. To ensure accurate measurements, exclusive access to the machines was ensured. As can be seen, the permutation implementation yields significant performance benefits, especially as the size of the matrix increases.

*The results of this experiment have been provided by Nick Soveiko, of Carleton University.

Table 3.1: Cost comparison of matrix-matrix multiplication

Matrix Size	AMD Opteron 2344HE			AMD Opteron 6172		
	Multiply	Permutation	Speedup	Multiply	Permutation	Speedup
100	38.6 μ s	18.1 μ s	2.13 $\times$	29.5 μ s	14.9 μ s	1.98 $\times$
300	97.0 μ s	53.3 μ s	1.82 $\times$	76.9 μ s	42.4 μ s	1.81 $\times$
1,000	313 μ s	176 μ s	1.78 $\times$	250 μ s	141 μ s	1.77 $\times$
3,000	1.11 ms	547 μ s	2.03 $\times$	894 μ s	430 μ s	2.08 $\times$
10,000	6.15 ms	2.36 ms	2.61 $\times$	4.20 ms	1.52 ms	2.76 $\times$
30,000	28.1 ms	7.57 ms	3.71 $\times$	18.5 ms	5.47 ms	3.38 $\times$
100,000	184 ms	28.8 ms	6.39 $\times$	95.6 ms	19.6 ms	4.88 $\times$
300,000	1.00 s	117 ms	8.55 $\times$	622 ms	77.4 ms	8.03 $\times$
1,000,000	4.42 s	512 ms	8.63 $\times$	3.17 s	360 ms	8.81 $\times$

The proposed technique can also be conceptually formulated as a bordered-block-diagonal matrix. The overall matrix will have the following form:

$$\begin{bmatrix} Y_1 & & & \Xi_1 \\ & Y_2 & & \Xi_2 \\ & & \ddots & \vdots \\ & & & Y_p & \Xi_p \\ \Xi_1^T & \Xi_2^T & \dots & \Xi_p^T & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \\ \gamma \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \\ 0 \end{bmatrix}, \quad (3.39)$$

and the above procedure can be used to solve this system.

3.4 Summary

In this chapter, a new technique for performing the time-domain simulation of nonlinear circuits has been developed. This technique is based on the existing frequency-domain technique of node tearing. To apply it to the problem of transient analysis, the companion form of circuits has been exploited, which allows any simulation problem to be represented as an equivalent frequency-domain problem.

In the next chapter, this algorithm will be examined for parallelism, and an initial parallel implementation will be performed. Its scalability compared to domain decomposition will be analysed, both analytically and through the consideration of circuit examples.

Chapter 4

Parallel Circuit Simulation with Node Tearing

In this chapter, an initial implementation of a parallel circuit simulation using node tearing is presented. Its performance is compared to traditional branch-tearing-based domain decomposition through simulation examples, and the results are justified through a comparative analysis of the node tearing method as compared to branch tearing.

4.1 Parallel Implementation of Node Tearing

In this section, a parallel algorithm is proposed to perform a complete Newton-Raphson iteration using the node tearing process described in Section 3.3. To do this, the Newton-Raphson iteration algorithm described in Section 3.3 will be broken into a set of tasks that must be completed in sequence. In each of the tasks, any available parallelism is identified to be exploited in the implementation. The description of each of these tasks follows.

4.1.1 Task 1: System Formulation

At the start of a Newton-Raphson (NR) iteration, the first task to be performed is to formulate the linear equations to be solved, in the form of (2.4). This involves constructing the system MNA Jacobian matrix, and to calculate the right-hand-side vector. As reviewed in Section 2.1, the contents of both are determined by evaluating the characteristics for each device model given the current state of the circuit's node voltages and branch currents.

From (2.4), it is seen that the evaluation of each device is done using the state of circuit at the previous iteration as an input. Therefore, the calculation of each device is completely independent and each can be performed in parallel. This provides for a large amount of potential parallelism, but it is fairly fine-grained. Therefore, if it were done this way, there might be too much overhead from the thread management. Because of this, it is useful to group the circuit devices into a certain number of blocks that will each be evaluated in parallel.

After some tuning, it was found that good results were obtained when the devices in each partition were distributed among a number of blocks equal to twice the number of processors in the system. To prevent the overhead from becoming too significant, never are there fewer than ten devices in a block. This stage of device evaluation will also be referred to as *Task 1a*.

To load the values into the actual Jacobian matrix and right-hand-side vector, different blocks are used. As multiple devices can touch the same matrix or vector elements, normally, locks would need to be used to protect these. Instead of this, the matrices and vectors are loaded as a separate task for each partition. This loading process will also be called *Task 1b*.

4.1.2 Task 2: Decoupled Partition Solution

Once all of the values of all the system Jacobian matrices and right-hand-side vectors for all partitions are known, the system can proceed to be solved using the method proposed in Section 3.3. The first step in this solution process is to solve each partitioned sub-circuit independently with its own independent sources activated but with all of its interconnections open-circuited. This consists of performing *Step-a* for each partition, which consists of solving (3.29), for each partition. This equation is repeated here for clarity, for the j^{th} partition:

$$\mathbf{Y}_j \hat{\mathbf{x}}_j = \mathbf{b}_j. \quad (4.1)$$

As this is a separate system of equations for each partition, this task can be performed in parallel for each partition.

4.1.3 Task 3: Determination of Partition Coupling Influence

Once Task 2 completes for a given partition, the next step (*Step-b*) from Section 3.3 is to solve for the influence of each of that partition's links on the partition itself. Recall from the description in Section 3.1 that in this step, we are connecting a unit current source to each of the partitions links, in turn, while deactivating its independent sources. This implies we have a system of equations with multiple right-hand sides, the number of which is equal to the number of connected links. The system to be solved, from (3.32) for partition j , is:

$$\mathbf{Y}_j \hat{\mathbf{\Gamma}}_j = \hat{\mathbf{\Xi}}_j. \quad (4.2)$$

As the system matrix is the same as that used in Task 2, the LU factors are simply conserved from the previous factorization. This task can be performed in parallel once

again for each partition. We can find additional parallelism in the fact that for a partition with l links, we have l right-hand sides, so l independent forward-backward substitutions are to be done. We can divide this work into blocks of reasonable size and process each of these, once again, in parallel.

4.1.4 Task 4: Construction of Interconnection Matrix

Once Task 3 has been completed for all partitions and all links, the overall interconnection matrix, $\mathbf{Z}$, for the partitioned system can be constructed. As discussed in Section 3.1.2, this is equivalent to the z -parameters of the interconnection network, consisting of all the links among the partitions. This is found using selection and permutation operations of equation (3.34), using the $\hat{\mathbf{\Gamma}}$ matrices from Task 3:

$$\mathbf{Z} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \mathbf{P}_j \hat{\mathbf{\Gamma}}_j. \quad (4.3)$$

This corresponds to *Step-c* from Section 3.3. In addition, the right-hand-side vector for the interconnection solution can also be constructed at the same time:

$$\boldsymbol{\beta} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \hat{\mathbf{x}}_j. \quad (4.4)$$

4.1.5 Task 5: Initial Solution of Partition Interconnection

Once Task 4 is complete, the next task is to solve for the actual link currents that will be present in the companion circuit for the given Newton-Raphson iteration. This is done using (3.35):

$$\mathbf{Z}\boldsymbol{\gamma} = \boldsymbol{\beta}, \quad (4.5)$$

where γ contains the link currents. To increase the task granularity for later discussions, in this task the ‘forward’ pass is performed, consisting of the matrix factorization and forward substitution.

4.1.6 Task 6: Final Solution of Partition Interconnection

In this task, the solution of γ is completed, with the backward substitution being performed. The set of Tasks 5 and 6 corresponds to *Step-d* in Section 3.3.

4.1.7 Task 7: Update of Full Partition Solution

The final task, corresponding to *Step-e*, is to update the solution of each partition of the circuit taking into account the effects of the coupling among the partitions. This is done with (3.36) and (3.37), for each of the i partitions:

$$Y_i \Delta x_i = \Xi_i \gamma \quad (4.6)$$

$$x_i = \hat{x}_i - \Delta x_i \quad (4.7)$$

As these equations are independent for each of the partitions, their individual updates can be performed in parallel.

4.1.8 Summary

In this section, an initial discovery of evident parallelism in the proposed algorithm was performed. The algorithm was examined at level of a single Newton-Raphson iteration, and it was broken up into a series of tasks. Within each of these tasks, possible parallelism was found.

In the following sections, the node tearing formulation is compared in detail with the branch tearing formulation, to give insight into the expected scalability characteristics. Afterwards, a series of benchmark examples is considered to compare the actual performance of the proposed algorithm against the existing branch-tearing-based domain decomposition approach.

4.2 Comparison with Domain Decomposition

In the following analysis, it is proven that the node tearing approach provides superior scalability characteristics compared to branch tearing. In a parallel implementation, there must be at least as many partitions as CPUs in order to fully exploit the system. It is shown the branch tearing suffers from an overhead of $O(l^2)$, where l is the number of links, which increases as the number of partitions increases. On the other hand, a proof is given to show the overhead in node tearing to be only $O(l)$, which allows for greater speedup in a parallel implementation. It is to be noted that the number of links is defined in both cases to be the same number: the number of links needed for the partitioning resolved using node tearing. The size of the border in branch tearing will usually be less than the number of links, but it will be shown that the overhead still scales with the number of links.

For the purposes of illustration, consider a system that has been partitioned into two parts that are connected with one link. Let the original system contain five nodes, and each

partition contain three. An example formulation with node tearing can be

$$\left[\begin{array}{ccc|ccc|c} a_{111} & a_{112} & a_{113} & & & & 1 \\ a_{121} & a_{122} & a_{123} & & & & 0 \\ a_{131} & a_{132} & a_{133} & & & & 0 \\ \hline & & & \mathbf{0} & & & \\ & & & & & & \\ & & & & & & \\ \hline & & & a_{211} & a_{212} & a_{213} & -1 \\ & & & a_{221} & a_{222} & a_{223} & 0 \\ & & & a_{231} & a_{232} & a_{233} & 0 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 & 0 \end{array} \right] \left[\begin{array}{c} v_{11} \\ v_{12} \\ v_{13} \\ v_{21} \\ v_{22} \\ v_{23} \\ x_1 \end{array} \right] = \left[\begin{array}{ccc|ccc|c} b_{11} & b_{12} & b_{13} & b_{21} & b_{22} & b_{23} & 0 \end{array} \right]^T. \quad (4.8)$$

An example such as this can be transformed into a branch tearing formulation using the following procedure:

1. Create a transformation matrix, $\mathbf{T}$ of dimension $m \times n$. m is the number of variables in the node tearing formulation, and n is the number of nodes. When l is the number of links in the node tearing formulation, $n = m - 2l$.
2. The matrix values are constructed row-by-row. First, all of the nodes that are not connected by a link to another node are processed. Each of these nodes correspond to one row in the transformation matrix. A one is placed on the assigned row in the column corresponding to this node in the original system. This constructs the blocks of the bordered-block-diagonal form.
3. The next part is to create the border. First, the links are arranged into a set of 'link chains'. Each link chain consists of a set of links that connect a set of nodes together. Each of these link chains correspond to one node in the original circuit, that has been torn a number of times equal to the number of links involved. In the branch tearing

formulation, this node will be represented by a single variable. Each link chain is assigned one of the remaining rows in the transformation matrix. For each, a 1 is inserted into each column corresponding to a connected node.

4. For a node tearing system given by:

$$\mathbf{A}^{\{N\}} \mathbf{x}^{\{N\}} = \mathbf{b}^{\{N\}}. \quad (4.9)$$

If the transformation matrix is introduced as follows:

$$\mathbf{T} \mathbf{A}^{\{N\}} \mathbf{T}^T (\mathbf{T} \mathbf{T}^T)^{-1} \mathbf{T} \mathbf{x}^{\{N\}} = \mathbf{T} \mathbf{b}^{\{N\}}, \quad (4.10)$$

the corresponding branch tearing system is given by:

$$\mathbf{A}^{\{B\}} \mathbf{x}^{\{B\}} = \mathbf{b}^{\{B\}}, \quad (4.11)$$

where $\mathbf{A}^{\{B\}} = \mathbf{T} \mathbf{A}^{\{N\}} \mathbf{T}^T$, $\mathbf{x}^{\{B\}} = (\mathbf{T} \mathbf{T}^T)^{-1} \mathbf{T} \mathbf{x}^{\{N\}}$, and $\mathbf{b}^{\{B\}} = \mathbf{T} \mathbf{b}^{\{N\}}$.

For the example of (4.8), the transformation matrix, $\mathbf{T}$, is developed with the above procedure to be:

$$\mathbf{T} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (4.12)$$

The resulting modified formulation of (4.8) can be written as

$$\left[\begin{array}{cc|c|c} a_{122} & a_{123} & \mathbf{0} & a_{121} \\ a_{132} & a_{133} & & a_{131} \\ \hline \mathbf{0} & a_{222} & a_{223} & a_{221} \\ & a_{232} & a_{233} & a_{231} \\ \hline a_{112} & a_{113} & a_{212} & a_{213} & a_{111} + a_{211} \end{array} \right] \left[\begin{array}{c} v_{12} \\ v_{13} \\ v_{22} \\ v_{23} \\ \frac{v_{11} + v_{21}}{2} = v_1 \end{array} \right] = \left[\begin{array}{c|c|c} b_{12} & b_{13} & b_{22} & b_{23} & b_{11} + b_{21} \end{array} \right]^T. \quad (4.13)$$

It must be determined which of these two formulations is the most efficient for reasonably-sized (large) circuits. To do this, the solution procedure for both formulations is compared.

To perform this comparison, a general circuit is considered. Let p be the number of partitions into which the circuit is divided. In performing this partitioning, let l be the number of links created. To distinguish between the formulations, an $\{N\}$ superscript is used to indicate the node tearing formulation, and $\{B\}$ is used for branch tearing.

To begin, the node tearing formulation for the circuit with p partitions is considered. In matrix form, this is

$$\left[\begin{array}{cccc} A_1^{\{N\}} & & & \Xi_1 \\ & A_2^{\{N\}} & & \Xi_2 \\ & & \ddots & \vdots \\ & & & A_p^{\{N\}} & \Xi_p \\ \Xi_1^T & \Xi_2^T & \dots & \Xi_p^T & \mathbf{0} \end{array} \right] \left[\begin{array}{c} x_1^{\{N\}} \\ x_2^{\{N\}} \\ \vdots \\ x_p^{\{N\}} \\ \gamma \end{array} \right] = \left[\begin{array}{c} b_1^{\{N\}} \\ b_2^{\{N\}} \\ \vdots \\ b_p^{\{N\}} \\ \mathbf{0} \end{array} \right]. \quad (4.14)$$

In the process of partitioning, l links were created, so the dimension of the border is l . Note that each Ξ_i is a purely binary matrix where each column consists of, at most, one nonzero.

Figure 4.1: Partitioning where border size is identical between node and branch tearing.

Next, the equivalent formulation with branch tearing is found as

$$\begin{bmatrix} A_1^{\{B\}} & & & F_1 \\ & A_2^{\{B\}} & & F_2 \\ & & \ddots & \vdots \\ & & & A_p^{\{B\}} & F_p \\ E_1 & E_2 & \dots & E_p & A_0^{\{B\}} \end{bmatrix} \begin{bmatrix} x_1^{\{B\}} \\ x_2^{\{B\}} \\ \vdots \\ x_p^{\{B\}} \\ x_0^{\{B\}} \end{bmatrix} = \begin{bmatrix} b_1^{\{B\}} \\ b_2^{\{B\}} \\ \vdots \\ b_p^{\{B\}} \\ b_0^{\{B\}} \end{bmatrix}. \quad (4.15)$$

There are several changes which happen in the border. Not only are the blocks in the bottom row (E_i) no longer guaranteed to be the transposes of those in the right column (F_i), they will also no longer be a purely binary matrices as was the case for the Ξ .

In addition, the size of the border is likely to have changed in the conversion process. If the same node is torn between $m \in \mathbb{N}$, $1 \leq m \leq p$ partitions, this will result in $m - 1$ links, and consequently $m - 1$ border variables for node tearing. In the case of branch tearing, only the node variable itself will appear in the border.

In the case where each node is only shared between two partitions, the dimension of the border in branch tearing will thus be l . This is illustrated in Figure 4.1. Here, we see a case where there are four nodes that are shared among the partitions, and each node is

Figure 4.2: Partitioning where border size is greater in node tearing than branch tearing.

only connected to two partitions. The size of the border for branch tearing is equal to the number of shared nodes, four, and the size for node tearing is equal to the number of links, which is also four. Therefore, we can see that in this case the border size is the same for both methods.

The most favourable case for branch tearing is where each node is shared among all partitions. In this case, there will be $l(p - 1)^{-1}$ border variables. This is demonstrated in Figure 4.2. In this case we once again have four nodes shared among the partitions, but in this case each node is connected to each partition. The size of the border for branch tearing is just equal to the number of shared nodes, which is four. The size of the border for node tearing is equal to the number of links, which can be counted to be eight (for each shared node, there is a corresponding link between partition 1 and 2, and another from partition 2 to 3). Thus, for this example, $l = 8$, and $p = 3$, and the size of the border for branch tearing is confirmed to be $l(p - 1)^{-1} = 8(3 - 1)^{-1} = 4$.

In general cases, the number of border variables in branch tearing will be somewhere between these two extremes. To formalize this, a node sharing factor, $\beta_1 \in [0, 1]$, is defined. When $\beta_1 = 0$, each of the nodes is shared between two partitions only (the first case), and the size of the border will be l . At the other extreme, when $\beta_1 = 1$, each node will be

shared among all partitions (the second case), and the number of border variables will be $l(p-1)^{-1}$. Therefore, the size of the border for the branch tearing formulation can be expressed as $l(1+\beta_1(p-2))^{-1}$. By simple substitution, it can be shown that this expression simplifies to the two extreme cases when β_1 is set to either 0 or 1.

4.2.1 Comparison of Computational Steps

Next, the solution procedure for both node tearing and branch tearing is described as a sequence of five steps. For node tearing, these are the steps as defined in Section 3.3, with the equivalent operations shown for the case of branch tearing.

Step a) The first step is to solve the main right-hand-side vector for each partition. The equations for node tearing and branch tearing, are, respectively

$$\mathbf{A}_i^{\{N\}} \hat{\mathbf{x}}_i^{\{N\}} = \mathbf{b}_i^{\{N\}}, \quad \text{and} \quad (4.16)$$

$$\mathbf{A}_i^{\{B\}} \hat{\mathbf{x}}_i^{\{B\}} = \mathbf{b}_i^{\{B\}}. \quad (4.17)$$

Step b) The next step in the solution process is to solve the right border, using the following equations for node tearing and branch tearing, respectively,

$$\mathbf{A}_i^{\{N\}} \boldsymbol{\Gamma}_i^{\{N\}} = \boldsymbol{\Xi}_i, \quad \text{and} \quad (4.18)$$

$$\mathbf{A}_i^{\{B\}} \boldsymbol{\Gamma}_i^{\{B\}} = \mathbf{F}_i. \quad (4.19)$$

Step c) In this step, the solution process involves the formulation of the matrix used to resolve the effect of the links between the partitions. In domain decomposition, this is commonly referred to as the Schur complement matrix. The formulations for node tearing

and branch tearing, are, respectively

$$\mathbf{Z}^{[N]} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \mathbf{\Gamma}_j^{[N]}, \quad \text{and} \quad (4.20)$$

$$\mathbf{Y}^{[B]} = \mathbf{A}_0^{[B]} - \sum_{j=1}^p \mathbf{E}_j \mathbf{\Gamma}_j^{[B]}. \quad (4.21)$$

The associated RHS vector is then constructed with the following equations for node tearing and branch tearing

$$\boldsymbol{\beta}^{[N]} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \hat{\mathbf{x}}_j^{[N]}, \quad \text{and} \quad (4.22)$$

$$\boldsymbol{\beta}^{[B]} = \mathbf{b}_0^{[B]} - \sum_{j=1}^p \mathbf{E}_j \hat{\mathbf{x}}_j^{[B]}. \quad (4.23)$$

Step d) After this work is done, the border variables are then found using the following relations, again, for node tearing and branch tearing respectively

$$\mathbf{Z}^{[N]} \boldsymbol{\gamma} = \boldsymbol{\beta}^{[N]}, \quad \text{and} \quad (4.24)$$

$$\mathbf{Y}^{[B]} \mathbf{x}_0^{[B]} = \boldsymbol{\beta}^{[B]}. \quad (4.25)$$

Step e) The final step is to update the variables in each partition to reflect the impact of the global system. The equations for each formulation are

$$\mathbf{x}_i^{[N]} = \hat{\mathbf{x}}_i^{[N]} - \left(\mathbf{A}_i^{[N]} \right)^{-1} \mathbf{\Xi}_i \boldsymbol{\gamma}, \quad \text{and} \quad (4.26)$$

$$\mathbf{x}_i^{[B]} = \hat{\mathbf{x}}_i^{[B]} - \left(\mathbf{A}_i^{[B]} \right)^{-1} \mathbf{F}_i \mathbf{x}_0^{[B]}. \quad (4.27)$$

In the rest of this section, the details of the computational cost analysis of the important steps of the solution process is provided.

4.2.2 Comparison of CPU Cost

To derive the scalability characteristics, the two steps that contribute the most to the overhead for each method will be examined. For node tearing, the most overhead comes from *Step-a*, and for branch tearing it comes from *Step-c*. First, the overhead for *Step-a* is compared. For this purpose, the following definitions are made:

- Let the LU operation for an $n \times n$ sparse matrix be $O(n^{\alpha_1})$, $1 \leq \alpha_1 \leq 2$
- β_1 be the node sharing factor,
- p be the number of partitions,
- l be the number of links among the partitions,
- n be the number of variables in the overall system, and
- k_1 be a constant linking the complexity of the LU factorization with the actual CPU cost.

Lemma 1. *The extra overhead of **Step-a** with node tearing over branch tearing is $O(l)$, and is given by:*

$$k_1 \frac{n^{(\alpha_1-1)}}{p^{\alpha_1}} \alpha_1 l \left(1 + \frac{1}{1 + \beta_1(p-2)} \right). \quad (4.28)$$

Proof. The operations for both cases are given in (4.16) and (4.17) and are similar; however, the size of the matrices are not. In the case of node tearing, each link adds one more variable to the non-border region of the matrix due to the duplication of the node. If the number of

variables in the unpartitioned system is n , the total number of non-border variables would be $n + l$. The average number of variables per partition would then be $(n + l)p^{-1}$.

For branch tearing, each node that is shared among partitions is moved into the border, resulting in a slight reduction of the overall size of the non-border region of the matrix. Therefore, the total number of variables excluding the border will be $n - l(1 + \beta_1(p - 2))^{-1}$, and the average number of variables per partition will be $(n - l(1 + \beta_1(p - 2))^{-1})p^{-1}$.

These equations will be solved using LU factorization and forward/backward (FB) substitution. The complexity of LU factorization is $O(n^{\alpha_1})$, where $1 < \alpha_1 < 3$, and that of FB substitution is $O(n^{\alpha_2})$, with $1 < \alpha_2 < 2$. For matrices with sparsities commonly seen in circuit matrices and efficient LU solvers such as KLU [2], both α_1 and α_2 commonly approach 1.1.

Considering the cost of the LU factorization for the case of node tearing, it is seen to be

$$k_1 \left[(n + l)p^{-1} \right]^{\alpha_1}, \quad (4.29)$$

where k_1 is a constant reflecting the CPU cost of the operation. This expression can be rearranged as

$$k_1 (n + l)^{\alpha_1} p^{-\alpha_1} \quad (4.30)$$

$$= k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(1 + \frac{l}{n} \right)^{\alpha_1} \quad (4.31)$$

$$\approx k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(1 + \alpha_1 \frac{l}{n} \right), \quad (4.32)$$

for the practical case of $l \ll n$.

For branch tearing, the cost is

$$k_1 \left[(n - l(1 + \beta_1(p - 2))^{-1}) p^{-1} \right]^{\alpha_1}, \quad (4.33)$$

which is rearranged as

$$k_1 (n - l(1 + \beta_1(p - 2))^{-1})^{\alpha_1} p^{-\alpha_1} \quad (4.34)$$

$$= k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(1 - \frac{l}{n(1 + \beta_1(p - 2))} \right)^{\alpha_1} \quad (4.35)$$

$$\approx k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(1 - \alpha_1 \frac{l}{n(1 + \beta_1(p - 2))} \right), \quad (4.36)$$

when $l \ll n$.

Taking the difference between these costs, the effect of the somewhat larger block diagonal matrices in the case of node tearing can be examined.

$$k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(1 + \alpha_1 \frac{l}{n} \right) - k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(1 - \alpha_1 \frac{l}{n(1 + \beta_1(p - 2))} \right) \quad (4.37)$$

$$= k_1 \left(\frac{n}{p} \right)^{\alpha_1} \left(\alpha_1 \frac{l}{n} + \alpha_1 \frac{l}{n(1 + \beta_1(p - 2))} \right) \quad (4.38)$$

$$= k_1 \frac{n^{(\alpha_1 - 1)}}{p^{\alpha_1}} \alpha_1 l \left(1 + \frac{1}{1 + \beta_1(p - 2)} \right) \quad (4.39)$$

$$= O(l), \quad (4.40)$$

when $\alpha_1 < 2$, which is typically the case for practical circuits. $\square$

Therefore, the increase in cost of the LU operation for node tearing is linearly proportional to the number of links. It is noted that this increase is still very small compared to the

overall cost of the LU. The same derivation and effect can be seen for the FB substitution.

Next, the overhead for *Step-c* is given by the following lemma.

Lemma 2. *The overhead in the calculation of **Step-c** for the branch tearing method compared to the node tearing method is $O(l^2)$.*

Proof. For node tearing, the operation to be performed is given in (4.20) and for branch tearing, it is given by (4.21). In both cases, the operations performed are the accumulation of a sparse matrix multiplied by a dense matrix. However, in this case node tearing has an advantage, since the nonzeros of the Ξ matrices are guaranteed to be ± 1 , with at most one nonzero per column. They are also guaranteed to be static across iterations. This implies that these equations can be solved with simple permutation and accumulate operations for the node tearing formulation, with no multiplications needed at all. This can therefore be implemented very efficiently, with fewer memory accesses needed. In terms of the number of multiplications involved, this operation is $O(0)$ for node tearing.

For branch tearing, each value to be accumulated in the $\mathbf{Y}^{(B)}$ matrix is the result of at least one multiplication. If n_i is the number of border nodes that have a branch connecting to the i^{th} partition, then the total number of accumulations that are performed is $\sum_{i=1}^p n_i^2$. It is known that n_i is related to the number of links based on the partitioning. Therefore, let $n_i = k_i l$, and the total number of accumulations will be

$$\sum_{i=1}^p n_i^2 \tag{4.41}$$

$$= \sum_{i=1}^p (k_i l)^2 \tag{4.42}$$

$$= \left(\sum_{i=1}^p k_i^2 \right) l^2. \tag{4.43}$$

The total number of multiplications involved in the construction of the $\mathbf{Y}^{(B)}$ matrix will be equal to the number of accumulations multiplied by the number of nonzeros per row of the sparse $\mathbf{F}_i$ matrices. If this proportionality coefficient is defined as k_0 , the total number of multiplications will be

$$k_0 \left(\sum_{i=1}^p k_i^2 \right) l^2. \quad (4.44)$$

As all of the constants k_j , $0 \leq j \leq p$, are static based on the circuit and partitioning under consideration, the overall complexity, in terms of multiplications, of forming the $\mathbf{Y}^{(B)}$ matrix is $O(l^2)$.

It can be shown that the multiplications for forming the RHS vector in the case of branch tearing will be $O(l)$. □

Finally, the computational cost overheads for the remaining steps: *Step-b*, *Step-d* and *Step-e*, are covered in the next lemma.

Lemma 3. *The overhead for the remaining steps (Step-b, Step-d and Step-e) is $O(l)$, for both the case of node tearing and for branch tearing.*

Proof. Step b) As the $\mathbf{A}$ matrices have already been factorized, this step consists of a number of additional FB substitutions. To calculate the cost of this, it must first be determined exactly how many substitutions must be done.

For node tearing, each link will normally require two FB substitutions: one for each partition the link connects. When a node is connected among m partitions, a total of $m - 1$ links are needed. At first glance, it would appear that these links would require a total of $2(m - 1)$ FB substitutions to solve. However, this is not the case. In fact, only m FB substitutions are required even though there are $m - 1$ links. To demonstrate why this is the case, consider the following example system matrix with three partitions and one node

split among these three partitions:

$$\left[\begin{array}{ccc|ccc|ccc} \left[\begin{array}{c} A_1 \end{array} \right] & & & & & & 0 & 0 \\ & & & & & & 1 & 0 \\ & & & & & & 0 & 0 \\ & & \left[\begin{array}{c} A_2 \end{array} \right] & & & & 0 & 0 \\ & & & & & & -1 & 1 \\ & & & & & & 0 & 0 \\ & & & \left[\begin{array}{c} A_3 \end{array} \right] & & & 0 & 0 \\ & & & & & & 0 & -1 \\ & & & & & & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array} \right], \quad (4.45)$$

where it is observed that

$$\Xi_1 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}; \quad \Xi_2 = \begin{bmatrix} 0 & 0 \\ -1 & 1 \\ 0 & 0 \end{bmatrix}; \quad \Xi_3 = \begin{bmatrix} 0 & 0 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}. \quad (4.46)$$

It is clear that the zero columns in each of these matrices do not require a FB substitution to find the solution. Next, it is observed that the two columns of Ξ_2 are identical except for their signs. Due to this, the solution of $\Gamma_2^{(N)}$ will consist of two columns which are identical, but with opposite signs. Due to this, only one FB substitution is needed to find both. Therefore, a total of three FB substitutions are needed.

For the branch tearing case, recall that the number of split nodes is $l(1 + \beta_1(p - 2))^{-1}$. If the average number of partitions for each split node is set to β_2 , then the total number of needed FB substitutions can be written as

$$\frac{\beta_2 l}{1 + \beta_1(p - 2)}. \quad (4.47)$$

Exactly the same number of FB substitutions will be required in the case of the branch tearing formulation. Therefore, from (4.40), it is known that there will be an overhead that is linearly proportional with the number of links for the case of node tearing. However, once again, this is small.

It is to be noted that the RHS matrices for both formulations are sparse. In the case of the node tearing formulation, it is guaranteed that each column has at most one nonzero. This fact can be exploited in the algorithm implementation to further reduce the cost of this step for node tearing compared to branch tearing.

Step d) The dimensions of the $\mathbf{Z}^{[N]}$ matrix for node tearing will be $l \times l$, while the $\mathbf{Y}^{[B]}$ matrix will also be square but with a size of $l(1 + \beta_1(p - 2))^{-1}$. Although the $\mathbf{Z}^{[N]}$ matrix may be slightly larger, it will be more sparse. Furthermore, through careful ordering of the columns in the border, it can be guaranteed to consist of several smaller dense matrices arranged in a specific block pattern. This allows the creation of an optimized solver for this type of matrix, which can be more efficient than the use of a general sparse solver for the $\mathbf{Y}^{[B]}$ matrix associated with branch tearing.

Although a similar ordering technique can be used for the $\mathbf{Y}^{[B]}$ matrix, it is less efficient when there are many nodes shared between multiple partitions. This becomes especially important when the solution of this matrix is scheduled within a parallel solver. With the block-based structure, the solution can be pipelined, which allows for more efficient scheduling and an improvement in scalability. This is less efficient in the case of branch tearing.

The cost of solving this matrix will be $O(l^{\alpha_3})$, where $\alpha_3 = 3$ when the matrix is fully dense, as is generally the case when there are 3 or fewer partitions. α_3 will decrease quickly as the number of partitions increases. Although this is asymptotically greater than the other

complexities considered so far, the overall cost of this operation is lower than the other operations discussed so far, especially when it is implemented in a parallel algorithm with pipelining.

Step e) In this case, the operations involved are the creation of a RHS vector, a FB substitution to find the vector needed to update the LHS vector, and finally a vector subtraction to update the LHS vector. For this stage, there is little difference between the node tearing and branch tearing formulations. However, there are some savings that can be found using the fact that in forming the RHS vector, in the case of node tearing, it is simply a binary permutation operation instead of a more costly sparse matrix-dense vector multiplication. Therefore, as found before, the extra cost for branch tearing, in terms of the multiplications, will be $O(l)$. Also, there will be an extra cost for node tearing in the FB substitution, which will also be $O(l)$.

□

From the preceding lemmas, the results are combined to give an overall CPU cost scaling relative to the number of links for branch tearing compared to node tearing.

Lemma 4. *For branch tearing, the increase in the CPU cost of an iteration as the number of links between the partitions (l) increases will be $O(l^2)$. The node tearing, on the other hand, exhibits superior scaling, as the complexity increases only as $O(l)$.*

Proof. To summarize, Table 4.1 enumerates the steps, identifies which formulation shows the higher cost, and gives the asymptotic complexity of the additional cost. The complexities of the individual overheads each step are as given in the above lemmas and the corresponding proofs.

Therefore, the node tearing formulation's overhead will be $O(l)$, and with branch tearing it will be $O(l^2)$. Due to this, the scalability of the node tearing method will be superior. □

Table 4.1: Cost comparison

Step	Equations		Extra Cost	
	Node Tearing	Branch Tearing	Node Tearing	Branch Tearing
a	(4.16)	(4.17)	$O(l)$	-
b	(4.18)	(4.19)	$O(l)$	-
c	(4.20)	(4.21)	-	$O(l^2)$
	(4.22)	(4.23)	-	$O(l)$
d	(4.24)	(4.25)	-	-
e	(4.26)	(4.27)	$O(l)$	$O(l)$
<i>Total:</i>			$O(l)$	$O(l^2)$

This comparison has considered only the basic form of domain decomposition and the basic form of node tearing. Many improvements to the domain decomposition formulation have been proposed over the years, as reviewed in Section 2.3.3. It is noted, however, that these improvements do not depend explicitly on the domain decomposition formulation and could therefore be applied to a node tearing formulation. For this reason, the most useful comparison is between pure domain decomposition and pure node tearing.

4.3 Numerical Examples

In this section, the performance and scalability of the proposed algorithm is demonstrated through numerical examples. Each example is executed with two implementations: one using node tearing and the other with branch tearing. To obtain a speedup graph, each algorithm is tested over a range of processor counts for each example. The partitioning used in each case was performed by hMETIS [19, 20], and is identical between the two versions, with the number of partitions matching the number of processors.

Each example was simulated on a machine using AMD Opteron 6172 processors, using up to 16 CPUs. In all cases, the speedup is measured relative to a standard simulation using

a traditional LU solver, with no parallelism or partitioning. To facilitate the comparison, a special simulator was created that allows for the changing of the solution method in a simple manner. To ensure a fair comparison between each method, a fixed step size is used to help ensure the same iterations are taken with each method. As such, the results can not be directly compared with those expected in a commercial simulator which uses a variable step size, as this will nearly always be faster.

4.3.1 DSP example

The first example consists of a system containing 20 cascaded DSP ALU blocks, as shown in Figure 4.3. This circuit consists of 31,706 nodes and 78,110 elements, of which 68,480 are nonlinear devices.

Figure 4.4 illustrates the behavior of the CPU overhead incurred by the proposed node tearing based algorithm for Steps a and c of analysis Section 4.2 with the increasing number of links, when executed on a single core. It is noted that these results correspond well with the conclusions drawn in that section.

The parallel speedup using the proposed algorithm is shown in Figure 4.5, corresponding to the results presented in Table 4.2. As can be seen, the proposed algorithm (shown as the solid line) is scaling quite well compared to the equivalent implementation with branch tearing (shown with the dash-dot line).

Finally, an accuracy comparison is presented for one of the interesting signals, in Figures 4.6. This signal is one of the digital outputs of a DSP block, representing an intermediate calculation. As can be seen, there is no change in accuracy between the two methods.

Figure 4.3: Topology for the DSP example.

Figure 4.4: Illustration of the behaviour of the CPU overhead cost for Steps a + c of the proposed node-tearing-based algorithm with an increasing number of links (for $l \ll n$) for the DSP example.

Figure 4.5: Parallel scalability for the DSP example.

Figure 4.6: Accuracy comparison for the DSP example, for a sample digital calculation result (X1.OUT6).

Table 4.2: Performance results for the DSP example

Partitions and CPUs	Standard Solver	Branch Tearing		Proposed (Node Tearing)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	998.5 s	—	—	—	—
2	—	758.0 s	1.3	617.6 s	1.6
4	—	582.0 s	1.7	340.7 s	3.0
6	—	562.6 s	1.8	238.7 s	4.2
8	—	529.2 s	1.9	194.2 s	5.1
10	—	516.0 s	1.9	160.3 s	6.2
12	—	521.5 s	1.9	140.6 s	7.1
14	—	492.3 s	2.0	127.3 s	7.8
16	—	483.9 s	2.1	115.9 s	8.6

4.3.2 Dual SRAM example

The third example consists of a 64 kb memory array, consisting of two banked 32 kb memory arrays, each in the form illustrated in Figure 4.7. The overall circuit has 156,612 nodes and 447,400 elements of which 446,316 are nonlinear elements.

The scalability measurement results are shown in Table 4.3 and Figure 4.8. As can be seen, the scalability of the proposed algorithm is far superior compared to the branch tearing approach. This is despite the partitioning being much more difficult.

Lastly, an accuracy comparison is once again presented for a couple of interesting signals, in Figures 4.9 and 4.10. The signals are output from the address and control signal decoder. The first, XRAM0.WRITE_ENABLE, is active high when a valid write is being performed. The second, XRAM0.VALID_COLUMN_ADR is active high when a valid column address has been decoded and presented to the memory array. As can be seen, there is no appreciable change in accuracy between the two methods.

Figure 4.7: Topology for the SRAM example.

Figure 4.8: Parallel scalability for the dual SRAM example.

Figure 4.9: Accuracy comparison for the dual SRAM example, for the XRAM0.WRITE_ENABLE signal.

Figure 4.10: Accuracy comparison for the dual SRAM example, for the XRAM0.VALID_COLUMN_ADR signal.

Table 4.3: Performance results for the dual SRAM example

Partitions and CPUs	Standard Solver	Branch Tearing		Proposed (Node Tearing)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	860.8 s	—	—	—	—
2	—	597.0 s	1.4	494.1 s	1.7
4	—	436.6 s	2.0	333.6 s	2.6
6	—	414.1 s	2.1	220.4 s	3.9
8	—	386.2 s	2.2	165.7 s	5.2
10	—	378.6 s	2.3	137.2 s	6.3
12	—	355.5 s	2.4	122.5 s	7.0
14	—	353.6 s	2.4	111.8 s	7.7
16	—	361.4 s	2.4	106.5 s	8.1

4.3.3 Array Multiplier example

This example consists of an interconnected network of eight 16 bit by 16 bit array multipliers, based on the design of the C6288 example from the ISCAS85 benchmark circuits [31], as described in [32]. The circuit for each multiplier is that shown in Figure 4.11, and the overall system is connected as shown in Figure 4.12. The number of nodes in the resulting system is 40,545, connected by 81,226 elements, of which 80,896 are nonlinear devices.

The performance results are given in Table 4.4 and Figure 4.13. Once again, the proposed algorithm is scaling quite well for this type of circuit, and the branch tearing implementation is reaching a maximum speedup of about 2.

An accuracy comparison is presented as for the previous circuits for a couple of interesting signals, in Figures 4.14 through 4.16. Once again, there is no observable difference between the two simulations.

Figure 4.11: Circuit block for the multiplier example.

Figure 4.12: Overall topology for the multiplier example.

Figure 4.13: Parallel scalability for the multiplier example.

Figure 4.14: Accuracy comparison for the multiplier example, for the final result signal P0.

Figure 4.15: Accuracy comparison for the multiplier example, for the intermediate result signal B0_A.

Figure 4.16: Accuracy comparison for the multiplier example, for the intermediate result signal A0_A.

Table 4.4: Performance results for the multiplier example

Partitions and CPUs	Standard Solver	Branch Tearing		Proposed (Node Tearing)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	1282.5 s	—	—	—	—
2	—	1042.7 s	1.2	902.1 s	1.4
4	—	772.4 s	1.7	510.4 s	2.5
6	—	730.2 s	1.8	366.5 s	3.5
8	—	676.7 s	1.9	289.4 s	4.4
10	—	671.3 s	1.9	251.3 s	5.1
12	—	666.9 s	1.9	224.9 s	5.7
14	—	634.4 s	2.0	214.7 s	6.0
16	—	623.8 s	2.0	196.4 s	6.5

4.3.4 Ring Oscillator example

The final example considered is that of a much smaller circuit: a ring oscillator. This circuit consists of a total of 704 transistors connected among 1044 nodes. The transient simulation was performed up to 15 ns, using a fixed step size of 0.01 ns.

The performance results are given in Table 4.5 and Figure 4.17. In this case, the proposed algorithm is scaling better than the branch tearing implementation which is reaching a maximum speedup of about 2. However, there is evidence of saturation which can be related to the size of the circuit.

Figure 4.17: Parallel scalability for the ring oscillator example.

Table 4.5: Performance results for the ring oscillator example

Partitions and CPUs	Standard Solver	Branch Tearing		Proposed (Node Tearing)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	89.2 s	—	—	—	—
2	—	60.7 s	1.5	50.9 s	1.8
4	—	51.2 s	1.7	33.0 s	2.7
6	—	50.7 s	1.8	23.5 s	3.8
8	—	47.4 s	1.9	19.4 s	4.6
10	—	46.1 s	1.9	18.8 s	4.8
12	—	48.2 s	1.9	16.6 s	5.4
14	—	48.0 s	1.9	16.6 s	5.4
16	—	48.6 s	1.8	17.2 s	5.2

4.4 Summary

In this chapter, an initial parallel implementation of the proposed algorithm was performed. A theoretical comparison of the scalability expectations between the proposed approach and the existing domain decomposition approach was given. Through the use of binary link formulations, node tearing achieves overheads that scale more favourably compared to branch tearing methods as the number of partitions and links increase. From this, it is expected that node tearing will provide more scalability as the number of CPUs increases.

To confirm this, several examples were considered that cover a variety of circuit topologies and connectivity patterns. For each of these, the node-tearing-based simulator produced superior scalability, with no sacrifice of accuracy.

In the following chapter, the proposed algorithm will be further improved by proposing some techniques to reduce the computational cost of the simulation and to increase the flexibility of the partitioning process by giving a means to remove some constraints.

Chapter 5

Addressing Partitioning Issues in Node Tearing

The node tearing method, as presented previously, imposes several constraints on the partitioning that can increase the number of nodes that are torn. This, in turn, reduces the attainable scalability. As presented in Section 2.2, for an algorithm that operates by splitting voltage nodes, any elements that reference the same current variable must be treated as a single meta-element (e.g., a set of coupled inductors, current-controlled sources and their probing element, etc.). In this section, the application of node tearing to current variables is explored to remove this constraint. In addition, an optimized handling of certain voltage sources is considered, so that they can be neglected during partitioning, simplifying the graph to be partitioned (this will help to greatly reduce the size of the interconnection matrix, especially for certain classes of circuits such as memory circuits). Finally, a means to handle subcircuits with no path to ground is proposed, so that the partitioner does not have to be constrained to not generate floating partitions.

Figure 5.1: Example circuit for demonstrating current variable tearing.

Figure 5.2: Corresponding hypergraph for the example circuit in Figure 5.1.

5.1 Extension of the Node Tearing Concept for non-Voltage Variables

It is to be noted that, in principle, node tearing operates on nodes, which by definition are voltage variables. However, a typical MNA formulation also involves other types of variables, such as current variables. For example, consider the circuit shown in Figure 5.1 and its hypergraph in Figure 5.2. Using the partitioning shown, the variable I_{L_2} will be split, and therefore needs to exist in both partitions.

Figure 5.3: Transformed example circuit.

Figure 5.4: Example circuit after splitting.

Such a situation can be addressed as follows. First, the mutual inductance is represented as a dependent voltage source, as shown in Figure 5.3. It is noted that the series combination of a voltage source and an inductor does not require a variable for the node between them, nor an additional variable for the current in the source. Therefore, the MNA equations of this manipulated circuit are identical.

To split this circuit as shown in the hypergraph, independent voltage sources are inserted into the node between the second inductor and its source, at node V_x . The source is then duplicated and the two are split apart, resulting in two separate circuits, as shown in Figure 5.4. This is a similar procedure as was followed before for node tearing, except now voltage sources are used instead of current sources.

The previous derivation for node tearing can thus be followed in exactly the same way to provide an equivalent procedure for processing shared current variables. This formulation is in every way compatible with the previous formulation, and results in a matrix with all the same characteristics as with basic node tearing. For example, for the circuit in Figure 5.4, the matrix formulation will be:

$$\begin{bmatrix} Y_1 & & \Xi_1 \\ & Y_2 & \Xi_2 \\ \Xi_1^T & \Xi_2^T & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \gamma \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \mathbf{0} \end{bmatrix}, \quad (5.1)$$

where

$$\mathbf{Y}_1 = \begin{bmatrix} g_1 & -g_1 & 1 & 0 & 0 \\ -g_1 & g_1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -sL_1 & -sM \\ 0 & 0 & 0 & -sM & 0 \end{bmatrix}; g_1 = R_1^{-1} \quad (5.2)$$

$$\mathbf{Y}_2 = \begin{bmatrix} g_2 & 1 \\ 1 & -sL_2 \end{bmatrix} \quad (5.3)$$

$$\mathbf{\Xi}_1 = [0 \ 0 \ 0 \ 0 \ 1]^\top \quad (5.4)$$

$$\mathbf{\Xi}_2 = [0 \ -1]^\top \quad (5.5)$$

$$\mathbf{x}_1 = [V_1 \ V_2 \ I_E \ I_{L_1} \ I_{L_2}]^\top \quad (5.6)$$

$$\mathbf{x}_2 = [V_3 \ I_{L_2}]^\top \quad (5.7)$$

$$\boldsymbol{\gamma} = [V_x] \quad (5.8)$$

$$\mathbf{b}_1 = [0 \ 0 \ E \ 0 \ 0]^\top \quad (5.9)$$

$$\mathbf{b}_2 = [0 \ 0]^\top \quad (5.10)$$

5.2 Optimized Handling of Voltage Sources

In many types of circuits, it is common to have a node that is distributed over the entire circuit, connected to many elements, but whose voltage is known as it is also connected to a grounded voltage source. The most prevalent example of this is a V_{DD} node in a digital CMOS integrated circuit, which will be connected to the source of most of the PMOS transistors and to the bulk of all of them.

While using traditional MNA formulations, this will result in a large number of nonzeros in the row and column associated with such a node due to its high connectivity among

Figure 5.5: Example circuit with a highly-connected V_E node.

different circuit elements. As the cost of the matrix factorization and the solution of the overall system is determined by the number of nonzeros, this can lead to excessive CPU cost.

When partitioning by node tearing is used, the impact is even greater, as this node will necessarily be torn among all partitions. This difficulty is illustrated in Figure 5.5, where V_E is a highly connected node. The MNA system corresponding to this circuit is:

$$\begin{bmatrix} \hat{\mathbf{A}}_x & \mathbf{e} & \mathbf{0} \\ \mathbf{f} & d & 1 \\ \mathbf{0} & 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_x \\ V_E \\ I_E \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{b}}_x \\ 0 \\ E \end{bmatrix}, \quad (5.11)$$

where $\hat{\mathbf{A}}_x \in \mathbb{R}^{(n-2) \times (n-2)}$, $\mathbf{e} \in \mathbb{R}^{(n-2) \times 1}$, $\mathbf{f} \in \mathbb{R}^{1 \times (n-2)}$ and $\hat{\mathbf{x}}_x, \hat{\mathbf{b}}_x \in \mathbb{R}^{(n-2) \times 1}$.

The work involved in tearing this node is unneeded, as its value is already known by the source connected to it, so it does not need to be part of the vector of unknowns. In fact,

it is more appropriate to include it in the right-hand side of the equation.

To accomplish this, (5.11) is simply rearranged so that $V_E (= E)$ is found on right-hand side, giving the following:

$$\begin{bmatrix} \hat{\mathbf{A}}_x & \mathbf{0} \\ \mathbf{f} & 1 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_x \\ I_E \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{b}}_x \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{e} \\ d \end{bmatrix} E \quad (5.12)$$

It is easy to see that (5.12) leads to the following two systems:

$$\hat{\mathbf{A}}_x \hat{\mathbf{x}}_x = \hat{\mathbf{b}}_x - \mathbf{e}E \quad (5.13)$$

$$I_E = -\mathbf{f} \hat{\mathbf{x}}_x - dE \quad (5.14)$$

The system in (5.13) solves the same circuit as the original system, but the variables V_E and I_E have been removed. Also, the non-zeros in d , $\mathbf{e}$ and $\mathbf{f}$ have been removed from the matrix, so the factorization and solution costs will be reduced. If the value of I_E is needed, (5.14) can be used at the end of the iteration to calculate its value.

This transformation can be equally applied to the companion form used with node tearing to simulate nonlinear circuits. In addition, as V_E is moved to the right-hand side and its value is known by all subcircuits, links among the partitions for this node are no longer needed. The only difference will be that there will be one value of I_E for each partition, and to get the true value, these are all summed.

Illustrative Example 2. An illustrative example is given to demonstrate the optimized handling of voltage sources. For this purpose, consider the example voltage-divider circuit of Figure 5.6. Using conventional MNA techniques, the MNA matrix will be of the

Figure 5.6: Example circuit for voltage source handling.

following form:

$$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ I_E \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}. \quad (5.15)$$

The solution of this system is:

$$\begin{bmatrix} V_1 \\ V_2 \\ I_E \end{bmatrix} = \begin{bmatrix} 2 \text{ V} \\ 1 \text{ V} \\ -1 \text{ A} \end{bmatrix}. \quad (5.16)$$

Even though the circuit contains 3 variables, one of them is known to be fixed by the voltage source ($V_1 = 2$). To apply the transformation of Section 5.2 to reduce the system

size, the following definitions are first made:

$$\mathbf{x}_x = V_2 \quad (5.17)$$

$$V_E = V_1 \quad (5.18)$$

$$I_E = I_E \quad (5.19)$$

$$\mathbf{b}_x = 0 \quad (5.20)$$

$$E = 2. \quad (5.21)$$

Applying these definitions, and rearranging (5.15) in the form of (5.11) gives:

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}_x \\ V_E \\ I_E \end{bmatrix} = \begin{bmatrix} \mathbf{b}_x \\ 0 \\ E \end{bmatrix} \quad (5.22)$$

Applying the matrix partitioning from (5.12) gives the following submatrices:

$$\hat{\mathbf{A}}_x = 2 \quad (5.23)$$

$$\mathbf{e} = -1 \quad (5.24)$$

$$\mathbf{f} = -1 \quad (5.25)$$

$$\mathbf{d} = 1 \quad (5.26)$$

Substituting these into (5.12):

$$\begin{bmatrix} 2 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} V_2 \\ I_E \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \end{bmatrix} 2 \quad (5.27)$$

And, finally, the system of (5.13) is given by:

$$2V_2 = 2 \quad (5.28)$$

To solve the system, only (5.28) needs to be solved. The order of the system has been reduced by 2: the variables for the node voltage and the current in the voltage source have been removed. If the current in the voltage source is needed, the second equation, as indicated by (5.14), can be evaluated afterwards as:

$$I_E = V_2 - 2 \quad (5.29)$$

5.3 Handling of Singular Subcircuits

One of the important steps of the node tearing process is to effectively measure the input impedance of the ports (at the tearing interface) of the subcircuits (to which the links are connected), using (3.9) and (3.10). However, in some cases, after partitioning, it is possible that, in the resulting companion form circuit, there exists no path to ground from a specific port of a subcircuit. In such cases, even after connecting a current source to that port, the resulting subcircuits can lead to MNA matrices that are singular. This is demonstrated through an illustrative example below.

Illustrative Example 3. An illustration of such a scenario is given in Figure 5.7(a). When the circuit is partitioned, ports are created with current sources attached in the manner of Figure 3.2(c) on page 33. Next, to solve for the input impedance of each port (as required by (3.9) and (3.10)), with each port being considered in turn (one at a time), a configuration like that shown in Figure 5.7(b) is used (note that the current source is retained in this case

Figure 5.7: Example case illustrating singular partitioning.

for only the port whose input impedance is being measured and the current sources of other ports are open circuited). In this case, it is clear that no solution exists for partition 2, and its MNA matrix is singular.

This case can be detected after partitioning by the detection of structural singularities during the initial matrix ordering and during the simulation if a zero pivot is obtained during the LU factorization process. One possible way to handle this is through the addition

of a very large resistance (on the order of 10^{12} to $10^{15} \Omega$) to the port. This will remove the singularity, but can have the following drawbacks:

- Even though the resistance is very large, it can have an impact on the calculated response of the circuit, and therefore the results can differ from the case where no partitioning is used. For example, if there are high-impedance elements near the port, such as the gate of a MOS transistor or a semiconductor device in cutoff, the difference can be significant.
- The input impedance at the port will be measured to be on the order of the added resistance, which will be several orders of magnitude greater than those of other ports. This can result in numerical issues when resolving the state of the combined system.

To address these problems, an alternative solution is proposed. For this purpose, note that, in this method, resistances are added as before, but their values are chosen carefully so that they are on the same order as the existing port impedances, and also so that they have no effect on the combined circuit. The steps in the method are as follows:

1. For the subcircuit without the singularity, z_{th} is measured at the port. Note that due to the use of the companion form, this will be a purely real number.
2. A compensating resistance, r_{comp} is chosen to be in the neighbourhood of z_{th} . A reasonable value is $0.5z_{th}$.
3. Next, an impedance of r_{comp} is added from the port to ground in the non-singular subcircuit, and $-r_{comp}$ is added to the singular subcircuit.
4. The resulting compensated circuit is analyzed as before.

This process is illustrated by the example in Figure 5.8. The parallel combination of $-r_{\text{comp}}$ and r_{comp} will be equivalent to an open circuit, as shown with

$$-r_{\text{comp}} || r_{\text{comp}} = \frac{-r_{\text{comp}} r_{\text{comp}}}{r_{\text{comp}} - r_{\text{comp}}} = \frac{-r_{\text{comp}}^2}{0} = \infty, \quad (5.30)$$

so it will have no effect on the circuit.

Figure 5.8: Example singular partitioning with compensation.

5.4 Summary

In this chapter, several algorithms were proposed to improve the quality of the partitioned subcircuits for efficient parallel simulation. The first technique is the extension of node-tearing-based partitioning to include non-voltage variables that arise in a general MNA formulation. Next, an optimized technique for handling voltage sources was presented, allowing the circuit to be simplified before partitioning. Finally, a means to handle singular subcircuits that may arise during partitioning was proposed.

In the next chapter, the previously presented parallel algorithm is optimized to better exploit the processors in a system, increasing their utilisation and obtaining better scalability. The results of those improvements, coupled with the ones from this chapter, will be analysed by considering the example circuit suite and comparing the results with those already obtained.

Chapter 6

Optimized Parallel Implementation for Simulation of General Circuits with Node Tearing

In the previous chapters, a new technique for solving a partitioned nonlinear circuit in the time domain was presented. A parallel implementation was developed, and then several improvements to improve scalability were presented. These improvements were done either to the fundamental algorithm, or by manipulating circuit structures. In this chapter, further improvements will be presented; however, they will take advantage of parallel machine characteristics to more fully exploit the processors in a system. In this way, more scalability will be obtained, but the same core mathematical operations will be performed.

6.1 Development of the Pipelined Multi-stage Solution of the Interconnection Matrix

One of the important stages of the solution of a circuit partitioned with node tearing is the resolution of the coupling between the partitions, as described by *Tasks 4-6* in Section 4.1,

and given in (3.34), and (3.35), repeated here for clarity:

$$\mathbf{Z} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \mathbf{P}_j \hat{\mathbf{\Gamma}}_j = - \sum_{j=1}^p \mathbf{\Xi}_j^T \mathbf{Y}_j^{-1} \mathbf{\Xi} \quad (6.1)$$

$$\mathbf{Z}\boldsymbol{\gamma} = - \sum_{j=1}^p \mathbf{\Xi}_j^T \hat{\mathbf{x}}_j \quad (6.2)$$

Ordinarily, the solution of (6.2) would be a sequential operation. While this is performed, there will be no other work available for other processors in the system (i.e., they will be idle during this time period). This can have an impact on scalability, especially for systems with a large number of links between partitions wherein the size of the $\mathbf{Z}$ matrix can become quite large.

In order to address this difficulty, a pipelined multi-stage algorithm for updating and solving the $\mathbf{Z}$ matrix is proposed in this section. For this purpose, first, the details of the structure and construction of the $\mathbf{Z}$ matrix are provided in Section 6.1.1. Section 6.1.2 describes the blockwise updating of the $\mathbf{Z}$ matrix and pipelined solution of (6.2). Section 6.1.3 describes the related implementation issues.

6.1.1 Construction and Structure of the Interconnection Matrix

Let the system consist of p partitions and l links among these p partitions. In the proposed algorithm, for an efficient pipelining strategy, first, the partitions are sorted in an ascending order of estimated computational complexity (based on the number of devices and number of nonzeros in their Jacobian matrices). Let the above ordered partitions be numbered from 1 to p .

Note that each of the l links (that is between two individual partitions) corresponds to one column of the $\mathbf{\Xi}_j$ matrices of (6.1), and thus, to a column of $\mathbf{Z}$. The $\mathbf{\Xi}_j$ matrices are

also reordered so that the links between lower-numbered partitions appear first. If the lower numbered partitions have their resolution prioritized during the iteration, the data needed to fill the $\mathbf{Z}$ matrix will become available from the top left of the matrix first, progressing to the bottom right (this will enable the development of a pipelined solution of the $\mathbf{Z}$ matrix as described in Section 6.1.2). Clearly, if the columns of this matrix (Ξ_j) are reordered to accomplish the above, it will have an effect on the structure of $\mathbf{Z}$. This effect is investigated by examining (6.1) and is explained below.

As each column or row of Ξ_j has a maximum of a single nonzero, and its value is either ± 1 , j can be viewed as a selection and permutation matrix. With this formulation, j is effectively performing both a row- and column-wise permutation of $\mathbf{Y}_j^{-1}$. Therefore, it can be seen that if the columns of Ξ_j are reordered (applied for each j), this will result in an equivalent $\mathbf{Z}$ matrix that has undergone a permutation.

The above formulation and structure of the $\mathbf{Z}$ matrix is described through an illustrative example below.

Illustrative Example 4. Structure of the $\mathbf{Z}$ matrix: Consider the example with four partitions of Figure 3.2(a). Here, let the required link ordering described above be defined as

$$\Xi_j = \begin{bmatrix} \Xi_{j12} & \Xi_{j13} & \Xi_{j23} & \Xi_{j14} & \Xi_{j24} & \Xi_{j34} \end{bmatrix}, \quad (6.3)$$

where Ξ_{jkl} is the part of Ξ for partition j containing links between partitions k and l . **It is noted that where $j \notin \{k, l\}$, Ξ_{jkl} will be a zero matrix.**

For this example, the resulting ordering results in a Ξ with the following order

$$\Xi = \begin{bmatrix} \xi_1 & \xi_2 & \xi_3 & \xi_4 & \xi_5 & \xi_6 & \xi_7 \end{bmatrix} \quad (6.4)$$

where ξ_i is the tearing vector (as described in Section 3.1.1) corresponding to the link for the node x_i in Figure 3.2(a).

The resulting partial Ξ_{jkl} matrices are thus

$$\Xi_{112} = \begin{bmatrix} \xi_{11} & \xi_{12} \end{bmatrix} ; \quad \Xi_{113} = \begin{bmatrix} \xi_{13} \end{bmatrix} ; \quad \Xi_{114} = \begin{bmatrix} \xi_{15} \end{bmatrix} \quad (6.5)$$

$$\Xi_{212} = \begin{bmatrix} \xi_{21} & \xi_{22} \end{bmatrix} ; \quad \Xi_{223} = \begin{bmatrix} \xi_{24} \end{bmatrix} ; \quad \Xi_{224} = \begin{bmatrix} \xi_{26} \end{bmatrix} \quad (6.6)$$

$$\Xi_{313} = \begin{bmatrix} \xi_{33} \end{bmatrix} ; \quad \Xi_{323} = \begin{bmatrix} \xi_{34} \end{bmatrix} ; \quad \Xi_{334} = \begin{bmatrix} \xi_{37} \end{bmatrix} \quad (6.7)$$

$$\Xi_{414} = \begin{bmatrix} \xi_{45} \end{bmatrix} ; \quad \Xi_{424} = \begin{bmatrix} \xi_{46} \end{bmatrix} ; \quad \Xi_{434} = \begin{bmatrix} \xi_{47} \end{bmatrix} \quad (6.8)$$

where ξ_{jm} is the portion of ξ_m corresponding to the nodes of partition j , and all other Ξ_{jkl} are appropriately dimensioned zero matrices.

The full formulation of the Z matrix using the above entities is as follows, obtained by substituting (6.3) in (6.1):

$$Z = - \sum_{j=1}^p \begin{bmatrix} \Xi_{j12}^T Y_j^{-1} \Xi_{j12} & \Xi_{j12}^T Y_j^{-1} \Xi_{j13} & \Xi_{j12}^T Y_j^{-1} \Xi_{j23} \\ \Xi_{j13}^T Y_j^{-1} \Xi_{j12} & \Xi_{j13}^T Y_j^{-1} \Xi_{j13} & \Xi_{j13}^T Y_j^{-1} \Xi_{j23} \\ \Xi_{j23}^T Y_j^{-1} \Xi_{j12} & \Xi_{j23}^T Y_j^{-1} \Xi_{j13} & \Xi_{j23}^T Y_j^{-1} \Xi_{j23} \\ \Xi_{j14}^T Y_j^{-1} \Xi_{j12} & \Xi_{j14}^T Y_j^{-1} \Xi_{j13} & \Xi_{j14}^T Y_j^{-1} \Xi_{j23} \\ \Xi_{j24}^T Y_j^{-1} \Xi_{j12} & \Xi_{j24}^T Y_j^{-1} \Xi_{j13} & \Xi_{j24}^T Y_j^{-1} \Xi_{j23} \\ \Xi_{j34}^T Y_j^{-1} \Xi_{j12} & \Xi_{j34}^T Y_j^{-1} \Xi_{j13} & \Xi_{j34}^T Y_j^{-1} \Xi_{j23} \\ \Xi_{j12}^T Y_j^{-1} \Xi_{j14} & \Xi_{j12}^T Y_j^{-1} \Xi_{j24} & \Xi_{j12}^T Y_j^{-1} \Xi_{j34} \\ \Xi_{j13}^T Y_j^{-1} \Xi_{j14} & \Xi_{j13}^T Y_j^{-1} \Xi_{j24} & \Xi_{j13}^T Y_j^{-1} \Xi_{j34} \\ \Xi_{j23}^T Y_j^{-1} \Xi_{j14} & \Xi_{j23}^T Y_j^{-1} \Xi_{j24} & \Xi_{j23}^T Y_j^{-1} \Xi_{j34} \\ \Xi_{j14}^T Y_j^{-1} \Xi_{j14} & \Xi_{j14}^T Y_j^{-1} \Xi_{j24} & \Xi_{j14}^T Y_j^{-1} \Xi_{j34} \\ \Xi_{j24}^T Y_j^{-1} \Xi_{j14} & \Xi_{j24}^T Y_j^{-1} \Xi_{j24} & \Xi_{j24}^T Y_j^{-1} \Xi_{j34} \\ \Xi_{j34}^T Y_j^{-1} \Xi_{j14} & \Xi_{j34}^T Y_j^{-1} \Xi_{j24} & \Xi_{j34}^T Y_j^{-1} \Xi_{j34} \end{bmatrix} \quad (6.9)$$

Next, the $\mathbf{Z}$ matrix can be further rewritten as

$$\mathbf{Z} = \sum_{j=1}^4 \mathbf{Z}_j, \quad (6.10)$$

where $\mathbf{Z}_j$ (with $j \in [1, 4]$) are as shown in (6.11)-(6.14), below. Note that the symbol ‘ $\mathbf{0}$ ’ represents an appropriately-sized zero matrix.

$$\mathbf{Z}_1 = \begin{bmatrix} \Xi_{112}^\top \mathbf{Y}_1^{-1} \Xi_{112} & \Xi_{112}^\top \mathbf{Y}_1^{-1} \Xi_{113} & \mathbf{0} & \Xi_{112}^\top \mathbf{Y}_1^{-1} \Xi_{114} & \mathbf{0} & \mathbf{0} \\ \Xi_{113}^\top \mathbf{Y}_1^{-1} \Xi_{112} & \Xi_{113}^\top \mathbf{Y}_1^{-1} \Xi_{113} & \mathbf{0} & \Xi_{113}^\top \mathbf{Y}_1^{-1} \Xi_{114} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \Xi_{114}^\top \mathbf{Y}_1^{-1} \Xi_{112} & \Xi_{114}^\top \mathbf{Y}_1^{-1} \Xi_{113} & \mathbf{0} & \Xi_{114}^\top \mathbf{Y}_1^{-1} \Xi_{114} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \quad (6.11)$$

$$\mathbf{Z}_2 = \begin{bmatrix} \Xi_{212}^\top \mathbf{Y}_2^{-1} \Xi_{212} & \mathbf{0} & \Xi_{212}^\top \mathbf{Y}_2^{-1} \Xi_{223} & \mathbf{0} & \Xi_{212}^\top \mathbf{Y}_2^{-1} \Xi_{224} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \Xi_{223}^\top \mathbf{Y}_2^{-1} \Xi_{212} & \mathbf{0} & \Xi_{223}^\top \mathbf{Y}_2^{-1} \Xi_{223} & \mathbf{0} & \Xi_{223}^\top \mathbf{Y}_2^{-1} \Xi_{224} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \Xi_{224}^\top \mathbf{Y}_2^{-1} \Xi_{212} & \mathbf{0} & \Xi_{224}^\top \mathbf{Y}_2^{-1} \Xi_{223} & \mathbf{0} & \Xi_{224}^\top \mathbf{Y}_2^{-1} \Xi_{224} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \quad (6.12)$$

$$\mathbf{Z}_3 = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \Xi_{313}^\top \mathbf{Y}_3^{-1} \Xi_{313} & \Xi_{313}^\top \mathbf{Y}_3^{-1} \Xi_{323} & \mathbf{0} & \mathbf{0} & \Xi_{313}^\top \mathbf{Y}_3^{-1} \Xi_{334} \\ \mathbf{0} & \Xi_{323}^\top \mathbf{Y}_3^{-1} \Xi_{313} & \Xi_{323}^\top \mathbf{Y}_3^{-1} \Xi_{323} & \mathbf{0} & \mathbf{0} & \Xi_{323}^\top \mathbf{Y}_3^{-1} \Xi_{334} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \Xi_{334}^\top \mathbf{Y}_3^{-1} \Xi_{313} & \Xi_{334}^\top \mathbf{Y}_3^{-1} \Xi_{323} & \mathbf{0} & \mathbf{0} & \Xi_{334}^\top \mathbf{Y}_3^{-1} \Xi_{334} \end{bmatrix} \quad (6.13)$$

$$\mathbf{Z}_4 = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \Xi_{414}^T \mathbf{Y}_4^{-1} \Xi_{414} & \Xi_{414}^T \mathbf{Y}_4^{-1} \Xi_{424} & \Xi_{414}^T \mathbf{Y}_4^{-1} \Xi_{434} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \Xi_{424}^T \mathbf{Y}_4^{-1} \Xi_{414} & \Xi_{424}^T \mathbf{Y}_4^{-1} \Xi_{424} & \Xi_{424}^T \mathbf{Y}_4^{-1} \Xi_{434} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \Xi_{434}^T \mathbf{Y}_4^{-1} \Xi_{414} & \Xi_{434}^T \mathbf{Y}_4^{-1} \Xi_{424} & \Xi_{434}^T \mathbf{Y}_4^{-1} \Xi_{434} \end{bmatrix} \quad (6.14)$$

6.1.2 Blockwise Updating of the Interconnection Matrix and its Pipelined Multi-stage Solution

The particular blockwise structure of $\mathbf{Z}$ matrix as discussed in Section 6.1.1 is exploited to develop the proposed pipelined multi-stage solution of (6.2). For this purpose, following two definitions are used.

Definition 1. Let the sub-blocks of the $\mathbf{Z}$ matrix be identified via a non-zero set $\{1 \dots p\}$ based on the contributions from the corresponding $\mathbf{Z}_j$, where $j \in [1, p]$.

An illustrative example is given below, demonstrating the usage of Definition 1.

Illustrative Example 5. For the purpose of illustration, consider the circuit in Figure 3.2 and (6.9)-(6.14). For example, if a given block has values contributed only from $\mathbf{Z}_1$ and $\mathbf{Z}_3$ (and a zero block for all other $\mathbf{Z}_j$), then the associated set will be $\{1, 3\}$. Adopting this method for all blocks, the following pattern is obtained for the blocks in the $\mathbf{Z}$ matrix

of (6.9) as:

$$\begin{bmatrix} \{1, 2\} & \{1\} & \{2\} & \{1\} & \{2\} & \emptyset \\ \{1\} & \{1, 3\} & \{3\} & \{1\} & \emptyset & \{3\} \\ \{2\} & \{3\} & \{2, 3\} & \emptyset & \{2\} & \{3\} \\ \{1\} & \{1\} & \emptyset & \{1, 4\} & \{4\} & \{4\} \\ \{2\} & \emptyset & \{2\} & \{4\} & \{2, 4\} & \{4\} \\ \emptyset & \{3\} & \{3\} & \{4\} & \{4\} & \{3, 4\} \end{bmatrix} \quad (6.15)$$

If the solution of the partitioned system is prioritized for the lowest numbered partition, the first partition will have its results available first, followed by the second and so on.

As the computation of the solution of each partition finishes, the corresponding values can be updated in each of the respective blocks and, when appropriate, the partial LU factorization of the $\mathbf{Z}$ matrix can begin immediately without waiting for the full construction of the $\mathbf{Z}$ matrix. To further explain the proposed algorithm for blockwise updating and pipelined solution of (6.2) the following definition is used.

Definition 2. In a $\mathbf{Z}$ matrix, let a block that has been partially accumulated be designated by $\square$ while one that has all of its values known be denoted by $\blacksquare$, the blocks whose values have still not been updated with any values with the symbol $\square$ and the blocks which remain permanently zero as $\circ$.

Next, the above process is illustrated via an example, below.

Illustrative Example 6. In this example, the block-wise updating and pipelined multistage solution of the $\mathbf{Z}$ matrix is illustrated in Figure 6.1 using the Definition 2 and the circuit in Figure 3.2 as well as (6.9)-(6.15).

As can be seen from (6.15), after the completion of the solution of the first partition, any block that contains '1' in its set can have values accumulated. Any block whose set

Figure 6.1: Process of the creation of the $\mathbf{Z}$ matrix.

is simply $\{1\}$ will have all of its values known. Using the above notations, after the first partition is solved, the resulting block pattern for $\mathbf{Z}$ is shown in Figure 6.1(a).

Next, the second partition is solved, and when its values are available, any blocks whose sets contain 2 can be accumulated. At this point, blocks having sets $\{1\}$, $\{2\}$, or $\{1, 2\}$ are fully accumulated. The resulting pattern is shown in Figure 6.1(b).

At this point, the values for the first block column and block row are fully known. Exploiting this fact, it is possible to immediately begin the factorization of this matrix using a left-looking block LU factorization process [4] and ending after the first block.

Continuing the above process, after the solution of the third partition, the resulting block pattern is shown in Figure 6.1(c). It can be seen that, since the next two block columns and rows are now completed, the LU factorization process can continue for these. Finally, after the fourth partition is solved, the resulting block pattern is shown in Figure 6.1(d) and the LU process can be completed.

The above solution process is given as a pseudocode in Figures 6.2 and 6.3 and also through a numerical example in the numerical results section (Section 6.4.1).

Exploiting this method, the solution of the $\mathbf{Z}$ matrix can start even before the complete solution of all partitions. This forms the basis for the development of a pipelined task scheduling, and is described in more detail in Section 6.2.


```

1 let  $n_b \leftarrow$  number of block rows or block columns of  $\mathbf{Z}$ 
2 let  $n_p \leftarrow$  number of partitions
3 let  $b_f \leftarrow 0$  (current forward block offset, reset at start of iteration)
4 let  $p_f \leftarrow 0$  (next forward partition number, reset at start of iteration)
5 let  $b_b \leftarrow n_b - 1$  (current backward block offset, reset at start of iteration)
6 let  $p_b \leftarrow n_p - 1$  (next backward partition number, reset at start of iteration)
7 let  $\mathbf{Z} \leftarrow$  partially accumulated matrix to be solved
8 let  $\mathbf{Z}_{ij} \leftarrow$  block at row  $i$  and column  $j$  of  $\mathbf{Z}$ , as in (6.9), with zero-based indices
9 let  $\{ \mathbf{L}_{ij}, \mathbf{U}_{ij} \} \leftarrow$  block LU factors of  $\mathbf{Z}$  for row  $i$  and column  $j$ 
10   (aliased with storage for  $\mathbf{Z}_{ij}$ )
11 let  $\boldsymbol{\beta} \leftarrow$  partially accumulated RHS to be solved
12 let  $\boldsymbol{\beta}_i \leftarrow$  corresponding block at row  $i$  of  $\boldsymbol{\beta}$ 
13 let  $\boldsymbol{\gamma} \leftarrow$  destination for solution of  $\mathbf{Z}\boldsymbol{\gamma} = \boldsymbol{\beta}$ 
14 let  $\boldsymbol{\gamma}_i \leftarrow$  corresponding block at row  $i$  of  $\boldsymbol{\gamma}$ 
15
16 function factorize_forward_solve_one_part
17   increment  $p_f$ 
18   return if  $p_f = 1$  or  $b_f \geq n_b$ 
19   let  $b_2 \leftarrow b_f + p_f - 1$ 
20   for  $b_f \leq i < b_2$ 
21     let  $\mathbf{A} \leftarrow \mathbf{Z}_{ii}$ 
22     let  $\boldsymbol{\gamma}_i \leftarrow \boldsymbol{\beta}_i$ 
23     for  $0 \leq j < i$ 
24       let  $\mathbf{A} \leftarrow \mathbf{A} - \mathbf{L}_{ij}\mathbf{U}_{ji}$ 
25       let  $\boldsymbol{\gamma}_i \leftarrow \boldsymbol{\gamma}_i - \mathbf{L}_{ij}\boldsymbol{\gamma}_j$ 
26     end for
27     { perform LU factorisation:  $\mathbf{L}_{ii}\mathbf{U}_{ii} = \mathbf{A}$  }
28     let  $\boldsymbol{\gamma}_i \leftarrow \mathbf{L}_{ii}^{-1}\boldsymbol{\gamma}_i$  using forward substitution
29     for  $i < j < n_b$ 
30       let  $\mathbf{L}_{ji} \leftarrow \mathbf{Z}_{ji}$ 
31       let  $\mathbf{U}_{ij} \leftarrow \mathbf{Z}_{ij}$ 
32       for  $0 \leq k < j$ 
33         let  $\mathbf{L}_{ji} \leftarrow \mathbf{L}_{ji} - \mathbf{L}_{jk}\mathbf{U}_{ki}$ 
34         let  $\mathbf{U}_{ij} \leftarrow \mathbf{U}_{ij} - \mathbf{L}_{ik}\mathbf{U}_{kj}$ 
35       end for
36       let  $\mathbf{L}_{ji} \leftarrow \mathbf{L}_{ji}\mathbf{U}_{ii}^{-1}\mathbf{L}_{ii}^{-1}$  using forward/back substitution
37     end for
38   let  $b_f \leftarrow b_2$ 
39   return
40 end function

```

Figure 6.2: Pseudocode for the pipelined factorization/forward substitution process of $\mathbf{Z}$.

```

1 let  $n_b \leftarrow$  number of block rows or block columns of  $\mathbf{Z}$ 
2 let  $n_p \leftarrow$  number of partitions
3 let  $b \leftarrow n_b - 1$  (current block offset, reset at start of iteration)
4 let  $p \leftarrow n_p - 1$  (next partition number, reset at start of iteration)
5 let  $\{L_{ij}, U_{ij}\} \leftarrow$  previously-computed block LU factors
6 let  $\gamma \leftarrow$  destination for solution of  $\mathbf{Z}\gamma = \beta$ 
7 let  $\gamma_i \leftarrow$  corresponding block at row  $i$  of  $\gamma$ 
8
9 function factorize_back_solve_one_part
10   return if  $p_b = 0$ 
11   let  $b_2 \leftarrow b_b - p_b$ 
12   decrement  $p_b$ 
13   for  $b_b \geq i > b_2$  (decreasing)
14     for  $i < j < n_b$ 
15       let  $\gamma_i \leftarrow \gamma_i - U_{ij}\gamma_j$ 
16     end for
17     let  $\gamma_i \leftarrow U_{ii}^{-1}\gamma_i$  using back substitution
18   end for
19   let  $b_b \leftarrow b_2$ 
20   return
21 end function

```

Figure 6.3: Pseudocode for the pipelined backward substitution process of $\mathbf{Z}$.

6.1.3 Addressing Block Fill-ins

One possible complication to the above method is that block fill-ins can be an issue depending on the ordering of the partitions. A block fill-in is an originally empty block in the matrix that receives some fill-ins during the block LU factorization of the reordered $\mathbf{Z}$ matrix. This can be addressed using the following two approaches.

Reduction of Block Fill-ins via Block Reordering

In this approach, knowledge of the pattern of the $\mathbf{Z}$ matrix, which is known immediately after the partitioning, is exploited to minimize the block fill-ins.

Before the simulation begins, the partitions can be reordered to minimize fill-ins. It can be seen from (6.11) through (6.14) that if all of the links are between adjacently numbered partitions, there will be no fill-ins. However, in a general circuit, links can occur between any two partitions. Therefore, one goal of the ordering is to minimize links between non-adjacent partitions. In cases where finding the specific ordering that gives the minimal number of links (a global optimization problem) is very expensive, a heuristic ordering technique such as approximate minimum degree [33] can be used.

Elimination of Block Fill-ins via Tri-diagonal $\mathbf{Z}$ Matrix

An alternative formulation can be exploited to eliminate the above-mentioned problem of block fill-in during the factorization of the $\mathbf{Z}$ matrix. This is based on formulating in terms of a block tri-diagonal matrix, which is guaranteed to have no block fill-ins during a block LU factorization (BLUF).

It can be shown by examining (6.9) that if it is ensured that there are no links between non-adjacent partitions, the structure of $\mathbf{Z}$ becomes block tri-diagonal. This can be done

with the following procedure:

- For each link between non-adjacent partitions i and j :
 - Insert a single new unconnected node to each partition between i and j , and
 - Replace the original link by a chain of links from i to j , connecting each intermediate partition through the newly added nodes.

This method has several merits and demerits compared with the previous approach.

The main advantages are:

1. Block fill-ins are eliminated.
2. The dimension of block grid is always $(p - 1)$ rows by $(p - 1)$ columns. With the first method, the block grid will grow quickly as more partitions are added.
3. BLUF on a block tri-diagonal matrix has a simpler implementation than general block LU factorization.

while the disadvantages are:

1. The overall dimension of Z will grow, due to the addition of more links.
2. There will be more non-zeros in Z , resulting in more needed storage space.

6.2 Development of the Proposed Parallel Task

Scheduler

In this section, the parallel algorithm of Section 4.1 is revisited. The algorithms presented in this chapter will be based on the same set of tasks defined in that section. To summarize them, the algorithm flow developed previously is shown as a flowchart in Figure 6.4.

Figure 6.4: Overall Newton-Raphson iteration flow.

As can be seen, there are two sections of the flow that can be executed with parallelism. First, *Tasks 1-3* are executed in parallel for each partition, then *Tasks 4-6* are executed sequentially, and finally *Task 7* is executed in parallel. The fact that *Tasks 4-6* are executed sequentially creates a bottleneck that can reduce scalability. However, the pipelining approach developed in the previous section can be applied to reduce this bottleneck and increase CPU utilisation efficiency, as will be shown in the following sections.

6.2.1 Tasks for Parallel Execution

To obtain highest parallelism, the defined tasks must be subdivided and scheduled optimally among the available processors. As can be seen in the flowchart, several of the tasks can be clearly executed in parallel for each subcircuit partition, as was done in the initial implementation. These include *Task 1*, *Task 2*, *Task 3*, and *Task 7*. Therefore, each of these tasks are subdivided into subtasks for each partition. However, and as described in more detail in Section 4.1, some of these tasks can be further subdivided to produce additional parallelism.

Noting that modern designs generally contain a large number of nonlinear devices, they can be efficiently evaluated independently and in parallel. Therefore, *Task 1a* can be further divided into blocks of circuit elements to be evaluated together. A reasonable size for the blocks must be chosen so that sufficient parallelism is created, but not too small as to create a large impact of multithreading overhead.

Task 3 can also be subdivided into several blocks. As the right-hand-side of (4.2) is a matrix, the columns can be solved using forward/back substitution independently and in parallel. A certain number of columns can be solved together within the same subtask, if it is efficient to solve multiple right-hand-sides at once. This can be the case, for example,

```

1 let  $p \leftarrow$  number of partitions
2 let task 1a next partition  $\leftarrow 0$ , task 1b next partition  $\leftarrow 0$ ,
3   task 2 next partition  $\leftarrow 0$ , task 3 next partition  $\leftarrow 0$ ,
4   task 5 next partition  $\leftarrow 0$ , task 7 next partition  $\leftarrow (p - 1)$ 
5 let task 1a next block  $\leftarrow 0$ , task 3 next block  $\leftarrow 0$ 
6 create cleared flags task 4 running, task 5 running, task 6 started
7 create cleared flag arrays, of size  $p$  (for each partition):
8   task 1a complete, task 1b complete, task 2 complete, task 3 complete,
9   task 4 complete, task 5 complete, task 7 ready
10 initialize array task 1a number of blocks with size  $p$  :
11   number of blocks of elements for each partition
12 initialize array task 3 number of blocks with size  $p$  :
13   number of blocks of rhs columns for each partition

```

Figure 6.5: Variable definitions for scheduling algorithms.

if the LU solver can exploit Single Instruction, Multiple Data (SIMD) vector instructions provided by modern processors in its forward/back substitution routines.

Finally, *Task 4*, *Task 5*, and *Task 6* can be divided into $(p - 1)$ subtasks (where p is the number of partitions) using the formulation and methods described in Section 6.1.

Using the above identified tasks and the pipelined solution approach described in Section 6.1, a multithreaded task scheduler is developed in this section to distribute the work to the available threads. This scheduler must respect the data flow requirements of the flowchart in Figure 6.4. However, to obtain more benefit from parallel execution, certain tasks can be prioritized higher than others. This allows them to begin executing as soon as their data is ready, instead of respecting the strict ordering of the flowchart. Two versions of the developed task scheduler are described below.

6.2.2 Scheduling Algorithm A

In this algorithm, the priority for various tasks matches relatively closely their sequential ordering in Figure 6.4. However, there are some differences. In order to effectively support

```

1 function execute_task_1a
2   let  $i \leftarrow$  task 1a next partition
3   let  $j \leftarrow$  task 1a next block
4   increment task 1a next block
5   if task 1a next block = task 1a number of blocks[ $i$ ]
6     let task 1a next block  $\leftarrow$  0
7     increment task 1a next partition
8   end if
9   { evaluate device models for block j of partition i }
10  if  $j + 1 =$  task 1a number of blocks[ $i$ ]
11    set flag task 1a complete[ $i$ ]
12  end if
13  return more_work_available
14 end function
15
16 function execute_task_1b
17   increment task 1b next partition
18   let  $j \leftarrow$  (task 1b next partition - 1)
19   { load values from model evaluation into  $Y_j$  and  $b_j$  }
20   set flag task 1b complete[ $j$ ]
21   return more_work_available
22 end function
23
24 function execute_task_2
25   increment task 2 next partition
26   let  $j \leftarrow$  (task 2 next partition - 1)
27   { solve  $Y_j \hat{x}_j = b_j$ , saving LU factors }
28   set flag task 2 complete[ $j$ ]
29   return more_work_available
30 end function

```

Figure 6.6: Pseudocode for support functions for scheduling algorithms (Part 1 of 3).


```

1 function execute_task_3
2   let  $i \leftarrow$  task 3 next partition
3   let  $j \leftarrow$  task 3 next block
4   increment task 3 next block
5   if task 3 next block = task 3 number of blocks[ $i$ ]
6     let task 3 next block  $\leftarrow$  0
7     increment task 3 next partition
8   end if
9   { solve  $\mathbf{Y}_i \mathbf{\Gamma}_i = \mathbf{\Xi}_i$ , using saved LU factors, for RHS block  $j$  }
10  if  $i =$  task 3 number of blocks[ $i$ ]
11    set flag task 3 complete[ $i$ ]
12  end if
13  return more_work_available
14 end function
15
16 function execute_task_4
17   set flag task 4 running
18   while task 4 next partition  $< p$  and task 3 complete[task 4 next partition]
19     let  $j \leftarrow$  task 4 next partition
20     { accumulate  $\mathbf{Z} \leftarrow \mathbf{\Xi}_j^T \mathbf{\Gamma}_j$ , using permutations
21       accumulate  $\boldsymbol{\beta} \leftarrow \mathbf{\Xi}_j^T \hat{\mathbf{x}}_j$ , using permutations }
22     set flag task 4 complete[ $j$ ]
23     increment task 4 next partition
24   end while
25   clear flag task 4 running
26   return more_work_available
27 end function

```

Figure 6.7: Pseudocode for support functions for scheduling algorithms (Part 2 of 3).

```

1 function execute_task_5
2   set flag task 5 running
3   while task 5 next partition < p and task 4 complete[task 5 next partition]
4     let j ← (task 5 next partition – 1)
5     { using the procedure of Figure 6.2 :
6       perform partial LU factorization of  $\mathbf{Z}$ , block j
7       perform partial forward substitution of  $\mathbf{Z}\boldsymbol{\gamma} = \boldsymbol{\beta}$ , block j } if j ≥ 0
8     set flag task 5 complete[j]
9     increment task 5 next partition
10  end while
11  clear flag task 5 running
12  return more_work_available
13 end function
14
15 function execute_task_6
16   set flag task 6 started
17   for i = p – 1 to 1
18     { using the procedure of Figure 6.3 :
19       perform partial back substitution of  $\mathbf{Z}\boldsymbol{\gamma} = \boldsymbol{\beta}$ , block i }
20     set flag task 7 ready[i]
21   end for
22   set flag task 7 ready[0]
23   return more_work_available
24 end function
25
26 function execute_task_7
27   let j ← task 7 next partition
28   { solve  $\mathbf{Y}_j \Delta \mathbf{x}_j = \boldsymbol{\Xi}_j \boldsymbol{\gamma}$ , using saved LU factors }
29   let  $\mathbf{x}_j \leftarrow \hat{\mathbf{x}}_j - \Delta \mathbf{x}_j$ 
30   decrement task 7 next partition
31   if task 7 next partition ≥ 0
32     return more_work_available
33   else
34     return all_work_complete
35   end if
36 end function

```

Figure 6.8: Pseudocode for support functions for scheduling algorithms (Part 3 of 3).

the pipelining of *Task 5*, the following modifications are made:

- For each task of *Task 1a*, *Task 1b*, *Task 2* and *Task 3*, each of these are performed

```

1 initialize variables as in Figure 6.5
2
3 function execute_next_task
4   if task 4 next partition < p and task 3 complete[task 4 next partition] and
5     not task 4 running
6     return call execute_task_4
7   else if task 5 next partition < p and task 4 complete[task 5 next partition] and
8     not task 5 running
9     return call execute_task_5
10
11
12
13
14
15
16
17
18
19   else if task 1a next partition < p and
20     task 1a next block < task 1a number of blocks[task 1a next partition]
21     return call execute_task_1a
22   else if task 1b next partition < p and task 1a complete[task 1b next partition]
23     return call execute_task_1b
24   else if task 2 next partition < p and task 1b complete[task 2 next partition]
25     return call execute_task_2
26   else if task 3 next partition < p and task 2 complete[task 3 next partition]
27     return call execute_task_3
28   else if task 5 complete[p - 2] and not task 6 started
29     return call execute_task_6
30   else if task 7 next partition ≥ 0 and task 7 ready[task 7 next partition]
31     return call execute_task_7
32   end if
33   return no_more_work_currently_available
34 end function

```

Figure 6.9: Pseudocode for Scheduling Algorithm A.

```

1 initialize variables as in Figure 6.5
2 let  $d_p \leftarrow$  pipeline depth
3 function execute_next_task
4   if task 4 next partition < p and task 3 complete[task 4 next partition] and
5     not task 4 running
6     return call execute_task_4
7   else if task 5 next partition < p and task 4 complete[task 5 next partition] and
8     not task 5 running
9     return call execute_task_5
10   else if task 1b next partition < (task 4 next partition +  $d_p$ ) and
11     task 1a complete[task 1b next partition]
12     return call execute_task_1b
13   else if task 2 next partition < (task 4 next partition +  $d_p$ ) and
14     task 1b complete[task 2 next partition]
15     return call execute_task_2
16   else if task 3 next partition < (task 4 next partition +  $d_p$ ) and
17     task 2 complete[task 3 next partition]
18     return call execute_task_3
19   else if task 1a next partition < p and
20     task 1a next block < task 1a number of blocks[task 1a next partition]
21     return call execute_task_1a
22   else if task 1b next partition < p and task 1a complete[task 1b next partition]
23     return call execute_task_1b
24   else if task 2 next partition < p and task 1b complete[task 2 next partition]
25     return call execute_task_2
26   else if task 3 next partition < p and task 2 complete[task 3 next partition]
27     return call execute_task_3
28   else if task 5 complete[p - 2] and not task 6 started
29     return call execute_task_6
30   else if task 7 next partition ≥ 0 and task 7 ready[task 7 next partition]
31     return call execute_task_7
32   end if
33   return no_more_work_currently_available
34 end function

```

Figure 6.10: Pseudocode for Scheduling Algorithm B.

fully for all partitions before moving to the next. That is, *Task 1a* will be performed for all partitions before advancing to *Task 1b*. This is because keeping the similar tasks together can have benefits for instruction caching.

- At any time, if any *Task 4* or *Task 5* can be executed, this is done at the highest priority. These tasks will begin executing as soon as *Task 3* is completed for the first partition.
- Finally, *Task 6* is performed at the next highest priority, followed by *Task 7*.

The corresponding task scheduler is described in the form of pseudocode in Figure 6.9, with supporting functions added for each task as described in Section 4.1 and with pseudocode in Figures 6.6, 6.7 and 6.8. First, the needed variables are initialized as shown in Figure 6.5. Next, each thread of execution calls the *execute_next_task* function. If it returns *more_work_available*, then the thread can immediately call the function again. If, instead, *no_more_work_currently_available* is returned, then the thread can sleep until at least one other thread completes its current task. Once *all_work_complete* is returned on any thread, the iteration is complete and all threads have no work to perform until the next iteration is started.

An illustration of the task scheduler based on Scheduling Algorithm A is given in Sections 6.4.3 and 6.4.4.

This scheduler has the drawback that if the CPU cost of doing model evaluations is significantly higher than that of resolving the system, *Task 4* and *Task 5* may not be executed sufficiently early to avoid having a significant time where the CPUs are idle. This limitation is addressed in the algorithm developed in the next section.

6.2.3 Scheduling Algorithm B

To further reduce the idle time associated with the “Scheduling Algorithm A”, a modification is presented in this section considering the case where the number of partitions exceeds the number of processors. In this new algorithm, the solution for a certain number of initial partitions is fully computed in order to start *Task 4* and *Task 5* even earlier than when compared to Scheduling Algorithm A of the previous section. The algorithm is given in the pseudocode of Figure 6.10, with the modifications compared to the previous algorithm shown in red.

The main difference in this algorithm is the addition of a new parameter, the *pipelining depth*. This parameter has the following effect:

- Initialize a counter to the value of the pipelining depth.
- Each time *Task 1a* is completed for a partition, if the partition’s number is less than the counter, allow the execution of *Task 1b*, *Task 2*, and *Task 3* to be performed at a higher priority than *Task 1a* for other partitions.
- Each time *Task 4* completes, increment this counter.

This modification allows the pipelined solution of *Task 5* to begin sooner. An appropriate value for the pipelining depth depends on the circuit in question and its partitioning. This value can be determined as outlined in the pseudocode of Figure 6.11.

An illustration of the task scheduler based on Scheduling Algorithm B is given in Sections 6.4.3 and 6.4.4.

```

1 let  $d_p \leftarrow 0$  (pipeline depth)
2 let  $t_{\min} \leftarrow \infty$  (minimum iteration time)
3 let done  $\leftarrow$  false
4
5 while simulation not finished
6   let  $t_0 \leftarrow$  current wall time
7   perform Newton-Raphson iteration
8   let  $t_1 \leftarrow$  current wall time
9   let  $\Delta t \leftarrow t_1 - t_0$ 
10  if not done
11    if  $t_{\min} > \Delta t$ 
12      let  $t_{\min} \leftarrow \Delta t$ 
13      let  $d_p \leftarrow d_p + 1$ 
14    else
15      let done  $\leftarrow$  true
16      let  $d_p \leftarrow d_p - 1$ 
17    end if
18  end if
19 end while

```

Figure 6.11: Pseudocode for pipeline depth selection.

6.3 Platform Specific Optimizations

To gain higher scalability, it is important to consider possible implementation choices that can exploit platform-specific features. These features are in addition to general parallel programming considerations such as data-structure design.

A common way to implement a parallel algorithm involving tasks is to have a master director thread which delegates tasks among worker threads. The way this works is as follows:

1. A set of initial work tasks is added to a queue. These tasks consist of all the tasks that have no data dependencies.
2. One task is taken from this queue and assigned to each work thread. A signal, usually

in the form of a semaphore*, is sent to each thread to begin processing the assigned work tasks.

3. Each time a work thread completes its work task, it signals this fact back to the master thread and then waits for another task to be assigned.
4. When the master thread receives a signal that a work task is complete, it will add any dependent tasks to the work queue. After doing this, it will assign any available work tasks to the currently idle threads, and signal them to begin processing.
5. This process repeats until all work tasks are complete.

In the implementation of the proposed algorithm, it was observed that using this style of task management resulted in a large reduction in the scalability as the size of the work tasks decreased. More specifically, it became hard to obtain speedup for cases where the circuit was of a small size. It is clearly still desirable to obtain speedup even in these cases, as the overall simulation time can still be quite long if many timepoints are required for these small circuits.

To determine the cause of this problem, first, the implementation of the task scheduler must be examined. The first problem that becomes apparent is that there will be a latency seen in the use of semaphores. Each time a work task finishes, the worker thread will go to sleep waiting for the next task from the scheduler in the main thread to assign it new work.

This is illustrated for the case of a simulation of a ring oscillator circuit on a quad-core AMD Phenom 9850 machine through the corresponding thread activity graph shown in Figure 6.12. Each time a thread is waiting for a semaphore, it switches to an idle state. This is shown as the thin grey period (semaphore wait) in the thread activity. As can be

*A semaphore is a multi-threading construct which allows one thread to pause its execution until it receives a signal from another thread.

Figure 6.12: Thread activity graph with POSIX thread synchronization.

Figure 6.13: Thread activity graph with hardware thread synchronization.

seen, for this example, the latency of these locking operations is larger than the cost of many of the work tasks. As a result, in a worst case, the execution could actually be longer in a multithreaded mode than in a direct sequential operation.

To avoid this problem, instead, the task delegation is moved from the master thread into the worker threads themselves. In this way, if more work is available, the worker thread can immediately begin executing it. The overall worker thread is consequently implemented along the lines of the pseudocode shown in Figure 6.14.

Examining the algorithm of Figure 6.10 (Scheduling Algorithm B), it is seen that there are a number of variables that are used to keep track of task completion. These variables must be shared among all worker threads, and thus access to them must be controlled with locks, to ensure only one thread modifies them at a given time.


```

1 function thread_worker
2   repeat
3     let status = call execute_next_task
4     switch status
5       case more_work_available :
6         post semaphore
7         break
8       case no_more_work_available :
9         wait for semaphore
10        break
11      case all_work_complete :
12        return
13    end switch
14  end repeat
15 end function

```

Figure 6.14: Pseudocode for worker thread implementation.

To see the needed modifications to accomplish this, the lines 4-7 of Figure 6.10 for *Task 4* are expanded, by including lines of 17-26 of the pseudocode of Figure 6.7. This is shown in Figure 6.15. To convert this to use POSIX locking primitives, the following is done:

1. The flag *task 4 running* ensures only one task of *Task 4* is executing at once. This is replaced with a lock *task 4 lock*, managing exclusive access to the task.
2. Access to *task 4 next partition*, Z and β is also managed by this lock.
3. Exclusive access to the array *task 3 complete* is managed with the lock *task 3 complete lock*.
4. Exclusive access to the array *task 4 complete* is managed with the lock *task 4 complete lock*.

The results of these changes are shown in Figure 6.16. This represents a large improve-

ment to scalability, as semaphores are only used rarely when the threads are starved for work. However, there are still other cases where a thread can be suspended, and needless latency can result.

Each time a `set lock` operation fails, the thread will sleep until it is woken by a corresponding `reset lock` operation. Due to the nature of POSIX thread implementations, to favour power efficiency and multitasking, a thread will be suspended while it is waiting for a lock. An example method to accomplish this is through the `futex` feature of the Linux kernel [34]. A suspended thread will require a certain amount of time before it will resume executing, and this results in a certain amount of latency each time a locking operation fails.

To avoid this latency, the suspension of the thread must be avoided. This means that CPU control will remain with the algorithm and not be passed to the OS kernel. To achieve this, the task scheduler was re-implemented without using POSIX thread library locking primitives. Instead, the data structures were changed so that they could be accessed in a thread-safe manner using atomic instructions exploiting hardware-level locking [35]. In the case locks were needed, they were replaced by spinlocks that also exploit hardware locking. In this way, the latency of a locking failure is avoided, and scalability is improved.

To accomplish this, two specific data structures were considered and exploited in this implementation, details of which are given below.

- **Bounded counters** – This is a counter that has maximum and minimum values. It can be incremented or decremented without needing locks. The counter can be used to distribute partitions to be processed to each thread in the scheduler, and can also be used to implement spinlocks. The x86 implementation used the common `lock cmpxchg` (locked compare and exchange) instruction for atomic operation.

```

1
2
3
4   if task 4 next partition < p and task 3 complete[task 4 next partition]
5       and not task 4 running
6       set flag task 4 running
7   while task 4 next partition < p and
8       task 3 complete[task 4 next partition]
9
10      let j ← task 4 next partition
11      { accumulate  $Z \leftarrow \Xi_j^T \Gamma_j$ , using permutations
12        accumulate  $\beta \leftarrow \Xi_j^T \hat{x}_j$ , using permutations }
13
14      set flag task 4 complete[j]
15
16      increment task 4 next partition
17
18      end while
19      clear flag task 4 running
20
21
22      return more_work_available
23
24
25
26
27 else if ...

```

Figure 6.15: Pseudocode for Scheduling Algorithm B, task 4 processing, with scheduling controlled by the master thread.

```

1   try set lock task 4 lock
2   if lock obtained
3       set lock task 3 complete lock
4       if task 4 next partition < p and task 3 complete[task 4 next partition]
5
6
7       while task 4 next partition < p and
8           task 3 complete[task 4 next partition]
9           reset lock task 3 complete lock
10          let j ← task 4 next partition
11          { accumulate  $Z \leftarrow \Xi_j^T \Gamma_j$ , using permutations
12            accumulate  $\beta \leftarrow \Xi_j^T \hat{x}_j$ , using permutations }
13          set lock task 4 complete lock
14          set flag task 4 complete[j]
15          reset lock task 4 complete lock
16          increment task 4 next partition
17          set lock task 3 complete lock
18          end while
19
20          reset lock task 3 complete lock
21          reset lock task 4 lock
22          return more_work_available
23      else
24          reset lock task 3 complete lock
25          end if
26          reset lock task 4 lock
27 end if

```

Figure 6.16: Lock-based implementation pseudocode for task 4 processing in Scheduling Algorithm B, where the scheduler runs in each thread.

In Figure 6.16, this is used as a spinlock for the main lock of *task 4 lock* and as a counter for *task 4 next partition*.

- **Priority sets** – This structure is used to keep track of the arrays of flags used in the previous pseudocodes to track task completion. It is implemented as a bitfield. When a partition's flag is to be set, the corresponding bit is set atomically with the `lock bts` (locked bit set) instruction. When the scheduler is looking for the next available partition, the lowest set bit is found, and is cleared atomically using the `lock btr` (locked bit reset) instruction. In Figure 6.16, this is used to replace the *task 3 complete* and *task 4 complete* arrays.

The method for replacing the arrays is as follows:

- Line 11 of Figure 6.7, where the flag is set in *task 3 complete*, is replaced by:

atomically set bit *i* in *task 3 complete*

- The check of this array in lines 4-8 of Figure 6.16 is replaced by:

```
while task 4 next partition < p
  atomically pop lowest bit of task 3 complete into j
  if j ≠ task 4 next partition
    atomically set bit j in task 3 complete
    break from while loop
  end if
...
```

- The *task 4 complete* array is treated in an identical manner.

One important consideration in the use of spinlocks is that the locking time be kept to a minimum. While a spinlock is being waited on, no other useful work can be done on the

given CPU. Therefore, it is critical that they are only used when the lock is expected to be held for only a minimal amount of time. If this is not the case, then lock testing should be used instead, as demonstrated by the `try set lock` operation in Figure 6.16.

The effect of these is shown with the revised thread activity graph shown in Figure 6.13. As can be seen, instead of the thread going idle when a lock fails, it actively waits for the lock to be released. This is shown by the black bars. The idle time is minimized, so scalability can be once again achieved.

6.4 Numerical Results

In this section five examples are presented to demonstrate the validity and efficiency of the proposed algorithms. The first example numerically illustrates and validates the steps associated with the pipelined multi-stage solution associated with the interface matrix. The remaining examples reconsider the circuit examples from Section 4.3. The earlier performance using the proposed algorithm is compared with that obtained with the improvements proposed in the last two chapters. In addition, two of the examples (the SRAM and Multiplier) are analysed in closer detail, comparing the performance between using the task scheduling algorithms A and B, when a large number of processors are used.

6.4.1 Pipelined Multi-stage Solution of Interconnection Matrix

In this example, the steps associated with the pipelined multi-stage updating of the Z matrix and solution of (3.15) are numerically illustrated and validated. For this purpose, the example circuit of Figure 6.17 consisting of 4 partitions with 3 links is considered.

Figure 6.17: Example circuit for illustration of $\mathbf{Z}$ matrix solution pipelining: (a) Before partitioning. (b) After partitioning.

The overall system matrix for this circuit is obtained as

$$\begin{bmatrix} \mathbf{Y}_1 & & & & \mathbf{\Xi}_1 \\ & \mathbf{Y}_2 & & & \mathbf{\Xi}_2 \\ & & \mathbf{Y}_3 & & \mathbf{\Xi}_3 \\ & & & \mathbf{Y}_4 & \mathbf{\Xi}_4 \\ \mathbf{\Xi}_1^T & \mathbf{\Xi}_2^T & \mathbf{\Xi}_3^T & \mathbf{\Xi}_4^T & \boldsymbol{\gamma} \end{bmatrix} \begin{bmatrix} V_{11} \\ V_{12} \\ V_{13} \\ V_{14} \\ \boldsymbol{\gamma} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \mathbf{0} \end{bmatrix}, \quad (6.16)$$

where, if the links are ordered with the methods of Section 6.1,

$$\mathbf{Y}_1 = 1 \ ; \ \mathbf{Y}_2 = 1 \ ; \ \mathbf{Y}_3 = 1 \ ; \ \mathbf{Y}_4 = 1 \quad (6.17)$$

$$\mathbf{\Xi}_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \ ; \ \mathbf{\Xi}_2 = \begin{bmatrix} -1 & 1 & 0 \end{bmatrix} \quad (6.18)$$

$$\mathbf{\Xi}_3 = \begin{bmatrix} 0 & -1 & 1 \end{bmatrix} \ ; \ \mathbf{\Xi}_4 = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix} \ ; \ \boldsymbol{\gamma} \in \mathbb{R}^{3 \times 1}. \quad (6.19)$$

To find the overall $\mathbf{Z}$ matrix, first, (3.15) is solved for each partition. If $\mathbf{Z}_i$ is the i^{th} component of $\mathbf{Z}$, then:

$$\mathbf{Z}_1 = -\mathbf{\Xi}_1^T \mathbf{Y}_1^{-1} \mathbf{\Xi}_1 = -\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} 1 \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.20)$$

$$\mathbf{Z}_2 = -\mathbf{\Xi}_2^T \mathbf{Y}_2^{-1} \mathbf{\Xi}_2 = \begin{bmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.21)$$

$$\mathbf{Z}_3 = -\mathbf{\Xi}_3^T \mathbf{Y}_3^{-1} \mathbf{\Xi}_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \end{bmatrix} \quad (6.22)$$

$$\mathbf{Z}_4 = -\mathbf{\Xi}_4^T \mathbf{Y}_4^{-1} \mathbf{\Xi}_4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad (6.23)$$

Assume that the partitions were pre-sorted based on their computational complexity (as discussed in Section 6.1.1), in the order $\mathbf{Z}_1$ to $\mathbf{Z}_4$. Correspondingly, consider that the solution of each partition occurs sequentially in time beginning with $\mathbf{Z}_1$, followed by $\mathbf{Z}_2$. Examining the $\mathbf{Z}_j$ matrices from (6.20)-(6.23) and using (6.10), it can be seen that all of the values in the first row and first column of the $\mathbf{Z}$ matrix come uniquely from these two

partial matrices, $\mathbf{Z}_1$ and $\mathbf{Z}_2$. The intermediate $\mathbf{Z}$ matrix will be:

$$\begin{bmatrix} -2 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.24)$$

At this stage, although the $\mathbf{Z}$ matrix is not yet fully computed, an LU decomposition process can begin on the first row and column. Using an in-place decomposition [4], this will result in the following matrix:

$$\begin{bmatrix} -2 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.25)$$

Next, $\mathbf{Z}_3$ becomes available, and it is accumulated into the partial $\mathbf{Z}$ matrix in (6.25), giving:

$$\begin{bmatrix} -2 & 1 & 0 \\ -\frac{1}{2} & -\frac{3}{2} & 1 \\ 0 & 1 & -1 \end{bmatrix} \quad (6.26)$$

All of the data for the second column and second row are now available, so the factorization can proceed to the second pivot, resulting in:

$$\begin{bmatrix} -2 & 1 & 0 \\ -\frac{1}{2} & -\frac{3}{2} & 1 \\ 0 & -\frac{2}{3} & -\frac{1}{3} \end{bmatrix} \quad (6.27)$$

Finally, $\mathbf{Z}_4$ becomes ready, and its values are accumulated:

$$\begin{bmatrix} -2 & 1 & 0 \\ -\frac{1}{2} & -\frac{3}{2} & 1 \\ 0 & -\frac{2}{3} & -\frac{4}{3} \end{bmatrix} \quad (6.28)$$

The final $\mathbf{L}$ and $\mathbf{U}$ factors are obtained by splitting this matrix as:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & -\frac{2}{3} & 1 \end{bmatrix} ; \quad \mathbf{U} = \begin{bmatrix} -2 & 1 & 0 \\ 0 & -\frac{3}{2} & 1 \\ 0 & 0 & -\frac{4}{3} \end{bmatrix} \quad (6.29)$$

To verify the above results from a pipelined multi-stage factorization, the product $\mathbf{LU}$ is computed as

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & -\frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} -2 & 1 & 0 \\ 0 & -\frac{3}{2} & 1 \\ 0 & 0 & -\frac{4}{3} \end{bmatrix} = \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix}, \quad (6.30)$$

which is exactly the same as $\mathbf{Z}$:

$$\mathbf{Z} = \sum_{i=1}^4 \mathbf{Z}_i = \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix}. \quad (6.31)$$

Thus, the proposed pipelined solution procedure for the $\mathbf{Z}$ matrix is shown to provide correct results even though it begins before the entire $\mathbf{Z}$ matrix is known.

6.4.2 DSP example

For this example, the DSP example, originally considered in Section 4.3.1, is run again with the simulator modified to include the improvements from Chapter 5 and with Scheduling Algorithm B.

A comparison of the speedup is shown in Figure 6.18, with the numerical values shown in Table 6.1. As can be seen, for this example, there were no major improvements brought for this circuit. This can be explained by the relatively low cost of partitioning (few links were needed) reducing the impact of pipelining. The circuit topology causes this

Figure 6.18: Parallel scalability for the DSP example.

eased partitioning, and thus is also such that the improvements of Chapter 5 did not bring improvements, either.

Table 6.1: Performance results for the DSP example

Partitions and CPUs	Standard Solver	Initial Proposed (Chapter 4)		Improved (Chapters 5+6)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	998.5 s	—	—	—	—
2	—	617.6 s	1.6	618.8 s	1.6
4	—	340.7 s	3.0	322.5 s	3.1
6	—	238.7 s	4.2	240.0 s	4.2
8	—	194.2 s	5.1	189.1 s	5.3
10	—	160.3 s	6.2	161.2 s	6.2
12	—	140.6 s	7.1	139.5 s	7.2
14	—	127.3 s	7.8	119.9 s	8.3
16	—	115.9 s	8.6	116.8 s	8.5

6.4.3 Dual SRAM example

For this example, the Dual SRAM example, originally considered in Section 4.3.2, is used to explore the performance and scalability characteristics of the two scheduling algorithms proposed in this chapter.

To this end, first, the activity of each of the threads during a Newton-Raphson iteration will be analyzed. To illustrate this clearly, a large number of partitions (32) will be used on a limited number of processors (8). This is to highlight the differences between the algorithms, which, as discussed previously, are more significant when there is a larger number of partitions.

To show the scalability differences between the two algorithms, a scalability exercise is run from 1 to 16 processors, each time with a number of partitions equal to twice the number of processors. Again, this is to highlight the differences between the algorithms.

Finally, to give a meaningful comparison against the initial implementation of Chapter 4, the simulations are run again using Scheduling Algorithm B with a number of partitions

equal to the number of CPUs (as done previously), and the results compared with those obtained previously.

Performance of Scheduling Algorithm A

In this section, the performance of Scheduling Algorithm A is examined. For the initial test and for the scheduling graphs presented in this section, the circuit was partitioned into 32 partitions and the execution was performed on 8 cores of a system using AMD Opteron 6172 processors (a large number of partitions is chosen here to emphasize the difference between the two algorithms). The partitions were determined by the knowledge of the using physical layout of this circuit as well as the graph partitioner [19, 20], which was used for other examples. The partitions were ordered to minimize the block fill-in during the Z-matrix factorization, based on the method in Section 6.1.3.

The raw CPU cost associated with the various tasks for one NR iteration is given in Table 6.2. The tasks used here are those defined in Section 4.

An illustrative thread utilization graph using Scheduling Algorithm A is given in Figure 6.19. As can be seen, most of the tasks are performed one at a time, with the following exception. At the point indicated by (1), *Task 4* begins executing in the pipelined mode. Shortly afterwards, *Task 5* begins executing at point (2). Without the pipelining active, all of the execution of *Tasks 4-6* would occur serially after *Task 3* (the purple section) and before *Task 7* (the green sections). So, there is already a clear gain here. However, in this example, the pipelining is shown not to be fully effective. There is a period of time towards the end where *Task 5* is the only task executing, and the remaining processors are idle.

In the next section, the performance using Scheduling Algorithm B will be compared.

Figure 6.19: Thread activity graph for one NR iteration of the dual SRAM example, with 32 partitions and 8 processors using Scheduling Algorithm A.

Performance of Scheduling Algorithm B

In this section, the dual SRAM circuit is run again in the same simulator, except that the scheduler is replaced with “Scheduling Algorithm B”, with an illustrative pipelining depth of 2. The resulting thread utilization graph is shown in Figure 6.20. In this case, the overall iteration takes less time, due to the reduction of idle time. This was accomplished due to *Task 4* and *Task 5* being able to begin execution earlier. The point where *Task 4* begins executing, indicated by ①, is clearly much sooner than with the previous approach. Similarly, *Task 5* also begins executing sooner, as indicated by ②. It is noted, however, that while the wall time is shorter, the overall CPU work is not reduced at all. The speedup is obtained solely by increasing the average CPU utilization.

The raw CPU cost associated with the various tasks for one NR iteration using the scheduling algorithm is compared in Table 6.2. As can be seen, they are virtually identical for both Scheduling Algorithms A and B. Therefore, any improvement to speedup or scalability using Scheduling Algorithm B is due to the efficient scheduling that is adopted in this approach which further exploits the available parallel platform.

In the next section, the scalability characteristics of both schedulers for a large number of partitions is compared.

Scalability comparison between Scheduling Algorithms A and B

To compare the scalability characteristics of the two schedulers, the simulations are run again, this time for a varying number of processors. To continue to emphasize the differences between the methods, a large number of partitions are used. For this test, the number of partitions is set to twice the number of processors used.

The wall time and the speed-up for the full simulation with an increasing number of

Figure 6.20: Thread activity graph for one NR iteration of the dual SRAM example, with 32 partitions and 8 processors using Scheduling Algorithm B.

Table 6.2: CPU cost of one NR iteration task for the dual SRAM example, 32 partitions

Task	Unpartitioned Algorithm	Scheduling Algorithm A	Scheduling Algorithm B
<i>Task 1a</i>	6.0301 s	7.3839 s	7.1612 s
<i>Task 1b</i>	1.6238 s	2.0665 s	2.1128 s
<i>Task 2</i>	432.31 ms	177.20 ms	189.43 ms
<i>Task 3</i>	–	1.0420 s	1.0066 s
<i>Task 4</i>	–	70.832 ms	69.378 ms
<i>Task 5</i>	–	302.28 ms	318.59 ms
<i>Task 6</i>	–	8.0313 ms	8.4986 ms
<i>Task 7</i>	–	27.001 ms	27.935 ms
Total CPU Work	8.0877 s	11.078 s	10.894 s
Wall Time	–	1.5325 s	1.41373 s
Speedup	–	5.28 $\times$	5.72 $\times$
Parallel Overhead	–	37.0 %	34.7 %
CPU Utilisation	–	90.36 %	96.33 %

cores, for both scheduling algorithms, is given in Table 6.3 (the speed-up was measured with respect to running the original circuit without partitioning and on a single core). The corresponding scalability of Scheduling Algorithm A can be seen in the speedup graph of Figure 6.21 as the solid line.

The improved scalability using Scheduling Algorithm B can be seen in Figure 6.21 as the dashed-dotted line. For this case, the optimized pipeline depth is shown in Table 6.3. For few processors, the results are very similar between both algorithms – which is expected as the pipeline depth was chosen to be 0. The improvement becomes more important for more processors and partitions, where the pipeline depth is optimized to a higher number. This demonstrates that Scheduling Algorithm B brings improvements to scalability, and has an effect for higher numbers of processors and partitions.

Figure 6.21: Parallel scalability for the dual SRAM example, partitions = $2 \times$ CPUs.

Table 6.3: Parallel scalability for the dual SRAM example, partitions = $2 \times$ CPUs

CPUs	Unpartitioned Algorithm	Scheduling Algorithm A		Scheduling Algorithm B		Pipeline Depth
	Wall Time	Wall Time	Speedup	Wall Time	Speedup	
1	839.0 s	851.3 s	0.98	848.4 s	0.98	0
2		499.0 s	1.68	510.6 s	1.64	0
4		257.0 s	3.26	262.6 s	3.19	0
6		184.6 s	4.54	189.0 s	4.43	0
8		147.0 s	5.70	145.7 s	5.75	0
10		125.5 s	6.68	123.7 s	6.78	1
12		115.1 s	7.29	111.3 s	7.53	1
14		109.0 s	7.69	104.3 s	8.04	4
16		109.7 s	7.64	101.1 s	8.30	4

Comparison with Initial Implementation

For this final test, the circuit is re-simulated using the same numbers of processors and partitions as in Section 4.3.2, but this time with the improvements from Chapter 5 and

Figure 6.22: Parallel scalability for the dual SRAM example.

using Scheduling Algorithm B.

A comparison of the speedup is shown in Figure 6.22, with the numerical values shown in Table 6.4. In contrast with the previous example, for this circuit, a global improvement to the speedup of about 10% is seen. Therefore, for this type of circuit structure, the improvements are having a noticable impact.

Table 6.4: Performance results for the dual SRAM example

Partitions and CPUs	Standard Solver	Initial Proposed (Chapter 4)		Improved (Chapters 5+6)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	860.8 s	—	—	—	—
2	—	494.1 s	1.7	461.8 s	1.9
4	—	333.6 s	2.6	277.9 s	3.1
6	—	220.4 s	3.9	190.0 s	4.5
8	—	165.7 s	5.2	147.6 s	5.8
10	—	137.2 s	6.3	124.8 s	6.9
12	—	122.5 s	7.0	108.2 s	8.0
14	—	111.8 s	7.7	100.4 s	8.6
16	—	106.5 s	8.1	93.68 s	9.2

6.4.4 Array Multiplier example

For this next example, the Array Multiplier circuit, originally considered in Section 4.3.3, is subjected to the same analyses as done for the previous example.

Performance of Scheduling Algorithm A

In this part of the experiment, the performance of Scheduling Algorithm A is reverified. An illustration of the execution of the related tasks is provided via a sample scheduling graph (for one iteration while simulating with 32 partitions and 8 cores) in Figure 6.23.

As before, most of the tasks are performed one at a time, with the following exception. At the point indicated by ①, *Task 4* begins executing in the pipelined mode. Shortly afterwards, *Task 5* begins executing at point ②. Once again, as before, in this example the pipelining is shown to give some improvement, but is not fully effective. There is a period of time towards the end where *Task 5* is the only task executing on a particular processor, and the remaining processors are idle, leading to a higher wall time.

Figure 6.23: Thread activity graph for one NR iteration of the multiplier example, with 32 partitions and 8 processors using Scheduling Algorithm A.

Performance of Scheduling Algorithm B

In this part of the experiment, the same circuit is executed with “Scheduling Algorithm B”, with an illustrative pipelining depth of 2. The resulting thread utilization graph is shown in Figure 6.24. In this case, the overall iteration takes less time, due to the reduction of idle time. This was accomplished due to *Task 4* and *Task 5* being able to begin execution earlier. The point where *Task 4* begins executing, indicated by ①, is clearly much sooner than with the previous approach. Similarly, *Task 5* also begins executing sooner, as indicated by ②. It is noted that, while the overall CPU work is not reduced, the wall time is reduced due to efficient scheduling. The speedup is obtained solely by increasing the average CPU utilization.

The raw CPU cost associated with the various tasks for one NR iteration using both of the scheduling algorithms A and B is compared in Table 6.5. As can be seen, the costs are approximately similar for both of the scheduling algorithms. However, the wall time in the case of Scheduling Algorithm B is smaller compared to Scheduling Algorithm A, mainly because of the efficient scheduling of *Task 5*, which minimizes CPU idle times and increases the average CPU utilization.

Scalability comparison between Scheduling Algorithms A and B

Next, to assess the speedup using both the algorithms, simulations for cases of different number of cores/partitions was performed. The corresponding results are given in Table 6.6, which gives the wall time and the speedup for the full simulations with an increasing number of cores (in each indicated case, the number of partitions is selected to be twice the number of CPU cores). As can be seen, the Algorithm B provides improved scalability, especially for higher numbers of processors and partitions.

Figure 6.24: Thread activity graph for one NR iteration of the multiplier example, with 32 partitions and 8 processors using Scheduling Algorithm B.

Table 6.5: CPU cost of one NR iteration task for the multiplier example, 32 partitions

Task	Unpartitioned Algorithm	Scheduling Algorithm A	Scheduling Algorithm B
<i>Task 1a</i>	1.0174 s	1.1552 s	1.1188 s
<i>Task 1b</i>	258.54 ms	302.93 ms	322.07 ms
<i>Task 2</i>	73.040 ms	44.940 ms	57.501 ms
<i>Task 3</i>	–	121.89 ms	106.00 ms
<i>Task 4</i>	–	9.8960 ms	9.9533 ms
<i>Task 5</i>	–	68.031 ms	70.481 ms
<i>Task 6</i>	–	1.7483 ms	1.7742 ms
<i>Task 7</i>	–	7.1525 ms	7.6391 ms
Total CPU Work	1.3495 s	1.7118 s	1.6942 s
Wall Time	–	260.43 ms	220.50 ms
Speedup	–	5.18 $\times$	6.12 $\times$
Parallel Overhead	–	26.8 %	25.5 %
CPU Utilisation	–	82.16 %	96.04 %

Figure 6.25: Parallel scalability for the multiplier example, partitions = $2 \times$ CPUs.

Table 6.6: Parallel scalability for the multiplier example, partitions = $2 \times$ CPUs.

CPUs	Unpartitioned Algorithm	Scheduling Algorithm A		Scheduling Algorithm B		Pipeline Depth
	Wall Time	Wall Time	Speedup	Wall Time	Speedup	
1	1285 s	1495 s	0.86	1491 s	0.86	0
2		926.2 s	1.39	925.7 s	1.39	0
4		472.3 s	2.72	477.1 s	2.69	0
6		355.0 s	3.62	357.2 s	3.60	0
8		280.2 s	4.59	276.4 s	4.65	1
12		237.8 s	5.40	211.2 s	6.08	3
16		246.2 s	5.22	180.9 s	7.10	7

Table 6.7: Performance results for the multiplier example

Partitions and CPUs	Standard Solver	Initial Proposed (Chapter 4)		Improved (Chapters 5+6)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	1282.5 s	—	—	—	—
2	—	902.1 s	1.4	897.1 s	1.4
4	—	510.4 s	2.5	498.2 s	2.6
6	—	366.5 s	3.5	365.4 s	3.5
8	—	289.4 s	4.4	279.2 s	4.6
10	—	251.3 s	5.1	229.6 s	5.6
12	—	224.9 s	5.7	215.8 s	5.9
14	—	214.7 s	6.0	199.1 s	6.4
16	—	196.4 s	6.5	188.3 s	6.8

Comparison with Initial Implementation

A comparison of the speedup is shown in Figure 6.26, with the numerical values shown in Table 6.7. While this circuit does show some improvement with the new implementation, it is not as great as that in the previous example, giving about a 5% improvement overall.

Figure 6.26: Parallel scalability for the multiplier example.

6.4.5 Ring Oscillator example

For this final example, the ring oscillator circuit, from Section 4.3.4, is reconsidered with the proposed improvements.

A comparison of the speedup is shown in Figure 6.27, with the numerical values shown in Table 6.8. As can be seen, for this example, there was a modest improvement in performance for higher numbers of processors. This indicates that for this circuit, there was a small improvement in scalability. This in turn implies that the cost of the partitioning is significant enough that pipelining provides a noticeable benefit, despite the small size of the circuit.

Figure 6.27: Parallel scalability for the ring oscillator example.

Table 6.8: Performance results for the ring oscillator example

Partitions and CPUs	Standard Solver	Initial Proposed (Chapter 4)		Improved (Chapters 5+6)	
	Wall Time	Wall Time	Speedup	Wall Time	Speedup
1	89.2 s	—	—	—	—
2	—	50.9 s	1.8	52.6 s	1.7
4	—	33.0 s	2.7	30.9 s	2.9
6	—	23.5 s	3.8	24.2 s	3.7
8	—	19.4 s	4.6	19.1 s	4.7
10	—	18.8 s	4.8	17.2 s	5.2
12	—	16.6 s	5.4	16.3 s	5.5
14	—	16.6 s	5.4	15.4 s	5.8
16	—	17.2 s	5.2	16.0 s	5.6

6.5 Summary

In this chapter, the proposed parallel algorithm was refined and optimized for a specific parallel environment. Previously considered circuit examples were re-analysed to measure the impact of these improvements, along with the improvements of Chapter 5.

In the next chapter, the application of the proposed method to a hierarchical partitioning and the construction of a hybrid algorithm (using both node and branch tearing) is studied.

Chapter 7

Hierarchical Circuit Formulations Exploiting Node Tearing

So far, node tearing has only been applied to a flat partitioning scheme. In this chapter, its application to other simulator architectures is explored. First, the design of a simulator using hierarchical partitioning with node tearing is considered. Next, a second initial exploration is performed to the case where both node tearing and branch tearing are used to resolve different parts of a single hierarchy.

7.1 Hierarchical Formulation with Node Tearing

The possibility of considering the circuit hierarchy while simulating is interesting in the fact that it can be viewed as a form of partitioning. Instead of using purely graph-based techniques, we might be able to exploit the design's natural partitioning. To that end, an initial exploration into the behaviour of node tearing when used with a hierarchical partitioning is considered. For this preliminary study, the actual circuit hierarchy will not be used, instead we will rely on the characteristics of the graph partitioner to develop a hierarchy.

7.1.1 Recursive Node Tearing

Consider the circuit in Figure 7.1 that has been partitioned using recursive bisection. The original circuit was first divided into two partitions (partitions 1 and 2), and each of those were divided into two subpartitions, in turn (partitions 1A and 1B from partition 1, 2A and 2B from 2). There are a total of four partitions, which can be represented by the tree shown in Figure 7.2.

To solve this system, first, the top-level partitioning split is considered. Taking the formulation of Section 3.3, for *Step-a*, we have the following two equations, from (3.29):

$$\mathbf{Y}_1 \hat{\mathbf{x}}_1 = \mathbf{b}_1 \quad (7.1)$$

$$\mathbf{Y}_2 \hat{\mathbf{x}}_2 = \mathbf{b}_2 \quad (7.2)$$

Similarly, for *Step-b*, (3.30), we have:

$$\mathbf{Y}_1 \mathbf{\Gamma}_1 = \mathbf{\Xi}_1 \quad (7.3)$$

$$\mathbf{Y}_2 \mathbf{\Gamma}_2 = \mathbf{\Xi}_2 \quad (7.4)$$

The next challenge is to define each of these matrices. Clearly, each $\mathbf{Y}_i$ must be defined in terms of each subtree. It can be seen that each of these subtrees is simply a node tearing problem of the flat hierarchy case, so we can take the matrix formulation of (3.39). Therefore, we have:

Figure 7.1: Circuit partitioning example using recursive bisection.

Figure 7.2: Circuit partitioning example using recursive bisection, tree representation.

$$\mathbf{Y}_1 = \begin{bmatrix} \mathbf{Y}_{1A} & \mathbf{\Xi}_{1A} \\ & \mathbf{Y}_{1B} & \mathbf{\Xi}_{1B} \\ \mathbf{\Xi}_{1A}^\top & \mathbf{\Xi}_{1B}^\top & \mathbf{0} \end{bmatrix}, \quad (7.5)$$

$$\mathbf{Y}_2 = \begin{bmatrix} \mathbf{Y}_{2A} & \mathbf{\Xi}_{2A} \\ & \mathbf{Y}_{2B} & \mathbf{\Xi}_{2B} \\ \mathbf{\Xi}_{2A}^\top & \mathbf{\Xi}_{2B}^\top & \mathbf{0} \end{bmatrix}, \quad (7.6)$$

$$\mathbf{x}_1 = \begin{bmatrix} \mathbf{x}_{1A} \\ \mathbf{x}_{1B} \\ \boldsymbol{\gamma}_1 \end{bmatrix}, \quad (7.7)$$

$$\mathbf{x}_2 = \begin{bmatrix} \mathbf{x}_{2A} \\ \mathbf{x}_{2B} \\ \boldsymbol{\gamma}_2 \end{bmatrix}, \quad (7.8)$$

$$\mathbf{b}_1 = \begin{bmatrix} \mathbf{b}_{1A} \\ \mathbf{b}_{1B} \\ \mathbf{0} \end{bmatrix}, \quad \text{and} \quad (7.9)$$

$$\mathbf{b}_2 = \begin{bmatrix} \mathbf{b}_{2A} \\ \mathbf{b}_{2B} \\ \mathbf{0} \end{bmatrix}, \quad (7.10)$$

where each leaf subcircuit, when considered independently, are described by the MNA equations:

$$\mathbf{Y}_{1A} \mathbf{x}_{1A} = \mathbf{b}_{1A}, \quad (7.11)$$

$$\mathbf{Y}_{1B} \mathbf{x}_{1B} = \mathbf{b}_{1B}, \quad (7.12)$$

$$\mathbf{Y}_{2A} \mathbf{x}_{2A} = \mathbf{b}_{2A}, \quad \text{and} \quad (7.13)$$

$$\mathbf{Y}_{2B} \mathbf{x}_{2B} = \mathbf{b}_{2B}, \quad (7.14)$$

and each of the Ξ matrices is formulated as described in Section 3.1.2, for the appropriate level of hierarchy. For this example, Ξ_{1A} and Ξ_{1B} will describe the links between partitions 1A and 1B, and Ξ_1 and Ξ_2 describe the links between partition 1 (either 1A or 1B) and partition 2 (either 2A or 2B).

Finally, the entire system can be re-expressed in the form of (3.39), as follows:

$$\left[\begin{array}{ccc|ccc|c} \mathbf{Y}_{1A} & \cdot & \Xi_{1A} & \cdot & \cdot & \cdot & \cdot & \Xi_1 \\ \cdot & \mathbf{Y}_{1B} & \Xi_{1B} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \Xi_{1A}^T & \Xi_{1B}^T & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \mathbf{Y}_{2A} & \cdot & \Xi_{2A} & \cdot & \Xi_2 \\ \cdot & \cdot & \cdot & \cdot & \mathbf{Y}_{2B} & \Xi_{2B} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \Xi_{2A}^T & \Xi_{2B}^T & \cdot & \cdot & \cdot \\ \hline \Xi_1^T & \cdot & \cdot & \Xi_2^T & \cdot & \cdot & \cdot & \cdot \end{array} \right] \begin{bmatrix} \mathbf{x}_{1A} \\ \mathbf{x}_{1B} \\ \gamma_1 \\ \mathbf{x}_{2A} \\ \mathbf{x}_{2B} \\ \gamma_2 \\ \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{1A} \\ \mathbf{b}_{1B} \\ \mathbf{0} \\ \mathbf{b}_{2A} \\ \mathbf{b}_{2B} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \quad (7.15)$$

It was mentioned previously that the recursive formulation simply replaces the standard linear solution processes of *Step-a* and *Step-b* of Section 3.3 with a node-tearing-based implementation. To implement the overall process in a general fashion, we need two functions: “*Factorize and Solve*” and “*Solve*”. These functions serve to solve a linear system for multiple right-hand sides. For the first right-hand side to be solved, in this case the one in *Step-a*, the system must be first factorized, so “*Factorize and Solve*” is used. For *Step-b*, the same system is solved with many different right-hand sides, so the factorization can be re-used, and only “*Solve*” is needed.

Exploiting these new function definitions, the overall node-tearing solution process, or the “*Factorize and Solve*” function for a node-tearing node of the tree, using the same steps defined in Section 3.3, becomes:

Factorize and Solve for Node Tearing

Step a) Call “*Factorize and Solve*” for each child partition and store the result (i.e., solve $Y_j \hat{x}_j = b_j$ for each partition j)

Step b) For each column of $\hat{\Xi}_j$, call the j^{th} child partition’s “*Solve*” function and store the results (i.e., solve $Y_j \hat{\Gamma}_j = \hat{\Xi}_j$ reusing the previous factorization).

Step c) Construct the Z matrix, as before: $Z = -\sum_{j=1}^p \Xi_j^T P_j \hat{\Gamma}_j$.

Step d) Solve the partition interconnections, γ , using $Z\gamma = -\sum_{j=1}^p \Xi_j^T \hat{x}_j$.

Step e) Finally, update the original solution, first by solving $Y_j \Delta x_j = \Xi_j \gamma$ using the “*Solve*” function for the j^{th} child and then updating its original solution with $x_j = \hat{x}_j - \Delta x_j$ to give the final x_j .

For this function, the right-hand side to be solved is simply the vertical concatenation

of the column vectors $\mathbf{b}_j$, and the solution is the same using the $\mathbf{x}_j$ vectors.

When another right-hand side is to be solved, any of the steps described above that do not depend on the contents of this vector can be simply skipped, using their stored results:

Solve for Node Tearing

Step a) Call “*Solve*” for each child partition and store the result (i.e., solve $\mathbf{Y}_j \hat{\mathbf{x}}_j = \mathbf{b}_j$ for each partition j , reusing the factorization from the previous call to “*Factorize and Solve*”)

Step b) Solve the partition interconnections, $\boldsymbol{\gamma}$, using $\mathbf{Z}\boldsymbol{\gamma} = -\sum_{j=1}^p \boldsymbol{\Xi}_j^T \hat{\mathbf{x}}_j$, reusing the previous factorization of $\mathbf{Z}$.

Step c) Update the original solution, first by solving $\mathbf{Y}_j \Delta \mathbf{x}_j = \boldsymbol{\Xi}_j \boldsymbol{\gamma}$ using the “*Solve*” function for the j^{th} child and then updating its original solution with $\mathbf{x}_j = \hat{\mathbf{x}}_j - \Delta \mathbf{x}_j$ to give the final $\mathbf{x}_j$.

For each leaf of the tree (i.e., unpartitioned child), the “*Factorize and Solve*” function is simply a standard LU factorization followed by a forward/backward substitution. The corresponding “*Solve*” function is simply a forward/backward substitution.

In this way, any possible circuit hierarchy tree can be solved using a node tearing formulation. In the actual implementation, the improvements presented in Sections 5 and 6 were exploited to increase parallelism and performance. Examples demonstrating the performance of this approach are given in Section 7.3.

Given the flexibility of this approach, any kind of partitioning can be included in the tree. In the next section, a hybrid formulation that includes both node tearing and branch tearing in the same tree will be considered.

7.2 Hierarchical Simulation with Hybrid

Implementation of Node and Branch Tearing

In the previous section, a recursive solution procedure was applied to solve a circuit partitioned hierarchically using node tearing. In this section, this method is extended to the case where some of the partitions are developed with branch tearing.

It is interesting to study this case as there may be circumstances where the performance of branch tearing might be interesting to compare to that of node tearing. As shown in Section 4.2, for small numbers of links, the complexity of domain decomposition is comparable to that of node tearing. When there are a small number of child partitions, the benefits of the pipelined scheduler from Chapter 6 will not be great. In these cases, it might be interesting to use branch tearing if it results in a border whose size is reduced.

To integrate branch tearing into the previously developed method, all that is needed is two functions: “*Factorize and Solve*” and “*Solve*”, designed this time for branch tearing. Take the bordered-block-diagonal form of (2.12), shown here for reference:

$$\begin{bmatrix} Y_1 & & & F_1 \\ & Y_2 & & F_2 \\ & & \ddots & \vdots \\ & & & Y_p & F_p \\ E_1 & E_2 & \dots & E_p & Y_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \\ x_0 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \\ b_0 \end{bmatrix}. \quad (7.16)$$

The solution of this system using the method of Section 2.3 gives the following functions:

Factorize and Solve for Branch Tearing

Step a) Call “*Factorize and Solve*” for each child partition and store the result (i.e., solve $Y_j \hat{x}_j = b_j$ for each partition j)

Step b) For each column of F_j , call the j^{th} child partition’s “*Solve*” function and store the results (i.e., solve $Y_j \Gamma_j = F_j$ reusing the previous factorization).

Step c) Construct the Y matrix: $Y = Y_0 - \sum_{j=1}^p E_j \Gamma_j$.

Step d) Solve the partition interconnections, x_0 , using $Y x_0 = b_0 - \sum_{j=1}^p E_j \hat{x}_j$.

Step e) Finally, update the original solution, first by solving $Y_j \Delta x_j = F_j x_0$ using the “*Solve*” function for the j^{th} child and then updating its original solution with $x_j = \hat{x}_j - \Delta x_j$ to give the final x_j .

Solve for Branch Tearing

Step a) Call “*Solve*” for each child partition and store the result (i.e., solve $Y_j \hat{x}_j = b_j$ for each partition j , reusing the factorization from the previous call to “*Factorize and Solve*”)

Step b) Solve the partition interconnections, x_0 , using $Y x_0 = b_0 - \sum_{j=1}^p E_j \hat{x}_j$, reusing the previous factorization of Y .

Step c) Update the original solution, first by solving $Y_j \Delta x_j = F_j x_0$ using the “*Solve*” function for the j^{th} child and then updating its original solution with $x_j = \hat{x}_j - \Delta x_j$ to give the final x_j .

In the next section, the performance of both of these hierarchical methods is examined by reconsidering the benchmark examples used in the previous sections.

7.3 Numerical Examples

In this section, the same four circuit examples are reconsidered using each of the two modified algorithms. First, each is run in the pure hierarchical mode, using only node tearing. The partitioning is effectively the same as in Section 6.4, but the partitions are arranged into a binary tree. As the partitioning was done with hMETIS, which functions using recursive bisection [19], arranging the tree in this way extracts the bisections done by hMETIS and thus should give a good link structure. This transformation is illustrated for an example having 8 partitions in Figure 7.3.

Next, each example was run in the hybrid hierarchical mode. For this, the same partitioning is once again used. It is transformed into a two-level tree. The partitions are grouped into pairs which are resolved using branch tearing, and the pairs are then resolved into the overall system using node tearing. This transformation is illustrated in Figure 7.4.

Figure 7.3: Circuit partitioning transformation to hierarchical recursive bisection form.

Figure 7.4: Circuit partitioning transformation to hybrid node-tearing/branch-tearing form.

Figure 7.5: Parallel scalability for the DSP example.

7.3.1 DSP example

In this experiment, the DSP example, last considered in Section 6.4.2, is run again with the two hierarchical modes developed in this section. The speedup graph is shown in Figure 7.5, with the tabulated results in Table 7.1. The ‘flat’ results used as a reference are those from Section 6.4.2. It can be seen that for this example, neither of the hierarchical modes considered are bringing any improvements.

Table 7.1: Performance results for the DSP example

Partitions and CPUs	Flat (Chapter 6)		Hierarchical Bisected		Hybrid Node + Branch Tearing	
	Wall Time	Speedup	Wall Time	Speedup	Wall Time	Speedup
2	618.8 s	1.6	605.9 s	1.6	737.7 s	1.4
4	322.5 s	3.1	333.6 s	3.0	394.4 s	2.5
6	240.0 s	4.2	266.9 s	3.7	304.1 s	3.3
8	189.1 s	5.3	205.0 s	4.9	238.4 s	4.2
10	161.2 s	6.2	194.8 s	5.1	192.9 s	5.2
12	139.5 s	7.2	159.3 s	6.3	182.7 s	5.5
14	119.9 s	8.3	139.4 s	7.2	153.0 s	6.5
16	116.8 s	8.5	141.9 s	7.0	135.3 s	7.4

7.3.2 Dual SRAM example

For this example, the Dual SRAM example, considered in Section 6.4.3, is run again with the same modified partitioning as for the last example. The resulting speedup graph is shown in Figure 7.6, with the corresponding tabulated results in Table 7.2.

Once again, it can be seen that the hierarchical modes are not bringing additional gains. The hierarchical bisected mode is showing additional overhead, as the scalability is much worse.

Figure 7.6: Parallel scalability for the dual SRAM example.

Table 7.2: Performance results for the dual SRAM example

Partitions and CPUs	Flat (Chapter 6)		Hierarchical Bisected		Hybrid Node + Branch Tearing	
	Wall Time	Speedup	Wall Time	Speedup	Wall Time	Speedup
2	461.8 s	1.9	464.6 s	1.9	589.0 s	1.5
4	277.9 s	3.1	286.6 s	3.0	311.8 s	2.8
6	190.0 s	4.5	262.1 s	3.3	251.0 s	3.4
8	147.6 s	5.8	206.8 s	4.2	194.6 s	4.4
10	124.8 s	6.9	200.8 s	4.3	160.5 s	5.4
12	108.2 s	8.0	190.8 s	4.5	135.8 s	6.3
14	100.4 s	8.6	167.0 s	5.2	124.8 s	6.9
16	93.68 s	9.2	166.8 s	5.2	109.9 s	7.8

Figure 7.7: Parallel scalability for the multiplier example.

7.3.3 Array Multiplier example

For this example, the Array Multiplier example, last considered in Section 6.4.4, is run again with the same modified partitioning scheme. The resulting speedup graph is shown in Figure 7.7, with the corresponding tabulated results in Table 7.3.

Much like in the last example, the hierarchical modes are not bringing additional gains. Once again, the hierarchical bisected mode is showing additional overhead, as the scalability is much worse.

Table 7.3: Performance results for the multiplier example

Partitions and CPUs	Flat (Chapter 6)		Hierarchical Bisected		Hybrid Node + Branch Tearing	
	Wall Time	Speedup	Wall Time	Speedup	Wall Time	Speedup
2	897.1 s	1.4	898.7 s	1.4	1031 s	1.2
4	498.2 s	2.6	582.9 s	2.2	596.6 s	2.1
6	365.4 s	3.5	538.1 s	2.4	444.3 s	2.9
8	279.2 s	4.6	421.8 s	3.0	348.6 s	3.7
10	229.6 s	5.6	423.0 s	3.0	292.9 s	4.4
12	215.8 s	5.9	390.9 s	3.3	252.0 s	5.1
14	199.1 s	6.4	293.7 s	4.4	238.4 s	5.4
16	188.3 s	6.8	327.9 s	3.9	206.1 s	6.2

7.3.4 Ring Oscillator example

For this final example, the ring oscillator circuit, last considered in Section 6.4.5, is run again with the same modified partitioning schemes. The resulting speedup graph is shown in Figure 7.8, with the corresponding tabulated results in Table 7.4.

In this circuit, the additional overhead resulting from the modified partitions is seen to be not as significant as in the others, as the performance is closer to that of the results from Chapter 6. However, there is still a degradation.

In the following section, some interpretations for these results are given, as well as some recommendations for future work to improve on them.

Figure 7.8: Parallel scalability for the ring oscillator example.

Table 7.4: Performance results for the ring oscillator example.

Partitions and CPUs	Flat (Chapter 6)		Hierarchical Bisected		Hybrid Node + Branch Tearing	
	Wall Time	Speedup	Wall Time	Speedup	Wall Time	Speedup
2	52.57 s	1.7	51.58 s	1.7	63.59 s	1.4
4	30.89 s	2.9	30.63 s	2.9	38.25 s	2.3
6	24.24 s	3.7	25.44 s	3.5	28.56 s	3.1
8	19.14 s	4.7	22.07 s	4.0	24.04 s	3.7
10	17.21 s	5.2	20.46 s	4.4	21.08 s	4.2
12	16.28 s	5.5	20.68 s	4.3	18.71 s	4.8
14	15.37 s	5.8	18.63 s	4.8	17.62 s	5.1
16	15.97 s	5.6	19.48 s	4.6	18.61 s	4.8

7.4 Interpretation and Future Directions

The results of the hierarchical formulation described in this section failed to yield performance or scalability improvements compared to the flat algorithm implemented in Chapter 6. Based on the results seen, the following observations are made.

1. For the hierarchical formulation using node tearing, the partitioning used for the flat simulation and the hierarchical simulation were identical. This means that the number of links were the same in both cases. The hierarchical approach differs only in that the $\mathbf{Z}$ (interconnection) matrix has been partitioned to be solved in a parallel manner. Because each node of the tree only has 2 children, the pipelining method of Section 6.1 is not applicable. This implies that, for equivalent sets of partitions obtained using graph partitioning methods, the pipelined approach to solving the interconnection matrix provides better results than solving it in parallel using a hierarchy. More work is needed to show if this is the case for other partitioning schemes.
2. For the hybrid node-tearing/branch-tearing formulation, the performance on average was better than the pure hierarchical node-tearing formulation. The scalability is likely better than that of the previous hierarchical formulation due to the fact that some pipelining was used while solving the partition interconnections. The branch tearing method did not seem to provide any benefit, however, this is not conclusive since no criteria were defined to determine whether branch tearing or node tearing would be best for the bottommost partition pairs. This is due to the fact that such an analysis would be quite involved and is beyond the scope of the present work.

Despite the performance of this initial analysis, there is still some interest in studying these formulations. The hierarchical form, for example, can permit the application of methods that depend on a hierarchy, such as a multirate formulation where different sections of the tree will be operating with a different timestep size.

To address the problems observed so far, some future areas for research are summarized in point 5 of Section 9.2.

7.5 Summary

In this chapter, the proposed parallel simulation algorithm based on node tearing was extended to the case of hierarchical partitioning, which exploits the existing structure of the overall system to improve the quality of the partitioning and to ideally provide additional parallelism. This hierarchical approach was then further extended by exploiting domain decomposition for the solution of the lowest levels of the hierarchy tree, which could theoretically yield performance improvements for specific cases.

For the currently considered circuit examples and simulator architecture, no improvement in simulation performance or scalability was observed. However, these methods are still interesting as it allows for the exploration of other techniques in future research.

Chapter 8

Parallel Simulation of Massively Coupled Interconnect Networks

In this chapter, a specific class of circuits, massively coupled interconnected transmission line networks, is considered. The motivation behind the need to simulate these networks is first presented, followed by a review of the physics behind these circuits. Finally, a parallel simulation algorithm suitable for this specific class of circuits, yielding higher efficiency (compared to the general parallel circuit simulation algorithms developed in the previous chapters), is developed and validated with numerical examples.

8.1 Motivation

As the operating frequencies of systems continue to rise, the accurate characterization and simulation of interconnect networks becomes a critical task. Formerly neglected effects such as ringing, delay, distortion, reflections and crosstalk can significantly influence the performance of a system. Without considering these high-speed effects during the design process through accurate and complete simulations, a fabricated device may not work as intended.

One of the difficulties in simulating a network of high-speed interconnects is that distributed interconnect models only have a direct representation in the frequency domain. Other nonlinear devices such as transistors can only be described in the time domain. However, a simultaneous formulation of both frequency- and time-domain equations cannot be handled directly by ordinary differential equation solvers found in traditional circuit simulators. Several methods have been proposed in literature to transform the interconnect models into a set of coupled ordinary differential equations that circuit simulators can accept [36–42].

The coupling between the lines is one of the major causes of the high computational cost of simulating large interconnect networks. The first component of this cost is that of the interconnect model. For example, a 64-line system requires the approximation of 8320 transfer functions. Each of these approximations must not only be accurate, but the resulting functions must preserve the passivity and the causality of the overall macromodel [43]. Once the model has been created, it is converted into an equivalent circuit and passed to a circuit simulator. It was shown in [13] that the average cost of simulating an n_L coupled line circuit is proportional to n_L^β where $3 \leq \beta \leq 4$. This results in a prohibitively time-consuming simulation task compared to the simple case of simulating a single line.

To more efficiently handle this problem, a new method of simulating these large coupled networks by employing the technique of waveform relaxation with transverse partitioning was introduced in [13]. The problem of simulating n_L coupled lines was effectively partitioned into a set of n_L subproblems each consisting of a single line. Sources are added to each of these lines to capture the coupling effects and the values of these sources are resolved using iterations. With this partitioning method, the simulation cost grows

linearly as n_L increases. Using this form of partitioning also ensures fast convergence of the waveform relaxation methods. It can be noted that each of the single-line subproblems depends only on the data from the previous iteration of the simulation.

By combining the techniques of transverse partitioning and waveform relaxation, a parallel algorithm has been developed and is presented in the following sections. To obtain maximum parallelism, a second algorithm is presented that included additional partitioning in the time domain. For each of these, an implementation tuned to a specific system architecture has been performed.

8.2 Multiconductor Transmission Line Equations and Transverse Partitioning

Consider the coupled interconnect system shown in Figure 8.1. This system is described by the set of partial differential equations known as the Telegrapher's Equations, as follows [44]:

$$\begin{aligned}\frac{\partial}{\partial x} \mathbf{v}(x, t) &= -\mathbf{R}\mathbf{i}(x, t) - \mathbf{L} \frac{\partial}{\partial x} \mathbf{i}(x, t), \\ \frac{\partial}{\partial x} \mathbf{i}(x, t) &= -\mathbf{G}\mathbf{v}(x, t) - \mathbf{C} \frac{\partial}{\partial x} \mathbf{v}(x, t),\end{aligned}\tag{8.1}$$

where $\mathbf{R} \in \mathbb{R}^{n_L \times n_L}$, $\mathbf{L} \in \mathbb{R}^{n_L \times n_L}$, $\mathbf{C} \in \mathbb{R}^{n_L \times n_L}$ and $\mathbf{G} \in \mathbb{R}^{n_L \times n_L}$ are the per-unit-length parameters which characterize the transmission line system. $\mathbf{v}(x, t) \in \mathbb{R}^{n_L}$ and $\mathbf{i}(x, t) \in \mathbb{R}^{n_L}$ represent the voltages and currents seen along each of the n_L coupled lines.

To accomplish faster simulation of such a coupled line system, transverse partitioning [13] is applied wherein each line is examined individually. The first step is to isolate

Figure 8.1: n_L coupled transmission line system.

the equations for the j^{th} line from (8.1):

$$\begin{aligned}
 \frac{\partial v_j}{\partial x} &= -R_{jj}i_j - L_{jj}\frac{\partial i_j}{\partial t} - \sum_{\substack{k=1 \\ k \neq j}}^{n_L} \left(R_{jk}i_k + L_{jk}\frac{\partial i_k}{\partial t} \right), \\
 \frac{\partial i_j}{\partial x} &= -\sum_{k=1}^{n_L} G_{jk}v_j - \sum_{k=1}^{n_L} C_{jk}\frac{\partial v_j}{\partial t} \\
 &\quad - \sum_{\substack{k=1 \\ k \neq j}}^{n_L} \left(G_{jk}(v_j - v_k) + C_{jk}\frac{\partial}{\partial t}(v_j - v_k) \right).
 \end{aligned} \tag{8.2}$$

Separating the coupling terms, (8.2) can be re-written as

$$\begin{aligned}
 \frac{\partial v_j}{\partial x} &= -R_{jj}i_j - L_{jj}\frac{\partial i_j}{\partial t} + e_j(x, t), \\
 \frac{\partial i_j}{\partial x} &= -\sum_{k=1}^{n_L} G_{jk}v_j - \sum_{k=1}^{n_L} C_{jk}\frac{\partial v_j}{\partial t} + q_j(x, t),
 \end{aligned} \tag{8.3}$$

where the new coupling terms $e_j(x, t)$ and $q_j(x, t)$ are defined as

$$e_j(x, t) = - \sum_{\substack{k=1 \\ k \neq j}}^{n_L} \left(R_{jk} i_k + L_{jk} \frac{\partial i_k}{\partial t} \right), \quad (8.4)$$

$$q_j(x, t) = - \sum_{\substack{k=1 \\ k \neq j}}^{n_L} \left(G_{jk} (v_j - v_k) + C_{jk} \frac{\partial}{\partial t} (v_j - v_k) \right). \quad (8.5)$$

The coupling effects are added to each line as new sources ($\tilde{v}_j(d, t)$, $\tilde{i}_j(d, t)$) at the far end as shown in Figure 8.2. The value of the sources is given by [13]

$$\begin{bmatrix} \tilde{v}_j(d, t) \\ \tilde{i}_j(d, t) \end{bmatrix} = \mathcal{F}^{-1} \left\{ \int_0^d e^{F_j(d-\eta)} \Psi_j(\eta, s) d\eta \right\}, \quad (8.6)$$

where

$$F_j = \begin{bmatrix} 0 & -R_{jj} - sL_{jj} \\ \sum_{k=1}^{n_L} (-G_{jk} - sC_{jk}) & 0 \end{bmatrix}, \quad (8.7)$$

$$\begin{aligned} \Psi_j(x, s) = \sum_{\substack{k=1 \\ k \neq j}}^{n_L} & \left(\begin{bmatrix} 0 & R_{jk} - sL_{jk} \\ -G_{jk} - sC_{jk} & 0 \end{bmatrix} \times \right. \\ & \left. \begin{bmatrix} V_j(x, s) - V_k(x, s) \\ I_k(x, s) \end{bmatrix} \right). \end{aligned} \quad (8.8)$$

Note that (8.3) still represents a coupled system. To achieve faster simulations, the lines must be fully decoupled so that they can be simulated as individual lines.

To decouple (8.3), initial guesses for $v_j(x, t)$ and $i_j(x, t)$ are assumed and the corresponding $\tilde{v}_j(d, t)$ and $\tilde{i}_j(d, t)$ are computed using (8.6)-(8.8). The solutions for $v_j(x, t)$ and $i_j(x, t)$ are updated using waveform-relaxation iterations [12, 45], by evaluating (8.6)-(8.8). For example, assuming that the solution of the r^{th} waveform relaxation iteration yields $e_j^{(r)}(x, t)$

and $q_j^{(r)}(x, t)$, (8.3) for the $(r + 1)^{\text{th}}$ iteration can be expressed as

$$\begin{aligned} \frac{\partial v_j^{(r+1)}}{\partial x} &= -R_{jj}i_j^{(r+1)} - L_{jj}\frac{\partial i_j^{(r+1)}}{\partial t} + e_j^{(r)}(x, t), \\ \frac{\partial i_j^{(r+1)}}{\partial x} &= -\sum_{k=1}^m G_{jk}v_j^{(r+1)} - \sum_{k=1}^m C_{jk}\frac{\partial v_j^{(r+1)}}{\partial t} \\ &\quad + q_j^{(r)}(x, t), \end{aligned} \tag{8.9}$$

where the superscript (r) represents the r^{th} iteration. Thus, each subcircuit in Figure 8.2 is solved independently and iteratively, while updating the sources $\tilde{v}_j(d, t)$ and $\tilde{i}_j(d, t)$ after each iteration. Convergence is usually obtained in 2-3 iterations even for coupling coefficients as high as 0.75. This fast convergence can be easily explained from the fact that the capacitance matrix $\mathbf{C}$ is diagonally dominant and the largest elements of the inductance matrix $\mathbf{L}$ are located on the diagonal, with the magnitude of the off-diagonal elements rapidly decreasing as the distance between the lines increases [44]. This process can also be adapted to different transmission line models, as described in detail in [13, 46].

Figure 8.2: Partitioned system for waveform relaxation.

8.3 Development of the Proposed Parallel Simulation

Algorithm

To perform a fast analysis of large coupled systems on a parallel platform, two potential alternatives are presented.

8.3.1 Parallel Simulation using Physical Partitioning (PS-PP)

This approach is based on physical transverse partitioning (described in Section 8.2) of the interconnect circuit. Correspondingly, each of the lines are simulated independently on a different processor during each WR iteration. Multiple independent simulations can be executed in parallel. After simulating each line, the corresponding relaxation sources are updated to prepare for the next iteration. Once this is complete for all of the lines, the next

Figure 8.3: Illustration of task flow requirements for the PS-PP approach.

iteration can begin.

For an n_L coupled line system, this means that there can be a maximum of n_L tasks executing in parallel. In other words, the maximum parallelism of this approach is n_L . In the case where the tasks do not divide equally among the available processors, some of the processors will be idle for a portion of the time, resulting in a loss of speedup.

To demonstrate the effect of this, each task is identified with the pair (r, l) where

- $r \in \mathbb{N}$, $1 \leq r \leq n_I$, where r is the iteration to which the task belongs, n_I is the total number of iterations, and
- $l \in \mathbb{N}$, $1 \leq l \leq n_L$, where l is the line being simulated in this task.

For a given iteration r , before any task (r, l) can be executed, all simulations for the previous iteration must have been completed. In other words, all tasks $(r-1, 1 \leq l \leq n_L)$ must be complete. This is illustrated in Figure 8.3. The concept of parallel processing using physical partitioning is demonstrated in the following illustrative example.

Illustrative Example 7. Consider a problem with five lines ($n_L = 5$) and two relaxation iterations ($n_I = 2$), run on a system with four processors ($n_P = 4$). For this example, to simplify the discussion, each of the lines will be identical with the same sources and

terminations. Cases where this is not true will be considered later, in Section 8.3.3. The CPU utilisation during each execution stage for this example is shown in Table 8.1. Note that three of the processors remain idle during execution stages 2 and 4.

Table 8.1: Execution stages for Illustrative Example 7 demonstrating the PS-PP approach
($n_I = 2, n_L = 5, n_P = 4$)

Execution Stage	Task			
	CPU 1	CPU 2	CPU 3	CPU 4
1	(1, 1)	(1, 2)	(1, 3)	(1, 4)
2	(1, 5)	<i>idle</i>	<i>idle</i>	<i>idle</i>
3	(2, 1)	(2, 2)	(2, 3)	(2, 4)
4	(2, 5)	<i>idle</i>	<i>idle</i>	<i>idle</i>

For this example, the processors are clearly underutilized. Three of the processors are idle 50% of the time, giving an average CPU utilisation of 62.5%.

Using the above observations, the following lemma is defined for obtaining a general expression for the CPU utilisation efficiency.

Lemma 5. *For the physical partitioning scheme, the CPU utilisation efficiency, η , is given by*

$$\eta = \frac{n_L}{n_P} \left\lceil \frac{n_L}{n_P} \right\rceil^{-1}, \quad (8.10)$$

where n_P is the number of available processors and $\lceil x \rceil^\dagger$ represents the ceiling operator applied to x .

Proof. The proof is derived using the following observations:

- a. Each iteration of the relaxation simulation must be executed sequentially; that is, one iteration must complete before the next iteration can begin.

[†] $\lceil x \rceil$, the ceiling operator applied to x , means to round up x to the next largest integer.

- b. Each iteration consists of a total maximum of n_L individual tasks (the simulations for n_L individual lines).
- c. When n_P processors are available, a total of n_P tasks can be processed at once.
- d. The tasks for each iteration are processed n_P tasks at a time, until fewer than n_P tasks remain. At that point, the remaining (fewer than n_P) tasks will be processed, in which case some processors may remain idle.

Based on the above observations, it can be concluded that:

- i. The number of execution stages is therefore equal to $\lceil n_L n_P^{-1} \rceil$.
- ii. The maximum number of tasks that could be executed during these stages is equal to $n_P \lceil n_L n_P^{-1} \rceil$.

Therefore, the overall efficiency is equal to $n_L (n_P \lceil n_L n_P^{-1} \rceil)^{-1}$. □

In the case where n_L divides evenly into n_P , $\eta = 100\%$. In any other case, there will be a loss of efficiency. To overcome this deficiency, a novel approach is presented in the following section.

8.3.2 Parallel Simulation using Physical and Time-Domain Partitioning (PS-PTP)

In this approach, a parallel simulation algorithm based on both physical and time-domain partitioning of the simulation task is developed. In the previous PS-PP approach, the cause for the loss of efficiency is due to the fact that each iteration needs to be performed sequentially. To overcome this difficulty, a means to start the next iteration before the

previous iteration is completed, is needed. To achieve this, the individual line simulation tasks are further divided by splitting the entire time duration into n_B time blocks. These blocks are treated as individual simulation tasks and can be scheduled more efficiently among the available processors. It is to be noted that time windowing has been previously suggested for accelerating the convergence of WR iterations [12]. However, in the context of this work, time partitioning has been adopted to increase the parallelism during analysis.

For the purpose of illustration, task identifiers for this partitioning scheme are defined as the triplet (r, l, b) , with r for the iteration, l for the line, and b for the time block. To execute any given task, certain data from the solution of other tasks are required. These are specified by the following.

1. For the given line and iteration, to proceed to the next time block, the corresponding iteration for the same line for the previous time block must have been completed. In other words, when $b > 1$, then the task $(r, l, b - 1)$ must be completed prior to executing the task (r, l, b) .
2. For the given line and time block, to proceed to the next iteration, the previous iteration for all lines for the same time block must have been completed. In other words, when $r > 1$, then the tasks $(r - 1, 1 \leq l \leq n_L, b)$ must be completed prior to executing the tasks $(r, 1 \leq l \leq n_L, b)$.

These two requirements are illustrated in Figure 8.4.

The main advantage of this approach is that the execution of an iteration can start before the previous iteration is complete for the entire time span of interest. As soon as the first time block of an iteration has been completed for all lines, the next iteration for that time block can begin. The above concepts are illustrated through an example, below.

Figure 8.4: Illustration of task flow requirements for the PS-PTP approach.

Illustrative Example 8. In this example, the experiment from Table 8.1 is reconsidered, with the time span partitioned into 4 blocks, so $n_B = 4$. The resulting task scheduling is shown in Table 8.2. As can be seen, the CPU utilisation efficiency increases to 100% in this case.

Table 8.2: Execution stages for Illustrative Example 8 demonstrating the PS-PTP approach, $n_L \geq n_P$ ($n_I = 2, n_L = 5, n_B = 4, n_P = 4$)

Execution Stage	Task			
	CPU 1	CPU 2	CPU 3	CPU 4
1	(1, 1, 1)	(1, 2, 1)	(1, 3, 1)	(1, 4, 1)
2	(1, 5, 1)	(1, 1, 2)	(1, 2, 2)	(1, 3, 2)
3	(1, 4, 2)	(1, 5, 2)	(2, 1, 1)	(2, 2, 1)
4	(2, 3, 1)	(2, 4, 1)	(2, 5, 1)	(1, 1, 3)
5	(1, 2, 3)	(1, 3, 3)	(1, 4, 3)	(1, 5, 3)
6	(2, 1, 2)	(2, 2, 2)	(2, 3, 2)	(2, 4, 2)
7	(2, 5, 2)	(1, 1, 4)	(1, 2, 4)	(1, 3, 4)
8	(1, 4, 4)	(1, 5, 4)	(2, 1, 3)	(2, 2, 3)
9	(2, 3, 3)	(2, 4, 3)	(2, 5, 3)	(2, 1, 4)
10	(2, 2, 4)	(2, 3, 4)	(2, 4, 4)	(2, 5, 4)

Noting observations from the above example and discussions, the following lemma is defined for obtaining an expression for the CPU utilisation efficiency for the general case of $n_L \geq n_P$.

Lemma 6. *For the PS-PTP approach, the CPU utilisation efficiency, η , is given by*

$$\eta = \frac{n_L n_B n_I}{n_P} \left\lceil \frac{n_L n_B n_I}{n_P} \right\rceil^{-1} ; \quad n_L \geq n_P. \quad (8.11)$$

Proof. The proof is derived using the following observations:

- a. For the first iteration of the first time block, n_P tasks are processed at a time until $(n_L \bmod n_P)^{\ddagger}$ tasks remain. As the tasks for the lines are processed in ascending numerical order, the final tasks to be processed from the first iteration of the first time block will be the highest numbered lines, corresponding to lines $(n_L + 1 - (n_L \bmod n_P))$ to n_L . These tasks will occupy the first $(n_L \bmod n_P)$ processors of the subsequent execution stage.
- b. Therefore, $n_P - (n_L \bmod n_P)$ processors are available for processing tasks from the first iteration of the second time block. Using these available processors, for the first iteration of the second time block, the initial lines to be processed are the lowest numbered lines, ranging from line 1 to line $(n_P - (n_L \bmod n_P))$. It is to be noted that, the first iteration of the second time block for a given line can be executed as soon as the first iteration of the first time block is complete for that line (as illustrated in Figure 8.4). Using these observations, and noting that $n_L \geq n_P$, all of the scheduled tasks for the second time block can be executed at this point since the highest numbered line from the second block, $(n_P - (n_L \bmod n_P))$, is less than the lowest numbered line from the first block, $(n_L + 1 - (n_L \bmod n_P))$.
- c. This logic can be applied to all tasks for all iterations, time blocks, and lines.

^{$\ddagger$} $(a \bmod b)$, the modulo operator, is equal to the remainder of the integer division of a by b . For example, $(11 \bmod 4) = 3$.

Based on the above observations, it can be concluded that:

- i. The total number of tasks is $n_I n_B n_L$, and these will be processed n_P tasks at a time.
- ii. The number of work stages required to process these tasks is $\lceil n_I n_B n_L n_P^{-1} \rceil$.
- iii. The maximum number of tasks that can be executed in this number of execution stages is $n_P \lceil n_I n_B n_L n_P^{-1} \rceil$.

Therefore, the CPU efficiency when $n_L \geq n_P$ is given by $n_I n_B n_L (n_P \lceil n_I n_B n_L n_P^{-1} \rceil)^{-1}$. $\square$

To minimize the idle CPU time, n_B is chosen to be equal to n_P . In this case, the efficiency will become $n_I n_L \lceil n_I n_L \rceil^{-1}$ which is equal to 100%, as was the case in the example of Table 8.2. It is to be noted that without the time domain partitioning, the efficiency would have been only 62.5%, as shown in Table 8.1.

In the case where $n_L < n_P$, this partitioning scheme can still improve efficiency, however, some idle time is necessary. This idle time will be seen in the initial and final execution stages. To illustrate why this idle time may be needed for such cases, the following example is considered.

Illustrative Example 9. In this example, consider $n_L = 3$, $n_P = 4$, $n_I = 2$ and $n_B = 4$. The scheduling for this case is shown in Table 8.3.

As can be seen, during the first execution stage, only n_L tasks are available, those of the first iteration for the first time block. The first iteration for the second time block cannot begin until at least the first time block for the first line is complete. This will not be until the beginning of the second execution stage, so there will be idle processors during this stage (stage 1). A similar effect can be seen during the final execution stages (stages 6 and 7).

Table 8.3: Execution stages for Illustrative Example 9 demonstrating the PS-PTP approach, $n_L < n_P$
 $(n_I = 2, n_L = 3, n_B = 4, n_P = 4)$

Execution Stage	Task			
	CPU 1	CPU 2	CPU 3	CPU 4
1	(1, 1, 1)	(1, 2, 1)	(1, 3, 1)	<i>idle</i>
2	(1, 1, 2)	(1, 2, 2)	(1, 3, 2)	(2, 1, 1)
3	(2, 2, 1)	(2, 3, 1)	(1, 1, 3)	(1, 2, 3)
4	(1, 3, 3)	(2, 1, 2)	(2, 2, 2)	(2, 3, 2)
5	(1, 1, 4)	(1, 2, 4)	(1, 3, 4)	(2, 1, 3)
6	(2, 2, 3)	(2, 3, 3)	(2, 1, 4)	<i>idle</i>
7	(2, 2, 4)	(2, 3, 4)	<i>idle</i>	<i>idle</i>

The total amount of idle time can be minimized by increasing the number of time blocks. This has the effect of shortening the task execution times. However, since some communication occurs between execution stages to update the relaxation sources, having too many time blocks can increase the cost of this communication. Therefore, a tradeoff needs to be considered between the amount of idle time and the cost of communication. Due to this, setting $n_B = n_P$ is still a good choice for the case where $n_L < n_P$, as illustrated in Table 8.3.

Without time domain partitioning (i.e., PS-PP approach), the efficiency given by (8.10) would have been 75%. On the other hand, for this example, the time domain partitioning increases the efficiency to 85.7%. Idle time is still present, but significantly reduced.

A pseudocode which implements the PS-PTP partitioning and the related parallel scheduling algorithms is given in Figures 8.5-8.7.

```

1 { Apply transverse partitioning to a coupled interconnect network (with  $n_L$  lines) to separate
2   the network into individual lines. }
3 { Divide the time domain of the solution into  $n_B$  blocks, where  $n_B = n_P$ , and  $n_P$  is the
4   number of processors in the system. }
5 let work queue  $\leftarrow$  a queue of work tasks to be executed, initially empty
6 let task results  $\leftarrow$  a table of simulation results, initially empty,
7   corresponding to each task
8 for  $r = 1$  to  $n_I$  (the number of iterations needed)
9   for  $b = 1$  to  $n_B$ 
10    for  $l = 1$  to  $n_L$ 
11      append task  $(r, l, b)$  to end of work queue
12    end for
13  end for
14 end for
15 while length of work queue  $> 0$ 
16   for  $p = 1$  to  $n_P$ 
17     if length of work queue  $> 0$ 
18       let current task  $\leftarrow$  head of work queue
19       let result  $\leftarrow$  call task_flow_verify(current task, task results)
20       if result = true
21         let task results  $\leftarrow$  call perform_wr_iteration(current task, task results)
22         on processor  $p$  in background
23         remove head of work queue
24       else
25         break out of for loop
26       end if
27     end if
28   end for
29   wait for processors to finish processing tasks
30 end while

```

Figure 8.5: Pseudocode for the proposed PS-PTP algorithm.

```

1 function task_flow_verify(current task, task results)
2   let  $(r, l, b) \leftarrow$  current task
3   if  $b > 1$ 
4     if task results does not contain results for task  $(r, l, b - 1)$ 
5       return false
6     end if
7   end if
8   if  $r > 1$ 
9     for  $j = 1$  to  $n_L$ 
10      if task results does not contain results for  $(r - 1, j, b)$ 
11        return false
12      end if
13    end for
14  end if
15  return true
16 end function

```

Figure 8.6: Pseudocode for the task flow verification for the PS-PTP algorithm.

```

1 function perform_wr_iteration(current task, task results)
2   let  $(r, l, b) \leftarrow$  current task
3   if  $r > 1$ 
4     load results for  $(r - 1, 1 < l < n_L, b)$  from task results
5     update  $\tilde{v}_l^{(r-1)}, \tilde{t}_l^{(r-1)}$  using (8.6)-(8.8) with loaded data
6   else
7     let  $\tilde{v}_l^{(0)}, \tilde{t}_l^{(0)} \leftarrow$  initial guesses
8   end if
9   if  $b > 1$ 
10    initialize circuit state using  $(r, l, b - 1)$  from task results
11  end if
12  perform simulation
13  save results to  $(r, l, b)$  in task results
14  return task results
15 end function

```

Figure 8.7: Pseudocode for the waveform relaxation iteration.

8.3.3 Parallel Task Scheduling for Networks with Varying Lengths and Differing Terminations

The above scheduling approach works optimally when the simulation of each line takes an equal amount of time. In the case where each line in the system has a different length or is terminated with networks of varying sizes or containing nonlinear elements, this assumption may not apply. To handle this situation, the above algorithm is modified to incorporate a different task scheduling method.

In Table 8.3, it can be seen that the individual tasks comprising the simulation are divided into execution stages. If one of the tasks in a stage takes longer than the others, the other processors will be idle while this task completes. To show how this occurs, the following example is considered.

Illustrative Example 10. In this example, consider $n_L = 3$, $n_P = 2$, $n_I = 2$ and $n_B = 2$. Let the circuit be of a particular structure where the simulation for the first line takes two units of time, while the simulations for the other lines take a single unit of time. The resulting scheduling for this case is shown in Table 8.4.

To avoid this problem, the work tasks are instead organized into a work queue. As each processor executes the algorithm, it will remove the task at the top of the queue and attempt to execute it. If the necessary data to execute a task is not available (as illustrated in Figure 8.4) the processor will wait for a signal from another processor indicating that the data is available. With this method, processors will only be idle when there is no work that can be done. The effect of this is shown in the following example.

Illustrative Example 11. The previous example is reconsidered, where $n_L = 3$, $n_P = 2$, $n_I = 2$ and $n_B = 2$. Once again, the lines are not all of the same length, nor do they have

Table 8.4: Execution process for Illustrative Example 10 demonstrating the PS-PTP approach with non-uniform line circuits

$$(n_I = 2, n_L = 3, n_B = 2, n_P = 2)$$

Time	Task	
	CPU 1	CPU 2
1	(1, 1, 1)	(1, 2, 1)
2		<i>idle</i>
3	(1, 3, 1)	(1, 1, 2)
4	<i>idle</i>	
5	(1, 2, 2)	(1, 3, 2)
6	(2, 1, 1)	(2, 2, 1)
7		<i>idle</i>
8	(2, 3, 1)	(2, 1, 2)
9	<i>idle</i>	
10	(2, 2, 2)	(2, 3, 2)

the same sources or loads. In this case, the effect is that the simulation of the first line takes twice as long as the simulations for the other lines. Using the modified scheduling algorithm, the resulting scheduling is shown in Table 8.5.

As can be seen, the idle time has been eliminated for this case. A pseudocode which implements this modified scheduling algorithm is given in Figure 8.8.

```

1 { Apply transverse partitioning to a coupled interconnect network (with  $n_L$  lines) to separate
2   the network into individual lines. }
3 { Divide the time domain of the solution into  $n_B$  blocks, where  $n_B = n_P$ , and  $n_P$  is the
4   number of processors in the system. }
5 let work queue  $\leftarrow$  a queue of work tasks to be executed, initially empty
6 let task results  $\leftarrow$  a table of simulation results, initially empty,
7   corresponding to each task
8 let semaphore  $\leftarrow$  a signal that is triggered when a work task is complete
9 for  $r = 1$  to  $n_I$  (the number of iterations needed)
10   for  $b = 1$  to  $n_B$ 
11     for  $l = 1$  to  $n_L$ 
12       append task( $r, l, b$ ) to end of work queue
13     end for
14   end for
15 end for
16 while length of work queue  $> 0$ 
17   for  $p = 1$  to  $n_P$ 
18     if length of work queue  $> 0$ 
19       let current task  $\leftarrow$  head of work queue
20       remove head of work queue
21       on processor  $p$  in background do
22         while call task_flow_verify(current task, task results) = false
23           wait for semaphore
24         end while
25         let task results  $\leftarrow$  call perform_wr_iteration(current task, task results)
26         trigger semaphore
27       end do
28     end if
29   end for
30 end while

```

Figure 8.8: Pseudocode for the proposed PS-PTP algorithm with modified scheduling for networks with varying line lengths and differing terminations.

Table 8.5: Execution process for Illustrative Example 11 demonstrating the PS-PTP approach with varying line circuits and modified scheduling algorithm

$$(n_I = 2, n_L = 3, n_B = 2, n_P = 2)$$

Time	Task	
	CPU 1	CPU 2
1	(1, 1, 1)	(1, 2, 1)
2		(1, 3, 1)
3	(1, 1, 2)	(1, 2, 2)
4		(1, 3, 2)
5	(2, 1, 1)	(2, 2, 1)
6		(2, 3, 1)
7	(2, 1, 2)	(2, 2, 2)
8		(2, 3, 2)

8.4 Implementation Details and Considerations

In this section, specific implementation details are considered. Section 8.4.1 provides further efficient implementation details when uniform lumped segmentation is used to model the transmission lines and Section 8.4.2 derives the computational complexity for this case. Section 8.4.3 analyses the effect that the system architecture has on the efficiency of the proposed algorithm.

8.4.1 Additional Partitioning for Uniform Lumped Segmentation

In the previous sections, the original n_L coupled line simulation problem has been decomposed into an iterative two-stage process. The first stage of each iteration is the simulation of n_L single lines in parallel, and the second stage consists of calculating the values for sources $\tilde{v}_j^{(r)}(d, t)$ and $\tilde{i}_j^{(r)}(d, t)$, for the next iteration. For certain transmission line modelling methods, the second stage of this process can be further partitioned. In the specific case where uniform lumped segmentation is used, the second stage is performed by the following

Figure 8.9: Uniform lumped segmentation model with relaxation sources, for the m^{th} segment.

set of equations:

$$\tilde{v}_{j,m}^{(r)}(t) = - \sum_{i=1}^{n_L} \left(\frac{L_{ij}}{L_{ii}} v_{L,i,m}^{(r)}(t) + R_{ij} \Delta x i_{L,i,m}^{(r)}(t) \right), \quad (8.12)$$

$$\tilde{i}_{j,m}^{(r)}(t) = - \sum_{i=1}^{n_L} \left(\frac{C_{ij}}{C_{ii}} i_{C,i,m}^{(r)}(t) + G_{ij} \Delta x v_{C,i,m}^{(r)}(t) \right), \quad (8.13)$$

where m identifies an individual segment corresponding to the model in Figure 8.9, Δx is the physical length covered by each segment, $\tilde{v}_{j,m}^{(r)}(t)$ and $\tilde{i}_{j,m}^{(r)}(t)$ are the values of the relaxation sources in the m^{th} segment of the j^{th} line to be used during the $(r+1)^{\text{th}}$ iteration, $v_{L,i,m}^{(r)}(t)$ and $i_{L,i,m}^{(r)}(t)$ are the voltage across and the current through the inductor element in the m^{th} segment of the i^{th} line measured during the r^{th} iteration, and $v_{C,i,m}^{(r)}(t)$ and $i_{C,i,m}^{(r)}(t)$ are the corresponding values for the capacitor.

Equations (8.12) and (8.13) can be reformulated as a pair of matrix-vector multiplications with dense matrices in $\mathbb{R}^{n_L \times 2n_L}$. These matrices contain constant values derived from the per-unit-length parameters of the lines and thus need only be formulated once. This multiplication is repeated for each segment in the model and for each time point. The cost of this operation can be reduced by taking advantage of one of the many highly efficient implementations that are available. Examples include the automatically optimized

version provided by the ATLAS BLAS [47], or a manually optimized version provided by a specific platform vendor.

An additional advantage of formulating this stage as a matrix-vector multiplication is that, it allows for a clear parallel partitioning scheme. Any matrix-vector multiplication can be performed in parallel with a row-wise partitioning of the matrix. To match the simulation stage of the algorithm, the matrix is partitioned into n_L sections, each corresponding to the relaxation sources for a specific line. As both stages of the algorithm now have the same number of partitions, they can be scheduled together, simplifying the parallel implementation.

8.4.2 Computational Complexity with Uniform Lumped Segmentation

The coupled line simulation is known to have a computational complexity of $O(n_L^\beta)$, $3 \leq \beta \leq 4$. For the proposed PS-PTP partitioning method to be an effective alternative for handling a higher number of coupled lines (higher n_L), the corresponding computational complexity must scale better than the above (while using more processors). For the first stage of the simulation, n_L individual lines are simulated. The cost of this will be n_L times the cost of simulating a single line. Therefore, the complexity of n_L independent line simulations is $O(n_L)$. When this is done in parallel on n_p processors using the proposed PS-PTP partitioning scheme, the complexity will be $O(n_L n_p^{-1})$. The actual computational cost of the simulations will be equal to $k_1 n_L n_p^{-1}$, where k_1 is a scaling coefficient.

As described earlier, the second stage (updating the relaxation sources) involves a series of n_K multiplications of a $2n_L \times n_L$ matrix by a $n_L \times 1$ vector, where n_K is the number of lumped sections used in the transmission line model. The cost of multiplying a matrix in

$\mathbb{R}^{f \times g}$ with a vector in $\mathbb{R}^g$ is known to be $O(fg)$ for a standard implementation. Therefore, this gives a computational complexity of $O(2n_K n_L^2)$. As $2n_K$ is a constant in this context, the complexity is $O(n_L^2)$ for this stage. When executed in parallel, the complexity becomes $O(n_L^2 n_P^{-1})$. The associated computational cost for this stage will be $k_2 n_L^2 n_P^{-1}$, where k_2 is a scaling coefficient.

Using the above observations, the following lemma can be defined to quantify the associated computational complexity.

Lemma 7. *When uniform lumped segmentation is used, the computational complexity of the proposed PS-PTP algorithm is $O(n_L n_P^{-1})$.*

Proof. It is known that the upper bound of the actual execution time will be proportional to the complexity. As the complexity of the simulation stage is $O(n_L n_P^{-1})$ and the complexity of the source update stage is $O(n_L^2 n_P^{-1})$, the execution time of the proposed algorithm will be equal to $n_I n_B (k_1 n_L + k_2 n_L^2) n_P^{-1}$. As the solution of a system of equations (the simulation) is generally a more costly operation than a matrix-vector multiplication for small matrices (the source update process), for all practical purposes $k_1 n_L > k_2 n_L^2$, so the simulation cost will dominate the overall cost. Figure 8.10 shows the relative cost of these two components for a coupled interconnect network with a varying number of lines and with single linear element terminations at the line ends. As is seen, the computational cost corresponding to the source evaluation stage is small compared to the cost of the simulation stage. For more complex circuits, this difference increases further. Therefore, the main contributing factor to the overall complexity of the proposed algorithm is $O(n_L n_P^{-1})$. $\square$

The circuit described in the above proof (Figure 8.10) was also used to compare the performance of the proposed algorithm and a full coupled-line simulation. Figure 8.11 shows the results. As is clearly seen, the computational cost of the proposed algorithm

Figure 8.10: Comparison of the computational cost of waveform relaxation iterations versus the number of lines (on a single CPU).

scales well as the number of lines increases, whereas the full coupled-line simulation quickly becomes impractical.

Figure 8.11: Comparison of the computational cost for a full coupled line simulation and the proposed PS-PTP approach.

8.4.3 System Architecture Considerations

In parallel processing platforms, the efficient management of data is a major concern for algorithms where a relatively small amount of processing is performed on a relatively large set of data. This is due to the fact that the modern CPUs are capable of processing data at a much higher rate than can be sustained by the memory system. This problem is aggravated in multi-core/multi-processor systems as the processors can easily be starved of data due to the fact that they can process the data faster; however, they may have to wait for further scheduling/synchronization due to the memory constraints. To analyse these concerns, the system architecture must be examined. In this section, the two popular memory architectures are reviewed and their effects on the implementation of the proposed algorithm are investigated.

The two most common multi-processor architectures used in PCs are known as Uniform Memory Access (UMA), often used by Intel, and Cache-Coherent Non-Uniform Memory

Access (ccNUMA), used by AMD and the newest Intel systems. An overview of these architectures has been considered previously, in Section 2.4.1.

To compare the performance on two architectures, the proposed algorithm was run on an Intel Xeon UMA system and an AMD Opteron ccNUMA system. The CPU time used by each thread was summed and this is referred to as *CPU work* (this concept was reviewed briefly in Section 2.4.2), and is used to compare the performances of the proposed algorithms.

Note that the proposed algorithm consists of a two-stage analysis process. The first part is a set of circuit simulations. In this part, no communications are needed among processors. Each processor simulates its own assigned circuit, so this is no different than multiple standard circuit simulations occurring in parallel. The only requirement in this case is memory bandwidth: as more tasks operate in parallel, more bandwidth is required. For this stage, it is expected that the ccNUMA architecture will show an advantage as its memory bandwidth is higher.

Figure 8.12 shows the CPU work scaling graph for the simulation stage of the proposed algorithm. As expected, the work done by UMA architecture increases with the number of CPUs. This is because the shared bus becomes more heavily loaded as more CPUs are active at once. For the ccNUMA architecture, the work decreases with the number of CPUs. This indicates that limited communication is being done and that the memory bandwidth is sufficient for this task.

The second stage of the proposed algorithm, the relaxation source updating, differs from the first stage. In this case, the communication requirements are high. The calculations performed here require the results from the simulations performed by each CPU, so they must be transferred between the physical CPUs. Therefore, it is not expected that ccNUMA

Figure 8.12: Comparison of CPU work scaling for the simulation stage.

will provide any benefits. Figure 8.13 shows the measurement results. As seen, UMA and ccNUMA architectures show practically similar results.

The conclusion drawn from these experiments is that when communication between threads is minimal, a ccNUMA architecture can provide additional memory bandwidth to the individual tasks to improve scaling and overall speedup. It was shown in Section 8.4.1 that the source update stage has significantly lower CPU cost compared to the simulation stage of the proposed algorithm. Figure 8.14 confirms this, showing an overall CPU work decrease for the ccNUMA architecture due to the simulation stage work decrease, despite the increase in the work in the source update stage. Therefore, the proposed algorithm provides better scaling and speedup on ccNUMA architectures.

Figure 8.13: Comparison of CPU work scaling for the source update stage.

Figure 8.14: Comparison of CPU work scaling for the overall simulation (including source update cost).

Figure 8.15: Example coupled line network.

8.5 Computational Results

In this section, two examples are considered to validate the accuracy and efficiency of the proposed parallel algorithm. The first experiment demonstrates the CPU scalability of the algorithm. The second example considers the application of the proposed parallel algorithm to the practical but CPU-intensive case of performing an eye diagram simulation for a signal integrity analysis.

8.5.1 Processor Scalability Example

To validate the accuracy and efficiency of the proposed parallel algorithm, it was tested on a 5-cm 16-line coupled interconnect (modeled using uniform lumped segmentation [36]) circuit, shown in Figure 8.15.

The implementation was done using the OpenMP multithreading extensions to C++, coupled with a custom circuit simulator. The simulator used the KLU package for solving the circuit equations [2]. A custom simulator was used so that direct internal access to the simulation results would be available for calculating the sources during subsequent iterations, avoiding external communications with the simulator. The implementation was run on a machine with dual quad-core AMD Opteron 2344HE processors (8 CPUs) using the ccNUMA architecture.

To evaluate the performance of the parallel algorithm, it is first considered with the proposed parallel physical partitioning algorithm (PS-PP, Section 8.3.1). Figure 8.16 compares the time domain response at the far end of the aggressor line ($V_{out,1}$). The solid line is the response of the proposed parallel algorithm, and the line with vertical thin dashes is the response of the sequential algorithm (which corresponds to the single processor implementation of WR+TP [13]). Figure 8.17 shows the response at the near end of the victim line ($V_{out,2}$). As seen, the proposed and sequential algorithm results match accurately. Convergence was observed after 4 iterations for both algorithms.

Figure 8.18 shows the speedup characteristics for this experiment. The dotted line represents the ideal case of speedup. The dash-dot-dashed line represents the speedup using the PS-PP approach. It can be seen that the speedup increases unevenly. This is because the partitioning gives a maximum parallelism of 16, which does not divide evenly among the available processors in all cases. This validates the discussion in Section 8.3.1.

Next, the parallel algorithm with the proposed physical and time-domain partitioning (PS-PTP, Section 8.3.2) was tested. In this case, the time domain span was partitioned into a number of blocks matching the number of processors used during the execution of each stage. The corresponding transient results (shown using dotted lines) are indistinguish-

Figure 8.16: Simulation results for the far end of aggressor line ($V_{out,1}$).

Figure 8.17: Simulation results for the near end of victim line ($V_{out,1}$).

Figure 8.18: Speedup analysis for the proposed parallel algorithms.

able from earlier experiments, as shown in Figures 8.16 and 8.17. With this additional partitioning, the parallelism significantly increased. Figure 8.18 shows the speedup of this approach (using the solid line). Clearly, improved speedup is seen even for the cases where the number of lines is not a multiple of the number of processors, compared to the PS-PP approach.

8.5.2 Long Coupled-Line Eye Diagram

This example is considered to demonstrate the advantage of using the proposed algorithm for practical but CPU-intensive applications. For this purpose, the case of generating an eye diagram for a signal integrity analysis of a long, large coupled interconnect system is considered.

The interconnect system consisted of 32 coupled lines of 50 cm length. To generate the eye diagram, a 50 Ω , 200 MHz random bit source with 0.3 ns risetime is connected to the near end of each line. The far ends are terminated with 1 pF capacitors. For this analysis,

the proposed parallel time-domain partitioning method was adopted. The analysis is run over 1000 cycles ($5\ \mu\text{s}$) and the far-end signal for each period is superimposed on a graph for each line. Representative eye diagram graphs are shown in Figures 8.19-8.20.

If the time-domain partitioning method was not adopted, since the simulation has to run over 1000 cycles, the memory requirements for keeping the simulation results for each iteration for the entire time span would have been too demanding. On the other hand, dividing the simulation time span into several time blocks using the proposed PS-PTP approach, memory can be efficiently handled since the intermediate simulation results can be discarded as soon as the corresponding previous time block simulation has been completed. In addition, it is to be noted that the PS-PTP approach enhances the parallelism of the task scheduling, improving speedup.

The simulation was run on a system with dual quad-core AMD Opteron 2344HE processors (8 CPUs) with 8 GB of memory. The total *CPU work* accounting for all the CPUs involved was 8175 CPU-seconds (6344 CPU-seconds for the line simulations and 1831 CPU-seconds for updating the relaxation sources). The proposed parallel analysis required a total of 1039 seconds of wall time. On the other hand, when the sequential WR+TP method [13] was run on a single CPU, the simulation failed due to insufficient memory. When this sequential algorithm was run with the time domain partitioning on a single CPU it required an analysis time of 6668 seconds. The observed parallel speedup is therefore 6.4 (on a parallel platform of 8 CPUs).

Figure 8.19: Simulation results for the far end of line 1.

Figure 8.20: Simulation results for the far end of line 17.

8.5.3 Network with Varying Line Lengths and Terminations

To demonstrate the effectiveness of the algorithm while handling circuits with varying line lengths and differing terminations, the example circuit shown in Figure 8.21 was studied. In this circuit, there are 16 coupled lines, with 2.5 cm lines interleaved with 5 cm lines. The short lines are driven by a linear source and terminated with a diode load. The longer lines have a capacitive termination at the far end and a diode termination at the near end. In addition, line 8 is driven by a CMOS inverter instead of a linear source. The transmission lines have the same per-unit-length properties as the lines used in Example A and are again modeled using uniform lumped segmentation [36].

For this example, the modified scheduling outlined in Figure 8.8 was used. The implementation was done using the same custom in-house circuit simulator core used previously, with POSIX threads used to manage the parallel tasks. As before, the implementation was run on a machine with dual quad-core AMD Opteron 2344HE processors (8 CPUs) using the ccNUMA architecture.

The scalability of the proposed algorithm for this example is shown in Table 8.6. This table shows the amount of CPU work done, as well as the wall time taken to perform the simulation. The ratio of these numbers gives the average CPU utilization. Any idle time in the simulation due to differences in individual simulation run times and as a consequence of the scheduling algorithm will appear as a utilization below 100%. As can be seen, idle time begins to be seen when 6 CPUs are used, which will impact speed-up slightly. The computational time of this example is relatively short, and for larger examples the impact of idle time will be even less.

Figure 8.22 shows the speedup characteristics for this experiment. The dotted line represents the ideal case of speedup. The solid line shows the results of the proposed

Figure 8.21: Example coupled line network, with varying line lengths and differing terminations.

Table 8.6: Performance of the proposed algorithm for Example C

Number of CPUs	CPU Work (CPU·s)	Wall Time (s)	Utilization
1	583.160	583.850	99.881%
2	510.160	256.118	99.594%
3	474.750	160.126	98.828%
4	447.090	113.547	98.437%
5	429.790	88.768	96.834%
6	420.010	73.924	94.694%
7	388.080	62.575	88.597%
8	400.330	60.613	82.558%

algorithm. In this case, the benefits of the additional memory bandwidth available in the ccNUMA platform can be easily seen, as the speedup is consistently above the ideal speedup. The effect of the previously mentioned idle time can be also seen in this graph in the change in the slope for higher numbers of CPUs.

Lastly, the simulated signals at the far ends of three of the lines are shown in Figures 8.23-8.25. The middle line is a victim line while the other two are aggressors.

Figure 8.22: Speedup analysis for the proposed parallel algorithms for varying line lengths, sources, and loads.

Figure 8.23: Simulation convergence, for the far end of line 6.

Figure 8.24: Simulation convergence, for the far end of line 7.

Figure 8.25: Simulation convergence, for the far end of line 8.

8.6 Summary

In this chapter, a novel parallel algorithm for simulating large coupled interconnect networks has been presented. New schemes based on physical and time-domain partitioning have been developed. In addition, the new algorithm exploits the waveform relaxation techniques to decouple the problem into multiple subproblems which can be executed in parallel. In addition to providing accurate results, it was also shown that the algorithm scales very well on shared-memory computer platforms. For a system of n_L coupled lines, the simulation cost has been reduced from $O(n_L^\beta)$, $3 \leq \beta \leq 4$ to $O(n_L n_p^{-1})$ when n_p processors are available. For very large coupled networks, this leads to considerable CPU savings.

Chapter 9

Conclusions and Future Work

9.1 Summary

In this thesis, a novel algorithm for the solution of partitioned circuits on a parallel architecture is developed. This algorithm does not depend on a certain circuit topology nor is it restricted to a specific class of circuits. Competing algorithms based on the domain decomposition approach have demonstrated issues that limit their scalability to systems containing many processors.

In Chapter 3, the algorithm is first developed as a sequential algorithm to solve a system consisting of partitions without resorting to a domain-decomposition-based approach. Instead, techniques derived from circuit theory are exploited. In Chapter 4, the proposed algorithm is recast as a parallel algorithm, and its performance and scalability are studied. In this chapter, a detailed comparison of the computational complexities between the proposed algorithm and the existing popular technique of domain decomposition was also provided.

Next, Chapter 5 proposed some algorithmic improvements to improve the quality of the partitioning. In addition, some more general proposals to improve performance were

also given.

In Chapter 6, the proposed parallel algorithm was analysed in more detail, this time with a view to optimizing processor utilisation. By more efficiently making use of the processors, even more scalability was obtained. This was demonstrated by reconsidering the circuit examples from the previous chapters.

The major advantages of the proposed parallel algorithm for the simulation of general circuits are summarized by the following:

1. In the process of resolving the effect of the interpartition links, domain decomposition consists of a number of matrix-matrix and matrix-vector multiplications. With node tearing, the matrices involved are permutation and selection matrices, which can be implemented with a much greater efficiency.
2. Both domain decomposition and node tearing have a necessary sequential part of their method: the solution of the $\mathbf{Z}$ matrix. In node tearing, however, it is practical to reorder the border columns so that the factorization and solution of the problem involving this matrix is pipelined. Therefore, the total amount of work that must be done which the other processors in the system are idle is greatly reduced, and scalability benefits as a result. This is especially important when the number of partitions is large.

Chapter 7 explored the extension of this algorithm to the case of hierarchical partitioning. In the current state of the implementation, no improvements were seen by this extension; however, they do open the door to exploring the use of other techniques which may benefit from a hierarchical structure.

Finally, in Chapter 8, the advantages of creating parallel algorithms to handle specific classes of circuits is examined. By exploiting peculiarities in the structures of a specific type

of circuits, additional efficiencies and parallelism can be found, resulting in a simulation implementation that scales even better than one constructed to handle general circuits.

9.2 Future Work

Through the work in the thesis, some directions for future research have become clear. Some suggestions for this future work are as follows:

1. **ccNUMA-optimized scheduler implementation:** the current scheduler implementations performed in Section 4.3 is not fully optimized for the ccNUMA architecture which is becoming the standard for PC architectures. Future work could be done to fully optimize the scheduler for this architecture and to see if significant performance improvements are seen.
2. **Combination with other parallel algorithmic improvements:** The numerous refinements that have been applied to domain decomposition simulations, summarized in Section 2.3.3, could be applied with the node tearing formulation and their respective performance improvements and impacts to scalability could be studied.
3. **Combination with alternative simulation methods:** This approach could also be combined with other simulation techniques, such as high-order integration methods [48], to obtain even higher simulation performance that can scale on parallel systems.
4. **Application to other problem domains:** Many other problems are based on the MNA formulation or other formulations based on sparse matrices representing a

network. These problems include such areas as power and signal integrity analyses, sensitivity analysis, and finite element methods.

5. In addition to these, the initial study of hierarchical methods in Section 7 could be continued. The related conclusions for future work are summarized below:

- (a) Advanced partitioning techniques for the hierarchical mode could be studied. It might be interesting to study the effects of generating a broader tree, if sufficiently large circuit examples are considered. Other ways of improving the partitioning could be studied as well, such as taking into account additional knowledge of the circuit by considering either levels of hierarchy used in the design process, or physical layout information.
- (b) A more detailed analysis of the performance of branch tearing compared to node tearing could be compared, so that an intelligent and automatic decision about which should be used for smaller leaf nodes of the tree. From the current results, it appears branch tearing does not bring any important benefits, but a conclusive verification of this would be interesting.

References

- [1] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, 2nd ed. Boston, MA: Kluwer, 1993.
- [2] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia: SIAM, 2006.
- [3] ———, “Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method,” *ACM Transactions on Mathematical Software*, vol. 30, no. 2, pp. 196–199, June 2004.
- [4] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: Johns Hopkins, 1996.
- [5] N. Frohlich, B. M. Riess, U. A. Wever, and Q. Zheng, “A new approach for parallel simulation of VLSI circuits on a transistor level,” *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, vol. 45, no. 6, pp. 601–613, June 1998.
- [6] P. F. Cox, R. G. Burch, D. E. Hocevar, P. Yang, and B. D. Epler, “Direct circuit simulation algorithms for parallel processing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 6, pp. 714–725, June 1991.

- [7] R. A. Saleh, K. A. Gallivan, M. Chang, I. N. Hajj, D. Smart, and T. N. Trick, "Parallel circuit simulation on supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1915–1931, Dec. 1989.
- [8] H. K. Thornquist, E. R. Keiter, R. J. Hoekstra, D. M. Day, and E. G. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2009) - Digest of Technical Papers*, San Jose, CA, Nov. 2009, pp. 410–417.
- [9] F. Magoulès and V. Savcenco, "Algorithmic requirements for simulation of large network extreme scenarios," PEGASE Consortium, Tech. Rep. D4.1, Apr. 2011.
- [10] G. Kron, *Tensor Analysis of Networks*. New York: Wiley, 1939.
- [11] R. A. Rohrer, "Circuit partitioning simplified," *IEEE Transactions on Circuits and Systems*, vol. 35, no. 1, pp. 2–5, Jan. 1988.
- [12] A. Ruehli and T. A. Johnson, *Circuit Analysis Computing by Waveform Relaxation in Encyclopedia of Electrical and Electronics Engineering*, J. Webster, Ed. New York: Wiley, 1999, vol. 3.
- [13] N. M. Nakhla, A. E. Ruehli, M. S. Nakhla, and R. Achar, "Simulation of coupled interconnects using waveform relaxation and transverse partitioning," *IEEE Transactions on Advanced Packaging*, vol. 29, no. 1, pp. 78–87, Feb. 2006.
- [14] M. Nakhla, R. Achar, D. Paul, and N. Nakhla, "Parallel simulation of general electrical and mixed-domain circuits," US Patent US8 543 360, Sept. 24, 2013.

- [15] D. Paul, M. S. Nakhla, R. Achar, and N. M. Nakhla, "Parallel circuit simulation via binary link formulations (PvB)," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 3, no. 5, pp. 768–782, May 2013.
- [16] D. Paul, R. Achar, M. S. Nakhla, and N. M. Nakhla, "Addressing partitioning issues in parallel circuit simulation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Jan. 2014, available on Early Access.
- [17] D. Paul, N. M. Nakhla, R. Achar, and M. S. Nakhla, "Parallel algorithm for analysis of high-speed interconnects," in *Proc. IEEE Topical Meeting on Electrical Performance of Electronic Packaging (17th EPEP)*, San Jose, CA, Oct. 2008, pp. 191–194.
- [18] —, "Parallel simulation of massively coupled interconnect networks," *IEEE Transactions on Advanced Packaging*, vol. 33, no. 1, pp. 115–127, Feb. 2010.
- [19] G. Karypis. hMETIS. [Online]. Available: <http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview>, Last Accessed: 2014-08-23.
- [20] N. Selvakumaran and G. Karypis, "Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 504–517, Mar. 2006.
- [21] R. Bisseling. Mondriaan partitioner. [Online]. Available: <http://www.staff.science.uu.nl/~bisse101/Mondriaan>, Last Accessed: 2014-08-23.
- [22] B. Vastenhouw and R. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, Mar. 2005.

- [23] S. H. Roosta, *Parallel Processing and Parallel Algorithms*. New York: Springer-Verlag, 2000.
- [24] H. Peng and C.-K. Cheng, “Parallel transistor level circuit simulation using domain decomposition methods,” in *Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*, Yokohama, Japan, Jan. 2009, pp. 397–402.
- [25] X. Chen, Y. Wang, and H. Yang, “NICS LU: An adaptive sparse matrix solver for parallel circuit simulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 261–274, Feb. 2013.
- [26] ———, “Parallel circuit simulation on multi/many-core systems,” in *Proc. IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW 2012)*, Shanghai, China, May 2012, pp. 2530–2533.
- [27] W. Dong, P. Li, and X. Ye, “WavePipe: Parallel transient simulation of analog and digital circuits on multi-core shared-memory machines,” in *Proc. 45th ACM/IEEE Design Automation Conference (DAC 2008)*, June 2008, pp. 238–243.
- [28] X. Ye, W. Dong, P. Li, and S. Nassif, “MAPS: Multi-algorithm parallel circuit simulation,” in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2008)*, San Jose, CA, Nov. 2008, pp. 73–78.
- [29] ———, “Hierarchical multialgorithm parallel circuit simulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 1, pp. 45–58, Jan. 2011.
- [30] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Harlow, UK: Addison-Wesley, 2003.

- [31] F. Brglez and H. Fujiwara, "A neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN," in *Proc. IEEE Int. Symp. Circuits and Systems*, June 1985, pp. 695–698.
- [32] M. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design and Test of Computers*, vol. 16, no. 3, pp. 72–80, Aug. 1999.
- [33] P. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software*, vol. 30, no. 3, pp. 381–388, Sept. 2004.
- [34] futex(7) manual page. [Online]. Available: <http://www.kernel.org/doc/man-pages/online/pages/man7/futex.7.html>, Last Accessed: 2014-08-23.
- [35] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Aug. 2012.
- [36] R. Achar and M. S. Nakhla, "Simulation of high-speed interconnects," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 693–728, May 2001.
- [37] A. Dounavis, R. Achar, and M. Nakhla, "Efficient passive circuit models for distributed networks with frequency-dependent parameters," *IEEE Transactions on Advanced Packaging*, vol. 23, no. 3, pp. 382–392, Aug. 2000.
- [38] A. Dounavis, N. Nakhla, R. Achar, and M. Nakhla, "Delay extraction and passive macromodeling of lossy coupled transmission lines," in *Proc. IEEE 12th Topical Meeting on Electrical Performance of Electronic Packaging*, Princeton, NJ, Oct. 2003, pp. 295–298.

- [39] Q. Chu, Y. Lau, and F. Y. Chang, "Transient analysis of microwave active circuits based on time domain characteristic models," *IEEE Transactions on Microwave Theory and Techniques*, vol. 48, no. 8, pp. 1097–1104, Aug. 1998.
- [40] D. Kuznetsov and J. E. Schutt-Ainé, "Optimal transient simulation of transmission lines," *IEEE Transactions on Circuits and Systems*, vol. 43, no. 2, pp. 110–121, Feb. 1996.
- [41] A. Woo, T. Yioultsis, and A. C. Cangellaris, "BestFit: a SPICE-compatible model for efficient, passive, broadband transmission-line analysis of dispersive interconnects," in *Proc. IEEE Topical Meeting on Electrical Performance of Electronic Packaging (12th EPEP)*, Princeton, NJ, Oct. 2003, pp. 255–258.
- [42] Q. Yu, J. M. L. Wang, and E. S. Kuh, "Passive multipoint moment-matching model order reduction algorithm on multiport distributed interconnect networks," *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, vol. 46, no. 1, pp. 140–160, Jan. 1999.
- [43] A. Dounavis, R. Achar, and M. Nakhla, "A general class of passive macromodels for lossy multiconductor transmission lines," *IEEE Transactions on Microwave Theory and Techniques*, vol. 49, no. 10, pp. 1686–1696, Oct. 2001.
- [44] C. R. Paul, *Analysis of Multiconductor Transmission Lines*. New York: Wiley, 1994.
- [45] E. Lelarasmee, A. Ruehli, and A. L. Sangiovanni-Vincentelli, "The waveform relaxation method for the time-domain analysis of large scale integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-1, no. 8, pp. 131–145, Aug. 1982.

- [46] N. Nakhla, A. E. Ruehli, M. S. Nakhla, R. Achar, and C. Chen, “Waveform relaxation techniques for simulation of coupled interconnects with frequency-dependent parameters,” *IEEE Transactions on Advanced Packaging*, vol. 30, no. 2, pp. 257–269, May 2007.
- [47] R. C. Whaley, A. Petitet, and J. J. Dongarra, “Automated empirical optimization of software and the ATLAS project,” Sept. 2000. [Online]. Available: <http://www.netlib.org/lapack/lawns/lawn147.ps>, Last Accessed: 2014-08-23.
- [48] E. Gad, M. Nakhla, R. Achar, and Y. Zhou, “A-stable and L -stable high-order integration methods for solving stiff differential equations,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 9, pp. 1359–1372, Sept. 2009.